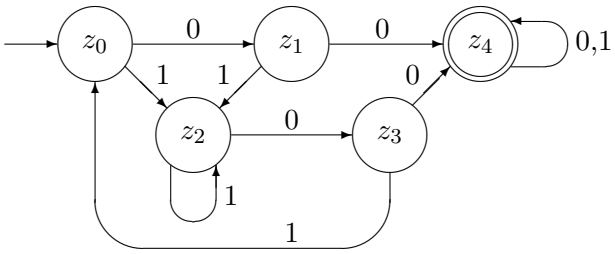
Heiko Körner

Einführung

in die

Theoretische Informatik

Automatentheorie, formale Sprachen, Berechenbarkeit und elementare Komplexitätstheorie



Skript zur Vorlesung Theoretische Informatik I/II

IWI Fakultät für Informatik und Wirtschaftsinformatik

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES
Näher dran.

Vorwort

Das vorliegende Skript soll die Vorlesung Theoretische Informatik I und II unterstützend begleiten, die ich an der Hochschule Karlsruhe laufend anbiete.

Im Kern behandelt die theoretische Informatik nur eine zentrale Frage:

Können bestimmte Problemstellungen überhaupt von Computern gelöst werden — und wenn ja, wie effizient?

Bestimmte Aufgaben sind für Computer offenbar sehr einfach. Denken Sie an das Addieren oder Sortieren von Millionen von Zahlen, was selbst ein preiswerter Rechner in Sekunden erledigen kann. Andere Problemstellungen scheinen dagegen viel schwerer zu sein.

Stellen Sie sich vor, Sie sollen eine Rundreise mit dem Auto durch die 15 größten Städte Deutschlands planen, so dass Sie keine Stadt doppelt besuchen. Sie können irgendwo starten und dann eine der verbleibenden 14 Städte aufsuchen. Von dort aus geht es zu einer der restlichen 13 Städte weiter, usw. Schließlich bleibt nur noch eine Stadt übrig, von der Sie wieder zur ersten Stadt zurückkehren. Unter den mehr als 80 Milliarden Möglichkeiten (!) sollen Sie eine Rundreise mit dem kürzesten Gesamtweg heraussuchen. Der Computer kann Ihnen mit einem geeigneten Programm sicher helfen, doch prinzipiell ist bis heute kein „wesentlich“ besseres Verfahren bekannt, als (fast) alle Möglichkeiten durchzuspielen. Wenn wir die Aufgabe auf eine kürzeste Rundreise durch alle Städte mit mehr als 20.000 Einwohnern ausdehnen (davon gibt es knapp 1.000 in Deutschland), so explodiert die Anzahl der Möglichkeiten weiter, und die Aufgabe scheint hoffnungslos zu sein.

Kern der *Komplexitätstheorie* bildet die Untersuchung solcher und anderer Problemstellungen auf ihre „Schwierigkeit“. Bei hartnäckigen Problemen kann es notwendig werden, auf eine bestmögliche Lösung zu verzichten, wenn im Gegenzug dadurch eine einfachere Berechnung möglich wird. Woanders ist es dagegen gerade gewünscht, dass sich bestimm-

te Aufgaben nur mit unvertretbar großem Aufwand lösen lassen, z.B. beim Entschlüsseln von Passwörtern. Wir werden einen Blick auf die Komplexitätstheorie am Ende des zweiten Semesters werfen.

In der *Berechenbarkeitstheorie* lassen wir den Aufwandsaspekt fallen und befassen uns mit der Frage, welche Probleme überhaupt lösbar sind. Die dafür aufgewendete Zeit ist unerheblich, solange nur in endlicher Zeit eine Lösung gefunden wird. Betrachten Sie dazu noch einmal das obige Rundreisenproblem. Wenn man für alle Rundreisen die Streckenlänge bestimmt und eine Reise mit der kürzesten Länge auswählt, so dauert dies wegen der Vielzahl von Möglichkeiten zwar lange, aber man bekommt immerhin in endlicher Zeit eine Antwort. Dies ist nicht bei allen Problemstellungen der Fall.

Vielleicht haben Sie z.B. schon in einer Programmiersprache wie Java, C oder Pascal Erfahrungen gesammelt und in eigenen oder fremden Programmen zeitaufwendig nach Fehlern gesucht. Man könnte auf die Idee kommen, sich hier und da von einem Computer helfen zu lassen, der diverse Aspekte automatisch überprüft. Leider aber stösst man bei diesem Ansatz schnell an prinzipielle Grenzen. Wir werden z.B. beweisen, dass kein noch so leistungsfähiger Computer im Allgemeinen prüfen kann, ob eine bestimmte Programmvariable korrekt initialisiert wird oder das Programm nach seinem Start überhaupt wieder anhält. Genauso wenig kann ein Computer den Wahrheitsgehalt einer beliebigen mathematischen Formel überprüfen. All diese Probleme sind *unentscheidbar*. Unentscheidbare Probleme werden ebenfalls im zweiten Semester besprochen.

Programmiersprachen wie Java sind mit umfangreichen Befehlssätzen und Bibliotheken ausgestattet. Wenn man nun zeigen möchte, dass ein Java–Programm die obigen unentscheidbaren Probleme nicht zu lösen vermag, so wäre die Beweisführung sehr aufwendig, denn es müsste gegen die kompletten Möglichkeiten dieser Sprache argumentiert werden. Man abstrahiert in der theoretischen Informatik deshalb bewusst von den heutigen Computern und arbeitet mit wesentlich einfacheren Verarbeitungsmodellen, die z.B. nur über einen kleinen Befehlssatz verfügen. Diese in der *Automatentheorie* eingeführten Modelle entsprechen Computern mit bestimmten Eigenschaften. Die prinzipiellen Fähigkeiten eines Computers werden hierdurch besser veranschaulicht. Es ist z.B. viel leichter zu sehen, was man mit einer begrenzten Menge an Hauptspeicher berechnen kann und was nicht. Die Automatentheorie bildet den Kernpunkt der Vorlesung im ersten Semester.

Bevor wir aber starten, werden in einer Einführung noch einige elementare Konzepte besprochen, wie z.B. Grundlagen zur Mengenlehre, bestimmte mathematische Schreibweisen oder auch Beweistechniken.

Hinweise zur Vorlesung

Theoretische Informatik ist ein interessantes, für viele Studierende aber auch ein sehr schweres Fach. Dies liegt an den vielen formalen Methoden, die für die meisten Studierende neu sind. In vielen anderen Vorlesungen und Laboren sieht und experimentiert man damit, was sich alles mit modernen Computern realisieren lässt. In der theoreti-

schen Informatik werden dagegen viele Negativresultate aufgestellt, was man also alles mit Computern *nicht* machen kann. Dies lässt sich natürlich nicht einfach ausprobieren, sondern man muss mit logischen Argumenten entsprechende Beweise herleiten, was anfangs vielen nicht leicht fällt.

Ich empfehle Ihnen, sich dieses Skript anfangs einmal auszudrucken und während der Vorlesung aufgeschlagen vor sich liegen zu haben. Trotzdem sollten Sie die Vorlesungsinhalte im Wesentlichen nur vorne an der Tafel mit verfolgen, d.h. Sie sollten nicht gleichzeitig im Skript mitlesen. Nutzen Sie das Skript in der Vorlesung zunächst nur für eigene Hinweise oder Ergänzungen, die Sie stichpunktartig am Rand notieren. Für Tipps zu Verbesserungsvorschlägen, gefundenen Fehlern usw. bin ich jederzeit dankbar.

Nachmittags oder abends sollten Sie das Gehörte am Skript dann nochmals nacharbeiten. Dies ist sehr wichtig, denn jede Vorlesung setzt den *gesamten* bereits vermittelten Stoff voraus, und ohne Nacharbeitung wird man sehr schnell „abgehängt“ — keine gute Voraussetzung, um die abschließende Klausur mit einer guten Leistung zu bestehen. Erfahrungsgemäß haben auch die Versuche wenig Erfolg, sich erst wenige Wochen vor dem Klausurtermin mit dem Lernstoff auseinander zu setzen.

Eine Verfolgung des Skriptes am Bildschirm eines Notebooks hat den Nachteil, dass Sie keine Notizen während der Vorlesung anbringen können. Gleiches gilt auch für Touchscreens, Tablets usw., auf denen man zwar mitschreiben, aber häufig doch erst die richtige Textstelle herscrollen und aufzoomen muss. Dabei verliert man schnell den Anschluss zu dem diskutierten Stoff an der Tafel, weil man bestimmte Details gerade nicht mitbekommen hat. Aus dem gleichen Grund sollten Sie immer möglichst pünktlich zu den Vorlesungen erscheinen.

Jede Woche wird ein Übungsblatt mit mehreren Aufgaben ausgeteilt, die Sie am besten innerhalb einer Dreier– oder Vierergruppe unter sich aufteilen, zusammen besprechen und dann gegebenenfalls zur Korrektur abgeben. Versuchen Sie nicht, alle Aufgaben allein zu lösen — dies ist allein vom Zeitaufwand her nicht machbar. Die Lösungen zu den Übungsblättern werden in den Tutorien besprochen und mit zeitlicher Verzögerung auch online gestellt. Die Bearbeitung der Übungsblätter ist keine Pflicht, aber im Hinblick auf die Klausur sehr sinnvoll. In den Tutorien können Sie zudem Fragen stellen, genauso wie vor, während oder nach der Vorlesung.

Die Klausuren am Ende des Semesters bestehen aus einer Zusammenstellung verschiedener Aufgaben, die in ähnlicher Form auch auf den Übungsblättern zu finden sind. Sobald das Online–Notensystem die Ergebnisse bereit hält, können Sie auf Wunsch Ihre Klausur zu den üblichen Öffnungszeiten im Sekretariat einsehen. Dort liegt im Allgemeinen auch eine Musterlösung aus. Wenn Sie darüber hinaus noch Fragen zu Ihrer Klausur haben, können Sie sich selbstverständlich jederzeit an mich wenden.

In der letzten Vorlesungswoche wird eine Probeklausur angeboten. Zu allen Klausuren dürfen Sie ein von beiden Seiten beschriftetes DIN A4–Blatt mitnehmen, welches Sie *mit Ihrer eigenen Handschrift* erstellt haben müssen.

Inhaltsverzeichnis

1

Mathematische Grundlagen

1.1. Aussagenlogik

In der theoretischen Informatik arbeiten wir viel mit wahren und falschen Aussagen („$2+2=4$“, „6 ist eine Primzahl“). Die beiden Wörter *wahr* (W) und *falsch* (F) werden häufig auch als die *booleschen*[1] *Werte* bezeichnet.

Boolesche Werte kann man manipulieren oder mit anderen booleschen Werten verknüpfen. Die resultierenden booleschen Werte hängen von der gewählten Operation und den vorherigen Werten ab. Die einfachste solche Operation ist die *Negation* (Verneinung), symbolisiert durch das Zeichen $\neg$. Sie dreht den Wahrheitsgehalt eines Wertes A um, d.h. wenn A vorher wahr war, so ist $\neg A$ nun falsch und umgekehrt. Wir können diesen Zusammenhang durch eine *Wahrheitstabelle* oder *Wahrheitstafel* darstellen. Links stehen die beiden möglichen Eingangswerte von A, rechts in der Ergebnisspalte die beiden resultieren Ergebniswerte von $\neg A$:

A	$\neg A$
W	F
F	W

Offenbar gilt $\neg(\neg A) = A$.

Die beiden nächsten (und auch alle weiteren) Verknüpfungen operieren jeweils auf zwei booleschen Werten A und B:

- Die *Konjunktion* (Und–Verknüpfung, als Zeichen $\wedge$) ist nur dann wahr, wenn beide Ausgangswerte wahr sind.

[1] George Boole, *1815 Lincoln, Vereinigtes Königreich, †1864 Ballintemple, Irland, britischer Mathematiker und Philosoph, ab 1848 Mathematikprofessor am Queens College in Cork.

- Die *Disjunktion* (Oder–Verknüpfung, als Zeichen $\vee$) ist dagegen schon dann wahr, wenn mindestens einer der beiden Ausgangswerte wahr ist.

Beachten Sie, dass es jetzt insgesamt vier Möglichkeiten für die Belegungen von A und B gibt:

A	B	$A \wedge B$
W	W	W
W	F	F
F	W	F
F	F	F

A	B	$A \vee B$
W	W	W
W	F	W
F	W	W
F	F	F

1.1.1 Beispiel Angenommen, A bezeichnet die Aussage „6 ist eine Primzahl“ sowie B die Aussage „9 ist eine Quadratzahl“. Dann ist A falsch und B richtig. Folglich ist $\neg A$ richtig („6 ist keine Primzahl“), $A \wedge B$ falsch („6 ist eine Primzahl und 9 ist eine Quadratzahl“) und $A \vee B$ richtig („6 ist eine Primzahl oder 9 ist eine Quadratzahl“).

Zwei weitere Verknüpfungen sind ebenfalls sehr wichtig. Die *Implikation* (Folgerung, als Zeichen $\Rightarrow$) drückt aus, dass man aus einer wahren Aussage A nur andere wahre Aussagen B folgern kann:

A	B	$A \Rightarrow B$
W	W	W
W	F	F
F	W	W
F	F	W

Die *Äquivalenz* (in Zeichen $\Leftrightarrow$) ist wahr, wenn der Austausch von A durch B nichts an dem Wahrheitsgehalt ändert. Beide booleschen Werte müssen also gleich sein:

A	B	$A \Leftrightarrow B$
W	W	W
W	F	F
F	W	F
F	F	W

1.1.2 Beispiel In Fortführung des obigen Beispiels 1.1.1 ist $A \Rightarrow B$ richtig („wenn 6 eine Primzahl ist, dann ist 9 eine Quadratzahl“). Insbesondere kann man aus einer falschen Aussage durchaus etwas Richtiges folgern. $A \Leftrightarrow B$ ist dagegen falsch („6 ist genau dann eine Primzahl, wenn 9 eine Quadratzahl ist“).

1.1.3 Satz Es gelten folgenden Regeln:

- $A \vee B = B \vee A$ (*Kommutativgesetze*)
  $A \wedge B = B \wedge A$

- $A \vee (B \vee C) = (A \vee B) \vee C$ (*Assoziativgesetze*)
  $A \wedge (B \wedge C) = (A \wedge B) \wedge C$
- $A \vee (A \wedge B) = A$ (*Absorptionsgesetze*)
  $A \wedge (A \vee B) = A$
- $(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$ (*Distributivgesetze*)
  $(A \vee B) \wedge C = (A \wedge C) \vee (B \wedge C)$
- $\neg(A \vee B) = \neg A \wedge \neg B$ (*Regeln von DeMorgan*)
  $\neg(A \wedge B) = \neg A \vee \neg B$

Hinweis: Bei Aussagen mit mehreren Operatoren wie z.B. den Regeln von DeMorgan[1] gibt es — sofern Klammern nicht etwas anderes vorgeben — bestimmte *Ausführungsprioritäten*. Zuerst werden immer die Negationen ausgewertet, gefolgt von den Konjunktionen und Disjunktionen. Am Ende werden Implikationen, Äquivalenzen und Gleichheiten ausgerechnet. Es schadet aber gegebenenfalls nie, zusätzliche Klammern zu setzen.

Beweis: Wir zeigen exemplarisch die erste Regel des Distributivgesetzes, indem wir alle Verknüpfungen nach und nach auswerten. Wir beginnen mit den einfachen Verknüpfungen, die am weitesten innen geklammert sind. Jede ausgewertete Verknüpfung ergibt so eine separate Spalte in der nachfolgenden Wahrheitstabelle:

A	B	C	$A \wedge B$	$A \vee C$	$B \vee C$
W	W	W	W	W	W
W	W	F	W	W	W
W	F	W	F	W	W
W	F	F	F	W	F
F	W	W	F	W	W
F	W	F	F	F	W
F	F	W	F	W	W
F	F	F	F	F	F

Mit Hilfe der neuen Spalten können wir auch die komplexeren Verknüpfungen links und rechts des Gleichheitszeichens auswerten:

A	B	C	$A \wedge B$	$A \vee C$	$B \vee C$	$(A \wedge B) \vee C$	$(A \vee C) \wedge (B \vee C)$
W	W	W	W	W	W	W	W
W	W	F	W	W	W	W	W
W	F	W	F	W	W	W	W
W	F	F	F	W	F	F	F
F	W	W	F	W	W	W	W
F	W	F	F	F	W	F	F
F	F	W	F	W	W	W	W
F	F	F	F	F	F	F	F

[1] Augustus De Morgan, *1806 Madurai, Indien, †1871 London, britischer Mathematiker. De Morgan gilt wie Boole als einer der Begründer der formalen Logik.

Wir sehen nun, dass die beiden letzten Spalten identisch sind. Die behauptete Gleichung ist also immer zutreffend, ganz gleich, wie A, B und C belegt sind.

Der Beweis aller anderen Behauptungen erfolgt analog. (Das kleine Quadrat am Ende dieser Zeile besagt, dass die Beweisführung damit vollständig ist.) □

1.1.4 Definition Die „Platzhalter“ A, B und C für Wahrheitswerte, die wir in den obigen Formeln und Beweisen verwendet haben, nennt man auch *boolesche Variablen*.

1.2. Grundlagen der Mengenlehre

Der Begriff der *Menge* wird in der Mathematik und Informatik häufig verwendet. Eine Menge fasst unterschiedliche Objekte zu einer neuen Einheit zusammen. Die Art dieser Objekte ist beliebig; es kann sich um Zahlen, Symbole oder auch andere Mengen handeln. So bezeichnen wir z.B. die Zusammenfassung von alphabetischen Zeichen oder auch die Zusammenfassung aller Hörer einer Vorlesung als Mengen. Um eine Menge formal zu notieren, kann man alle enthaltenen Objekte als Liste mit umgebenden geschweiften Klammern aufschreiben. Die Menge der ersten fünf Primzahlen sieht also wie folgt aus:

$$\{2, 3, 5, 7, 11\} \ .$$

Ein Objekt, das zu einer Menge A gehört, wird als *Element* von A bezeichnet. Falls x ein Element von A ist, so schreiben wir $x \in A$, andernfalls schreiben wir $x \notin A$. Es gilt also $7 \in \{2, 3, 5, 7, 11\}$ und $6 \notin \{2, 3, 5, 7, 11\}$. Falls jedes Element einer Menge A in einer Menge B enthalten ist, so nennen wir A eine *Teilmenge* von B bzw. B eine *Obermenge* von A und notieren dies durch $A \subseteq B$ bzw. $B \supseteq A$. Zwei Mengen A und B sind *gleich* oder *identisch* (in Zeichen $A = B$), wenn sie exakt dieselben Elemente enthalten. Sonst sind sie *ungleich* ($A \neq B$). So sind beispielsweise $\{2, 3, 5\}$ und $\{3, \sqrt{25}, 3, 2, 2\}$ gleich, $\{2, 3, 5\}$ und $\{3, 7, 2\}$ aber nicht. Gilt $A \subseteq B$, aber $A \neq B$, so ist A eine *echte Teilmenge* von B bzw. B eine *echte Obermenge* von A, und wir schreiben $A \subset B$ bzw. $B \supset A$.

Wir führen folgende Notationen ein:

- Falls alle Objekte $x_1, x_2, \ldots, x_n$ Elemente der Menge A sind, so schreiben wir auch $x_1, x_2, \ldots, x_n \in A$ statt $x_1 \in A$, $x_2 \in A$, $\ldots, x_n \in A$.
- Falls keines der Objekte $x_1, x_2, \ldots, x_n$ ein Element von A ist, so schreiben wir $x_1, x_2, \ldots, x_n \notin A$.
- Die *leere Menge*, die kein Element enthält, bezeichnen wir mit dem Symbol $\emptyset$.

Wir haben oben gesehen, wie man eine Menge durch die Auflistung aller ihrer Elemente aufschreiben kann. Dies funktioniert natürlich nur mit *endlichen Mengen*, also Mengen, die nur endlich viele Elemente enthalten. Bei unendlichen Mengen kann man sich manchmal auf die Angabe einiger Elemente beschränken, falls die Beschaffenheit der fehlenden Elemente vom Kontext her klar ist. Die folgenden häufigen Mengen lassen sich

so definieren (die Definition/Neueinführung wird durch die Zeichenkombination „:=“ angedeutet):

- $\mathbb{N} := \{1, 2, 3, 4, 5, \dots\}$ ist die Menge der *natürlichen Zahlen*.
- $\mathbb{N}_0 := \{0, 1, 2, 3, 4, 5, \dots\}$ ist die Menge der natürlichen Zahlen einschließlich der Null.
- $\mathbb{Z} := \{\dots, -2, -1, 0, 1, 2, \dots\}$ beschreibt die Menge der *ganzen Zahlen*.

Eine weitere Notationsmöglichkeit bietet sich an, falls jedes Element einer Menge A eine bestimmte Eigenschaft erfüllt. Dann schreiben wir:

$$A = \{x \,|\, \text{Eigenschaft von } x\} \ .$$

Statt x ist natürlich auch ein anderes Symbol verwendbar. Für die Menge P der Primzahlen können wir beispielsweise schreiben:

$$P = \{n \,|\, n \in \mathbb{N} \wedge n > 1 \wedge n \text{ besitzt außer 1 und } n \text{ keine weiteren Teiler}\} \ .$$

Vor dem senkrechten Strich können wir auch (einfache) Formeln verwenden. So beschreibt

$$\{n^2 \,|\, n \in \mathbb{N}\}$$

die Menge aller Quadratzahlen.

1.2.1 Definition Seien A und B Mengen.

- Die Anzahl der Elemente in A heißt *Kardinalität* oder auch *Mächtigkeit* von A und wird mit $|A|$ notiert. Falls A endlich ist, so ist $|A| \in \mathbb{N}_0$; falls A unendlich ist, gilt $|A| = \infty$. Die Kardinalität der leeren Menge ist Null, d.h. $|\emptyset| = 0$. Die beiden Mengen A und B sind von gleicher Kardinalität, falls $|A| = |B|$ gilt.
- Unter der *Vereinigungsmenge* $A \cup B$ versteht man die Zusammenfassung der Elemente von sowohl A als auch B, d.h.

  $$A \cup B := \{x \,|\, x \in A \vee x \in B\} \ .$$

  Es gilt also stets $x \in (A \cup B) \Longleftrightarrow x \in A \vee x \in B$.

  Wenn wir k Mengen $A_1, A_2, \dots, A_k$ vereinigen wollen, so notieren wir die Menge $A_1 \cup A_2 \cup \dots \cup A_k$ durch

  $$\bigcup_{i=1}^{k} A_i \ .$$

- Unter der *Schnittmenge* $A \cap B$ versteht man die Menge aller Elemente, die sowohl in A als auch in B vorkommen, d.h.

  $$A \cap B := \{x \,|\, x \in A \wedge x \in B\} \ .$$

  Es gilt also stets $x \in (A \cap B) \Longleftrightarrow x \in A \wedge x \in B$.

Wenn wir k Mengen $A_1, A_2, \ldots, A_k$ miteinander schneiden wollen, so schreiben wir statt $A_1 \cap A_2 \cap \cdots \cap A_k$ auch

$$\bigcap_{i=1}^{k} A_i \ .$$

- In der *Differenz* $A \setminus B$ sind nur diejenigen Elemente von A enthalten, die nicht zugleich auch in B vorkommen:

  $$A \setminus B := \{x \,|\, x \in A \wedge x \notin B\} \ .$$

  Es gilt also stets $x \in (A \setminus B) \Longleftrightarrow x \in A \wedge x \notin B$.
- Das *Komplement* $\overline{A}$ von A ist die Menge aller Objekte, die A nicht enthält, d.h. $\overline{A} = \{x \,|\, x \notin A\}$. Hier haben wir $x \in \overline{A} \Longleftrightarrow x \notin A$ bzw. $x \notin \overline{A} \Longleftrightarrow x \in A$.

  Häufig gibt man eine Obermenge B von A an, gegenüber der das Komplement zu bilden ist. So ist z.B. das Komplement von $\{1, 4\}$ gegenüber den ersten fünf natürlichen Zahlen die Menge $\{2, 3, 5\}$, und gegenüber den ersten fünf Quadratzahlen die Menge $\{9, 16, 25\}$. Das Komplement von A gegenüber B entspricht dann also der Differenz $B \setminus A$.
- A und B heißen *elementfremd* oder *disjunkt*, wenn sie kein gemeinsames Element besitzen, also $A \cap B = \emptyset$ gilt. Die Vereinigung $A \cup B$ nennt man dann auch *disjunkte Vereinigung* und schreibt $A \dot\cup B$.
- Unter der *Potenzmenge* $\mathcal{P}(A)$ von A versteht man die Menge aller Teilmengen von A, d.h.

  $$\mathcal{P}(A) := \{X \,|\, X \subseteq A\} \ .$$

1.2.2 Beispiel Sei $A := \{2, 3, 5\}$ und $B := \{2, 4\}$. Dann ist $|A| = 3$, $A \cap B = \{2\}$, $A \cup B = \{2, 3, 4, 5\}$, $A \setminus B = \{3, 5\}$ und

$$\mathcal{P}(A) = \Big\{\emptyset, \{2\}, \{3\}, \{5\}, \{2, 3\}, \{2, 5\}, \{3, 5\}, \{2, 3, 5\}\Big\} \ .$$

1.2.3 Satz Seien A, B und C beliebige Mengen. Dann gelten:

- $\overline{\overline{A}} = A$
- $A \cup B = B \cup A$ (*Kommutativgesetze*)
  $A \cap B = B \cap A$
- $A \cup (B \cup C) = (A \cup B) \cup C$ (*Assoziativgesetze*)
  $A \cap (B \cap C) = (A \cap B) \cap C$
- $A \cup (A \cap B) = A$ (*Absorptionsgesetze*)
  $A \cap (A \cup B) = A$
- $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$ (*Distributivgesetze*)
  $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$

- $\overline{A \cup B} = \overline{A} \cap \overline{B}$ (*Regeln von DeMorgan*)
  $\overline{A \cap B} = \overline{A} \cup \overline{B}$

Beweis: Viele der obigen Regeln leuchten einem auf Anhieb sofort ein. Wir betrachten z.B. das erste Absorptionsgesetz. Der Schnitt von A und B ist sicherlich eine Teilmenge von A. Wenn man aber A mit einer Teilmenge von A vereinigt, bleibt A offenbar unverändert erhalten. Deshalb ist die Regel $A \cup (A \cap B) = A$ offensichtlich. Trotzdem wollen wir auch hier zeigen, wie man die Regel anhand der formalen Definitionen beweisen kann. Es gilt nämlich:

$$A \cup (A \cap B) = \{x \,|\, x \in A \vee x \in (A \cap B)\} = \{x \,|\, x \in A \vee (x \in A \wedge x \in B)\} \ .$$

Nun haben wir in Satz 1.1.3 bereits das Absorptionsgesetz für boolesche Operationen bewiesen (es gilt $X \vee (X \wedge Y) = X$ für boolesche Variablen X und Y). Wir setzen für X den Wahrheitswert von $x \in A$ ein und verfahren genauso für Y und $x \in B$. Dann besagt das Absorptionsgesetz, dass wir die Mengeneigenschaft $x \in A \vee (x \in A \wedge x \in B)$ durch $x \in A$ ersetzen können. Damit erhalten wir:

$$A \cup (A \cap B) = \{x \,|\, x \in A\} \ .$$

Wegen $\{x \,|\, x \in A\} = A$ erhalten wir so die Behauptung.

Alle anderen Kommutativgesetze, Assoziativgesetze usw. lassen sich analog beweisen, weil die entsprechenden Regeln auch für boolesche Operationen gelten (siehe Satz 1.1.3). Wir zeigen noch $\overline{\overline{A}} = A$:

$$\overline{\overline{A}} = \{x \,|\, x \notin \overline{A}\} = \{x \,|\, \neg(x \in \overline{A})\}$$
$$= \{x \,|\, \neg(x \notin A)\} = \{x \,|\, \neg(\neg(x \in A))\} = \{x \,|\, x \in A\} = A \ .$$

Damit sind alle Behauptungen gezeigt. □

1.3. Kartesische Produkte

Eine *Sequenz* von Objekten ist eine Liste dieser Objekte in einer bestimmten Reihenfolge. Im Gegensatz zu Mengen umklammert man Sequenzen mit runden statt mit geschweiften Klammern. Die Sequenz der ersten fünf Primzahlen sieht demgemäß wie folgt aus:

$$(2, 3, 5, 7, 11) \ .$$

Weitere Unterschiede zu Mengen:

- Bei Sequenzen kommt es auf die Reihenfolge der Objekte an, d.h. $(2, 3, 5, 7, 11)$ ist nicht dasselbe wie $(3, 2, 5, 7, 11)$. Bei Mengen gilt stattdessen

$$\{2, 3, 5, 7, 11\} = \{3, 2, 5, 7, 11\} \ .$$

- Wiederholungen machen bei Mengen keinen Sinn, z.B. ist $\{2, 2, 3, 3, 5\}$ identisch mit $\{2, 3, 5\}$. Bei Sequenzen sind Wiederholungen dagegen möglich, und es gilt

$$(2, 2, 3, 3, 5) \neq (2, 3, 5) \ .$$

Eine Sequenz mit k Objekten heißt auch *k–Tupel*. Die Sequenz $(2, 3, 5, 7, 11)$ ist demnach ein Beispiel für ein 5–Tupel. Ein 2–Tupel wird auch *Paar* genannt, ein 3–Tupel auch *Tripel*.

1.3.1 Definition Das *kartesische Produkt* $A \times B$ zweier Mengen A und B ist die Menge aller Paare (a, b) mit $a \in A$ und $b \in B$.

1.3.2 Beispiel Ist $A := \{1, 2\}$ und $B := \{\text{rot}, \text{gelb}, \text{grün}\}$, so gilt:

$$A \times B = \{(1, \text{rot}), (2, \text{rot}), (1, \text{gelb}), (2, \text{gelb}), (1, \text{grün}), (2, \text{grün})\} \ .$$

Allgemeiner kann man auch das kartesische Produkt $A_1 \times A_2 \times \cdots \times A_k$ von k Mengen $A_1, A_2, \ldots, A_k$ bilden, welches alle k–Tupel $(a_1, a_2, \ldots, a_k)$ mit $a_i \in A_i$ enthält.

1.3.3 Beispiel Ist $A := \{\delta, \varphi\}$ und $B := \{\Sigma, \Gamma\}$, so ist $A \times B \times A$ die Menge

$$\{(\delta, \Sigma, \delta), (\delta, \Sigma, \varphi), (\delta, \Gamma, \delta), (\delta, \Gamma, \varphi), (\varphi, \Sigma, \delta), (\varphi, \Sigma, \varphi), (\varphi, \Gamma, \delta), (\varphi, \Gamma, \varphi)\} \ .$$

Wenn man kartesische Produkte von einer Menge A mit sich selbst bildet, so schreibt man statt

$$\underbrace{A \times A \times \cdots \times A}_{k\text{–mal}}$$

auch einfach A^k. Die Menge $\mathbb{N}^3$ enthält also alle möglichen Tripel mit natürlichen Zahlen:

$$\mathbb{N}^3 = \{(a, b, c) \mid a, b, c \in \mathbb{N}\} \ .$$

Bei endlichen Mengen $A_1, A_2, \ldots, A_k$ gilt offenbar $|A_1 \times A_2 \times \cdots \times A_k| = |A_1| \cdot |A_2| \cdot \cdots \cdot |A_k|$, insbesondere also auch $|A^k| = |A|^k$ für endliche Mengen A.

1.4. Beweistechniken

Wir haben bereits einige Beweise in den letzten Abschnitten geführt. Beweise werden uns während der gesamten Vorlesung begleiten, denn Behauptungen kann man nicht „einfach so“ stehenlassen, sondern man muss sich von deren Richtigkeit überzeugen. Aber wie findet man passende Beweise? Hier haben sich bestimmte Techniken als nützlich erwiesen, die im Folgenden vorgestellt werden.

1.4.1. Beweis durch Konstruktion

Häufig gelingt es, eine Behauptung der Form „es gibt ein bestimmtes Objekt x mit einer bestimmten Eigenschaft ...“ dadurch zu beweisen, indem man konkret aufzeigt, wie man x konstruieren und sich somit von seiner Existenz überzeugen kann.

Betrachten wir dazu folgendes Beispiel (ein *Lemma* ist ein Hilfssatz, der nur bewiesen wird, um damit eine weiterführende Aussage zu beweisen):

1.4.1 Lemma Zu jeder Zahl $n \in \mathbb{N}$ existiert eine größere Primzahl p.

Beweis: Wir betrachten die *Fakultät* $n!$ von n, die wie folgt definiert ist:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 \ .$$

Beispielsweise gilt $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Natürlich ist $n!$ viel größer als n.

$n!$ ist sicherlich durch alle Zahlen $2, 3, \ldots, n$ teilbar. Die Zahl $m := n!+1$ ist deshalb durch *keine* dieser Zahlen teilbar, denn dabei bleibt stets der Rest 1 übrig. Wir betrachten nun zwei Fälle:

- m ist eine Primzahl: Dann können wir einfach $p := m$ wählen, denn m ist ja größer als $n!$, also insbesondere auch größer als n.
- m ist keine Primzahl: Dann können wir m als das Produkt von (nicht notwendigerweise verschiedenen) Primzahlen darstellen:

  $$m = p_1 \cdot p_2 \cdot \cdots \cdot p_r, \text{ für passende Primzahlen } p_1, p_2, \ldots, p_r \ .$$

  Jede der Primzahlen p_i muss aber größer als n sein, da ja $2, 3, \ldots, n$ keine Teiler von m sind. Also können wir eine beliebige dieser Primzahlen als p wählen.

Damit ist die Behauptung in beiden Fällen (konstruktiv) bewiesen. □

1.4.2. Beweis durch Widerspruch

Eine zweite Möglichkeit, eine bestimmte Aussage A zu beweisen, besteht darin, zunächst das *Gegenteil* (also $\neg A$) anzunehmen und daraus dann einen Widerspruch abzuleiten. Die gegenteilige Annahme kann demnach nicht richtig gewesen sein, d.h. die Behauptung muss doch stimmen. Man spricht dann auch von *indirekten Beweisen*. Auch hierzu ein Beispiel:

1.4.2 Satz Es gibt unendlich viele Primzahlen.

Beweis: Wir gehen von dem Gegenteil aus, d.h. wir nehmen an, dass es nur endlich viele Primzahlen gibt. Unter diesen ist dann auch eine bestimmte größte Primzahl, die wir mit n bezeichnen. Zu n können wir jedoch gemäß Lemma 1.4.1 eine noch größere

Primzahl p finden. Dies ist aber ein Widerspruch zur Wahl von n, denn n sollte bereits die größte Primzahl sein.

Also muss es doch unendlich viele Primzahlen geben. □

1.4.3. Weitere Beweistechniken

Neben den beiden vorgestellten Methoden gibt noch weitere Beweistechniken, die gerade in der theoretischen Informatik häufig verwendet werden. Hierzu zählen vor allem sog. *Diagonalisierungen* sowie Beweise durch *vollständige Induktion*. Aus Zeitgründen müssen wir auf eine Ausführung dieser Techniken hier verzichten. Für interessierte Leser wird jedoch eine Einführung in Induktionsbeweise im Anhang A gegeben.

1.5. Funktionen und Relationen

Aus der Mathematik sind Ihnen bereits *Funktionen* bekannt. Vereinfacht ausgedrückt ist eine Funktion ein Objekt, das eine Eingabe entgegennimmt und daraus eine Ausgabe produziert. Dieselbe Eingabe erzeugt dabei immer wieder dieselbe Ausgabe. Produziert eine Funktion f bei Eingabe a die Ausgabe b, so schreiben wir $f(a) = b$ und sagen: „f bildet a auf b ab“. Eine Funktion heißt deshalb auch *Abbildung*. Das Ergebnis b ist der *Funktionswert* und a das *Argument*. Bei den Argumenten und Funktionswerten kann es sich um Zahlen, aber auch um Zeichen, Symbole, usw. handeln.

Die Ausgabevorschrift kann man durch eine direkte mathematische Formel angeben, also z.B.

$$f(x) := 5x - 4 \ .$$

Bei endlichen Funktionen sind alternativ auch Wertetabellen möglich:

x	$g(x)$
0	rot
1	gelb
2	grün
3	blau
4	schwarz

Hier gilt beispielsweise $g(3) =$ blau.

Auch *Fallunterscheidungen* werden häufig verwendet:

$$h(n) := \begin{cases} n^2 & \text{falls } n \geq 2 \\ n & \text{falls } -2 \leq n < 2 \\ -n^3 & \text{sonst} \end{cases}$$

Für die Funktion h gilt z.B. $h(4) = 16$, $h(1) = 1$ und $h(-3) = 27$.

1.5.1 Definition Die Menge der möglichen Argumente heißt *Definitionsbereich*, und die Funktionswerte entstammen dem *Bildbereich* (engl. „range“). Für eine Funktion f mit Definitionsbereich D und Bildbereich R schreiben wir auch

$$f : D \longrightarrow R \ .$$

Die obige Funktion f mit $f(x) = 5x - 4$ könnte z.B. von $\mathbb{N}$ nach $\mathbb{N}$ oder von $\mathbb{Z}$ nach $\mathbb{Z}$ abbilden, d.h. wir könnten $f : \mathbb{N} \longrightarrow \mathbb{N}$ oder $f : \mathbb{Z} \longrightarrow \mathbb{Z}$ schreiben. Beachten Sie, dass nicht unbedingt alle Werte aus dem Bildbereich „getroffen“ werden müssen — im Fall $f : \mathbb{N} \longrightarrow \mathbb{N}$ gibt es z.B. kein Argument $x \in \mathbb{N}$ mit $f(x) = 4 \in \mathbb{N}$. Jedes mögliche Ergebnis muss jedoch im Bildbereich liegen. Eine Angabe wie $f : \mathbb{Z} \longrightarrow \mathbb{N}$ wäre also nicht korrekt, denn z.B. würde für das Argument $x = -3 \in \mathbb{Z}$ das Ergebnis $f(-3) = -19$ nicht im Bildbereich $\mathbb{N}$ liegen.

Der Definitionsbereich einer Funktion kann auch ein kartesisches Produkt

$$A_1 \times A_2 \times \cdots \times A_k$$

sein. Die Funktion besitzt dann k Argumente, wobei das i-te Argument A_i entstammt. Wir können beispielsweise die Addition zweier Zahlen formal durch die Angabe einer zweistelligen Funktion mit dem Namen $plus : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ beschreiben:

$$plus(x, y) := x + y \ .$$

Gibt es wie hier zwei Argumente, so handelt es sich um eine *binäre* Funktion. Bei nur einem Argument sprechen wir von einer *unären* Funktion.

1.5.2 Definition Ein *Prädikat* ist eine Funktion, deren Bildbereich nur die beiden booleschen Werte *wahr* und *falsch* umfasst.

1.5.3 Beispiel Wir können ein unäres Prädikat *prim* über $\mathbb{N}$ definieren, welches genau dann wahr ist, wenn sein Argument eine Primzahl darstellt:

$$prim(n) := \begin{cases} \text{W} & \text{falls } n \text{ eine Primzahl ist} \\ \text{F} & \text{sonst} \end{cases}$$

Somit ist $prim(11)$ wahr und $prim(12)$ falsch.

Wir haben Prädikate bereits kennengelernt, nämlich als Eigenschaften bei der Definition von Mengen. Auf Seite 13 hatten wir beispielsweise die Menge der Primzahlen wie folgt festgelegt:

$$P = \{n \,|\, n \in \mathbb{N} \wedge n > 1 \wedge n \text{ besitzt außer 1 und } n \text{ keine weiteren Teiler}\} \ .$$

Mit dem Prädikat aus dem letzten Beispiel können wir auch einfach schreiben:

$$P = \{n \,|\, prim(n)\} \ .$$

Im Zusammenhang mit Prädikaten verwenden wir auch häufig die beiden folgenden *Quantoren*:

- $\forall$ ist der sogenannte *Allquantor*. $\forall x\colon \varphi(x)$ bedeutet: „Für alle x ist das Prädikat $\varphi(x)$ wahr."
- $\exists$ ist der sogenannte *Existenzquantor*. $\exists x\colon \varphi(x)$ bedeutet: „Es gibt ein x, so dass das Prädikat $\varphi(x)$ wahr ist."

Manchmal schreiben wir auch $\forall x \in A\colon \varphi(x)$ oder $\exists x \in A\colon \varphi(x)$, wenn x nur aus einer vorgegebenen Menge A stammen soll.

1.5.4 Beispiel Wir benutzen erneut das Prädikat *prim* aus dem letzten Beispiel.

- $\exists n \in \mathbb{N}\colon \mathit{prim}(n)$ ist wahr, denn es gibt eine natürliche Zahl, die eine Primzahl ist.
- $\forall n \in \mathbb{N}\colon \mathit{prim}(n)$ ist falsch, denn nicht alle natürlichen Zahlen sind auch Primzahlen.
- $\forall n \in \mathbb{N}\colon \exists p \in \mathbb{N}\colon p > n \wedge \mathit{prim}(p)$ ist wahr. Die Ausdruck besagt, dass für jede natürliche Zahl n eine größere Primzahl p existiert. Diese wurde vor kurzem in Lemma 1.4.1 bewiesen und ist deshalb richtig.
- $\exists p \in \mathbb{N}\colon \forall n \in \mathbb{N}\colon p > n \wedge \mathit{prim}(p)$ besagt, dass es eine Primzahl gibt, die größer als alle natürlichen Zahlen ist — dies ist natürlich falsch.

1.5.5 Definition Sei A eine Menge. Ein zweistelliges Prädikat R über $A \times A$ heißt (binäre) *Relation* über A. Statt „$R(x, y) =$ wahr" schreibt man auch einfach nur $R(x, y)$ oder noch einfacher xRy. Man sagt dann: „x und y stehen in Relation zueinander."

1.5.6 Beispiel Die Ordnung „kleiner als" ($<$) auf den natürlichen Zahlen („$2 < 4$", „$17 < 21$") ist ein Beispiel für eine Relation über $\mathbb{N}$.

Wir können eine Relation R über A auch mit derjenigen Menge aller Paare aus $A \times A$ identifizieren, für die R wahr ist. Für $A := \{1, 2, 3\}$ ist die „kleiner als"–Relation also nichts anderes als die Menge $\{(1, 2), (1, 3), (2, 3)\}$.

1.5.7 Definition Sei R eine Relation über einer Menge A.

- R ist *reflexiv*, falls $\forall x \in A\colon xRx$ gilt. Eine Ordnung wie „$<$" über $\mathbb{N}$ ist also nicht reflexiv, da z.B. 2 nicht kleiner als 2 ist. Die Ordnung „$\leq$" ist dagegen reflexiv.
- R ist *symmetrisch*, falls $\forall x, y \in A\colon xRy \Rightarrow yRx$ gilt. Weder „$<$" noch „$\leq$" sind symmetrisch, denn z.B. ist $2 < 4$ bzw. $2 \leq 4$, aber es gilt nicht $4 < 2$ bzw. $4 \leq 2$.
- R ist *transitiv*, falls $\forall x, y, z \in A\colon xRy \wedge yRz \Rightarrow xRz$ gilt. Ordnungen wie „$<$" sind stets transitiv, denn aus $x < y$ und $y < z$ folgt immer $x < z$.
- R ist eine *Äquivalenzrelation*, wenn sie reflexiv, symmetrisch und transitiv ist.

1.5.8 Beispiel Wir legen eine Relation $\equiv$ über $\mathbb{Z}$ wie folgt fest:

$$x \equiv y \ :\Longleftrightarrow\ x - y \text{ ist durch 3 teilbar } .$$

(Die Relation wird durch diese Festlegung gleichzeitig definiert, deshalb steht hier „$:\Longleftrightarrow$“ statt nur einfach „$\Longleftrightarrow$“.) Wir zeigen, dass $\equiv$ eine Äquivalenzrelation ist:

- Für alle $x \in \mathbb{Z}$ ist $x - x = 0$, und 0 ist durch 3 teilbar. Also gilt $x \equiv x$, d.h. $\equiv$ ist reflexiv.
- Für alle $x, y \in \mathbb{Z}$ gilt: wenn $x - y$ durch 3 teilbar ist, dann ist $y - x$ vom Betrag her die gleiche Zahl (nur mit anderem Vorzeichen) und deshalb ebenfalls durch 3 teilbar. Also gilt $x \equiv y \Rightarrow y \equiv x$, d.h. $\equiv$ ist symmetrisch.
- Für alle $x, y, z \in \mathbb{Z}$ gilt: wenn sowohl $x - y$ als auch $y - z$ durch 3 teilbar sind, dann ist $(x-y)+(y-z)$ die Summe aus zwei durch 3 teilbaren Zahlen, also selbst durch 3 teilbar. Wegen $(x-y)+(y-z) = x-z$ ist dann also $x-z$ durch 3 teilbar, d.h. es gilt $x \equiv z$. Also ist $\equiv$ auch transitiv.

Etwas abstrakter ist das folgende Beispiel:

1.5.9 Beispiel Sei $\varphi : A \times B \longrightarrow \{\mathrm{W}, \mathrm{F}\}$ ein beliebiges zweistelliges Prädikat. Dann definiert die Relation R über A mit

$$xRy \ :\Longleftrightarrow\ \forall b \in B\colon\ (\varphi(x,b) \ \Leftrightarrow\ \varphi(y,b))$$

stets eine Äquivalenzrelation über A, ganz gleich, wie φ konkret aussieht. Denn für jedes $x \in A$ und $b \in B$ ist offenbar

$$\varphi(x,b) \ \Leftrightarrow\ \varphi(x,b)$$

immer erfüllt, da links und rechts von dem Äquivalenzzeichen der gleiche Ausdruck steht. Also gilt

$$\forall x \in A\colon\ \forall b \in B\colon\ (\varphi(x,b) \ \Leftrightarrow\ \varphi(x,b)) \ ,$$

was nicht anderes als

$$\forall x \in A\colon\ xRx$$

bedeutet, d.h. R ist reflexiv. Für je zwei Elemente $x, y \in A$ und ein Element $b \in B$ sind ferner

$$\varphi(x,b) \ \Leftrightarrow\ \varphi(y,b)$$

und

$$\varphi(y,b) \ \Leftrightarrow\ \varphi(x,b)$$

gleichwertige Aussagen, da die Äquivalenz kommutativ ist. Aus

$$\forall b \in B\colon\ \varphi(x,b) \ \Leftrightarrow\ \varphi(y,b)$$

folgt also immer

$$\forall b \in B\colon\ \varphi(y,b) \ \Leftrightarrow\ \varphi(x,b) \ ,$$

d.h. es gilt

$$xRy \implies yRx \ .$$

Da dies für alle Elemente $x, y \in A$ der Fall ist, haben wir insgesamt

$$\forall x, y \in A\colon\ xRy \implies yRx \ ,$$

d.h. R ist auch symmetrisch. Bzgl. der Transitivität betrachtet man nun noch drei Elemente $x, y, z \in A$ mit xRy und yRz, d.h. wir haben

$$\forall b \in B\colon\ \varphi(x,b) \Leftrightarrow \varphi(y,b)$$

und

$$\forall b \in B\colon\ \varphi(y,b) \Leftrightarrow \varphi(z,b) \ .$$

Für jedes beliebige Element $b \in B$ gilt dann also immer

$$\varphi(x,b) \Leftrightarrow \varphi(y,b)$$

und ebenso

$$\varphi(y,b) \Leftrightarrow \varphi(z,b) \ ,$$

also insgesamt

$$\varphi(x,b) \Leftrightarrow \varphi(y,b) \Leftrightarrow \varphi(z,b) \ ,$$

und damit (ohne das mittlere Zwischenergebnis):

$$\varphi(x,b) \Leftrightarrow \varphi(z,b) \ .$$

Da das Element $b \in B$ frei wählbar ist, erhalten wir

$$\forall b \in B\colon\ \varphi(x,b) \Leftrightarrow \varphi(z,b)$$

und damit xRz. Insgesamt haben wir so — für beliebige Elemente $x, y, z \in A$ — von xRy und yRz auf xRz geschlossen. Also gilt sogar

$$\forall x, y, z \in A\colon\ xRy \wedge yRz \implies xRz \ ,$$

d.h. R ist auch transitiv.

1.5.10 Definition Sei R eine Äquivalenzrelation über einer Menge A. Für jedes Element $a \in A$ ist $[a] := \{x \in A \mid aRx\}$ die *Äquivalenzklasse* aller zu a äquivalenten Elemente.

1.5.11 Beispiel Für die Äquivalenzrelation $\equiv$ aus dem Beispiel 1.5.8 ergeben sich drei Äquivalenzklassen, nämlich

$$\begin{aligned} [1] &= \{\dots, -8, -5, -2, 1, 4, \dots\} \\ [2] &= \{\dots, -7, -4, -1, 2, 5, \dots\} \\ [3] &= \{\dots, -6, -3, 0, 3, 6, \dots\} \end{aligned}$$

Natürlich gilt auch z.B. $[-7] = [5] = [2]$, $[-18] = [0]$, usw.

1.5.12 Satz Sei R eine Äquivalenzrelation über einer Menge A. Dann gilt:

a) Jede Äquivalenzklasse $[a]$ ist nicht leer.

b) Die Vereinigung aller Äquivalenzklassen ergibt A.

c) Zwei verschiedene Äquivalenzklassen sind stets disjunkt.

Die Äquivalenzklassen bilden also eine *Zerlegung* der zugrunde liegenden Menge A.

Beweis: Die einzelnen Behauptungen lassen sich wie folgt zeigen:

a) Wegen der Reflexivität von R gilt zumindest immer $a \in [a]$.

b) Aus $a \in [a]$ folgt $\{a\} \subseteq [a]$, d.h. wir haben

$$A = \bigcup_{a \in A} \{a\} \subseteq \bigcup_{a \in A} [a] \ .$$

Natürlich gilt auch umgekehrt

$$\bigcup_{a \in A} [a] \subseteq A \ ,$$

denn die Vereinigung von Teilmengen von A kann immer wieder nur eine Teilmenge von A ergeben. Also gilt insgesamt

$$A = \bigcup_{a \in A} [a] \ .$$

c) Wir betrachten für zwei verschiedene Elemente $a, b \in A$ die zugehörigen Äquivalenzklassen $[a]$ und $[b]$. Wir beweisen indirekt und nehmen deshalb an, dass $[a] \cap [b] \neq \emptyset$ gilt. Dann existiert zumindest ein Element $x \in [a] \cap [b]$, und es gilt aRx wegen $x \in [a]$ sowie bRx wegen $x \in [b]$. Sei nun $y \in [a]$ ein beliebiges weiteres Element von $[a]$, d.h. es gilt aRy. Dann gilt wegen der Symmetrie von R und aRx auch xRa, zusammen mit aRy und der Transitivität von R also auch xRy. Wegen bRx und erneut der Transitivität von R folgt weiter bRy, d.h. es gilt $y \in [b]$. Aus $y \in [a]$ folgt also immer $y \in [b]$, d.h. $[a]$ ist eine Teilmenge von $[b]$. Genauso gut kann man bei der obigen Argumentation aber auch die Rollen von $[a]$ und $[b]$ vertauschen, so dass man analog $[b] \subseteq [a]$ erhält. Die beiden Teilmengenbeziehungen $[a] \subseteq [b]$ und $[b] \subseteq [a]$ sind jedoch nur im Fall $[a] = [b]$ möglich.

Falls also zwei Mengen $[a]$ und $[b]$ nicht disjunkt sind, so sind sie gleich. Oder, was dasselbe ist, wenn sie nicht gleich sind, so sind sie disjunkt.

Damit ist alles gezeigt. □

1.6. Formale Sprachen

Im Vorwort wurde die Frage gestellt, was Computer prinzipiell berechnen können. Bevor wir diese Frage angehen, müssen wir zunächst festlegen, wie wir uns eine Verarbeitung

überhaupt vorstellen, d.h. mit was für Objekten ein Computer „gefüttert“ werden kann und was für Objekte er als Ausgabe produzieren soll. Neben Zahlen sind hierfür *Zeichenketten* („Strings“) sehr praktisch.

Ein *Wort* ist eine endliche Zeichenkette, wobei jedes Zeichen aus einem bestimmten *Alphabet* stammt. Die meisten Wörter der deutschen Sprache werden beispielsweise über dem Alphabet

$$\{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}, \mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}, \ddot{\mathrm{a}}, \ddot{\mathrm{o}}, \ddot{\mathrm{u}}, \ss, \ddot{\mathrm{A}}, \ddot{\mathrm{O}}, \ddot{\mathrm{U}}\}$$

gebildet. Formal gesehen ist ein Alphabet einfach eine endliche Menge von Zeichen oder *Symbolen*. Wir bezeichnen Alphabete üblicherweise mit den griechischen Großbuchstaben Σ („Sigma“) und Γ („Gamma“). Beispiele sind $\Sigma := \{a, b, c, \$, \bot\}$ oder $\Gamma := \{0, 1\}$. Man sagt dann, dass man Wörter *über* Σ und Γ bildet, also z.B. $\$aacba\bot$ bzw. 0001011.

1.6.1 Definition Sei Σ ein Alphabet. Die *Länge* eines Wortes w über Σ ist die Anzahl aller hintereinander geschriebenen Symbole und wird mit $|w|$ notiert. Falls w die Länge n hat, so schreiben wir $w = w_1 w_2 \ldots w_n$, wobei jedes w_i in Σ enthalten ist. Das sogenannte *leere Wort* ε (sprich: „Epsilon“) hat die Länge $|\varepsilon| = 0$.

1.6.2 Beispiel Sei wie oben $\Sigma := \{a, b, c, \$, \bot\}$ und $w := \$aacba\bot$. Dann ist $|w| = 7$, $w_1 = \$$ und $w_4 = c$.

Die sogenannte *Konkatenation* oder *Verkettung* (d.h. „Hintereinanderschreibung“) zweier Wörter $x = x_1 \ldots x_m$ und $y = y_1 \ldots y_n$ ist das Wort

$$x \cdot y := x_1 \ldots x_m y_1 \ldots y_n \ .$$

Statt $x \cdot y$ schreiben wir meistens kurz xy. Ist beispielsweise $x = 0001$ und $y = 011$, so ist $xy = 0001011$ und $yx = 0110001$. Offenbar gilt $x\varepsilon = \varepsilon x = x$ für alle Wörter x. Falls wir ein Wort x mehrmals mit sich selbst konkatenieren wollen, so verwenden wir die Schreibweise

$$x^k := \underbrace{xx \cdots x}_{k\text{–mal}} \ .$$

So entspricht z.B. für $x = 011$ der Ausdruck x^4 dem Wort 011011011011.

Ein Wort y ist ein *Teilwort* oder *Infix* von einem anderen Wort x, falls y irgendwo in x ohne Unterbrechungen vorkommt. So ist 011 ein Infix von 1001110, aber nicht von 1000101. Das leere Wort ε ist immer ein Teilwort von einem beliebigen anderen Wort.

1.6.3 Definition Für ein Alphabet Σ bezeichnet Σ^* die Menge aller Wörter, die man über Σ bilden kann. $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$ ist die Menge aller nichtleeren Wörter über Σ. Für $k \in \mathbb{N}_0$ bezeichnet

$$\Sigma^k := \{w \mid w \in \Sigma^* \wedge |w| = k\}$$

die Menge aller Wörter über Σ mit der Länge k. Eine *formale Sprache* (oder einfach *Sprache*) ist eine Menge von Wörtern, d.h. eine bestimmte Teilmenge von Σ^*. Eine

Sprache kann *endlich* oder *unendlich* sein, d.h. endlich oder unendlich viele Wörter enthalten. Natürlich sind Σ^*, Σ^+ und auch alle Σ^k selbst Sprachen.

1.6.4 Beispiel Sei $\Sigma := \{a, b\}$. Dann gilt $\Sigma^2 = \{aa, ab, ba, bb\}$ sowie

$$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \dots\}$$

und

$$\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \dots\} \ .$$

Weitere Sprachen über Σ sind z.B.

$$A := \{a, aaaa, aaaaa\}$$

oder

$$B := \{ab, abab, ababab, abababab, \dots\} \ .$$

A ist endlich, B ist unendlich.

Offenbar gilt stets $\Sigma^0 = \{\varepsilon\}$, $\Sigma^1 = \Sigma$ sowie

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i \qquad \text{und} \qquad \Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i \ .$$

Meist bezeichnen Kleinbuchstaben am Ende des deutschen Alphabets $\{\dots, w, x, y, z\}$ Wörter aus Σ^*, während Kleinbuchstaben am Anfang des Alphabets $\{a, b, c, \dots\}$ häufig einzelne Symbole aus Σ darstellen.

Die Begriffe *Länge*, *Konkatenation* und *Infix* werden zur Übung nun noch einmal definiert, diesmal aber formal exakt.

Länge: Sei Σ ein Alphabet und $x \in \Sigma^*$. Die Länge von x ist Null, falls $x = \varepsilon$ ist. Andernfalls trennen wir das letzte Symbol von x ab und legen die Länge des resultierenden Wortes w plus Eins als die Länge von x fest. Dies läßt sich formal wie folgt notieren:

$$|x| := \begin{cases} 0 & \text{falls } x = \varepsilon \\ |w| + 1 & \text{falls } \exists w \in \Sigma^* \colon \exists a \in \Sigma \colon x = wa \end{cases}$$

Die Länge von $abb \in \{a, b\}^*$ berechnet sich so zu

$$|abb| = |ab| + 1 = |a| + 1 + 1 = |\varepsilon| + 1 + 1 + 1 = 0 + 1 + 1 + 1 = 3 \ .$$

Konkatenation: Für $x = x_1 \dots x_m$ und $y = y_1 \dots y_n$ aus Σ^* legen wir formal xy als dasjenige Wort $z = z_1 \dots z_{m+n} \in \Sigma^{m+n}$ fest, für das gilt:

$$z_i := \begin{cases} x_i & \text{falls } 1 \le i \le m \\ y_{i-m} & \text{falls } m < i \le m + n \end{cases}$$

Infix: Wir definieren:

$$w \text{ ist ein Infix von } y :\Longleftrightarrow \exists x, z \in \Sigma^* \colon xwz = y \ .$$

Falls in obiger Beziehung $x = \varepsilon$ gilt, so ist w ein *Präfix* von y, d.h. w taucht am Anfang von y auf. Analog ist w ein *Suffix* von y, falls $z = \varepsilon$ gilt, d.h. w steht am Ende von y. Demnach ist 1101 sowohl ein Präfix also auch ein Suffix von 1101101.

1.7. Das O–Kalkül

In der Einführung wurde bereits erwähnt, dass in der theoretischen Informatik auch die Komplexität diverser Problemstellungen untersucht wird. Konkret wird dabei gefragt, wie viel Zeit ein Computer zur Ermittlung einer bestimmten Lösung benötigt. Dabei ist klar, dass eine größere Probleminstanz im Allgemeinen auch mehr Rechenzeit verlangt als eine kleinere Probleminstanz. Wenn zum Beispiel eine Menge von Zahlen sortiert werden soll, so ist dies offenbar für 20 Zahlen schneller zu erledigen als für 10000 Zahlen. Man setzt deshalb die benötigte Rechenzeit in ein Verhältnis zur Größe der konkret gestellten Aufgabe.

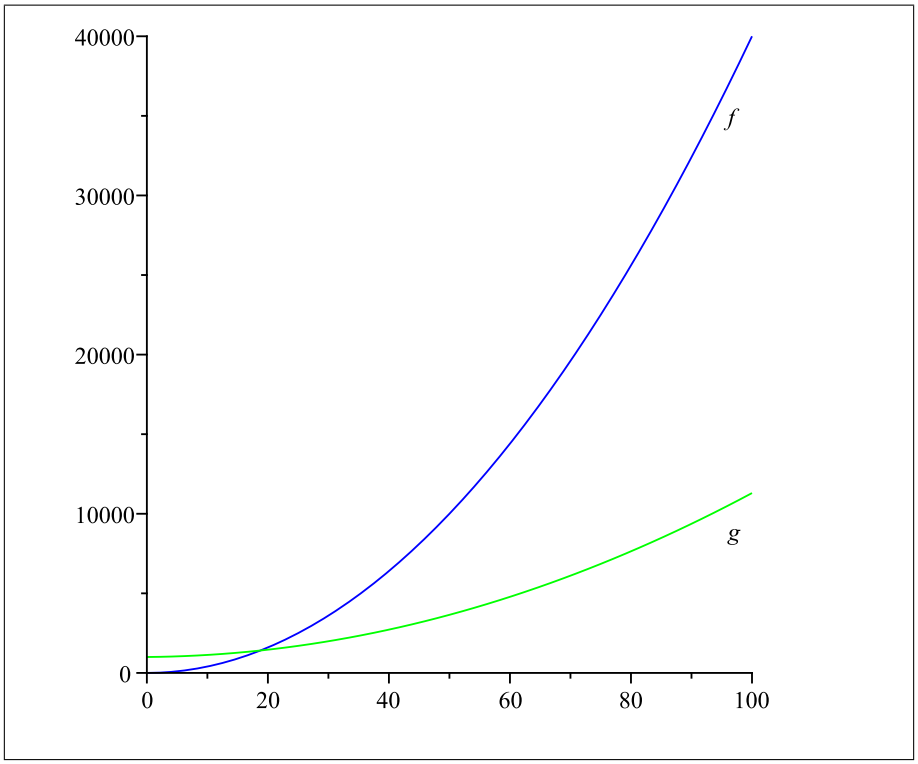
Angenommen, wir haben je zwei Programme F und G auf zwei (sehr langsamen) Rechnern zur Verfügung, die eine Menge von n Zahlen sortieren. Das erste Programm F benötigt dazu $f(n) := 4n^2$ Sekunden, das andere $g(n) := n^2 + 3n + 1000$ Sekunden. (Natürlich sind heutige Rechner viel schneller, aber wir bleiben der Einfachheit halber bei diesem Beispiel.) Um zu ermitteln, welches Programm das bessere (also schnellere) ist, können wir f und g für verschiedene Werte von n vergleichen:

n	$f(n)$	$g(n)$
3	36	1018
5	100	1040
10	400	1130
20	1600	1460
100	40000	11300

Alternativ kann man auch die Graphen der beiden Funktionen miteinander vergleichen, so wie Sie es aus der Mathematik sicherlich kennen. Die beiden Graphen sind auf der nächsten Seite dargestellt.

Anfangs ist also F vermutlich das bessere Programm, weil es weniger Zeit als G benötigt. Für größere Werte von n (genauere Untersuchungen zeigen: ab $n = 19$) ist aber G schneller, und wie man an den Tabellenwerten bzw. den Funktionsgraphen erkennt, wird der Abstand zwischen $f(n)$ und $g(n)$ mit größer werdenden Argumenten für n immer deutlicher. Ist also G das bessere Programm?

Zunächst einmal ist es so, dass man sich für kleinere Probleminstanzen in der Informatik nicht wirklich interessiert (3, 5 oder 10 Zahlen kann man ja auch noch von Hand



sortieren, eine Million Zahlen aber nicht). Deshalb ist immer entscheidend, wie sich die Rechenzeiten für große Instanzen entwickeln. Der Begriff „groß“ ist dabei dehnbar: das Sortieren von 1000 Zahlen ist für Menschen bereits eine sehr langwierige Aufgabe, aber für moderne Computer ist diese Größenordnung selbst bei der Wahl von schlechten Algorithmen eine Angelegenheit von Millisekunden. Man hat sich deshalb darauf geeinigt, dass man einfach eine beliebig große (aber endliche) Anzahl von Fällen am Anfang ausschließt und erst danach mit dem Vergleich beginnt. Also ist so gesehen G tatsächlich das bessere Programm.

Trotzdem muss aber paradoxerweise G nicht unbedingt besser als F sein. Es gilt nämlich die Abschätzung

$$f(n) = 4n^2 \leq 4n^2 + 12n + 4000 \leq 4(n^2 + 3n + 1000) = 4 \cdot g(n) \ ,$$

also kurzgefasst

$$f(n) \leq 4 \cdot g(n) \ ,$$

d.h. F ist höchstens viermal langsamer als G, und diese Tatsache muss nicht unbedingt etwas mit der Güte des Verfahrens von F zu tun haben. Vielleicht läuft F ja auf einem älteren Computer, oder ist in einer „langsameren“ Programmiersprache geschrieben worden. Konstante Faktoren sind daher keine verlässliche Aussage für den Vergleich von Algorithmen und werden deshalb vernachlässigt.

Unter diesen Gesichtpunkten kann man die Angabe der „genauen“ Zeitangaben nun vereinfachen. Wir führen dies gleich am Beispiel unserer zweiten *Komplexitätsfunktion* $g(n) = n^2 + 3n + 1000$ vor. Ziel ist es, für $g : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ eine vereinfachende Abschätzung $h : \mathbb{N}_0 \longrightarrow \mathbb{R}$ zu finden, die eine *asymptotische obere Schranke* von g darstellt ($\mathbb{R}$ ist dabei die Menge der *reellen Zahlen*). Die Funktion h soll also den Zeitverbrauch g nach oben hin begrenzen, d.h. das Programm soll für alle Eingabegrößen n nicht mehr als $h(n)$ viel Zeit verbrauchen. Es soll also

$$\forall n \in \mathbb{N}_0 \colon\ g(n) \leq h(n)$$

gelten. Allerdings besagt unsere erste Vereinbarung, dass wir endlich viele Werte am Anfang ignorieren können. Man darf also einen Startwert $n_0 \in \mathbb{N}$ wählen und betrachtet die Vergleichsbedingung nur für alle Argumente n, die größer oder gleich n_0 sind:

$$\exists n_0 \in \mathbb{N} \colon\ \forall n \geq n_0 \colon\ g(n) \leq h(n)\ .$$

Weiterhin haben wir gesehen, dass konstante Faktoren $c \in \mathbb{N}$ im Geschwindigkeitsvergleich keine Rolle spielen. Es ist beispielsweise egal, ob wir Sekunden oder Minuten messen, da sich beide Einheiten nur durch einen konstanten (also unwichtigen) Faktor von 60 unterscheiden. Wir sind also schon zufrieden, wenn

$$\exists c \in \mathbb{N} \colon\ \exists n_0 \in \mathbb{N} \colon\ \forall n \geq n_0 \colon\ g(n) \leq c \cdot h(n)$$

gilt und schreiben dann $g = O(h)$ gemäß der folgenden Definition.

1.7.1 Definition Sei $h : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion. Mit $O(h)$ (sprich: „groß Oh von h“ oder einfach „Oh von h“) bezeichnen wir die Menge aller Funktionen, die — abgesehen von einem konstanten Faktor c und endlich vielen Ausnahmen $n_0 \in \mathbb{N}$ am Anfang — *höchstens* so schnell wachsen wie h:

$$O(h) := \{g : \mathbb{N}_0 \longrightarrow \mathbb{R} \mid \exists c \in \mathbb{N} \colon\ \exists n_0 \in \mathbb{N} \colon\ \forall n \geq n_0 \colon\ g(n) \leq c \cdot h(n)\}\ .$$

Für $g \in O(f)$ sagt man auch: „g ist von der Ordnung f“.

Statt $g \in O(f)$ schreibt man üblicherweise $g = O(f)$, aber dies ist nur eine Notation und keine wirkliche Gleichung (denn auf der linken Seite steht eine einzelne Funktion, und auf der rechten Seite eine Menge von Funktionen). Man darf beide Seiten auch nicht tauschen, d.h. die Notation $O(f) = g$ ist unzulässig.

1.7.2 Beispiel Als erstes betrachten wir die Funktion $g(n) = n^2 + 3n + 1000$ vom Anfang dieses Abschnitts. Dann können wir Folgendes beobachten:

- Für alle $n \geq 3$ ist $3n \leq n^2$.
- Für alle $n \geq 32$ ist $1000 \leq n^2$.
- Wir nehmen nun das Maximum von 3 und 32 und wissen dann, dass für alle $n \geq 32$ beide obigen Abschätzungen richtig sind. Für alle $n \geq 32$ ist also der Funktionswert $g(n) = n^2 + 3n + 1000$ kleiner oder gleich $n^2 + n^2 + n^2 = 3n^2$.

Mit der Wahl $c := 3$, $n_0 := 32$ und $h(n) := n^2$ ist demnach die Abschätzung

$$\forall n \geq n_0 \colon\ g(n) \leq c \cdot h(n)$$

richtig. Also gilt $g = O(n^2)$.

An dem vorstehenden Beispiel erkennt man gut, wie die O–Notation in der Praxis funktioniert. Man hat es bei einer exakten Laufzeitangabe üblicherweise mit einer Summe von verschiedenen Termen zu tun (im Beispiel waren dies n^2, $3n$, und 1000). Die O–Notation sucht sich nun im Endeffekt unter diesen Termen den am schnellsten wachsenden heraus (dies war hier n^2). Denn durch eine genügend große Wahl von n_0 (hier 32) dominiert dieser Term für $n \geq n_0$ alle anderen, so dass ein Vielfaches davon (im Beispiel war es das Dreifache) eine obere Schranke des ursprünglichen Funktionswertes darstellt. Der zugehörige konstante Faktor kann dann abschließend durch eine entsprechende Wahl der Konstanten c ausgeglichen werden.

Auch bzgl. der anderen Funktion $f(n) = 4n^2$ erhalten wir $f = O(n^2)$. Zum Beweis könnte man wie im Beispiel 1.7.2 formal $c := 4$, $n_0 := 1$ und $h(n) := n^2$ wählen und dann zeigen, dass immer

$$\forall n \geq n_0 \colon\ f(n) \leq c \cdot h(n)$$

gilt. Einfacher ist jedoch die Argumentation mit der unter dem Beispiel angedeuteten Methode: es gibt hier sowieso nur einen Term, konstante Faktoren können fallen gelassen werden — fertig.

Sowohl f als auch g sind also von sog. *quadratischer Komplexität*, und damit sind die beiden zugrundeliegenden Programme F und G im Endeffekt gleich gut.

Wir können das für $g(n) = n^2 + 3n + 1000$ vorgebrachte Argument verallgemeinern:

1.7.3 Satz Sei $f : \mathbb{N}_0 \to \mathbb{R}$ ein *Polynom* vom *Grad* k, d.h. für passende (evtl. negative) *Koeffizienten* $a_0, a_1, \ldots, a_k \in \mathbb{R}$ ist f von der Form

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

bzw. (in Summenschreibweise)

$$f(n) = \sum_{i=0}^{k} a_i n^i \ .$$

(Für das Polynom $n^2 + 3n + 1000$ wäre z.B. $a_0 = 1000$, $a_1 = 3$ und $a_2 = 1$.)

Dann gilt $f = O(n^k)$.

Beweis: Zunächst verändern wir das Polynom und wandeln alle negativen Koeffizienten in positive um (aus z.B. $7n^3+17n^2-23n+2$ wird so $7n^3+17n^2+23n+2$.) Das modifizierte Polynom ist dann offenbar eine obere Schranke der ursprünglichen Funktion, da alle Terme, die vorher abgezogen wurden, jetzt addiert werden. Ein verbleibender Term der Form $a_i n^i$ wächst nun umso schneller, je größer der Exponent i ist. Folglich ist n^k der dominierende Term. □

Es gilt also z.B.

$$7n^3 + 17n^2 - 23n + 2 = O(n^3) \ .$$

Natürlich gilt theoretisch auch $7n^3 + 17n^2 - 23n + 2 = O(n^4)$, denn n^4 wächst stärker als n^3, d.h. n^4 ist erst recht eine asymptotische obere Schranke. (Beachten Sie, dass man das Polynom auch als $0n^4 + 7n^3 + 17n^2 - 23n + 2$ schreiben könnte, womit die eben bewiesene Regel wieder direkt anwendbar wäre.) Wir werden jedoch später sehen, warum man trotzdem normalerweise immer $7n^3 + 17n^2 - 23n + 2 = O(n^3)$ schreibt.

Die O–Notation kommt nicht nur bei Polynomen zum Einsatz, sondern auch z.B. bei der Vereinfachung logarithmischer Terme. (Zur Erinnerung: $\log_b y$ bezeichnet den *Logarithmus* von y zur *Basis* b, d.h. $\log_b y$ ist diejenige Zahl x, für die $b^x = y$ gilt. Beispielsweise ist $\log_2 8 = 3$ und $\log_{\sqrt{2}} 4 = 4$.) Im Zusammenhang mit der O–Notation können wir nun bei der Verwendung von Logarithmen die Basisangaben stets weglassen. Denn für zwei Logarithmen $\log_a n$ und $\log_b n$ gilt immer $\log_a n = \log_a b \cdot \log_b n$, und die Größe $\log_a b$ ist eine Konstante. Diese wiederum ist in der O–Notation irrelevant, d.h. es gilt $log_a n = O(log_b n)$ und $\log_b n = O(\log_a n)$. Wir schreiben zukünftig einfach nur noch $O(\log n)$.

Bei der Vereinfachung von Laufzeiten durch die O–Notation treten bestimmte *Komplexitätsklassen* wie z.B. das obige $O(n^2)$ öfter auf. Folgende Klassen werden häufig benutzt:

Ordnung	Wachstum
$O(1)$	konstant
$O(\log n)$	logarithmisch
$O(n)$	linear
$O(n \log n)$	„*n–log–n*“
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(n^k)$ für ein $k > 1$	polynomiell
$O(d^n)$ für ein $d > 1$	exponentiell

Die Tabelle ist dabei nach steigendem Zeitaufwand sortiert. (Ausnahme: für $1 < k < 2$ liegt die Komplexität von $O(n^k)$ zwischen $O(n \log n)$ und $O(n^2)$, für $2 < k < 3$ liegt sie zwischen $O(n^2)$ und $O(n^3)$.) Da jeder O–Ausdruck eine Menge von Funktionen darstellt, kann man die wachsende Komplexität auch durch eine Kette von echten Teilmengenbeziehungen ausdrücken (dabei sind wie oben d und k Zahlen, die größer als Eins sind, wohingegen ε eine „kleine“ Zahl bezeichnet, die echt zwischen 0 und 1 liegt):

$$O(1) \subset O(\log n) \subset O(n^\varepsilon) \subset O(n) \subset O(n \log n) \subset O(n^k) \subset O(d^n) \ .$$

Für $\ell > 0$ gilt zudem

$$O(n^\ell) \subset O(n^{\ell+\varepsilon})$$

sowie für $d > 1$

$$O(d^n) \subset O((d+\varepsilon)^n) \ .$$

Besonders die zweite Regel führt zuweilen zur Verwirrung. Es gilt z.B. *nicht* $3^n = O(2^n)$, obwohl sich die beiden Funktionen scheinbar nur durch verschiedene führende Zahlen unterscheiden. Die Zahlen 2 und 3 sind hier jedoch keine multiplikativen Konstanten, sondern gehen als unterschiedliche Basen in die zugehörigen Exponentialfunktionen ein. Man kann leicht nachprüfen, dass 3^n viel schneller als 2^n wächst. Dies kann auch nicht durch eine noch so große multiplikative Konstante c ausgeglichen werden. Egal, wie man c auch wählt: für genügend große n gilt stets $3^n > c \cdot 2^n$.

Wie „gut“ ist nun ein Programm, wenn es eine bestimmte (z.B. quadratische) Komplexität hat? Dies kann man sich anhand der Auswirkungen auf die Laufzeit klarmachen, wenn man die Eingabegröße verändert. Hierzu einige Beispiele:

- Bei quadratischer Komplexität führt die *Verdopplung* der Eingabegröße n (wenn man also z.B. doppelt so viele Zahlen sortiert) zu einer *Vervierfachung* der Laufzeit, nämlich von n^2 auf $(2n)^2 = 4n^2$.
- Bei logarithmischer Komplexität sieht dies viel besser aus: hier führt die Verdopplung der Eingabegröße n nur zu konstant vielen Zusatzschritten, da z.B. für den Zweierlogarithmus $\log_2 2n = \log_2 n + 1$ gilt.
- Umgekehrt sind bei exponentieller Komplexität selbst die schnellsten Rechner der Welt machtlos, denn schon eine Vergrößerung der Eingabe um nur wenige neue Elemente führt zu einer Verdopplung der Laufzeit. Kann z.B. ein Computer ein bestimmtes Problem für 1000 Eingabeelemente in nur einer Mikrosekunde lösen, so dauert bei einer Zeitkomplexität von $O(2^n)$ die Lösung des gleichen Problems für 1050 Elemente schon knapp 36 Jahre (!), obwohl sich die Eingabe nur um 5% vergrößert hat. Solche Probleme werden daher häufig als „hartnäckig“ bezeichnet. Das in der Einleitung erwähnte *Problem des Handlungsreisenden* gehört nach dem heutigen Wissensstand mit dazu.

Jetzt wird auch klar, warum man $7n^3 + 17n^2 - 23n + 2$ zu $O(n^3)$ und nicht zu $O(n^4)$ vereinfacht. Denn die Laufzeit eines Verfahrens ist ein zentrales „Gütesiegel“ — je größer die Laufzeit, desto schlechter das Verfahren. Also versucht man die durch die O–Notation vorgenommene obere Abschätzung möglichst weit nach unten zu drücken, und durch die Angabe $7n^3 + 17n^2 - 23n + 2 = O(n^4)$ würde man freiwillig das zentrale Bewertungskriterium verschlechtern.

Neben der O–Notation gibt es noch weitere sog. *Landausche*[1] *Symbole*. Wir werden vor allem noch das Ω–Symbol verwenden. Die Idee und Verwendung der Ω–Notation ist dabei im Prinzip genau dieselbe wie bei der O–Notation, jedoch verwendet man das Ω–Symbol bei asymptotischen *unteren* Schranken. Wenn also ein Programm in $\Omega(n^2)$ viel Zeit abläuft, so drückt man damit aus, dass es *mindestens* (und nicht wie bei der O–Notation *höchstens*) quadratisch viele Rechenschritte benötigt.

Zum Abschluss sei noch bemerkt, dass man mit der O–und Ω–Notation nicht nur Laufzei-

[1] Edmund Landau, *1877 Berlin, †1938 Berlin, deutscher Mathematiker, ab 1909 Professor für reine Mathematik in Göttingen.

ten beschreiben kann, sondern auch z.B. den Verbrauch von Speicherplatz. Man spricht dann auch von der *Platzkomplexität* eines Programms.

Alle notwendigen mathematischen Grundlagen sind damit gelegt. Wir können uns nun mit einem ersten Teilbereich der theoretischen Informatik befassen, der sogenannten *Automatentheorie*.

2

Reguläre Sprachen

2.1. Grammatiken und die Chomsky–Hierarchie

Wir haben in der Einführung formale Sprachen vorgestellt. Eine solche Sprache L beschreibt häufig die Menge aller Lösungen zu einem bestimmten Problem. Wenn z.B. ein Computer für eine Zeichenkette x testen soll, ob sie nur aus Nullen besteht und zudem einen Länge zwischen 2 und 7 besitzt, so können wir zunächst die Sprache aller Lösungen

$$L := \{00, 000, 0000, 00000, 000000, 0000000\}$$

definieren und dann den Computer prüfen lassen, ob x in L vorkommt. Allerdings ist L im Allgemeinen unendlich groß und liegt deshalb nicht in Form einer Auflistung seiner Elemente vor (sonst könnten wir bzw. der Computer ja auch einfach in dieser Liste nachsehen und nach dem passenden Eintrag suchen). Stattdessen muss L durch ein endliches System von Syntaxregeln in möglichst konstruktiver Weise beschrieben werden. Maschinen, die nach entsprechenden Algorithmen programmiert sind, müssen anhand dieser Regeln z.B. in der Lage sein, ähnlich wie oben zu entscheiden, ob ein gegebenes Wort x zu L gehört oder nicht.

2.1.1 Beispiel Sei $\Sigma := \{(,), +, -, *, /, a\}$. Die Sprache L sei die Sprache aller vollständig und korrekt geklammerten arithmetischen Ausdrücke mit der Variablen a, z.B. sind $((a + a) * a)$ und $((a - (a + a))/a)$ Elemente von L. Wenn nun ein Computer ermitteln soll, ob eine bestimmte Zeichenkette y einen solchen arithmetischen Ausdruck darstellt, so muss er die Frage „Ist y in L enthalten?“ beantworten.

Beachten Sie, dass wir zwar eine umgangssprachliche, aber keine präzise Definition gegeben haben, welche Elemente (in diesem Fall Ausdrücke) aus Σ^* zu $L \subseteq \Sigma^*$ gehören. Wir wissen aus unserer Erfahrung, was korrekt geklammerte Ausdrücke sind, aber ein Computer besitzt diese Erfahrung nicht. Wir werden das oben erwähnte fomale Regelwerk, welches alle korrekt geklammerten Ausdrücke erzeugt, in Beispiel 2.1.10 noch nachreichen.

2.1.2 Beispiel Sei L die Menge aller fehlerfrei kompilierbaren Java–Programme. Das Alphabet Σ umfasst hier alle Buchstaben, Zahlen und sonstigen Zeichen, die man benötigt, um Java–Programme eintippen zu können. Ein Programm ist dann nichts anderes als ein (langes) Wort $x \in \Sigma^*$, und die Frage nach einem korrekten Java–Programm x ist wieder äquivalent zu „Ist $x \in L$?“. Wieder stellt sich die Frage nach einer präzisen Definition von L und einem zugehörigen Erzeugendensystem.

2.1.3 Beispiel Eine Folge von Nullen und Einsen $b_k b_{k-1} b_{k-2} \dots b_0$ mit $b_i \in \{0, 1\}$ kann man wie folgt als eine Zahl n aus $\mathbb{N}_0$ interpretieren:

$$n = \sum_{i=0}^{k} b_i 2^i \ .$$

Man spricht dann auch von einer *binären Kodierung* der Zahl n. Beispielsweise entspricht $101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 5$ der Zahl 5 und 001101 der Zahl 13. Wir wollen allerdings keine unnötigen führenden Nullen zulassen, d.h. 001101 ist keine zulässige Kodierung von 13, sondern nur 1101. Als einzige Ausnahme hiervon sei erlaubt, die Zahl 0 durch den Code 0 (und nur durch diesen Code) zu repräsentieren.

Man kann zeigen, dass dann jede Zahl $n \in \mathbb{N}_0$ genau eine binäre Kodierung besitzt und bezeichnet diese mit $bin(n)$. Wir können so die Sprache

$$L := \{bin(n) \,|\, n \in \mathbb{N}_0\} = \{0, 1, 10, 11, 100, \dots\}$$

aller binär kodierten Zahlen über dem Alphabet $\Sigma = \{0, 1\}$ definieren.

Wenn nun ein Computer bestimmen soll, ob eine bestimmte Zeichenkette x eine gültige binäre Kodierung darstellt, so muss er formal die Frage „Ist x in L enthalten?“ beantworten. Ein Erzeugendensystem von L können wir — anders als beim vorherigen Beispiel — diesmal leicht angeben:

- Das Wort „0“ ist eine gültige binäre Kodierung.
- Jedes Wort aus Nullen und Einsen, welches mit einer führenden Eins beginnt, ist eine gültige Kodierung.

Ein Computer kann nun eine Eingabe $x \in \Sigma^*$ daraufhin überprüfen, ob x eine dieser Regeln erfüllt und so die Frage „Ist $x \in L$?“ (das sogenannte *Wortproblem*) beantworten.

Das Generieren oder Beschreiben von L geschieht oft mit *Grammatiken*, die Lösung des Wortproblems ($w \in L$?) durch *Automaten*. Wir besprechen zunächst die Grammatiken.

2.1.4 Definition Eine *Grammatik* ist ein Viertupel $G = (V, \Sigma, P, S)$, wobei die einzelnen Komponenten folgende Bedeutung haben:

- V ist eine endliche Menge von *Nicht–Terminalsymbolen* oder *Variablen*.
- Σ ist eine endliche Menge von *Terminalsymbolen*. Jedes Wort, welches man später aus G erzeugen kann, ist in Σ^* enthalten.

- P ist eine endliche Menge von *Produktionen* oder *Regeln* der Form

$$(\ell, r) \in (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$$

  mit $\ell \notin \Sigma^+$.
- $S \in V$ bezeichnet die *Startvariable*.

Außerdem gilt $V \cap \Sigma = \emptyset$, d.h. die Menge der Variablen und die Menge der Terminalsymbole sind disjunkt. Für eine Regel (ℓ, r) schreibt man auch $\ell \to r$.

2.1.5 Beispiel Betrachten Sie die Grammatik $G = (V, \Sigma, P, S)$ mit $V = \{S, B, C\}$, $\Sigma = \{a, b, c\}$ und

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

Variablen werden üblicherweise mit Großbuchstaben am Anfang des Alphabets notiert $(A, B, C, \dots)$, Terminalsymbole dagegen mit Kleinbuchstaben $(a, b, c, \dots)$. Kleinbuchstaben am Ende des Alphabets $(u, v, w, x, \dots)$ bezeichnen häufig Wörter aus Σ^*, während kleine Buchstaben am Anfang des griechischen Alphabets $(\alpha, \beta, \gamma, \dots)$ Wörter aus $(V \cup \Sigma)^*$ repräsentieren.

Die Idee bei Grammatiken besteht aus der Bildung von *Ableitungen*. Das Prinzip demonstrieren wir anhand der Grammatik aus dem Beispiel 2.1.5.

Wir geben zunächst ein beliebiges Wort $\alpha \in (V \cup \Sigma)^+$ vor, z.B. $aabCBC$. Nun können wir versuchen, die linke Seite einer Regel als Teilwort in α wiederzufinden. Zum Beispiel kommt in α das Teilwort CB vor, und CB ist der linke Teil der Regel $CB \to BC$. Dieses Teilwort ersetzt man nun durch die rechte Seite der Regel und erhält so $aabBCC$. Man sagt, dass man $aabBCC$ aus $aabCBC$ *ableitet* und schreibt

$$aabCBC \Rightarrow_G aabBCC\ .$$

Nun wiederholt man diesen Schritt mit derselben oder irgendeiner anderen Regel, bis in dem generierten Wort keine einzige Variable mehr vorkommt, es also aus Σ^* stammt. In diesem Fall ist auch keine Regel $\ell \to r$ mehr anwendbar, denn ℓ darf ja nicht in Σ^+ enthalten sein.

Bei dem Wort $aabBCC$ können wir nacheinander die Regeln $bB \to bb$, $bC \to bc$ und $cC \to cc$ anwenden und so das Wort $aabbcc \in \Sigma^*$ erzeugen.

Ein Ableitungsprozeß beginnt immer mit der Startvariablen S. Eine mögliche komplette Ableitung wäre also

$$S \Rightarrow_G aSBC \Rightarrow_G aaBCBC \Rightarrow_G aaBBCC$$
$$\Rightarrow_G aabBCC \Rightarrow_G aabbCC \Rightarrow_G aabbcC \Rightarrow_G aabbcc\ .$$

Alle aus S ableitbaren „Zwischenprodukte“ $aSBC$, $aaBCBC$, usw. (auch S und $aabbcc$ selbst) werden *Satzformen* genannt.

Wir definieren das Ableitungsprinzip nun formal.

2.1.6 Definition Durch eine Grammatik $G = (V, \Sigma, P, S)$ wird auf $(V \cup \Sigma)^*$ wie folgt eine Relation $\Rightarrow_G$ definiert:

$$(\alpha \Rightarrow_G \beta) :\Longleftrightarrow \exists \gamma, \ell, r, \delta \in (V \cup \Sigma)^* \colon \alpha = \gamma\ell\delta \wedge \beta = \gamma r\delta \wedge (\ell, r) \in P \ .$$

Man sagt, α geht unter G *unmittelbar über* nach β. Im Beispiel 2.1.5 und dem Übergang

$$aabCBC \Rightarrow_G aabBCC$$

gilt demnach $\alpha = aabCBC$, $\beta = aabBCC$, $\ell = CB$, $r = BC$, $\gamma = aab$, und $\delta = C$.

2.1.7 Definition Für $\alpha, \beta \in (V \cup \Sigma)^*$ schreiben wir $\alpha \Rightarrow_G^* \beta$, falls man β aus α in endlich vielen Schritten ableiten kann. Es gilt also $\alpha = \beta$, oder es gibt eine Folge $(\gamma_0, \gamma_1, \ldots, \gamma_n)$ mit $\gamma_i \in (V \cup \Sigma)^*$ und

$$\alpha = \gamma_0 \Rightarrow_G \gamma_1 \Rightarrow_G \ldots \Rightarrow_G \gamma_n = \beta \ .$$

Die Relation $\Rightarrow_G^*$ ist die sogenannte *reflexiv transitive Hülle* von $\Rightarrow_G$. Die Folge

$$\alpha = \gamma_0 \Rightarrow_G \gamma_1 \Rightarrow_G \ldots \Rightarrow_G \gamma_n = \beta$$

heißt (G–)*Ableitung* von α nach β. Ein Wort $\alpha \in (V \cup \Sigma)^*$ mit $S \Rightarrow_G^* \alpha$ heißt *Satzform* von G.

2.1.8 Definition Die durch die Grammatik G dargestellte *Sprache* ist

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

Die Elemente aus $L(G)$ werden auch *Sätze* genannt. Das sog. *Wortproblem* ist die Aufgabe zu entscheiden, ob ein gegebenes Wort $x \in \Sigma^*$ einen gültigen Satz von $L(G)$ darstellt oder nicht.

Wir vereinbaren folgende Schreibvereinfachungen:

- Ist G aus dem Kontext ersichtlich, so schreiben wir auch $\Rightarrow$ statt $\Rightarrow_G$.
- $\ell \to r_1 \mid r_2 \mid \ldots \mid r_k$ ist die Kurzform für $\ell \to r_1$, $\ell \to r_2, \ldots, \ell \to r_k$.

2.1.9 Beispiel Wir greifen auf die Sprache $L := \{bin(n) \mid n \in \mathbb{N}_0\}$ aller binär kodierten natürlichen Zahlen zurück, die in Beispiel 2.1.3 eingeführt wurde. Sei $G = (V, \Sigma, P, S)$ mit $V = \{S, B\}$, $\Sigma = \{0, 1\}$ und $P = \{S \to 0 \mid 1 \mid 1B,\ B \to 0 \mid 1 \mid 0B \mid 1B\}$. Wir behaupten, dass diese Grammatik L erzeugt.

Beweis: Wir erinnern uns zunächst an die im Beispiel 2.1.3 aufgeführten Regeln:

- Das Wort „0“ ist eine gültige binäre Kodierung.
- Jedes Wort aus Nullen und Einsen, welches mit einer führenden Eins beginnt, ist eine gültige Kodierung.

L ist also die Sprache aller Wörter, die diesen Regeln genügen. Es ist nun $L = L(G)$ zu zeigen. Wir beginnen mit dem Beweis der Teilmengenbeziehung $L \subseteq L(G)$ und führen die folgenden Abkürzungen ein:

$$S_0\colon\ S \to 0\ , \qquad S_1\colon\ S \to 1\ , \qquad S_B\colon\ S \to 1B\ ,$$

$$B_0\colon\ B \to 0\ , \qquad B_1\colon\ B \to 1\ , \qquad B^0\colon\ B \to 0B\ , \qquad B^1\colon\ B \to 1B\ .$$

Wegen S_0 und S_1 sind 0 und 1 in $L(G)$ enthalten. Für jedes andere Wort $1a_1a_2\ldots a_n \in L$ mit $a_i \in \{0,1\}$ können wir folgende Ableitung angeben:

$$S \overset{S_B}{\Longrightarrow} 1B \overset{B^{a_1}}{\Longrightarrow} 1a_1B \overset{B^{a_2}}{\Longrightarrow} 1a_1a_2B \overset{B^{a_3}}{\Longrightarrow} \cdots \overset{B^{a_{n-1}}}{\Longrightarrow} 1a_1a_2\ldots a_{n-1}B \overset{B_{a_n}}{\Longrightarrow} 1a_1a_2\ldots a_n\ .$$

Also gilt in der Tat $L \subseteq L(G)$.

Nun zeigen wir umgekehrt $L(G) \subseteq L$. Aus S kann man entweder direkt die Kodierung 0 (mit S_0) oder direkt die Kodierung 1 (mit S_1) erzeugen. Beide Kodierungen liegen in L. Die einzige weitere Möglichkeit besteht in der Wahl der Startregel $S \to 1B$. Die führende Eins bleibt dann immer an dieser Position stehen, ganz gleich, was für ein Wort man sonst noch aus B ableitet. Folglich genügt das generierte Wort am Ende mit Sicherheit der zweiten Regel und ist damit ebenfalls eine gültige Kodierung.

Also gilt insgesamt $L \subseteq L(G)$ und $L(G) \subseteq L$. Dies ist nur möglich, wenn beide Mengen gleich sind. □

2.1.10 Beispiel Es sei $G = (V, \Sigma, P, S)$ mit $V = \{S\}$, $\Sigma = \{(,), +, -, *, /, a\}$, und

$$P = \{S \to a \,|\, (S+S) \,|\, (S-S) \,|\, (S*S) \,|\, (S/S)\}\ .$$

Dann können wir z.B. den Ausdruck $((a-(a+a))/a)$ wie folgt ableiten:

$$S \Rightarrow (S/S) \Rightarrow ((S-S)/S) \Rightarrow ((S-(S+S))/S) \Rightarrow ((a-(S+S))/S)$$

$$\Rightarrow ((a-(a+S))/S) \Rightarrow ((a-(a+a))/S) \Rightarrow ((a-(a+a))/a)\ .$$

Es leuchtet einem ein, dass $L(G)$ genau die Sprache aller vollständig geklammerten arithmetischen Ausdrücke in der Variablen a ist (vgl. Beispiel 2.1.1 auf Seite 33). Auf einen formalen Beweis werden wir hier aus Zeitgründen aber verzichten. □

2.1.11 Definition Die *Chomsky–Hierarchie*[1] unterteilt Grammatiken und die zugehörigen Sprachen in die folgenden vier Typen:

- Jede Grammatik ist vom *Typ 0* (eine sogenannte *Phrasenstrukturgrammatik*).

[1] Noam Chomsky, *1928 Philadelphia, US–amerikanischer Sprachwissenschaftler, seit 1955 Professor für Linguistik am MIT.

- Eine Grammatik ist vom *Typ 1* oder *kontextsensitiv*, falls für alle Regeln $\ell \to r \in P$ stets $|\ell| \leq |r|$ gilt. Eine Regel wie z.B. $S \to aSBC$ erfüllt wegen $|S| = 1$ und $|aSBC| = 4$ demnach diese Bedingung, eine Regel wie z.B. $AC \to a$ dagegen nicht. Alle Grammatiken aus den Beispielen 2.1.5, 2.1.9 und 2.1.10 sind also kontextsensitiv.
- Eine Typ 1–Grammatik ist vom *Typ 2* oder *kontextfrei*, falls für alle Regeln $\ell \to r$ stets $\ell \in V$ gilt. Auf der linken Seite einer Regel steht also immer nur ein einzelnes Nichtterminal. Die Grammatik aus dem Beispiel 2.1.5 ist also nicht kontextfrei, wohl aber die beiden anderen Grammatiken aus den Beispielen 2.1.9 und 2.1.10.

  In einer kontextfreien Grammatik kann eine Variable A unabhängig vom Kontext, in dem A steht, durch eine passende rechte Seite r ersetzt werden: $\alpha A\beta \Rightarrow_G \alpha r\beta$ (sofern natürlich eine passende Regel $A \to r$ existiert). Dies ist bei kontextsensitiven Grammatiken nicht so. Hier sind Regeln der Form $uAv \to urv$ möglich, d.h. A kann nur im Kontext zwischen u und v durch r ersetzt werden.
- Eine Typ 2–Grammatik ist vom *Typ 3* oder *regulär*, falls für jede Regel $\ell \to r$ die rechte Seite r entweder aus einem einzelnen Terminalzeichen oder aus einem Terminalzeichen gefolgt von einer Variablen besteht. Ein Blick auf unsere Beispielgrammatiken zeigt, dass nur die Grammatik aus Beispiel 2.1.9 regulär ist. Die dort besprochenen Ableitungen sind typisch für reguläre Grammatiken. Jede Satzform einer solchen Grammatik hat immer das Aussehen $a_1a_2 \ldots a_iA$ (abgesehen von dem am Ende erzeugten Wort). Dabei stellt $a_1a_2 \ldots a_i$ einen Präfix des später generierten Wortes $a_1a_2 \ldots a_n$ dar. Es gibt also immer nur eine Variable in jeder Satzform, und diese ist ganz am Ende positioniert. Bei jedem Ableitungsschritt wird ein weiteres Zeichen an den bisher gebildeten Präfix angehängt und dabei evtl. das Nichtterminal am Ende ausgetauscht. Nur im letzten Schritt wird abweichend davon die Variable durch ein einzelnes Terminalsymbol (also durch das letzte Zeichen a_n) ersetzt.
- Eine Sprache L heißt vom Typ $i \in \{0, 1, 2, 3\}$, falls eine Typ i–Grammatik G mit $L = L(G)$ existiert. Typ 1–Sprachen heißen kontextsensitiv, Typ 2–Sprachen heißen kontextfrei, und Typ 3–Sprachen heißen regulär.

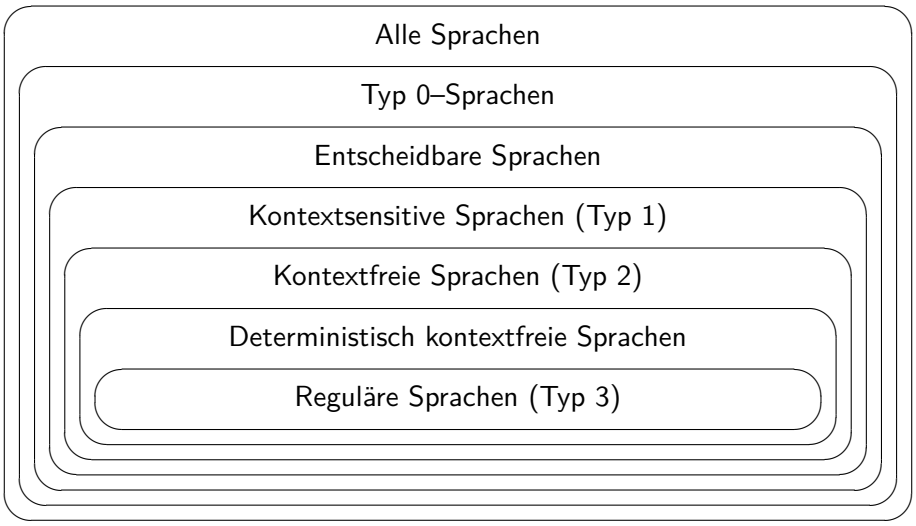
Die Abbildung auf der nächsten Seite veranschaulicht die Inklusionsbeziehungen zwischen den einzelnen Sprachklassen (die sogenannten *entscheidbaren* bzw. *deterministisch kontextfreien* Sprachen werden später noch genauer besprochen).

Alle Teilmengenbeziehungen sind echt. Es gibt also z.B. kontextsensitive Sprachen, die nicht kontextfrei sind. Wir haben sogar schon eine kennengelernt:

2.1.12 Beispiel Die in Beispiel 2.1.5 vorgestellte Grammatik $G = (V, \Sigma, P, S)$ mit den Komponenten $V = \{S, B, C\}$, $\Sigma = \{a, b, c\}$ und

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

generiert die Sprache $L(G) = \{a^nb^nc^n \,|\, n \geq 1\}$. Davon kann man sich leicht überzeugen:



Die Chomsky–Hierarchie bildlich

- Zuerst wendet man die erste Regel $(n-1)$–mal und anschließend die zweite Regel einmal an. Man erhält die Satzform

$$\underbrace{a \ldots a}_{n\text{–mal}} \underbrace{BC \ldots BC}_{n\text{–mal}} \ .$$

- Jetzt wendet man die dritte Regel $CB \to BC$ so lange wie möglich an. Dadurch wandern alle B–Variablen nach vorn und alle C-Variablen nach hinten:

$$\underbrace{a \ldots a}_{n\text{–mal}} \underbrace{B \ldots B}_{n\text{–mal}} \underbrace{C \ldots C}_{n\text{–mal}} \ .$$

- Durch die restlichen Regeln verwandelt man die Variablen B und C in die entsprechenden Kleinbuchstaben b und c, so dass man schließlich das Wort $a^n b^n c^n$ erhält.
- Wenn man sich noch genauer mit der Grammatik beschäftigt, so sieht man, dass man prinzipiell nichts anderes machen kann. Es lassen sich also keine anderen Wörter erzeugen.

G ist kontextsensitiv, aber nicht kontextfrei. Nun könnte es prinzipiell sein, dass wir „verschwenderisch“ mit der Regelfreiheit umgegangen sind und man mit einer kontextfreien Grammatik dieselben Wörter produzieren könnte. Wir werden jedoch später sehen (siehe Beispiel 3.4.2 auf Seite 112), dass es keine kontextfreie Grammatik gibt, die $\{a^n b^n c^n \mid n \geq 1\}$ erzeugt.

Ist G vom Typ 1, so wird bei einer Ableitung die linke Seite einer Regel durch eine längere rechte Seite ersetzt. Wenn also eine Satzform β aus α ableitbar ist, so folgt immer

$|\alpha| \leq |\beta|$. Da schon die Startsatzform S die Länge eins besitzt, ist es demnach unmöglich, das leere Wort ε aus S abzuleiten. Diese Einschränkung wird in der Literatur häufig dadurch umgangen, indem ausnahmsweise eine sog. „ε–Regel“ der Form $S \to \varepsilon$ erlaubt wird, obwohl sie wegen $|S| = 1$ und $|\varepsilon| = 0$ in kontextsensitiven Grammatiken eigentlich unzulässig ist (S sei wie immer die Startvariable). Wir werden uns aus Zeitgründen um diese Problematik nicht weiter kümmern und stattdessen einfach akzeptieren, dass kontextsensitive Grammatiken das leere Wort nicht erzeugen können.

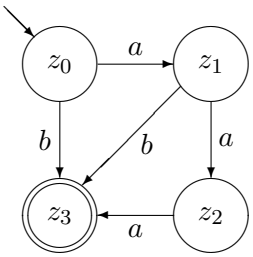
Unser Ziel ist es nun, für jede Typklasse möglichst einfache Maschinenmodelle anzugeben, die entscheiden können, ob ein Wort zu der von einer Grammatik generierten Sprache gehört. Die Modelle sollen also das Wortproblem lösen. Wir beginnen mit den regulären Sprachen und den sog. *endlichen Automaten*.

2.2. Endliche Automaten

Endliche Automaten sind das einfachste Rechenmodell in der theoretischen Informatik. Wir werden sehen, dass solche Automaten genau dann das Wortproblem für eine Sprache L lösen können, wenn L regulär ist.

Die Arbeitsweise von endlichen Automaten kann leicht veranschaulicht werden.

2.2.1 Beispiel Betrachten Sie den folgenden *Zustandsgraph* eines endlichen Automaten:



Das Diagramm symbolisiert vier verschiedene *Zustände* z_0, z_1, z_2 und z_3. Der Automat befindet sich immer in einem dieser vier Zustände. Zwischen zwei Zuständen kann gewechselt werden, wenn es einen Pfeil (man sprich auch von einer *gerichteten Kante*) von dem einen zu dem anderen Zustand gibt und das dort angegebene Symbol (hier jeweils a oder b) als führendes Zeichen von einem Eingabewort gelesen werden kann. Danach ist dieses Zeichen „verbraucht“ und kann nicht erneut benutzt werden. Mit jedem Zustandswechsel wird das restliche Eingabewort also um ein Zeichen kürzer. Falls beispielsweise als Eingabe das Wort ab zur Verfügung steht und der Automat sich gerade in dem Zustand z_0 befindet, so wechselt er unter Verwendung des ersten Symbols a in den Zustand z_1 und unter Verwendung des zweiten Symbols b in den nächsten Zustand z_3. Danach ist das Eingabewort vollständig verbraucht, und die Verarbeitung stoppt. Selbst bei einem

längeren Eingabewort hätte man nicht fortfahren können, denn z_3 ist eine „Sackgasse“, aus der keine gerichtete Kante mehr heraus führt.

Zwei Zustände sind in dem Diagramm besonders gekennzeichnet. Der Zustand z_0 hat einen kleinen eingehenden Pfeil am linken oberen Rand, der nicht aus einem anderen Zustand austritt. Daran erkennt man den sogenannten *Startzustand*, d.h. hier beginnt anfangs die Verarbeitung. Ferner besitzt der Zustand z_3 einen doppelten Rand, was ihn als *Endzustand* kennzeichnet (es kann auch mehrere Endzustände geben). Wenn sich der Automat nach der Verarbeitung in einem solchen Endzustand befindet, so *akzeptiert* er das Eingabewort, ansonsten *verwirft* er die Eingabe. Wir sehen also, dass unser Eingabewort *ab* akzeptiert wird, denn wie oben gesehen wechselt der Automat vom Startzustand z_0 über z_1 in den Endzustand z_3. Der Automat verwirft dagegen die Eingabe *aa*, da er sich nach der kompletten Verarbeitung in z_2 befindet, also in keinem Endzustand. Auch das Wort *aab* wird verworfen, denn nach der Verarbeitung der ersten beiden *a*–Symbole befindet sich der Automat wie zuvor in Zustand z_2, und von dort aus gibt es keinen Übergang mit dem Symbol *b*. Selbst *aba* wird nicht akzeptiert, obwohl wir diesmal in dem Endzustand z_3 hängenbleiben.

Start– und Endzustände sind (abgesehen von ihrer zusätzlichen Bedeutung) normale Zustände, d.h. sie dürfen während der Verarbeitung des Eingabewortes durchaus mehrmals betreten und auch wieder verlassen werden.

Beachten Sie, dass endliche Automaten im Endeffekt handelsübliche Computer mit endlich viel Speicherplatz repräsentieren. Die (extrem vielen) möglichen Zustände eines solchen Computers ergeben sich durch alle möglichen verschiedenen Belegungskombinationen des Hauptspeichers und der Festplatte(n). Jeder interne Verarbeitungsschritt des Prozessors verändert diverse Speicherzellen und überführt den Computer somit von einem Zustand in den nächsten. Wir werden sehen, dass auf Grund ihrer Einfachheit endliche Automaten schon an relativ elementaren Aufgaben scheitern — und Computer deshalb ebenfalls.

Wir geben jetzt eine formale Definition eines endlichen Automaten an.

2.2.2 Definition Ein *deterministischer endlicher Automat* (abgekürzt *DEA*) ist gegeben durch ein 5–Tupel $M = (Z, \Sigma, \delta, z_0, E)$, wobei die einzelnen Komponenten folgende Bedeutung haben:

- Z enthält alle *Zustände* des DEAs, d.h. Z ist die endliche *Zustandsmenge* von M.
- Σ ist ein endliches *Eingabealphabet* mit $Z \cap \Sigma = \emptyset$.
- $\delta : Z \times \Sigma \longrightarrow Z$ ist die (partielle) *Zustandsübergangsfunktion*. „Partiell“ bedeutet hier, dass die Funktion δ evtl. nicht für alle möglichen Argumente aus $Z \times \Sigma$ definiert ist, d.h. man kommt nicht unbedingt von jedem Zustand mit jedem Symbol zu einem nächsten Zustand weiter.
- $z_0 \in Z$ ist der *Startzustand*.
- $E \subseteq Z$ ist die Menge der (akzeptierenden) *Endzustände*.

Einen DEA und seine Arbeitsweise kann man sich wie in Beispiel 2.2.1 durch seinen *Zustandsgraph* veranschaulichen. Dabei handelt es sich um einen *gerichteten, beschrifteten Graphen* mit den folgenden Eigenschaften:

- Für jeden Zustand $z \in Z$ gibt es einen *Knoten*, symbolisiert durch einen Kreis, den man in der Mitte mit z markiert.
- Der Startzustands–Knoten z_0 ist durch einen (kleinen) Extra–Pfeil markiert, der auf z_0 zeigt.
- Die Endzustands–Knoten aus E sind durch eine Doppelumrandung gekennzeichnet.
- Für jeden *Übergang* $\delta(z, a) = z'$ gibt es einen „Pfeil“ oder besser eine *gerichtete Kante*, die von z nach z' führt und mit a beschriftet ist, also $z \xrightarrow{a} z'$. Falls δ für das Argument (z, a) nicht definiert ist, so wird auch keine Kante eingefügt.

Der Automat aus Beispiel 2.2.1 entspricht also formal dem 5–Tupel $(Z, \Sigma, \delta, z_0, E)$ mit $Z = \{z_0, z_1, z_2, z_3\}$, $\Sigma = \{a, b\}$, und $E = \{z_3\}$. Für die Übergangsfunktion δ gilt

$$\delta(z_0, a) = z_1 \ , \quad \delta(z_0, b) = z_3 \ , \quad \delta(z_1, a) = z_2 \ , \quad \delta(z_1, b) = z_3 \ , \quad \delta(z_2, a) = z_3 \ .$$

Die Funktion ist partiell, da z.B. für $\delta(z_3, b)$ kein Übergang definiert ist.

Die Arbeitsweise eines endlichen Automaten M haben wir bereits angedeutet. Mit der Eingabe $a_1 a_2 \ldots a_n \in \Sigma^*$ beginnt M im Zustand z_0, liest a_1, geht über in den durch $\delta(z_0, a_1) =: z_1$ festgelegten Zustand, liest dann a_2, geht über in den durch $\delta(z_1, a_2) =: z_2$ festgelegten nächsten Zustand, usw. Am Ende erreicht M den Zustand $z_n := \delta(z_{n-1}, a_n)$.

M akzeptiert die Eingabe $a_1 a_2 \ldots a_n$, falls z_n in E liegt, d.h. z_n muss ein (akzeptierender) Endzustand sein. Andernfalls *verwirft* M die Eingabe. M verwirft auch dann, falls für einen zwischenzeitlich eingenommenen Zustand der nächste Übergang gar nicht definiert ist, selbst dann, falls zwischendurch bereits ein Endzustand betreten wurde.

2.2.3 Definition Die von einem DEA $M = (Z, \Sigma, \delta, z_0, E)$ *akzeptierte Sprache* $L(M)$ wird durch

$$L(M) \ := \ \{x \in \Sigma^* \mid M \text{ angesetzt auf } x \text{ geht zum Schluß in einen Endzustand über}\}$$

festgelegt.

Wir können die Übergangsfunktion δ verallgemeinern, indem wir nicht nur die Verarbeitung einzelner Zeichen, sondern auch die Verarbeitung ganzer Wörter zulassen.

2.2.4 Definition Zu einem gegebenen DEA $M = (Z, \Sigma, \delta, z_0, E)$ legt man die *verallgemeinerte Übergangsfunktion* $\hat{\delta} : Z \times \Sigma^* \longrightarrow Z$ für alle $z \in Z$ und $x \in \Sigma^*$ wie folgt fest:

$$\hat{\delta}(z, x) := \begin{cases} z & \text{falls } x = \varepsilon \\ \hat{\delta}(\delta(z, a), y) & \text{falls } \exists a \in \Sigma\colon \exists y \in \Sigma^*\colon x = ay \end{cases}$$

Offenbar erweitert $\hat{\delta}$ die Funktion δ von Einzelzeichen aus Σ zu Worten über Σ^*. Es gilt:

- $\hat{\delta}(z, a) = \delta(z, a)$ für alle $a \in \Sigma$ und
- $\hat{\delta}(z, a_1 a_2 \dots a_n) = \delta(\dots \delta(\delta(\delta(z, a_1), a_2), a_3), \dots)$.

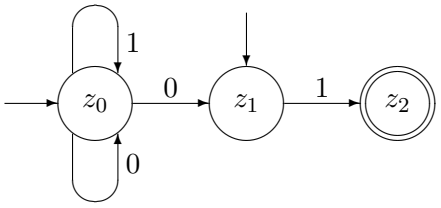
Für die von M akzeptierte Sprache gilt also $L(M) = \{x \in \Sigma^* \mid \hat{\delta}(z_0, x) \in E\}$. In Beispiel 2.2.1 ist z.B. $\hat{\delta}(z_0, aa) = z_2$ und $L(M) = \{aaa, ab, b\}$.

Neben *deterministischen* Automaten gibt es auch *nichtdeterministische* Varianten. Solche *nichtdeterministische endliche Automaten* (*NEAs*) verallgemeinern die DEAs wie folgt:

- Im Gegensatz zu einem DEA darf es bei einem NEA mehrere Startzustände geben.
- Im Gegensatz zu einem DEA dürfen bei einem NEA von einem Zustand aus mehrere verschiedene Folgezustände mit demselben Zeichen erreichbar sein.
- Für die Akzeptanz eines Wortes aus Σ^* ist es ausreichend, wenn mindestens eine Zustandsfolge von dem NEA so durchlaufen werden kann, dass ein Endzustand erreicht wird. Weitere Zustandsfolgen führen möglicherweise nicht zum Ziel, dies hat aber keine Auswirkung auf die Akzeptanz eines Wortes. Das Wort wird also nur dann verworfen, wenn es keine einzige Zustandsfolge gibt, die zum Schluß zu einem Endzustand führt.

Die ersten beiden Eigenschaften erlauben es einem NEA, unter mehreren Möglichkeiten beim Start bzw. bei der Verarbeitung des nächsten Eingabesymbols zu wählen, d.h. der nächste Schritt ist i.A. *nichtdeterministisch*. Dies ist bei einem DEA nicht der Fall. Hier gibt es immer nur (wenn überhaupt) eine eindeutig (also *deterministisch*) festgelegte Fortsetzungsmöglichkeit.

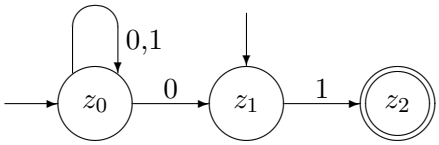
2.2.5 Beispiel Das folgende Diagramm zeigt den Zustandsgraph eines NEA M.



Der NEA M kann im Zustand z_1 beginnen. Von hier aus ist nur noch mit einer Eins der Endzustand z_2 erreichbar, und von z_2 aus geht es nicht weiter. Wenn also nur z_1 ein Startzustand wäre, so würde M nur das eine Wort „1“ akzeptieren. Der Zustand z_0 ist jedoch ebenfalls ein Startzustand. M kann beliebig lange unter Verbrauch von Nullen und Einsen in diesem Zustand verbleiben. Alternativ kann durch Verbrauch einer Null auch der Zustand z_1 erreicht werden. Von hier aus führt, wie bereits bekannt, nur noch eine einzelne Eins zum Ziel. Also erkennt der NEA M insgesamt die Sprache

$$L(M) = \{1\} \cup \{x01 \mid x \in \{0, 1\}^*\} \ .$$

Um Platz zu sparen, darf man (übrigens auch beim Zustandsgraph eines DEAs) mehrere Übergangsalternativen durch Kommata getrennt an eine Kante schreiben. Den obigen NEA könnte man also auch wie folgt darstellen:



In der folgenden Definition wird ein NEA formal beschrieben.

2.2.6 Definition Ein *nichtdeterministischer endlicher Automat* (*NEA*) ist ein 5–Tupel $M = (Z, \Sigma, \delta, \mathcal{S}, E)$ mit den folgenden Komponenten:

- Z ist eine endliche Zustandsmenge.
- Σ ist ein endliches Eingabealphabet mit $Z \cap \Sigma = \emptyset$.
- $\delta : Z \times \Sigma \longrightarrow \mathcal{P}(Z)$ bezeichnet die Übergangsfunktion.
- $\mathcal{S} \subseteq Z$ legt die Menge der Startzustände fest (sinnvollerweise gilt $\mathcal{S} \neq \emptyset$).
- $E \subseteq Z$ ist die Menge der akzeptierenden Endzustände.

Im Beispiel 2.2.5 gilt demnach $Z = \{z_0, z_1, z_2\}$, $\Sigma = \{0, 1\}$, $\mathcal{S} = \{z_0, z_1\}$ sowie

$$\delta(z_0, 0) = \{z_0, z_1\} \ , \quad \delta(z_0, 1) = \{z_0\} \ , \quad \delta(z_1, 1) = \{z_2\} \ .$$

Bei NEAs kann man δ als eine totale (also überall definierte) Funktion auffassen. Falls es nämlich für einen bestimmten Zustand $z \in Z$ sowie ein bestimmtes Symbol $a \in \Sigma$ keine Übergänge gibt, so gilt $\delta(z, a) = \emptyset$. In Beispiel 2.2.5 gilt dies implizit für alle vorhin nicht aufgeführten Übergänge, z.B. für $\delta(z_2, 1)$.

Ähnlich wie beim DEA wird auch beim NEA die Übergangsfunktion $\delta : Z \times \Sigma \longrightarrow \mathcal{P}(Z)$ wie folgt zu einer Funktion $\hat{\delta} : \mathcal{P}(Z) \times \Sigma^* \longrightarrow \mathcal{P}(Z)$ verallgemeinert:

$$\hat{\delta}(s, x) := \begin{cases} s & \text{falls } x = \varepsilon \\ \hat{\delta}\Big(\bigcup\limits_{z \in s} \delta(z, a), y \Big) & \text{falls } \exists a \in \Sigma \colon \exists y \in \Sigma^* \colon \ x = ay \end{cases}$$

Offenbar ist $\hat{\delta}(s, x)$ die Menge aller Zustände, die man ausgehend von den Zuständen in s nach Abarbeitung der Eingabe x erreichen kann. In Beispiel 2.2.5 gilt z.B.

$$\hat{\delta}(\{z_0, z_2\}, 101) = \hat{\delta}(\{z_0\}, 01) = \hat{\delta}(\{z_0, z_1\}, 1) = \{z_0, z_2\} \ .$$

2.2.7 Definition Die von einem NEA $M = (Z, \Sigma, \delta, \mathcal{S}, E)$ *akzeptierte Sprache* ist definiert als

$$L(M) \ := \ \{x \in \Sigma^* \,|\, \hat{\delta}(\mathcal{S}, x) \cap E \neq \emptyset\} \ ,$$

d.h. nach Verarbeitung der gesamten Eingabe x muss es unter allen erreichbaren Zuständen mindestens einen Endzustand geben.

Das Verhalten eines DEA kann man recht einfach durch ein entsprechendes Programm nachbilden (also *simulieren*). In der Programmiersprache Java würde man z.B. in einem ersten Schritt die Zustandsmenge als auch das Eingabealphabet durch die Indexmengen $\{0,\ldots,|Z|-1\}$ bzw. $\{0,\ldots,|\Sigma|-1\}$ ersetzen. Die Übergangsfunktion wäre dann nichts anderes als eine Abbildung

$$\delta : \{0,\ldots,|Z|-1\} \times \{0,\ldots,|\Sigma|-1\} \longrightarrow \{0,\ldots,|Z|-1\} \ ,$$

die leicht durch ein zweidimensionales `int`–Array implementiert werden kann. Insbesondere lässt sich bei einem DEA M das Wortproblem („ist ein Wort $x \in \Sigma^*$ in $L(M)$ enthalten?“) in $O(|x|)$ vielen Schritten, also in linearer Zeit lösen. Hierzu führt man einfach die Verarbeitung von x unter M konkret durch. Bei einem NEA ist dies nicht so einfach, den man müsste die unterschiedlichen Wahlmöglichkeiten bei den Übergängen nacheinander durchprobieren, was zu exponentieller Laufzeit führen kann. Glücklicherweise lässt sich dieses Problem überraschend einfach aus der Welt schaffen. Wir werden nämlich nachfolgend sehen, dass DEAs und NEAs gleich mächtige Konzepte sind und insbesondere zu jedem NEA ein gleichwertiger DEA existiert, mit dem man das Wortproblem dann wieder in linearer Zeit berechnen kann.

Zunächst einmal ist jeder DEA de facto auch gleichzeitig ein NEA, da man die zusätzlich erlaubten Möglichkeiten eines NEAs ja nicht unbedingt ausnutzen muss. Der Zustandsgraph eines DEA kann daher ohne Änderung auch als Zustandsgraph eines NEA angesehen werden. Überraschenderweise kann aber auch jeder NEA in einen *äquivalenten* DEA (der also die gleiche Sprache erkennt) umgewandelt werden. Der wichtigste Schritt hierzu basiert auf der verallgemeinerten Übergangsfunktion $\hat{\delta}$ des NEA. Man überlegt sich, welche Teilmenge von Zuständen man von einer anderen Teilmenge von Zuständen mit einem einzelnen Symbol aus Σ erreichen kann. Falls man auf eine neue Teilmenge von Zuständen stößt, die noch nicht untersucht wurde, so prüft man auch für diese Menge, welche weiteren Mengen man von dort aus erreichen kann. Als erste solche Menge betrachtet man die Menge aller Startzustände $\mathcal{S}$.

2.2.8 Beispiel Die Menge aller Startzustände $\mathcal{S}$ im Beispiel 2.2.5 ist $\{z_0, z_1\}$. Mit einer Null erreicht man von z_0 aus die Zustände z_0 und z_1, aber von z_1 aus gibt es keinen passenden Übergang. Insgesamt gilt also

$$\hat{\delta}(\{z_0, z_1\}, 0) = \{z_0, z_1\} \ ,$$

d.h. wir erreichen von $\{z_0, z_1\}$ aus mit einer Null wieder die gleiche Menge. Für das andere Symbol „1“ gilt dies nicht; hier haben wir

$$\hat{\delta}(\{z_0, z_1\}, 1) = \{z_0, z_2\} \ .$$

Für die neue Zustandsmenge $\{z_0, z_2\}$ führen wir deshalb die Analyse fort. Es gilt

$$\hat{\delta}(\{z_0, z_2\}, 0) = \{z_0, z_1\}$$

(d.h. hier haben wir nichts weiter zu tun, da wir die Menge $\{z_0, z_1\}$ bereits untersucht haben) und

$$\hat{\delta}(\{z_0, z_2\}, 1) = \{z_0\} \ .$$

Mit der Eins sind wir also wieder auf eine neue Zustandsmenge gestoßen. Wegen

$$\hat{\delta}(\{z_0\}, 0) = \{z_0, z_1\}$$

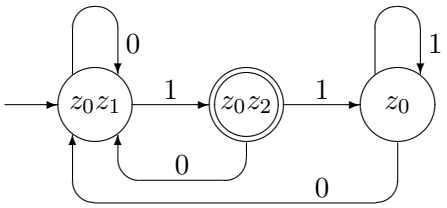
und

$$\hat{\delta}(\{z_0\}, 1) = \{z_0\}$$

ergeben sich jedoch keine weiteren unbekannten Zustandsmengen, und unsere Analyse endet zunächst an dieser Stelle.

Wenn man als Zustandsmenge jetzt nicht Z, sondern die Potenzmenge $\mathcal{P}(Z)$ betrachtet, so ist $\hat{\delta}$ eine *deterministische* Übergangsfunktion auf dieser neuen Zustandsmenge. Damit ist die wichtigste Komponente des zu erzeugenden DEA bereits bekannt. Als Startzustand wählt man die Menge $\mathcal{S} \in \mathcal{P}(Z)$. Ferner wird jede bisherige Menge von Zuständen, die zumindest einen Endzustand aus E enthält, als ein Endzustand des neuen DEA angesehen.

2.2.9 Beispiel In Fortführung des Beispiels 2.2.8 ist der neue Startzustand $\mathcal{S} = \{z_0, z_1\}$, und der einzige Endzustand ist $\{z_0, z_2\}$, weil dieser als einziger den bisherigen Endzustand z_2 enthält. Zusammen mit der bereits konstruierten Übergangsfunktion ergibt sich daraus der folgende Automat (eine Zustandsmenge wie $\{z_0, z_1\}$ wird aus Platzgründen durch z_0z_1 notiert):



Dieser Automat ist jetzt deterministisch und akzeptiert, wie man leicht sieht, ebenfalls die Menge $\{1\} \cup \{x01 \mid x \in \{0,1\}^*\}$.

Wir halten die beschriebene Technik in dem folgenden Satz fest.

2.2.10 Satz (Rabin[1], Scott[2]) Zu jedem NEA M existiert ein äquivalenter DEA M' mit $L(M) = L(M')$.

[1] Michael O. Rabin, *1931 Breslau, israelischer Informatiker, Professor für Infomatik in Harvard.

[2] Dana S. Scott, *1932 Berkeley, US–amerikanischer Mathematiker und Logiker, emeritierter Professor für Infomatik, zuletzt in Pittsburgh.

Beweis: Sei $M = (Z, \Sigma, \delta, \mathcal{S}, E)$ ein NEA mit verallgemeinerter Übergangsfunktion $\hat{\delta}$. Ein DEA $M' = (Z', \Sigma, \delta', z_0', E')$ mit $L(M) = L(M')$ kann durch die sog. *Potenzmengenkonstruktion* erzeugt werden. Zu diesem Zweck setzt man

- $Z' := \mathcal{P}(Z)$,
- $\delta'(s, a) := \bigcup_{z \in s} \delta(z, a) = \hat{\delta}(s, a)$,
- $z_0' := \mathcal{S}$, (man beachte: $\mathcal{S} \subseteq Z$, also $z_0' \in Z' = \mathcal{P}(Z)$), und
- $E' := \{z \in Z' \mid z \cap E \neq \emptyset\}$.

Sei nun $x \in \Sigma^*$ beliebig. Sei n die Länge von x, also $x = a_1 a_2 \dots a_n$ für passende $a_i \in \Sigma$ $(i = 1, \dots, n)$. Dann gilt:

$$
\begin{aligned}
& x \in L(M) \\
\iff\ & \hat{\delta}(\mathcal{S}, x) \cap E \neq \emptyset \\
\iff\ & \text{es gibt eine Folge von Teilmengen } z_1, \dots, z_n \subseteq Z \text{ mit} \\
& \hat{\delta}(\mathcal{S}, a_1) = z_1,\ \hat{\delta}(z_1, a_2) = z_2,\ \dots,\ \hat{\delta}(z_{n-1}, a_n) = z_n \text{ und } z_n \cap E \neq \emptyset \\
\iff\ & \text{es gibt eine Folge von (neuen) Zuständen } z_1, \dots, z_n \in Z' \text{ mit} \\
& \delta'(z_0', a_1) = z_1,\ \delta'(z_1, a_2) = z_2,\ \dots,\ \delta'(z_{n-1}, a_n) = z_n \text{ und } z_n \in E' \\
\iff\ & x \in L(M') \ .
\end{aligned}
$$

Wenn aber für alle $x \in \Sigma^*$ stets $x \in L(M) \iff x \in L(M')$ gilt, so sind die beiden Sprachen $L(M)$ und $L(M')$ identisch. □

Es sei bemerkt, dass evtl. viele der Zustände in $Z' = \mathcal{P}(Z)$ von z_0' aus mit der Übergangsfunktion δ' gar nicht erreichbar sind. Dies kann man z.B. gut an der bereits durchgeführten Konstruktion in den Beispielen 2.2.8 und 2.2.9 sehen. Hier sind wir mit nur drei Zuständen ausgekommen, obwohl nach der Potenzmengenkonstruktion eigentlich $|\mathcal{P}(Z)| = 2^3 = 8$ Zustände zu generieren gewesen wären. Es ist daher immer sinnvoll, zunächst wie beschrieben bei $z_0' = \mathcal{S}$ zu beginnen und die sich durch $\hat{\delta}$ ergebenden Zustandsmengen aufzuschreiben. Nur falls man dabei auf neue Zustandsmengen stößt, die noch nicht untersucht wurden, macht man von dort aus entsprechend weiter.

Prinzipiell brauchen wir also zwischen DEAs und NEAs nicht mehr zu unterscheiden und können sie in der Klasse der *endlichen Automaten* (*EA*) zusammenfassen. Man beachte jedoch die evtl. stark anwachsende Anzahl von Zuständen bei dieser Simulation. Im schlimmsten Fall kann bei der Potenzmengenkonstruktion aus einem NEA mit n Zuständen ein deterministischer Automat mit 2^n Zuständen entstehen.

Wir zeigen als nächstes, dass endliche Automaten (fast) genau die regulären Sprachen erkennen.

2.2.11 Satz Jede durch einen EA M erkannte Sprache L mit $\varepsilon \notin L$ ist regulär.

Beweis: Wir können *ohne Beschränkung der Allgemeinheit* (abgekürzt *o.B.d.A*) davon ausgehen, dass M deterministisch ist, denn ansonsten könnten wir M gemäß Satz 2.2.10 in einen entsprechenden DEA umformen. Sei also M durch einen DEA $(Z, \Sigma, \delta, z_0, E)$ gegeben. Dann definieren wir eine Typ 3–Grammatik $G = (V, \Sigma, P, S)$, deren Komponenten wie folgt festgelegt sind:

- $V := Z$
- $S := z_0$
- Für alle Übergänge $\delta(z, a) = z'$ existiert in P die Regel $z \to az'$.
- Für alle Übergänge $\delta(z, a) = z'$ mit $z' \in E$ gibt es in P eine weitere Regel $z \to a$.

Für das Beispiel 2.2.1 auf Seite 40 wäre also $V = \{z_0, z_1, z_2, z_3\}$, $\Sigma = \{a, b\}$, $S = z_0$, und

$$P = \{z_0 \to az_1 \,|\, bz_3 \,|\, b,\ z_1 \to az_2 \,|\, bz_3 \,|\, b,\ z_2 \to az_3 \,|\, a\}\ .$$

Wir zeigen nun, dass bei Einhaltung dieser Konstruktionsvorschrift stets $L(G) = L(M)$ erfüllt ist. Es gilt nämlich für alle nichtleeren Wörter $a_1a_2 \ldots a_n \in \Sigma^+$:

$$\begin{aligned}
& a_1a_2 \ldots a_n \in L(M) \\
\iff\ & \text{es gibt eine Folge von Zuständen } z_1, \ldots, z_n \in Z \text{ mit} \\
& \delta(z_0, a_1) = z_1,\ \delta(z_1, a_2) = z_2,\ \ldots,\ \delta(z_{n-1}, a_n) = z_n \text{ sowie } z_n \in E \\
\iff\ & \text{es gibt eine Folge von Variablen } z_1, \ldots, z_n \in V \text{ mit} \\
& z_0 \to a_1z_1,\ \ z_1 \to a_2z_2,\ \ \ldots,\ \ z_{n-2} \to a_{n-1}z_{n-1},\ \ z_{n-1} \to a_n\ \in P \\
\iff\ & \text{es gibt eine Folge von Variablen } z_1, \ldots, z_n \in V \text{ mit} \\
& z_0 \Rightarrow_G a_1z_1 \Rightarrow_G a_1a_2z_2 \Rightarrow_G \ldots \Rightarrow_G a_1a_2 \ldots a_{n-1}z_{n-1} \Rightarrow_G a_1a_2 \ldots a_n \\
\iff\ & S \Rightarrow_G^* a_1a_2 \ldots a_n \\
\iff\ & a_1a_2 \ldots a_n \in L(G)\ .
\end{aligned}$$

Ein Wort wird also genau dann von dem DEA M akzeptiert, wenn es durch die skizzierte Grammatik erzeugbar ist. Folglich sind die zugehörigen beiden Sprachen identisch.

Beachten Sie, dass ein Automat prinzipiell sehr wohl das leere Wort akzeptieren kann (falls nämlich ein Startzustand gleichzeitig ein Endzustand ist). Wie jede reguläre Grammatik wäre jedoch auch die konstruierte Grammatik niemals in der Lage, das leere Wort zu erzeugen. Aus diesem Grund wurde im Satz die Bedingung $\varepsilon \notin L$ vorausgesetzt. □

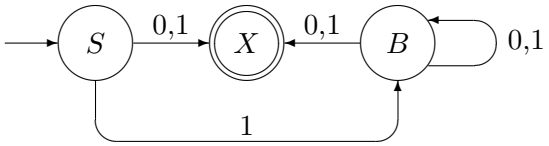
Die Umkehrung von Satz 2.2.11 gilt ebenfalls:

2.2.12 Satz Zu jeder regulären Grammatik G gibt es einen NEA M mit $L(M) = L(G)$.

Beweis: Sei $G = (V, \Sigma, P, S)$ eine reguläre Grammatik. Wir konstruieren nun einen NEA $M = (Z, \Sigma, \delta, \mathcal{S}, E)$, der wie gewünscht $L(M) = L(G)$ erfüllen wird (was wir im Anschluss beweisen werden). Dazu legen wir fest:

- $Z := V \dot\cup \{X\}$ (dabei ist X ein neues Nichtterminal)
- $\mathcal{S} := \{S\}$
- $E := \{X\}$
- Ist $A \to aB$ eine Produktion aus P, so existiert ein Übergang $B \in \delta(A, a)$.
- Ist $A \to a$ eine Produktion aus P, so existiert ein Übergang $X \in \delta(A, a)$.

Wenn wir z.B. nochmals die Grammatik $G = (V, \Sigma, P, S)$ mit $V = \{S, B\}$, $\Sigma = \{0, 1\}$ und $P = \{S \to 0 \,|\, 1 \,|\, 1B,\ B \to 0 \,|\, 1 \,|\, 0B \,|\, 1B\}$ aus Beispiel 2.1.9 (Seite 36) betrachten, so ergibt sich mit der angegebenen Konstruktionsvorschrift der folgende NEA:



Dieser Automat akzeptiert gerade genau die von der Grammatik erzeugte Sprache

$$L(G) = \{0\} \cup \{1x \,|\, x \in \{0, 1\}^*\} \ .$$

Dies ist wie bereits erwähnt kein Zufall, denn es gilt allgemein (unabhängig von diesem konkreten Beispiel) für alle nichtleeren Wörter $a_1 a_2 \ldots a_n \in \Sigma^+$:

$$\begin{aligned}
& a_1 a_2 \ldots a_n \in L(G) \\
\iff\ & \text{es gibt eine Folge von Variablen } A_1, A_2, \ldots, A_{n-1} \in V \text{ mit} \\
& S \Rightarrow_G a_1 A_1 \Rightarrow_G a_1 a_2 A_2 \Rightarrow_G \ldots \Rightarrow_G a_1 a_2 \ldots a_{n-1} A_{n-1} \Rightarrow_G a_1 a_2 \ldots a_n \\
\iff\ & \text{es gibt eine Folge von Variablen } A_1, A_2, \ldots, A_{n-1} \in V \text{ mit} \\
& S \to a_1 A_1,\ A_1 \to a_2 A_2,\ \ldots,\ A_{n-2} \to a_{n-1} A_{n-1},\ A_{n-1} \to a_n \in P \\
\iff\ & \text{es gibt eine Folge von Zuständen } A_1, A_2, \ldots, A_{n-1} \in Z \text{ mit} \\
& A_1 \in \delta(S, a_1),\ A_2 \in \delta(A_1, a_2),\ \ldots,\ A_{n-1} \in \delta(A_{n-2}, a_{n-1}),\ X \in \delta(A_{n-1}, a_n) \\
\iff\ & a_1 a_2 \ldots a_n \in L(M) \ .
\end{aligned}$$

Für das leere Wort ε ist die Äquivalenz $\varepsilon \in L(G) \iff \varepsilon \in L(M)$ ebenfalls korrekt, denn beide Seiten sind mit Sicherheit falsch: eine reguläre Grammatik kann niemals das leere Wort ableiten, und der konstruierte NEA M besitzt nur einen Endzustand, der nicht zugleich Startzustand ist, d.h. M wird das leere Wort nicht akzeptieren. Also gilt sogar für alle Wörter $x \in \Sigma^*$ die Äquivalenz $x \in L(G) \iff x \in L(M)$ und damit $L(G) = L(M)$. Unsere Konstruktion ist also auch hier korrekt. □

Insgesamt haben wir damit die anfangs behauptete Äquivalenz zwischen regulären Grammatiken und endlichen Automaten konstruktiv gezeigt. Es sei jedoch bemerkt, dass die

konstruierten Automaten und Grammatiken aus den Sätzen 2.2.11 und 2.2.12 zuweilen überflüssigen Ballast mit sich „herumschleppen“. Beispielsweise kann es generierte Produktionen geben, die dann doch nie benötigt werden. Es empfiehlt sich daher immer, nach einer solchen Konstruktion gewissenhaft zu prüfen, ob noch Vereinfachungen möglich sind.

2.3. Reguläre Ausdrücke

Reguläre Sprachen lassen sich auch durch sog. *reguläre Ausdrücke* beschreiben. Reguläre Ausdrücke sind weit verbreitet und tauchen z.B. bei vielen Texteditoren in den Suchfunktionen auf. Für viele Programmiersprachen existieren Bibliotheken, mit denen sich reguläre Ausdrücke direkt verwenden lassen (auch für Java, siehe z.B. das Paket `java.util.regex.Pattern`). Reguläre Ausdrücke kommen auch bei dem Programm `grep` zum Einsatz, welches unter dem Betriebssystem Unix und seinen Varianten zum Suchen von Zeichenketten in Dateien benutzt wird.

Wir definieren als Erstes die *Syntax* und *Semantik* solcher Ausdrücke. Die Syntax gibt dabei die Regeln an, die ein regulärer Ausdruck erfüllen muss. Diese Regeln bauen auf einem bestimmten Grundalphabet Σ auf. Alle regulären Ausdrücke über Σ werden in der Menge $\mathit{Reg}(\Sigma)$ zusammengefasst. Jeder gültige Ausdruck in $\mathit{Reg}(\Sigma)$ repräsentiert dann eine bestimmte Sprache über Σ, die durch die Semantikvorschriften festgelegt wird. Formal bedienen wir uns dazu einer *semantischen Funktion* φ, die von $\mathit{Reg}(\Sigma)$ nach $\mathcal{P}(\Sigma^*)$ abbildet. Für jeden regulären Ausdruck $\gamma \in \mathit{Reg}(\Sigma)$ ist $\varphi(\gamma)$ also ein Element aus $\mathcal{P}(\Sigma^*)$, also eine Teilmenge von Σ^* und damit eine Sprache über Σ.

2.3.1 Definition Die Menge $\mathit{Reg}(\Sigma)$ aller *regulären Ausdrücke* über einem Alphabet Σ wird durch die folgenden Regeln gebildet:

- Jedes Wort w aus Σ^* ist ein regulärer Ausdruck (also $\Sigma^* \subseteq \mathit{Reg}(\Sigma)$) und bezeichnet die Sprache $\varphi(w) := \{w\}$, also diejenige Sprache, die nur aus dem einem Wort w besteht. Wenn also z.B. $\Sigma := \{0, 1\}$ gilt, so sind ε, 01101 und 11110 reguläre Ausdrücke, die die Sprachen $\{\varepsilon\}$, $\{01101\}$ und $\{11110\}$ bezeichnen.
- Um die leere Sprache $\emptyset$ auszudrücken, benutzt man das gleiche Symbol $\emptyset$. Es gilt also $\emptyset \in \mathit{Reg}(\Sigma)$ und $\varphi(\emptyset) := \emptyset$.
- Sind bereits α und β zwei reguläre Ausdrücke (also $\alpha, \beta \in \mathit{Reg}(\Sigma)$), so ist auch $\alpha \,|\, \beta$ ein regulärer Ausdruck. Er bezeichnet die Sprache

  $$\varphi(\alpha \,|\, \beta) := \varphi(\alpha) \cup \varphi(\beta) \ ,$$

  d.h. die Sprachen der beiden Ausdrücke werden durch den „|“ Operator vereinigt. Ist also z.B. wieder $\Sigma := \{0, 1\}$, so ist $010 \,|\, 111$ ein gültiger regulärer Ausdruck, der die Sprache $\{010\} \cup \{111\} = \{010, 111\}$ bezeichnet. Ebenso ist $0011 \,|\, \varepsilon \,|\, \emptyset$ ein gültiger Ausdruck, welcher die Sprache $\{0011\} \cup \{\varepsilon\} \cup \emptyset = \{0011, \varepsilon\}$ repräsentiert.

Natürlich könnte man diese Sprache auch einfacher durch $0011 \mid \varepsilon$ angeben, da die Vereinigung mit der leeren Menge keine Änderungen bewirkt.

- Sind bereits α und β zwei reguläre Ausdrücke (also $\alpha, \beta \in \mathit{Reg}(\Sigma)$), so ist auch $\alpha\beta$ ein regulärer Ausdruck. Er bezeichnet die Sprache

$$\varphi(\alpha\beta) := \varphi(\alpha)\varphi(\beta) := \{xy \mid x \in \varphi(\alpha), y \in \varphi(\beta)\} \ ,$$

  d.h. die durch α und β repräsentierten Sprachen werden *konkateniert*. Jedes Wort in der konkatenierten Sprache wird gebildet, indem man an ein Wort aus der ersten Sprache ein Wort aus der zweiten Sprache anhängt. Im Fall $\Sigma := \{0, 1\}$ wäre z.B. $(0 \mid 1)(11 \mid 101)$ ein regulärer Ausdruck, der die vierelementige Sprache $\{011, 111, 0101, 1101\}$ beschreibt. Dabei ergibt sich z.B. das Wort 111 aus der Konkatenation von $1 \in \varphi(0 \mid 1)$ und $11 \in \varphi(11 \mid 101)$.

- Ist bereits α ein regulärer Ausdruck (also $\alpha \in \mathit{Reg}(\Sigma)$), so ist auch $(\alpha)^\star$ ein regulärer Ausdruck. Er bezeichnet die *Kleenesche Hülle* der Sprache $\varphi(\alpha)$, also

$$\varphi((\alpha)^\star) := (\varphi(\alpha))^* := \{x_1 x_2 \dots x_k \mid k \geq 0, x_i \in \varphi(\alpha)\} \ .$$

  Man fügt also mehrere Wörter aus $\varphi(\alpha)$ durch Konkatenation zu neuen Wörtern zusammen. Im Fall $\Sigma := \{0, 1\}$ repräsentiert zum Beispiel $(00)^\star$ die Sprache $\{\varepsilon, 00, 0000, 000000, \dots\}$, denn da der reguläre Ausdruck 00 nur das Wort 00 repräsentiert, kann man nur Zeichenketten aus Nullen mit gerader Länge bilden. (Beachten Sie, dass auch *null* Wörter „hintereinander“ geschrieben werden dürfen, so dass sich das leere Wort ε ergibt.) Wichtig: der reguläre Ausdruck $00^\star$ erzeugt etwas anderes, nämlich die Sprache $\{0, 00, 000, 0000, \dots\}$. Der „*“ Operator wirkt nämlich immer nur auf das unmittelbar davorstehende Zeichen (es sei denn, man verwendet wie im vorherigen Beispiel Klammern).

- Nicht unbedingt notwendig, aber bequem ist auch die Verwendung des „$^+$“ Operators. Ist bereits α ein regulärer Ausdruck (also $\alpha \in \mathit{Reg}(\Sigma)$), so ist auch $(\alpha)^+$ ein regulärer Ausdruck, der den Ausdruck $(\alpha)(\alpha)^\star$ abkürzt, d.h. es gilt

$$\varphi((\alpha)^+) = (\varphi(\alpha))^+ := \{x_1 x_2 \dots x_k \mid k \geq 1, x_i \in \varphi(\alpha)\} \ .$$

  Im Gegensatz zu $\varphi((\alpha)^\star)$ muss also mindestens ein Wort der Ausgangssprache in den neuen Wörtern verwendet werden. Beispielsweise repräsentiert $(00)^+$ die Sprache $\{00, 0000, 000000, \dots\}$. Das leere Wort ε ist also nur dann in $\varphi((\alpha)^+)$ enthalten, wenn bereits $\varepsilon \in \varphi(\alpha)$ gilt.

2.3.2 Beispiel Häufig kann man reguläre Ausdrücke vereinfachen. Wie geben hierfür einige Beispiele an (das zugrundeliegende Alphabet sei $\Sigma := \{a, b\}$):

- $b \mid b^+ \mid \varepsilon$ erzeugt die Sprache $\{b\} \cup \{b\}^+ \cup \{\varepsilon\} = \{\varepsilon, b, bb, bbb, \dots\} = \{b\}^*$, die man auch einfacher durch $b^\star$ beschreiben könnte.

- $(a \mid b)\emptyset$ erzeugt die leere Sprache $\emptyset$. Denn jedes Wort aus $\varphi(a \mid b)$ ist mit einem Wort aus $\varphi(\emptyset) = \emptyset$ zu konkatenieren. Letzteres ist aber unmöglich, da die leere Menge überhaupt kein einziges Wort enthält. Also kann man auch insgesamt keine Wörter bilden. Der reguläre Ausdruck $\emptyset$ würde also völlig ausreichen.
- $\emptyset^\star$ erzeugt die Sprache $\{\varepsilon\}$. Denn jedes Wort aus $\varphi(\emptyset^\star)$ wird gebildet, indem man endlich viele Wörter aus $\varphi(\emptyset) = \emptyset$ hintereinanderschreibt — aber in der leeren Menge sind wie gesagt keine Wörter verfügbar. Trotzdem kann man natürlich immer noch null Wörter hintereinander schreiben und damit das leere Wort ε erzeugen. Insgesamt könnte man also einfach den regulären Ausdruck ε verwenden.
- $(\alpha)^{\star\star}$ entspricht dem Ausdruck $(\alpha)^\star$, ganz gleich, um welchen konkreten Ausdruck es sich bei α handelt. Der Ausdruck α erzeugt nämlich irgendeine beliebige Sprache $L := \varphi(\alpha)$, und $(\alpha)^\star$ erzeugt somit L^*. Folglich erzeugt $((\alpha)^\star)^\star = (\alpha)^{\star\star}$ die Sprache $(L^*)^*$, und diese ist wiederum mit L^* identisch (siehe Übungsblatt #4).

Manchmal erfordert das Vereinfachen von regulären Ausdrücken jedoch etwas tiefergehende Überlegungen.

2.3.3 Beispiel Sei $\Sigma = \{0, 1\}$. Der reguläre Ausdruck $(01^\star \mid 0)^+$ soll vereinfacht werden. Wir schauen uns zunächst den inneren Ausdruck $01^\star \mid 0$ an. Es gilt:

$$\varphi(01^\star \mid 0) = \varphi(01^\star) \cup \varphi(0) = \{0, 01, 011, 0111, 01111, \dots\} \cup \{0\} \ ,$$

d.h. die einzelne Null ist in der Vereinigung doppelt vertreten und somit überflüssig. Also können wir den inneren Ausdruck zu $01^\star$ und den ursprünglichen Ausdruck zu $(01^\star)^+$ vereinfachen. Eine weitere Vereinfachung ist möglich, wenn wir uns die von dem inneren Ausdruck $01^\star$ beschriebene Sprache $\{0, 01, 011, 0111, \dots\}$ noch etwas genauer ansehen. Schreibt man mehrere dieser Wörter hintereinander, so beginnt das resultierende Wort sicherlich immer noch mit einer führenden Null. Folglich ist $\varphi((01^\star)^+)$ eine Teilmenge von $\varphi(0(0 \mid 1)^\star)$, denn der Ausdruck $0(0 \mid 1)^\star$ beschreibt die Sprache aller Wörter, bei denen sich die Folge hinter der führenden Null beliebig aus Einsen und Nullen zusammensetzen darf. Vielleicht etwas überraschend gilt jedoch die Teilmengenbeziehung auch in der umgekehrten Richtung, d.h. beide Ausdrücke beschreiben die gleiche Sprache. Ist nämlich $y \in \varphi(0(0 \mid 1)^\star)$ ein Wort aus der zweiten Sprache, so können wir y problemlos aus Wörtern von $\varphi(01^\star)$ zusammensetzen, indem wir bei jeder Null mit einem neuen solchen Wort beginnen. Diese sind nachfolgend für das Beispiel $y = 0110010111$ einzeln unterstrichen:

$$0110010111 \ = \ \underline{011}\,\underline{0}\,\underline{01}\,\underline{0111} \ .$$

Also ist der anfängliche Ausdruck $(01^\star \mid 0)^+$ insgesamt gesehen zu dem Ausdruck $0(0 \mid 1)^\star$ gleichwertig.

Alle Sprachen über Σ mit nur endlich vielen Wörtern sind durch reguläre Ausdrücke über Σ beschreibbar. Gilt nämlich $L := \{x_1, x_2, \dots, x_k\} \subseteq \Sigma^*$, so ist

$$\gamma := x_1 \mid x_2 \mid \dots \mid x_k$$

ein regulärer Ausdruck mit $\varphi(\gamma) = L$.

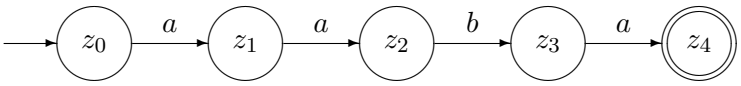
Reguläre Ausdrücke sind selbst auch Zeichenketten, also Wörter über einem bestimmten Alphabet. Wie man sieht, haben wir neben den Symbolen aus Σ auch die Zeichen $\emptyset, \varepsilon, (,), |, ^\star$ und $^+$ verwendet. Reguläre Ausdrücke werden also insgesamt über dem Alphabet $\Sigma \dot\cup \{\emptyset, \varepsilon, (,), |, ^\star, ^+\}$ gebildet. Allerdings sind natürlich nicht alle Wörter über dem Alphabet $\Sigma \dot\cup \{\emptyset, \varepsilon, (,), |, ^\star, ^+\}$ korrekte reguläre Ausdrücke. Beispielsweise ist $^+\emptyset\,|$ kein regulärer Ausdruck, denn vor dem „$^+$“ Operator und hinter dem „$|$“ Operator muss immer etwas stehen. Man kann aber sagen: $Reg(\Sigma)$ ist die kleinste Teilmenge von $(\Sigma \dot\cup \{\emptyset, \varepsilon, (,), |, ^\star, ^+\})^*$, die die obigen Eigenschaften erfüllt.

2.3.4 Satz Jede durch einen regulären Ausdruck $\gamma \in Reg(\Sigma)$ beschriebene Sprache $L := \varphi(\gamma)$ wird durch einen passenden DEA akzeptiert.

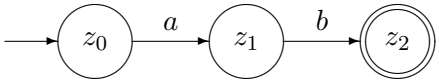
Beweis: Der Beweis erfolgt konstruktiv. Wir analysieren dazu den Ausdruck γ von „innen nach außen“ (so wie man auch einen geklammerten mathematischen Term von innen nach außen ausrechnet) und erstellen dabei für die bis dahin analysierten Teile nach und nach immer größer werdende DEAs, die die jeweils zugehörigen Sprachen akzeptieren. Am Ende wird sich ein DEA ergeben, der genau die Sprache $\varphi(\gamma)$ erkennt. Wir werden die Konstruktion aktiv an einem einfachen Beispiel mitverfolgen, es soll nämlich für den regulären Ausdruck $\gamma := ((ab\,|\,b)b)^\star$ ein passender DEA gefunden werden.

In einem ersten Schritt bildet man DEAs für alle in γ auftauchenden Ausdrücke α der Form ε und $\emptyset$ sowie für alle vorkommenden Wörter $w \in \Sigma^+$. Passende Automaten lassen sich jeweils leicht angeben:

- Im Fall $\alpha = \varepsilon$ besteht er nur aus einem einzigen Zustand, der zugleich Start– und Endzustand ist. Es gibt keine Übergänge.
- Im Fall $\alpha = \emptyset$ besteht der Automat ebenfalls nur aus einem einzigen Zustand, der diesmal zwar wieder Startzustand, aber kein Endzustand ist.
- Falls $\alpha = w$ für ein $w \in \Sigma^+$ gilt, so konstruiert man einen DEA mit $|w| + 1$ Zuständen, der nacheinander die einzelnen Symbole von w verarbeitet. Der erste Zustand ist der Startzustand, und der letzte Zustand der einzige Endzustand. Für das Wort $aaba$ ergibt sich so z.B. dieser DEA:



In unserem Beispiel $((ab\,|\,b)b)^\star$ sind die drei Wörter ab, b und b enthalten. Das b ist dabei zwar doppelt vorhanden, es muss jedoch trotzdem jeweils eine individulle Kopie angefertigt werden. Somit startet unsere Konstruktion mit diesen drei Automaten:



Weil die Automaten später miteinander verkettet werden, wurden die Zustandsnamen vorsorglich bereits unterschiedlich gewählt.

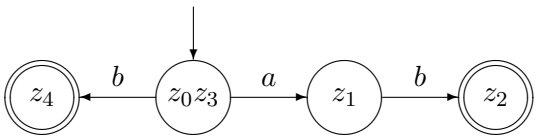
Ausgehend von seinen Basiskomponenten entsteht ein regulärer Ausdruck nun dadurch, dass wiederholt Verknüpfungsregeln aus Def. 2.3.1 zur Anwendung kommen. Es handelt sich dabei um die Konkatenation, die Vereinigung und die Kleenesche Hülle. In unserem Beispiel $((ab \,|\, b)b)^\star$ wurden z.B. erst die beiden Basisausdrücke ab und b mit dem Vereinigungsoperator $|$ zu $(ab \,|\, b)$ zusammengefasst, anschließend wurde dieser Ausdruck mit dem dritten Basisausdruck b konkateniert, und zum Schluss kam der Kleenesche Hüllenoperator zur Anwendung. Wir werden nun zeigen, dass man diese Schritte bei den Automaten parallel nachvollziehen kann.

Als erstes behandeln wir den Fall, dass zwei reguläre Ausdrücke α und β (für die wir bereits passende DEAs M_α und M_β vorliegen haben) durch den Vereinigungsoperator $|$ zu einem Ausdruck $\alpha \,|\, \beta$ zusammen gefasst werden. Dann braucht man sich in Gedanken nur die Zustandsgraphen von M_α und M_β nebeneinander hinzumalen. Das entstehende Gebilde kann man als einen NEA mit zwei Startzuständen auffassen, wobei ein Startzustand durch M_α und ein weiterer durch M_β beigesteuert wird. Wenn man nun ein Wort durch diesen NEA verarbeiten lassen möchte, so muss man entweder in dem Startzustand von M_α oder in dem Startzustand von M_β anfangen. Im ersten Fall werden genau die Wörter aus $L(M_\alpha) = \varphi(\alpha)$ akzeptiert, im zweiten Fall die Wörter aus $L(M_\beta) = \varphi(\beta)$.

Insgesamt akzeptiert der NEA also alle Wörter aus $\varphi(\alpha) \cup \varphi(\beta) = \varphi(\alpha \,|\, \beta)$. Wir wandeln den NEA dann anschließend wieder in einen DEA um.

Unser Beispiel $((ab \,|\, b)b)^\star$ enthält den Teilausdruck $(ab \,|\, b)$, der sich aus den beiden Basisausdrücken $\alpha := ab$ und $\beta := b$ zusammensetzt. Die beiden zugehörigen DEAs M_α und M_β sind bereits aus dem ersten Schritt bekannt:

Fasst man beide DEAs als einen gemeinsamen NEA auf, so akzeptiert dieser wie gewünscht die Wörter ab und b. Wir wandeln den NEA in einen DEA um und schreiben aus Platzgründen wieder Mengen wie z.B. $\{z_0, z_3\}$ als z_0z_3:



Nun betrachten wir den Fall, dass zwei reguläre Ausdrücke α und β (für die wir wieder bereits passende DEAs M_α und M_β vorliegen haben) zu dem Ausdruck $\alpha\beta$ konkateniert werden. Auch dann malt man sich in Gedanken die Zustandsgraphen von M_α und M_β nebeneinander hin. Für jeden Übergang von einem Zustand z von M_α zu einem Endzustand von M_α wird nun zusätzlich ein genauso beschrifteter Übergang von z zu dem Startzustand von M_β hinzugefügt. Wenn man also vom Startzustand von M_α ausgehend ein Wort $v \in \varphi(\alpha)$ verarbeitet und deshalb einen Endzustand von M_α erreicht (da M_α ja genau die Sprache $\varphi(\alpha)$ akzeptiert), so sorgen die zusätzlichen Übergänge dafür, dass man mit dem letzten Symbol von v genauso gut einen Startzustand von M_β erreichen kann.

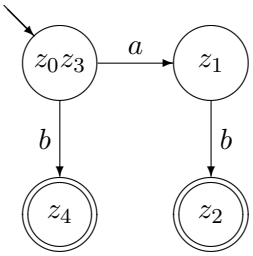
Sind alle neuen Übergänge eingezeichnet, setzen wir abschließend die Endzustände von M_α zu normalen Zuständen zurück.

Wenn man nun von dem Startzustand von M_α aus ausgehend ein Wort verarbeitet, so ist man zwischendurch gezwungen, einen der neuen Übergänge zu benutzen, bei dem Startzustand von M_β weiterzumachen und von dort aus einen der Endzustände von M_β zu erreichen (denn andere Endzustände gibt es ja nicht mehr). Jedes Wort x, welches von dem Startzustand von M_α aus akzeptiert wird, setzt sich also aus der Konkatenation zweier Wörter v und w zusammen, wobei das erste aus $\varphi(\alpha)$ und das zweite aus $\varphi(\beta)$ stammt. Also stammt das Wort $x = vw$ wie gewünscht aus $\varphi(\alpha)\varphi(\beta) = \varphi(\alpha\beta)$.

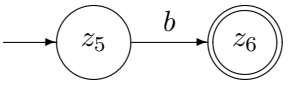
Das letzte Zeichen von $v \in \varphi(\alpha)$ wird immer für einen der neuen Übergänge verwendet, um den Startzustand von M_β zu erreichen. Falls $\varphi(\alpha)$ das leere Wort enthält, könnte aber $v = \varepsilon$ gelten, so dass es gar kein solches Zeichen gibt. Dies ist jedoch kein Problem, denn in dem konstruierten NEA ist der Startzustand von M_β immer noch aktiv, so dass die Verarbeitung auch direkt dort beginnen kann. Im Fall $\varepsilon \notin \varphi(\alpha)$ muss jedoch der Startzustand von M_β deaktiviert werden, um die Verarbeitung eines nichtleeren Präfixes aus M_α zu erzwingen.

Wie schon bei der vorherigen Konstruktionsvorschrift wandeln wir den NEA anschließend noch in einen DEA um.

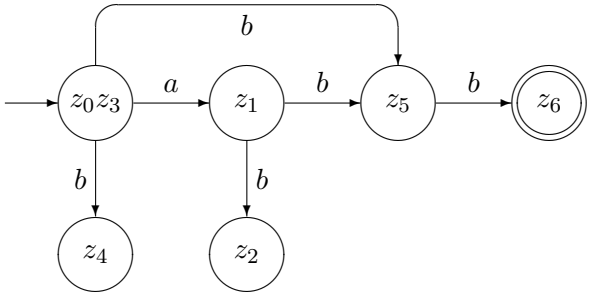
In unserem Beispiel $((ab \mid b)b)^\star$ werden die beiden Teilausdrücke $(ab \mid b)$ und b konkateniert. Für den ersten Ausdruck hatten wir uns den zugehörigen DEA M_α soeben hergeleitet:



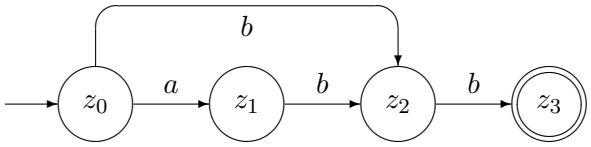
Der zweite DEA M_β ist uns noch aus dem ersten Schritt bekannt:



Die angegebene Konstruktion ergibt dann den folgenden Automat:



Die beiden früheren Endzustände z_2 und z_4 sind nun nutzlos und können entfernt werden. Wir nehmen dies zum Anlass, alle Zustände nochmals neu durchzunummerieren:

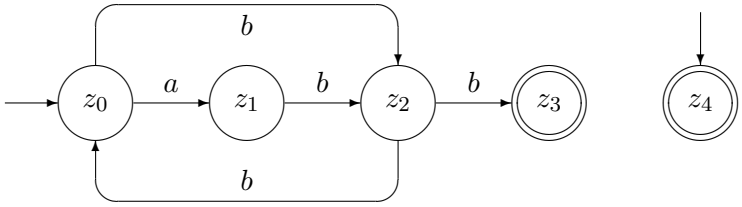


Dieser Automat ist zufälligerweise bereits deterministisch, eine Umwandlung in einen DEA ist daher entbehrlich.

Als letztes untersuchen wir jetzt noch die Anwendung des Kleeneschen Sternoperators. Der Ausdruck sei also von der Form $(\alpha)^\star$, wobei wir wieder von der Existenz eines bereits konstruierten DEAs M_α ausgehen können. Gesucht ist dann ein NEA für die Sprache $\varphi((\alpha)^\star) = (\varphi(\alpha))^*$. Wir verändern dazu den Zustandsgraph von M_α wie folgt. Zu jedem Übergang von einem Zustand z zu einem Endzustand wird ein identisch beschrifteter Übergang von z zu dem Startzustand hinzugefügt. Durch diese „Rückkopplung“ erreicht man folgenden Effekt. Immer dann, wenn man mit einem Wort $w \in \varphi(\alpha)$ von dem Startzustand aus einen Endzustand erreichen könnte, kann man entweder genau dies so machen (und der Automat akzeptiert), oder man kann alternativ mit dem letzten Zeichen von w wieder zum Startzustand zurück gehen. Von dort aus kann man dann ein weiteres Wort aus $\varphi(\alpha)$ verarbeiten, usw. Es ist klar, dass der so modifizierte Automat die Sprache $(\varphi(\alpha))^+$ akzeptiert.

Nun müssen wir nur noch sicherstellen, dass der Automat auch das leere Wort erkennt, sofern dies noch nicht der Fall ist. Dazu wird ein zusätzlicher Startzustand eingefügt, der gleichzeitig auch Endzustand ist und keine Übergänge zu anderen Zuständen besitzt. Anschließend erfolgt wieder die Umwandlung des gesamten NEAs in einen DEA.

Der Sternoperator wird auch in unserem Beispiel $((ab\,|\,b)b)^\star$ als letzter Schritt angewendet. Ausgehend von dem hergeleiteten DEA für den Teilausdruck $(ab\,|\,b)b$ erhalten wir:



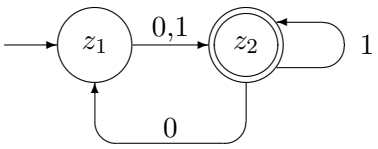
Die an sich noch fällige Umwandlung in einen DEA lassen wir der Einfachheit halber weg, da wir bei diesem Beispiel nichts weiter konstruieren müssen.

Allgemein kann man mit den gezeigten Techniken jeden noch so komplizierten regulären Ausdruck nach und nach in einen Automaten umwandeln. Somit ist der Beweis komplett. □

Die Umkehrung von Satz 2.3.4 gilt ebenfalls:

2.3.5 Satz Jede durch einen DEA akzeptierte Sprache kann regulär ausgedrückt werden.

Beweis: Der Beweis basiert auf einer trickreichen und interessanten Konstruktion. Sei M ein DEA und $L := L(M)$ die von ihm akzeptierte Sprache. O.B.d.A. sei M von der Form $M = (\{z_1, z_2, \ldots, z_n\}, \Sigma, \delta, z_1, E)$, d.h. der Startzustand trägt diesmal ausnahmsweise die Bezeichnug z_1. Ein solcher DEA könnte also z.B. so aussehen:



Wir werden den konstruktiven Beweis anhand dieses DEAs nachverfolgen.

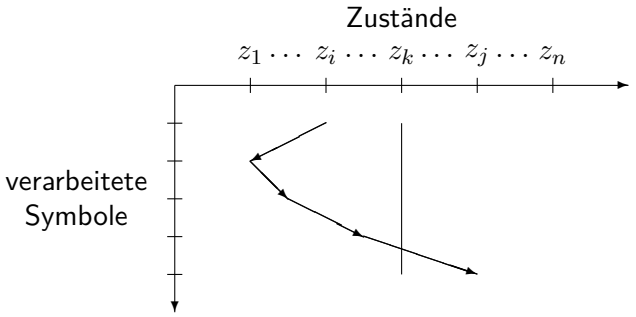
Die zentrale Idee besteht in der Konstruktion von regulären Ausdrücken für bestimmte Sprachen $R^k_{i,j}$, wobei für die Indizes $i, j \in \{1, 2, \ldots, n\}$ und $k \in \{0, 1, \ldots, n\}$ gilt. L wird sich später als Vereinigung gewisser $R^k_{i,j}$ ergeben.

Wir legen $R^k_{i,j}$ als die folgende Sprache fest:

$$\{x \in \Sigma^* \,|\, \hat{\delta}(z_i, x) = z_j \wedge \text{kein zwischenzeitlicher Zustand hat einen Index} > k\} \ .$$

Das Prinzip der $R^k_{i,j}$–Mengen wird durch die nachstehende Skizze verdeutlicht. Beginnend mit z_i wird bei jedem Pfeil nach unten ein Zeichen des Eingabewortes verarbeitet

und ein neuer Zustand erreicht. Alle Zustände mit einem Index größer als k sind dabei nicht zugelassen, d.h. die „Pfeilschlange“ darf sich nur links von der senkrechten Sperre bei z_k bewegen. Mit dem letzten Übergang wird schließlich z_j erreicht. Dabei darf j durchaus größer als k sein (so wie hier), da das Verbot für Zustände rechts von der Sperre nur für die eingenommenen Zwischenzustände gilt. Der erste Zustand z_i und der letzte Zustand z_j sind von dieser Beschränkung ausgenommen.



Wir zeigen jetzt nacheinander für $k = 0, 1, \ldots, n$, dass sich alle $R^k_{i,j}$–Mengen durch reguläre Ausdrücke $\alpha^k_{i,j}$ beschreiben lassen.

Sei also zunächst $k = 0$. Beachten Sie, dass jeder Zustand einen Index größer als 0 besitzt, d.h. es sind im Endeffekt überhaupt keine Zwischenzustände erlaubt. Für $i \neq j$ ist demnach

$$R^0_{i,j} = \{a \in \Sigma \,|\, \delta(z_i, a) = z_j\} \ ,$$

und für $i = j$ gilt

$$R^0_{i,i} = \{\varepsilon\} \cup \{a \in \Sigma \,|\, \delta(z_i, a) = z_i\} \ .$$

Alle $R^0_{i,j}$ sind also endlich und lassen sich daher durch reguläre Ausdrücke beschreiben. Für unseren einfachen DEA gilt z.B.

$$R^0_{1,1} = \{\varepsilon\} \ , \quad R^0_{1,2} = \{0, 1\} \ , \quad R^0_{2,1} = \{0\} \ , \quad R^0_{2,2} = \{\varepsilon, 1\} \ .$$

Die zugehörigen regulären Ausdrücke lauten

$$\alpha^0_{1,1} = \varepsilon \ , \quad \alpha^0_{1,2} = 0 \,|\, 1 \ , \quad \alpha^0_{2,1} = 0 \ , \quad \alpha^0_{2,2} = \varepsilon \,|\, 1 \ .$$

Als nächstes wollen wir die regulären Ausdrücke $\alpha^k_{i,j}$ für $k = 1$, $k = 2$, usw. herleiten. Wir werden dazu jeweils auf die regulären Ausdrücke der vorherigen Stufe zurückgreifen, die man bis dahin schon konstruiert hat. Wir müssen uns dazu klarmachen, wie sich die Wörter in den $R^k_{i,j}$–Mengen zusammensetzen.

Sei also $k \geq 0$ und $x \in R^{k+1}_{i,j}$, d.h. es gilt $\hat{\delta}(z_i, x) = z_j$, und zwischenzeitlich werden nur Zustände aus $\{z_1, \ldots, z_{k+1}\}$ besucht. Wir werfen jetzt konkret einen Blick darauf, wie oft der Zustand z_{k+1} eingenommen wird. Falls z_{k+1} überhaupt nicht benutzt wird, so ist

x offensichtlich auch schon in der Sprache $R^k_{i,j}$ enthalten. Falls der Zustand z_{k+1} dagegen einmal eingenommen wird, so lässt sich x in zwei Teile $x = uy$ zerlegen, so dass

$$\hat{\delta}(z_i, u) = z_{k+1} \quad \text{und} \quad \hat{\delta}(z_{k+1}, y) = z_j$$

gilt. Dann ist aber u in $R^k_{i,k+1}$ und y in $R^k_{k+1,j}$ enthalten. Es kann aber auch sein, dass z_{k+1} noch öfter eingenommen wird, z.B. gleich dreimal. Dann gibt es analog eine Zerlegung $x = uvwy$ mit

$$\hat{\delta}(z_i, u) = z_{k+1}\,, \;\; \hat{\delta}(z_{k+1}, v) = z_{k+1}\,, \;\; \hat{\delta}(z_{k+1}, w) = z_{k+1} \quad \text{und} \quad \hat{\delta}(z_{k+1}, y) = z_j \;\; .$$

Dementsprechend gilt dann $u \in R^k_{i,k+1}$, $v \in R^k_{k+1,k+1}$, $w \in R^k_{k+1,k+1}$, und $y \in R^k_{k+1,j}$. Allgemeiner kann man also sagen: falls der Zustand z_{k+1} genau n–mal (mit $n \in \mathbb{N}$) durchlaufen wird, so existiert eine Zerlegung $x = uv_1v_2 \ldots v_{n-1}y$ mit

$$u \in R^k_{i,k+1}\,, \;\; v_1, v_2, \ldots, v_{n-1} \in R^k_{k+1,k+1} \quad \text{und} \quad y \in R^k_{k+1,j} \;\; .$$

Oder anders ausgedrückt: es gibt eine Zerlegung $x = uvy$ mit

$$u \in R^k_{i,k+1}\,, \;\; v \in (R^k_{k+1,k+1})^* \quad \text{und} \quad y \in R^k_{k+1,j} \;\; .$$

Diese Überlegungen führen dann zu der Gleichung

$$R^{k+1}_{i,j} \;=\; R^k_{i,j} \cup (R^k_{i,k+1}(R^k_{k+1,k+1})^* R^k_{k+1,j}) \;\; ,$$

denn um von dem Zustand z_i aus den Zustand z_j zu erreichen, wird entweder der Zustand z_{k+1} nicht benutzt (dann reicht $R^k_{i,j}$ zur Beschreibung aus), oder aber z_{k+1} wird ein– oder mehrfach (in Schleifen) durchlaufen. Dies wird durch den anderen Teilausdruck $(R^k_{i,k+1}(R^k_{k+1,k+1})^* R^k_{k+1,j})$ beschrieben.

Nehmen Sie nun an, dass bereits reguläre Ausdrücke $\alpha^k_{i,j}$ für jede Sprache $R^k_{i,j}$ zur Verfügung stehen. Die obige Gleichung können wir dann dazu verwenden, um die Sprache $R^{k+1}_{i,j}$ ebenfalls regulär auszudrücken:

$$\alpha^{k+1}_{i,j} \;=\; (\alpha^k_{i,j} \,|\, \alpha^k_{i,k+1}(\alpha^k_{k+1,k+1})^\star \alpha^k_{k+1,j}) \;\; .$$

Also können wir reguläre Ausdrücke $\alpha^k_{i,j}$ für alle $R^k_{i,j}$–Mengen angeben, denn für $k = 0$ sind uns diese bereits bekannt, und für $k = 1, 2, \ldots, n$ können wir sie nacheinander konstruieren. Für unseren DEA ergeben sich auf der Stufe $k = 1$ z.B. die folgenden Ausdrücke:

$$\begin{aligned}
\alpha^1_{1,1} &= \alpha^0_{1,1} \,|\, \alpha^0_{1,1}(\alpha^0_{1,1})^\star \alpha^0_{1,1} = \varepsilon \,|\, \varepsilon\varepsilon^\star\varepsilon = \varepsilon \\
\alpha^1_{1,2} &= \alpha^0_{1,2} \,|\, \alpha^0_{1,1}(\alpha^0_{1,1})^\star \alpha^0_{1,2} = (0 \,|\, 1) \,|\, \varepsilon\varepsilon^\star(0 \,|\, 1) = (0 \,|\, 1) \\
\alpha^1_{2,1} &= \alpha^0_{2,1} \,|\, \alpha^0_{2,1}(\alpha^0_{1,1})^\star \alpha^0_{1,1} = 0 \,|\, 0\varepsilon^\star\varepsilon = 0 \\
\alpha^1_{2,2} &= \alpha^0_{2,2} \,|\, \alpha^0_{2,1}(\alpha^0_{1,1})^\star \alpha^0_{1,2} = (\varepsilon \,|\, 1) \,|\, 0\varepsilon^\star(0 \,|\, 1) = \varepsilon \,|\, 1 \,|\, 00 \,|\, 01
\end{aligned}$$

Für die nächste Stufe $k = 2$ ergibt sich analog:

$$\begin{aligned}
\alpha^2_{1,1} &= \alpha^1_{1,1} \,|\, \alpha^1_{1,2}(\alpha^1_{2,2})^\star \alpha^1_{2,1} = \varepsilon \,|\, (0 \,|\, 1)(\varepsilon \,|\, 1 \,|\, 00 \,|\, 01)^\star 0 = \varepsilon \,|\, (0 \,|\, 1)(1 \,|\, 00 \,|\, 01)^\star 0 \\
\alpha^2_{1,2} &= \alpha^1_{1,2} \,|\, \alpha^1_{1,2}(\alpha^1_{2,2})^\star \alpha^1_{2,2} = (0 \,|\, 1) \,|\, (0 \,|\, 1)(\varepsilon \,|\, 1 \,|\, 00 \,|\, 01)^+ = (0 \,|\, 1)(1 \,|\, 00 \,|\, 01)^\star \\
\alpha^2_{2,1} &= \alpha^1_{2,1} \,|\, \alpha^1_{2,2}(\alpha^1_{2,2})^\star \alpha^1_{2,1} = 0 \,|\, (\varepsilon \,|\, 1 \,|\, 00 \,|\, 01)^+ 0 = (1 \,|\, 00 \,|\, 01)^\star 0 \\
\alpha^2_{2,2} &= \alpha^1_{2,2} \,|\, \alpha^1_{2,2}(\alpha^1_{2,2})^\star \alpha^1_{2,2} = (\varepsilon \,|\, 1 \,|\, 00 \,|\, 01) \,|\, (\varepsilon \,|\, 1 \,|\, 00 \,|\, 01)(\varepsilon \,|\, 1 \,|\, 00 \,|\, 01)^+ \\
&= (1 \,|\, 00 \,|\, 01)^\star
\end{aligned}$$

Beachten Sie, dass für $k = n$ die obige Definition für $R^n_{i,j}$ von

$$\{x \in \Sigma^* \,|\, \hat{\delta}(z_i, x) = z_j \wedge \text{kein zwischenzeitlicher Zustand hat einen Index} > n\}$$

auf

$$R^n_{i,j} := \{x \in \Sigma^* \,|\, \hat{\delta}(z_i, x) = z_j\}$$

zusammenschmilzt, da es überhaupt keine Zustände mit einem höheren Index als n gibt. Insbesondere ist $R^n_{1,i}$ die Menge aller Wörter, die den Startzustand z_1 in den Zustand z_i überführen. Also gilt auch

$$L(M) = \bigcup_{z_i \in E} R^n_{1,i} \ .$$

Diese Gleichung kann man ebenfalls durch einen regulären Ausdruck beschreiben. Ist nämlich $\{i_1, i_2, \ldots, i_m\}$ die Menge aller Indizes der Endzustandsmenge E (d.h. wir haben $E = \{z_{i_1}, z_{i_2}, \ldots, z_{i_m}\}$), so können wir mit

$$\gamma := (\alpha^n_{1,i_1} | \alpha^n_{1,i_2} | \ldots | \alpha^n_{1,i_m})$$

einen regulären Ausdruck γ bilden, für den $\varphi(\gamma) = L(M)$ gilt.

In unserem Beispiel ist z_2 der einzige Endzustand, so dass $\alpha^2_{1,2} = (0 \,|\, 1)(1 \,|\, 00 \,|\, 01)^\star$ bereits dem gesuchten regulären Ausdruck entspricht. Ein kurzer Blick auf den Ausgangs–DEA genügt, um die Korrektheit dieses Ausdrucks nachzuprüfen.

Also ist jede von einem Automat erkannte Sprache durch einen regulären Ausdruck beschreibbar. □

Die beiden Sätze 2.3.5 und 2.3.4 sind insgesamt als Satz von Kleene bekannt:

2.3.6 Korollar (Kleene[1]) Eine Sprache L wird genau dann durch einen endlichen Automat akzeptiert, wenn sie durch einen regulären Ausdruck beschreibbar ist. □

[1] Stephen C. Kleene, *1909 Hartford, Connecticut, †1994, Madison, Wisconsin, US–amerikanischer Mathematiker und Logiker, 1937–1979 Professor an der University of Wisconsin–Madison.

Es sei zum Abschluss bemerkt, dass die gezeigten Techniken zum Umwandeln konkreter Automaten oder Ausdrücke für die Praxis meistens ungeeignet sind. Häufig kommt man durch „direktes“ Nachdenken schneller zum Ziel.

Zusammenfassend halten wir fest:

2.3.7 Korollar Sei L eine beliebige Sprache. Dann sind die folgenden Aussagen äquivalent:

- L ist regulär.
- L wird von einer regulären Grammatik erzeugt.
- L wird von einem DEA akzeptiert.
- L wird von einem NEA akzeptiert.
- L kann durch einen regulären Ausdruck beschrieben werden.

Beweis: Aus der zweiten Aussage folgt die vierte (Satz 2.2.12), aus der vierten folgt die dritte (Satz 2.2.10), und aus der dritten folgt wieder die zweite (Satz 2.2.11). Zwischen der zweiten, dritten und vierten Aussage existiert also ein „Kreisverkehr“ von Implikationen, so dass alle gleichwertig sein müssen. Ferner sind die ersten beiden Aussagen äquivalent, denn dies ist gerade die Definition regulärer Sprachen. Schließlich belegt noch Satz 2.3.6 die Äquivalenz zwischen der dritten und fünften Aussage, so dass im Endeffekt alle Aussagen miteinander äquivalent sind.

Strenggenommen sind einige dieser Äquivalenzen nur korrekt, wenn zusätzlich $\varepsilon \notin L$ vorausgesetzt wird (sonst gelten tlw. die zitierten Sätze nicht). Von nun an werden wir diesen Sonderfall aber vernachlässigen und z.B. eine Sprache L auch dann als regulär bezeichnen, wenn sie zwar das leere Wort enthält, „ansonsten“ aber regulär ist. □

Somit können wir z.B. folgern:

2.3.8 Korollar Eine endliche Sprache ist immer regulär.

Beweis: Für eine endliche Sprache $L = \{w_1, w_2, \ldots, w_k\}$ ist $w_1 \,|\, w_2 \,|\, \ldots \,|\, w_k$ ein passender regulärer Ausdruck, und nach dem letzten Satz muss L somit regulär sein. □

2.4. Das Pumping–Lemma für reguläre Sprachen

Natürlich sind nicht alle Sprachen regulär, denn die Chomsky–Hierarchie besagt ja gerade, dass es z.B. Sprachen gibt, die zwar kontextfrei, aber nicht regulär sind. Kann man einer Sprache ihre Nichtregularität ansehen? Eine Möglichkeit dazu ergibt sich durch den folgenden interessanten Satz, welcher eine bestimmte Eigenschaft aller regulären Sprachen beschreibt. Wenn eine Sprache diese Eigenschaft nicht erfüllt, kann sie demnach nicht regulär sein.

2.4.1 Satz Ist L regulär, so existiert ein $n \in \mathbb{N}$ mit den folgenden Eigenschaften:

Ist $x \in L$ mit $|x| \geq n$, so ist x zerlegbar in $x = uvw$ mit

1) $|v| \geq 1$
2) $|uv| \leq n$
3) $\forall i \geq 0\colon\ uv^i w \in L$

Wenn also L regulär ist, so gilt formal die folgende Aussage:

$$\exists n \in \mathbb{N}\colon\ \forall x \in L\colon\ (|x| \geq n) \Longrightarrow (\exists u, v, w\colon\ x = uvw\ \wedge\ 1.\ \wedge\ 2.\ \wedge\ 3.)$$

Bevor wir diesen Satz beweisen, sehen wir uns zwei Beispiele an:

2.4.2 Beispiel Wir betrachten nochmals die Sprache aller binär kodierten Zahlen:

$$L := \{bin(m) \mid m \in \mathbb{N}_0\} = \{0, 10, 11, 100, 101, \dots\}\ .$$

Diese Sprache war regulär (vgl. Beispiel 2.1.9 auf Seite 36). Also ist die Eigenschaft von Satz 2.4.1 erfüllt, und zwar für die Zahl $n = 2$, was man wie folgt einsieht. Jede binär kodierte Zahl $x \in L$ mit $|x| \geq 2$ beginnt zwingend mit einer führenden Eins, da eine führende Null nur bei der Kodierung der Zahl Null selbst vorkommt, und diese Kodierung hat nur die Länge eins. Also ist jedes Wort $x \in L$ der Länge k mit $k \geq 2$ von der Form $1x_2x_3 \dots x_k$. Nun existiert tatsächlich eine Zerlegung $x = uvw$, die alle drei Bedingungen erfüllt, nämlich $u = 1$, $v = x_2$, und $w = x_3 \dots x_k$. (Im Fall $k = 2$ ist w also das leere Wort.) Denn es gilt wie gewünscht $|v| = 1 \geq 1$, $|uv| = |1x_2| = 2 \leq 2 = n$, und jedes Wort der Form $uv^i w = 1(x_2)^i x_3 \dots x_k$ beginnt mit einer führenden Eins, d.h. es handelt sich nach wie vor um eine korrekt kodierte binäre Zahl. Also ist für jedes $i \in \mathbb{N}_0$ das Wort $uv^i w$ in L enthalten.

Im nächsten Beispiel kann man nicht mehr so einfach sehen, dass sich die Bedingungen erfüllen lassen.

2.4.3 Beispiel Die Sprache L aller binär kodierten Zahlen, die durch drei teilbar sind, ist ebenfalls eine reguläre Menge:

$$L := \{bin(3m) \mid m \in \mathbb{N}_0\} = \{0, 11, 110, 1001, 1100, \dots\}$$

Wir zeigen, dass auch hier Satz 2.4.1 gilt, diesmal für die Zahl $n = 3$. Jede binäre Zahl mit mindestens drei Ziffern beginnt auch hier immer mit einer Eins, aber viel interessanter sind die beiden dahinter liegenden Ziffern. Falls die binäre Zahl $x = x_1x_2 \dots x_k \in L$ mit $100\dots$ beginnt, so erfüllt die Zerlegung $x = uvw$ mit $u = x_1 = 1$, $v = x_2x_3 = 00$, und $w = x_4 \dots x_k$ die drei Bedingungen (im Fall $x = 1001 = bin(9)$ würde man also $u = 1$, $v = 00$ und $w = 1$ erhalten). Dann gilt zunächst natürlich wieder $|v| = |00| = 2 \geq 1$ sowie $|uv| = |100| = 3 \leq n = 3$, aber es ist nicht unmittelbar offensichtlich, warum jede binäre Kodierung der Form $1(00)^i x_4 \dots x_k$ eine durch drei teilbare Zahl darstellt (im obigen

Fall $x = 1001 = bin(9)$ gilt beispielsweise $100001 = bin(33)$, $10000001 = bin(129)$, usw.) Wir sehen uns dazu die Zahlenwerte der einzelnen Bits in der ursprünglichen Kodierung $x = x_1 \dots x_k = 100x_4 \dots x_k$ genauer an.

Das in u enthaltene führende Bit kodiert den Wert k_u irgendeiner Zweierpotenz (im Fall $x = 1001$ gilt z.B. $k_u = 8$). Die Bits in v sind dagegen beide gleich 0 und tragen daher nichts zum Gesamtwert bei. Also wird der restliche Wert ausschließlich durch die Bits in w kodiert. Diesen Wert bezeichnen wir mit k_w, wobei wir evtl. vorhandene führende Nullen von w ignorieren (im Fall $x = 1001 = bin(9)$ war $w = 1$, also $k_w = 1$, und zur Kontrolle können wir die Summe $k_u + k_w = 8 + 1 = 9$ berechnen). Es kann auch sein, dass $w = \varepsilon$ gilt, in diesem Fall setzen wir $k_w := 0$.

Wir betrachten nun z.B. das Wort $uv^2w = 10000w$, d.h. wir fügen zwei zusätzliche Nullen hinter die führende Eins ein. Dann rutscht das führende Bit um zwei Stellen nach links und kodiert somit einen viermal höheren Wert, d.h. das veränderte Wort kodiert nicht mehr die Zahl $k_u + k_w$, sondern $4 \cdot k_u + k_w$. Wegen

$$4 \cdot k_u + k_w = 3 \cdot k_u + (k_u + k_w)$$

setzt sich der neue Zahlenwert also aus der Summe einer durch drei teilbaren Zahl und der Ausgangszahl zusammen. Da die ursprüngliche Zahl aber ebenfalls durch drei teilbar war (denn es galt ja $x \in L$), ist somit auch die neue Zahl durch drei teilbar. Die neu erzeugte Kodierung ist folglich ebenfalls in L enthalten. Analog kann man zeigen, dass alle Kodierungen der Form $1(00)^i x_4 \dots x_k$ durch drei teilbar und in L enthalten sind.

Was aber, wenn x nicht mit $100\dots$ beginnt? Auch dann ist immer eine passende Zerlegung möglich:

- Falls $x = 101\dots$ gilt, so erfüllt die Zerlegung $x = uvw$ mit $u = 10$, $v = 1$, und $w = x_4 \dots x_k$ die drei Bedingungen.
- Falls $x = 110\dots$ gilt, so kann man die Zerlegung $x = uvw$ mit $u = 11$, $v = 0$, und $w = x_4 \dots x_k$ wählen.
- Im letzten Fall $x = 111\dots$ leistet die Zerlegung $u = 1$, $v = 11$, und $w = x_4 \dots x_k$ das Gewünschte.

Mit ähnlichen Argumenten wie oben kann man auch in diesen Fällen die Korrektheit der angegebenen Zerlegungen nachweisen. Aus Zeitgründen werden wir an dieser Stelle aber darauf verzichten.

Nun zum Beweis des Satzes 2.4.1:

Beweis: Da L regulär ist, existiert ein DEA $M = (Z, \Sigma, \delta, z_0, E)$ mit $L = L(M)$. Wir wählen n als die Anzahl der Zustände von M, also $n := |Z|$. Sei jetzt $x \in L$ mit $|x| \geq n$. Wir werfen einen Blick auf alle Zustände, die M während der Verarbeitung von x einnimmt. M beginnt anfangs in dem Startzustand z_0 und durchläuft für jedes Symbol in x einen weiteren Zustand. Wegen $|x| \geq n$ werden also mindestens $n + 1$ Zustände eingenommen. Diese Zustände können nicht alle verschieden sein, da es überhaupt nur n Zustände in Z gibt (sog. *Schubfachprinzip*). M muss daher beim Verarbeiten der ersten

n Zeichen von x zumindest einen Zustand z doppelt betreten und dabei zustandsmäßig eine Schleife durchlaufen. Sei nun die Zerlegung $x = uvw$ so gewählt, dass der Zustand z zum ersten Mal nach der Verarbeitung des Präfixes u und zum zweiten Mal nach der Verarbeitung des Infixes v angenommen wird. Dies ist unter Berücksichtigung der beiden ersten obigen Bedingungen $|v| \geq 1$ und $|uv| \leq n$ möglich.

Ausgehend vom Zustand z könnte der Automat noch mehrfach das Infix v verarbeiten und dabei die Zustandsschleife durchlaufen — der resultierende Zustand wäre immer wieder z, so als wäre nichts geschehen.

Wegen der Gleichheit der Zustände nach dem Lesen von u und Lesen von uv sind auch die Zustände nach der Verarbeitung von uw und uvw gleich. Da es sich dabei wegen $uvw = x \in L$ um einen Endzustand handelt, gilt auch $uw = uv^0w \in L$. Dasselbe trifft dann auch für $uvvw, uvvvw, \ldots$ und allgemein für alle uv^iw mit $i \in \mathbb{N}_0$ zu. Damit ist auch die dritte Bedingung erfüllt. □

Satz 2.4.1 ist in der Fachwelt als *Pumping–Lemma* bekannt. Der Begriff „Pumping“ bezieht sich auf die Eigenschaft des mittleren Teilwortes v, welches beliebig oft „gepumpt“ (d.h. wiederholt) werden kann, ohne an der Akzeptanz des Automaten etwas zu ändern.

Die eigentliche Anwendung des Pumping–Lemmas liegt in dem Führen von Widerspruchsbeweisen, um die Nichtregularität von diversen Sprachen zu zeigen. Wir geben vier Beispiele an:

2.4.4 Beispiel $L := \{a^m b^m \mid m \geq 1\}$ ist nicht regulär. Der Beweis erfolgt durch Widerspruch. Angenommen, L sei regulär. Dann existiert ein $n \in \mathbb{N}$, so dass sich alle $x \in L$ mit $|x| \geq n$ wie gezeigt in $x = uvw$ zerlegen lassen. Man betrachte nun die Zerlegung uvw von dem Wort $a^n b^n \in L$.

Wegen $|uv| \leq n$ und $|v| \geq 1$ besteht v nur aus einer nichtleeren Folge von a's. Laut Bedingung 3 im Pumping–Lemma wäre dann (für $i = 0$) auch $uw = a^{n-|v|}b^n$ in L enthalten, was natürlich nicht der Fall ist. Also ist L nicht regulär.

2.4.5 Beispiel $L := \{0^m \mid m \text{ ist eine Quadratzahl}\}$ ist nicht regulär. Falls doch, so existiert ein $n \in \mathbb{N}$, so dass sich jedes 0^m, $m = k^2 \geq n$, zerlegen lässt in $0^m = uvw$ mit den Eigenschaften 1 – 3.

Wegen $n^2 \geq n$ kann insbesondere das Wort $x = 0^{n^2} \in L$ in $x = uvw$ zerlegt werden. Aus den Bedingungen 1 und 2 des Pumping–Lemmas ergibt sich $1 \leq |v| \leq |uv| \leq n$. Mit der Bedingung 3 für den Fall $i = 2$ folgt ferner $uv^2w = uvvw \in L$. Wegen

$$n^2 = |x| = |uvw| < |uvvw| = |uvw| + |v| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$$

gilt aber $n^2 < |uvvw| < (n+1)^2$, d.h. die Länge von $uvvw$ ist keine Quadratzahl. Also ist L nicht regulär.

2.4.6 Beispiel $L := \{0^p \mid p \text{ ist eine Primzahl}\}$ ist nicht regulär. Denn andernfalls wählt man eine Primzahl p mit $p \geq n+2$, wobei n wieder die Zahl aus dem Pumping–Lemma ist. Die Zahl p existiert, da es unendlich viele Primzahlen gibt (siehe Satz 1.4.2). Sei nun eine Zerlegung $0^p = uvw$ gemäß den Regeln 1 – 3 gegeben. Nach Regel 3 müsste für jede Zahl $i \in \mathbb{N}_0$ auch das Wort uv^iw in L liegen, d.h. alle Zahlen der Form $|uw|+i\cdot|v|$ müssten prim sein. Für $i = |uw|$ gilt jedoch $|uw| + |uw| \cdot |v| = |uw| \cdot (1 + |v|)$. Beide Faktoren sind größer als Eins, denn wir haben $|uw| \geq |w| = |uvw| - |uv| = p - |uv| \geq (n+2) - n = 2$ sowie $1 + |v| \geq 2$. Also ist für den Fall $i = |uw|$ die Länge von uv^iw doch keine Primzahl, und das Pumping–Lemma gilt demnach nicht. Also ist auch hier L nicht regulär.

2.4.7 Beispiel $L := \{bin(p) \mid p \text{ ist eine Primzahl}\} = \{10, 11, 101, 111, 1011, \dots\}$ ist keine reguläre Sprache. Um dies zu beweisen, benötigen wir fünf einfache Resultate aus der Zahlentheorie:

2.4.8 Lemma Für je drei ganze Zahlen i, j und p mit $p > 0$ gilt stets

$$(i \cdot j) \bmod p = ((i \bmod p) \cdot (j \bmod p)) \bmod p \ ,$$

wobei „mod“ den *Modulo*–Operator bezeichnet, also den Rest bei einer ganzzahligen Division durch p. Beispielsweise gilt

$$10 \cdot 11 \bmod 7 = ((10 \bmod 7) \cdot (11 \bmod 7)) \bmod 7 \ ,$$

denn es gilt

$$10 \cdot 11 \bmod 7 = 110 \bmod 7 = 5$$

und ebenso

$$((10 \bmod 7) \cdot (11 \bmod 7)) \bmod 7 = (3 \cdot 4) \bmod 7 = 12 \bmod 7 = 5 \ .$$

Beweis: Es seien $r_i := i \bmod p$ und $r_j := j \bmod p$ die Reste bei der ganzzahligen Division von i und j durch p. Dann gilt $i = k_i \cdot p + r_i$ und analog $j = k_j \cdot p + r_j$ für passende Zahlen $k_i, k_j \in \mathbb{N}_0$ (k_i und k_j sind die zugehörigen Quotienten der ganzzahligen Divisionen). Dann gilt

$$\begin{aligned} & i \cdot j \\ = \ & (k_i \cdot p + r_i) \cdot (k_j \cdot p + r_j) \\ = \ & k_i \cdot p \cdot k_j \cdot p + r_i \cdot k_j \cdot p + k_i \cdot p \cdot r_j + r_i \cdot r_j \\ = \ & (k_i \cdot p \cdot k_j + r_i \cdot k_j + k_i \cdot r_j) \cdot p + r_i \cdot r_j \ . \end{aligned}$$

Der erste Summand ist ein Produkt von p mit einer anderen Zahl und deshalb klar durch p teilbar. Für einen evtl. Rest bei der Division durch p ist also nur der andere Summand $r_i \cdot r_j$ von Bedeutung. Also folgt

$$(i \cdot j) \bmod p = (r_i \cdot r_j) \bmod p = ((i \bmod p) \cdot (j \bmod p)) \bmod p$$

und damit die Behauptung. □

2.4.9 Lemma Für jede Primzahl $p > 2$ ist $2^{p-1} - 1$ immer durch p teilbar. (Beispielsweise ist 5 ein Teiler von $2^4 - 1 = 15$, ferner ist 7 ein Teiler von $2^6 - 1 = 63$, usw.)

Beweis: Sei eine Primzahl $p > 2$ fest vorgegeben. Wir betrachten die Zahlen

$$2 \cdot 0 \bmod p\,, \quad 2 \cdot 1 \bmod p\,, \quad 2 \cdot 2 \bmod p\,, \quad \ldots\,, \quad 2 \cdot (p-1) \bmod p\ .$$

Im Fall $p = 5$ würden wir z.B. die Ergebnisse

$$2 \cdot 0 \bmod 5 = 0\,, \quad 2 \cdot 1 \bmod 5 = 2\,, \quad 2 \cdot 2 \bmod 5 = 4\,,$$
$$2 \cdot 3 \bmod 5 = 1 \quad \text{und} \quad 2 \cdot 4 \bmod 5 = 3$$

erhalten.

Kein Ergebnis in dieser Zahlenreihe kommt doppelt vor — nicht in diesem Beispiel und auch nicht für andere Primzahlen p. Wären nämlich für zwei verschiedene Zahlen i und j die zugehörigen Ergebnisse $2 \cdot i \bmod p$ und $2 \cdot j \bmod p$ gleich, so würden $2 \cdot i$ und $2 \cdot j$ beim Teilen durch p den gleichen Rest hinterlassen, d.h. die Differenz $2 \cdot i - 2 \cdot j = 2(i - j)$ wäre durch p teilbar. Da p eine Primzahl ist, müsste p dann sogar einen der beiden Faktoren teilen, was aber nicht sein kann: zum einen gilt nach Voraussetzung $p > 2$, zum anderen gilt auch $i \neq j$ und $0 \leq i, j \leq p - 1$, also $0 < |i - j| < p$.

Da wir in der obigen Zahlenreihe p Ergebnisse berechnet haben und es überhaupt nur p verschiedene Ergebnisse für den Rest bei einer ganzzahligen Division durch p gibt (nämlich die Zahlen $0, 1, 2, \ldots, p - 1$), folgt sofort, dass jede Zahl von 0 bis $p - 1$ genau einmal in der Zahlenreihe auftreten muss — in irgendeiner uns unbekannten Reihenfolge. Nur die Position der Zahl 0 ist immer fest vorgegeben. Sie steht wegen $2 \cdot 0 \bmod p = 0$ immer an der ersten Stelle. Wenn wir diese aus der obigen Zahlenreihe entfernen, so erhalten wir als erstes Zwischenergebnis, dass die Zahlenreihe

$$2 \cdot 1 \bmod p\,, \quad 2 \cdot 2 \bmod p\,, \quad 2 \cdot 3 \bmod p\,, \quad \ldots \quad 2 \cdot (p-1) \bmod p$$

alle Zahlen von 1 bis $p - 1$ genau einmal enthält.

Nun bilden wir das Produkt von all diesen Zahlen:

$$(2 \cdot 1 \bmod p) \cdot (2 \cdot 2 \bmod p) \cdot (2 \cdot 3 \bmod p) \cdot \ldots \cdot (2 \cdot (p-1) \bmod p)\ .$$

Nach unserem bisherigen Kenntnisstand erhalten wir dann als Ergebnis die *Fakultät* $(p - 1)!$, die bereits auf Seite 17 definiert wurde:

$$(p-1)! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1)\ .$$

Also muss die Gleichung

$$(2 \cdot 1 \bmod p) \cdot (2 \cdot 2 \bmod p) \cdot \ldots \cdot (2 \cdot (p-1) \bmod p) = (p-1)!$$

gelten. Nun teilen wir beide Seiten ganzzahlig durch p. Dann verbleibt auf beiden Seiten der gleiche Rest:

$$\Big((2 \cdot 1 \bmod p) \cdot (2 \cdot 2 \bmod p) \cdot \ldots \cdot (2 \cdot (p-1) \bmod p)\Big) \bmod p = (p-1)! \bmod p\ .$$

Die linke Seite können wir dabei wie folgt vereinfachen (bei der ersten Umformung benutzen wir dabei Lemma 2.4.8):

$$\begin{aligned}
& \Big((2 \cdot 1 \bmod p) \cdot (2 \cdot 2 \bmod p) \cdot \ldots \cdot (2 \cdot (p-1) \bmod p)\Big) \bmod p \\
= \; & \Big((2 \cdot 1) \cdot (2 \cdot 2) \cdot \ldots \cdot (2 \cdot (p-1))\Big) \bmod p \\
= \; & \Big(2 \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot 3 \cdot 2 \cdot 4 \cdot \ldots \cdot 2 \cdot (p-1)\Big) \bmod p \\
= \; & \Big(\underbrace{2 \cdot 2 \cdot 2 \cdots 2}_{(p-1)\text{–mal}} \cdot \; 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1)\Big) \bmod p \\
= \; & \Big(2^{p-1} \cdot (p-1)!\Big) \bmod p \; .
\end{aligned}$$

Also gilt

$$\Big(2^{p-1} \cdot (p-1)!\Big) \bmod p = (p-1)! \bmod p \; ,$$

d.h. die Differenz

$$2^{p-1} \cdot (p-1)! - (p-1)! = (2^{p-1} - 1) \cdot (p-1)! = (2^{p-1} - 1) \cdot 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1)$$

muss glatt durch p teilbar sein. Genauer gesagt muss p dann sogar einen der beteiligten Faktoren teilen, da p eine Primzahl ist. Aber alle Faktoren $1, 2, 3, \ldots, p-1$ sind kleiner als p und kommen daher dafür nicht in Frage. Somit bleibt nur der Faktor $2^{p-1} - 1$ übrig. Also teilt p diese Zahl, und die Behauptung ist bewiesen. □

Das Lemma 2.4.9 kann nicht nur für die Basis 2, sondern völlig analog auch für jede andere Zahl $a < p$ bewiesen werden (beispielsweise ist 5 nicht nur, wie oben gesehen, ein Teiler von $2^4 - 1 = 15$, sondern auch von z.B. $3^4 - 1 = 80$ oder $4^4 - 1 = 255$). Dies ergibt dann den sog. *kleinen Satz von Fermat*[1]. Für unsere Zwecke reicht der Fall $a = 2$ aber völlig aus.

2.4.10 Lemma Seien $n, m, p \in \mathbb{N}$ drei natürliche Zahlen. Sind dann sowohl $n - 1$ als auch $m - 1$ durch p teilbar, so ist auch $nm - 1$ durch p teilbar.

Beweis: Ist $m - 1$ durch p teilbar, so gilt dies auch für die Zahl $n(m-1)$. Ist zudem $n - 1$ durch p teilbar, muss folglich auch die Summe von $n(m-1)$ und $n - 1$ durch p teilbar sein, welche vereinfacht genau

$$n(m-1) + (n-1) = nm - n + n - 1 = nm - 1$$

beträgt. □

[1] Pierre de Fermat, *Anfang des 17. Jahrhunderts, Beaumont–de–Lomagne, †12.01.1665, Castres, französischer Mathematiker und Jurist. Nach ihm sind zahlreiche Beiträge aus der Zahlentheorie benannt.

2.4.11 Lemma Sei $k \in \mathbb{N}$ und p eine Primzahl mit $p > 2$. Dann ist $2^{k(p-1)} - 1$ stets durch p teilbar.

Beweis: Gemäß Lemma 2.4.9 ist $2^{p-1} - 1$ immer durch p teilbar, was die Aussage für $k = 1$ beweist. Außerdem können wir zwei Zahlen m und n als die gleiche Zahl 2^{p-1} festlegen und dann mittels Lemma 2.4.10 folgern, dass auch

$$mn - 1 = 2^{p-1} \cdot 2^{p-1} - 1 = 2^{(p-1)+(p-1)} - 1 = 2^{2(p-1)} - 1$$

durch p teilbar sein muss, was die Aussage für $k = 2$ beweist. Wir benutzen dieses Ergebnis sofort wieder, indem wir n durch $2^{2(p-1)}$ ersetzen und nochmals Lemma 2.4.10 anwenden. Dann ist nämlich auch

$$mn - 1 = 2^{p-1} \cdot 2^{2(p-1)} - 1 = 2^{(p-1)+2(p-1)} - 1 = 2^{3(p-1)} - 1$$

durch p teilbar, und dies ist die Aussage für $k = 3$. Genauso fahren wir für alle weiteren Zahlen $k = 4, 5, 6, \ldots$ fort. Ist nämlich bereits $2^{k(p-1)} - 1$ als durch p teilbar bekannt, so folgt mit der Wahl von $m = 2^{p-1}$ und $n = 2^{k(p-1)}$ aus Lemma 2.4.10 die Aussage für die nächste Zahl $k + 1$, da dann auch

$$mn - 1 = 2^{p-1} \cdot 2^{k(p-1)} - 1 = 2^{(p-1)+k(p-1)} - 1 = 2^{(k+1)(p-1)} - 1$$

durch p teilbar ist. Wir haben die Aussage also *induktiv* bewiesen (eine Einführung in Induktionsbeweise ist im Anhang A zu finden). □

2.4.12 Lemma Sei $k \in \mathbb{N}$ und p eine Primzahl mit $p > 2^k$. Dann ist die Summe

$$q := 2^k + 2^{2k} + 2^{3k} + \cdots + 2^{(p-1)k}$$

durch p teilbar.

Beweis: Wir zeigen zunächst, dass das Produkt $q(2^k - 1)$ durch p teilbar ist. Es gilt nämlich

$$\begin{aligned}
& q(2^k - 1) \\
= \; & (2^k + 2^{2k} + 2^{3k} + \cdots + 2^{(p-1)k})(2^k - 1) \\
= \; & (2^k + 2^{2k} + 2^{3k} + \cdots + 2^{(p-1)k}) \cdot 2^k - (2^k + 2^{2k} + 2^{3k} + \cdots + 2^{(p-1)k}) \\
= \; & (2^{k+k} + 2^{2k+k} + 2^{3k+k} + \cdots + 2^{(p-1)k+k}) - (2^k + 2^{2k} + 2^{3k} + \cdots + 2^{(p-1)k}) \\
= \; & (2^{2k} + 2^{3k} + 2^{4k} + \cdots + 2^{pk}) - (2^k + 2^{2k} + 2^{3k} + \cdots + 2^{(p-1)k}) \\
= \; & (2^{2k} + 2^{3k} + 2^{4k} + \cdots + 2^{(p-1)k}) + 2^{pk} - 2^k - (2^{2k} + 2^{3k} + 2^{4k} + \cdots + 2^{(p-1)k}) \\
= \; & 2^{pk} - 2^k \\
= \; & 2^{(p-1)k+k} - 2^k \\
= \; & 2^{(p-1)k} \cdot 2^k - 2^k \\
= \; & (2^{(p-1)k} - 1) \cdot 2^k \;\; ,
\end{aligned}$$

und nach Lemma 2.4.11 ist p ein Teiler von $2^{(p-1)k} - 1$, also auch von $(2^{(p-1)k} - 1) \cdot 2^k$ und somit von $q(2^k - 1)$.

Aber dann muss p als Primzahl sogar mindestens einen der beiden Faktoren teilen, also entweder q oder $2^k - 1$ oder beide. Gemäß der Voraussetzung gilt aber $2^k - 1 < 2^k < p$, so dass p kein Teiler von $2^k - 1$ sein kann. Folglich muss p die Zahl q teilen. □

Nun haben wir alle Hilfsmittel zusammen, um das Beispiel 2.4.7 abschließen zu können:

Beweis: Angenommen, die Sprache $L = \{bin(p) \,|\, p \text{ ist eine Primzahl}\}$ wäre regulär. Sei n wieder die Pumping–Lemma–Zahl. Sei weiter p eine Primzahl mit $p > 2^n$ (eine solche Primzahl existiert, da es nach Satz 1.4.2 unendlich viele Primzahlen gibt). Wir betrachten nun die Binärdarstellung von p. Diese muss wegen $p > 2^n$ mindestens $n + 1$ Bits lang sein (für $n = 4$ müsste beispielsweise $p > 16$ gelten, und schon die Darstellung $bin(16) = 10000$ umfasst $5 = n + 1$ Bits). Also lässt sich die Binärdarstellung $bin(p)$ zerlegen in $bin(p) = uvw$ mit $|v| \geq 1$ und $|uv| \leq n$, so dass für alle Zahlen $i \in \mathbb{N}_0$ das Wort uv^iw in L enthalten und damit die Binärdarstellung einer Primzahl sein muss. Ähnlich wie in Beispiel 2.4.3 seien k_u, k_v und k_w die binär kodierten Werte von u, v und w, wobei wir wieder führende Nullen von v und w ignorieren. Ein evtl. leeres Wort u interpretieren wir als den Zahlenwert 0.

Mit diesen Festlegungen beträgt der Wert des Ausgangswortes $bin(p) = uvw$ also

$$p = k_u \cdot 2^{|vw|} + k_v \cdot 2^{|w|} + k_w \ .$$

Für alle Zahlen $i \in \mathbb{N}_0$ ist nun das Wort uv^iw in L enthalten. Speziell für die Wahl $i = p$ ist demnach uv^pw ein Wort aus L, also müsste dessen binär kodierter Wert

$$k_u \cdot 2^{|v^pw|} + k_v \cdot (1 + 2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} + k_w$$

eine Primzahl sein. Diese Primzahl bezeichen wir mit q. Natürlich gilt $q > p$, da die durch das „zusätzliche Pumpen“ entstandene Binärdarstellung von q länger als diejenige von p ist. Wir berechnen nun die Differenz von q und p:

$$\begin{aligned}
& q - p \\
= \ & k_u \cdot 2^{|v^pw|} + k_v \cdot (1 + 2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} + k_w \\
& \quad -(k_u \cdot 2^{|vw|} + k_v \cdot 2^{|w|} + k_w) \\
= \ & k_u \cdot 2^{|v^pw|} + k_v \cdot (1 + 2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} - k_u \cdot 2^{|vw|} - k_v \cdot 2^{|w|} \\
= \ & k_u \cdot (2^{|v^pw|} - 2^{|vw|}) + k_v \cdot (1 + 2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} - k_v \cdot 1 \cdot 2^{|w|} \\
= \ & k_u \cdot (2^{p|v|+|w|} - 2^{|vw|}) + k_v \cdot (2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} \\
= \ & k_u \cdot (2^{(p-1)|v|+|v|+|w|} - 2^{|vw|}) + k_v \cdot (2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} \\
= \ & k_u \cdot (2^{(p-1)|v|+|vw|} - 2^{|vw|}) + k_v \cdot (2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} \\
= \ & k_u \cdot (2^{(p-1)|v|} \cdot 2^{|vw|} - 2^{|vw|}) + k_v \cdot (2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} \\
= \ & k_u \cdot (2^{(p-1)|v|} - 1) \cdot 2^{|vw|} + k_v \cdot (2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|} \ .
\end{aligned}$$

Nach den Bedingungen des Pumping–Lemmas gilt $|uv| \leq n$ und somit insbesondere $|v| \leq n$. Also gilt auch $2^{|v|} \leq 2^n < p$ gemäß unserer Wahl von p. Mit der Wahl $k := |v|$ können wir also Lemma 2.4.12 anwenden und damit folgern, dass

$$2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}$$

durch p teilbar ist. Dies gilt demnach auch für

$$k_v \cdot (2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) \cdot 2^{|w|}$$

und damit für den zweiten Summanden der berechneten Differenz von q und p.

Auch der erste Summand ist durch p teilbar. Es gilt nämlich $p > 2^n \geq 2$, so dass nach Lemma 2.4.11 die Zahl $2^{k(p-1)} - 1$ für alle $k \in \mathbb{N}$ durch p teilbar ist, inbesondere für unsere Wahl $k := |v| \geq 1$. Also ist p immer ein Teiler von $2^{(p-1)|v|} - 1$ und damit von dem ersten Summanden

$$k_u \cdot (2^{(p-1)|v|} - 1) \cdot 2^{|vw|} \ .$$

Aber dann ist $q - p$ die Summe zweier durch p teilbaren Zahlen, also selbst durch p teilbar. Gleiches gilt folglich auch für $q = (q - p) + p$, d.h. q kann wegen $q > p$ keine Primzahl sein. Also gilt $bin(q) \notin L$. Die Bedingungen des Pumping–Lemmas sind somit nicht erfüllt, und L ist demnach keine reguläre Sprache. □

2.5. Minimierung endlicher Automaten

Als nächstes soll ein Verfahren hergeleitet werden, mit dem man endliche Automaten *minimieren* kann, d.h. zu einem gegebenen endlichen Automaten wird ein gleichwertiger Automat konstruiert, der möglichst wenig Zustände besitzt. Wir werden dabei in diesem Abschnitt davon ausgehen, dass alle verwendeten DEAs über *totale* Übergangsfunktionen verfügen, d.h. sie sind für alle Übergänge definiert. Es ist also nicht möglich, dass man von einem bestimmten Zustand aus mit einem Eingabesymbol aus Σ nicht mehr weiterkommt. (Sollte dies noch nicht der Fall sein, so kann man einen neuen „toten“ Zustand t mit $\delta(t, a) = t$ für alle $a \in \Sigma$ einführen und alle bislang fehlenden Übergänge in diesen Zustand wechseln lassen.) O.B.d.A. soll weiterhin jeder Zustand $z \in Z$ durch mindestens ein Wort $x \in \Sigma^*$ erreichbar sein, also $\hat{\delta}(z_0, x) = z$ gelten. Wenn dies nicht der Fall ist, kann man offenbar den Zustand einfach weglassen, ohne an der akzeptierten Sprache etwas zu ändern.

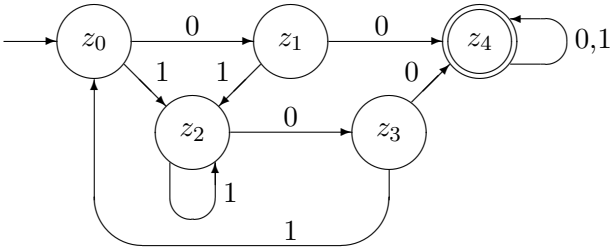
Der Lösungsansatz zur Minimierung eines DEAs $M = (Z, \Sigma, \delta, z_0, E)$ basiert auf einer bestimmten Relation $\equiv_M$, die über der Zustandsmenge Z gebildet wird:

$$z \equiv_M z' \ :\Longleftrightarrow\ \forall x \in \Sigma^* \colon (\hat{\delta}(z, x) \in E \ \Leftrightarrow\ \hat{\delta}(z', x) \in E) \ .$$

Die zentrale Eigenschaft von zwei miteinander in Relation stehenden Zuständen z und z' ist die folgende: wenn man während der Verarbeitung von einem Wort bei dem Zustand

z vorbeikommt, so könnte man ab dann auch bei dem Zustand z' mit der restlichen Verarbeitung fortfahren, ohne dass sich an der Akzeptanz des Gesamtwortes etwas ändern würde. Denn egal wie das restliche Wort x auch aussieht: die Eigenschaft $z \equiv_M z'$ besagt gerade, dass man von z aus mit dem Wort x genau dann einen Endzustand erreicht, wenn dies auch von z' aus der Fall ist.

2.5.1 Beispiel Sei M der folgende DEA über $\Sigma := \{0, 1\}$:



In diesem DEA sind z.B. die Zustände z_1 und z_3 äquivalent. Dies kann man wie folgt sehen:

- Wenn ein Wort x mit einer Null beginnt, so befindet sich der Automat nach der Verarbeitung der führenden Null beidesmal in z_4. Folglich ist die weitere Verarbeitung von x in beiden Fällen identisch, denn man verbleibt für den restlichen Verlauf in diesem gemeinsamen Endzustand.
- Wenn ein Wort x nicht mit einer Null beginnt, so erreicht der Automat den Endzustand z_4 genau dann, wenn zwei Nullen in x hintereinander verarbeitet werden. Ab dann verläuft die Verarbeitung wieder gleich.
- Wenn es sich bei x um das leere Wort handelt, so verbleibt der Automat in z_1 bzw. z_3, also beidesmal in einem Nicht–Endzustand.

Ähnliche Überlegungen zeigen, dass auch $z_0 \equiv_M z_2$ gilt. Es ist also im Endeffekt egal, ob sich der DEA momentan in z_0 oder z_2 (bzw. z_1 oder z_3) befindet, da ja von z_0 aus die restliche Verarbeitung genau dann in einem Endzustand mündet, wenn dies auch für z_2 zutrifft. Folglich sind z_0 und z_2 (und analog auch z_1 und z_3) gleichwertig, und es liegt nahe, solche Zustände jeweils zu einem neuen Zustand zu verschmelzen. Dadurch verringert sich dann entsprechend die Anzahl der Zustände im Automaten.

Bevor wir aber diese Idee verwirklichen, halten wir zunächst einige Eigenschaften der Relation $\equiv_M$ fest.

2.5.2 Lemma $\equiv_M$ ist eine Äquivalenzrelation.

Beweis: Im Beispiel 1.5.9 auf Seite 21 wurde gezeigt, dass für beliebige Mengen A und B sowie für ein beliebiges Prädikat $\varphi : A \times B \longrightarrow \{\mathrm{W}, \mathrm{F}\}$ die Relation R über A mit

$$zRz' \;:\Longleftrightarrow\; \forall x \in B\colon\; (\varphi(z, x) \;\Leftrightarrow\; \varphi(z', x))$$

eine Äquivalenzrelation ist. Also ergibt sich durch den Spezialfall $A := Z$, $B := \Sigma^*$ und $\varphi(z, x) := \hat{\delta}(z, x) \in E$ sofort die Behauptung. □

Z zerfällt demnach unter $\equiv_M$ in Äquivalenzklassen (siehe Definition 1.5.10 auf Seite 22). Ist z ein beliebiger Zustand, so bezeichnet $[z]_M$ die zugehörige Äquivalenzklasse mit $z \in [z]_M$, d.h. z ist ein Repräsentant von $[z]_M$.

2.5.3 Beispiel Im Beispiel 2.5.1 waren die Zustände z_0 und z_2 bzw. z_1 und z_3 miteinander äquivalent. Die zugehörigen Äquivalenzklassen sind also

$$\{z_0, z_2\} \ , \quad \{z_1, z_3\} \quad \text{und} \quad \{z_4\} \ .$$

2.5.4 Lemma Seien $z, z' \in Z$ mit $z \equiv_M z'$. Dann gilt $\hat{\delta}(z, w) \equiv_M \hat{\delta}(z', w)$ für alle $w \in \Sigma^*$.

Beweis: $z \equiv_M z'$ bedeutet, dass für alle Wörter $x \in \Sigma^*$ stets

$$\hat{\delta}(z, x) \in E \Leftrightarrow \hat{\delta}(z', x) \in E$$

gilt. Insbesondere gilt dies für Wörter der Form $x = wx'$, wobei x' frei wählbar ist, d.h. wir haben

$$\forall x' \in \Sigma^* \colon (\hat{\delta}(z, wx') \in E \Leftrightarrow \hat{\delta}(z', wx') \in E) \ .$$

Wegen $\hat{\delta}(z, wx') = \hat{\delta}(\hat{\delta}(z, w), x')$ und $\hat{\delta}(z', wx') = \hat{\delta}(\hat{\delta}(z', w), x')$ gilt deshalb

$$\forall x' \in \Sigma^* \colon (\hat{\delta}(\hat{\delta}(z, w), x') \in E \Leftrightarrow \hat{\delta}(\hat{\delta}(z', w), x') \in E)$$

und daher $\hat{\delta}(z, w) \equiv_M \hat{\delta}(z', w)$. □

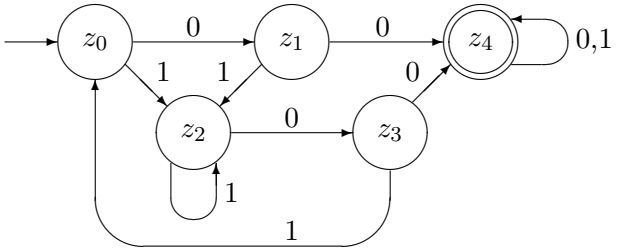
2.5.5 Lemma Jede Äquivalenzklasse von $\equiv_M$ ist eine Teilmenge von E oder von $Z \setminus E$.

Beweis: In der Definition von $\equiv_M$ folgt durch die Wahl von $x = \varepsilon$, dass für zwei Zustände z und z' mit $z \equiv_M z'$ stets $z \in E \Leftrightarrow z' \in E$ gilt. Innerhalb einer Äquivalenzklasse sind nun gerade alle Zustände paarweise miteinander äquivalent. Folglich liegt auch jede Äquivalenzklasse ganz in E oder ganz in $Z \setminus E$. □

Ausgehend von einem DEA M können wir nun mit Hilfe der Äquivalenzklassen von $\equiv_M$ einen sog. *Äquivalenzklassenautomat* $M_L := (Z_L, \Sigma, \delta_L, z_L, E_L)$ konstruieren. Die Komponenten sind dabei wie folgt gewählt:

- $Z_L := \{[z]_M \mid z \in Z\}$
- $\delta_L([z]_M, a) := [\delta(z, a)]_M$
- $z_L := [z_0]_M$
- $E_L := \{[z]_M \in Z_L \mid z \in E\}$

2.5.6 Beispiel Wir betrachten nochmals den DEA aus dem Beispiel 2.5.1:



Die Äquivalenzklassen bzgl. $\equiv_M$ waren die folgenden (wir werden später sehen, wie man diese berechnet):

$$\{z_0, z_2\} \ , \quad \{z_1, z_3\} \quad \text{und} \quad \{z_4\} \ .$$

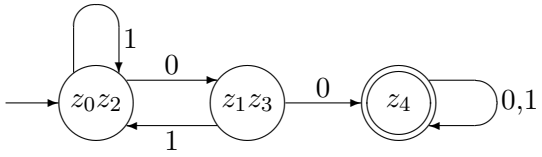
Diese Äquivalenzklassen stellen nun zugleich die Zustände von M_L dar. Die neue Übergangsfunktion δ_L ergibt sich daraus wie folgt. Um z.B. die Übergänge des Zustandes $\{z_0, z_2\}$ zu bilden, wählen wir einen (alten) Zustand aus der Menge aus, also beispielsweise z_2, und wenden dann auf z_2 die alte Übergangsfunktion an. Mit der 0 wurde beispielsweise bislang z_3 erreicht, und da dieser Zustand jetzt in $\{z_1, z_3\}$ liegt, erhalten wir so $\delta_L(\{z_0, z_2\}, 0) = \{z_1, z_3\}$. Genauso gut hätten wir anfangs z_0 statt z_2 wählen können, denn mit der alten Übergangsfunktion und der 0 erreichte man bislang z_1, d.h. der neu konstruierte Übergang wäre wegen $z_1 \in \{z_1, z_3\}$ der gleiche gewesen. Dies ist natürlich kein Zufall. Beachten Sie, dass allgemein ein Übergang wie $\delta_L([z]_M, a) := [\delta(z, a)]_M$ zwar von einer Äquivalenzklasse in eine andere Äquivalenzklasse wechselt, aber mit Hilfe eines Repräsentanten der Äquivalenzklasse $[z]_M$ (nämlich z selbst) definiert worden ist. Dies ist nur dann möglich, wenn die Definition von der konkreten Wahl des Repräsentanten unabhängig ist. Tatsächlich würde ein anderer Repräsentant $z' \in [z]_M$ zwar formal zu der Klasse $[\delta(z', a)]_M$ führen, diese ist jedoch mit $[\delta(z, a)]_M$ identisch, da aus $z \equiv_M z'$ stets $\delta(z, a) \equiv_M \delta(z', a)$ folgt (siehe Lemma 2.5.4). Die Zustände $\delta(z, a)$ und $\delta(z', a)$ gehören also immer zu der gleichen Klasse.

Gleiche Überlegungen kann man auch für die Endzustandsmenge E anstellen. Falls es zwei Zustände $z \in E$ und $z' \notin E$ mit $z \equiv_M z'$ geben würde, so wäre wegen $z \in E$ zunächst $[z]_M \in E$ und wegen $z' \notin E$ analog $[z']_M \notin E$. Dies wäre wegen $[z]_M = [z']_M$ ein Widerspruch. Wir haben uns aber bereits klar gemacht, dass es zwei solche Zustände nicht geben kann (siehe Lemma 2.5.5). Man sagt deshalb, dass δ und E *wohldefiniert* sind.

Analog kann man nun alle übrigen Übergänge ermitteln und erhält:

$$\delta_L(\{z_0, z_2\}, 0) = \{z_1, z_3\} \ , \quad \delta_L(\{z_0, z_2\}, 1) = \{z_0, z_2\} \ , \quad \delta_L(\{z_1, z_3\}, 0) = \{z_4\} \ ,$$
$$\delta_L(\{z_1, z_3\}, 1) = \{z_0, z_2\} \ , \quad \delta_L(\{z_4\}, 0) = \{z_4\} \ , \quad \delta_L(\{z_4\}, 1) = \{z_4\} \ .$$

Der Zustandsgraph von M_L sieht also wie folgt aus:



Dieser Automat beschreibt offenbar die Sprache aller Wörter, in denen 00 als Teilwort vorkommt. Wenn man sich etwas genauer mit dem Ausgangsautomaten M beschäftigt, so sieht man, dass M genau die gleiche Sprache akzeptiert. Dies ist ebenfalls kein Zufall, es gilt nämlich der folgende Satz:

2.5.7 Satz M_L ist zu M äquivalent, d.h. es gilt $L(M_L) = L(M) = L$.

Beweis: Die Definition von δ_L impliziert direkt, dass für die zugehörige verallgemeinerte Übergangsfunktion $\hat{\delta}_L$ stets

$$\forall x \in \Sigma^* \colon \hat{\delta}_L([z_0]_M, x) = [\hat{\delta}(z_0, x)]_M$$

gilt. Zusammen mit der Definition von z_L und E_L ergibt sich so für beliebige $x \in \Sigma^*$:

$$\begin{aligned} x \in L(M_L) &\iff \hat{\delta}_L(z_L, x) \in E_L \iff \hat{\delta}_L([z_0]_M, x) \in E_L \\ &\iff [\hat{\delta}(z_0, x)]_M \in E_L \iff \hat{\delta}(z_0, x) \in E \iff x \in L \ . \end{aligned}$$

Also gilt $L(M_L) = L(M) = L$. □

Immer wenn mehrere Zustände bei der Konstruktion von M_L in einem neuen gemeinsamen Zustand zusammengefasst werden, nimmt die Anzahl der resultierenden Zustände in M_L gegenüber M ab. Also ist M_L „kleiner“ als M (bzw. höchstens genauso groß, nämlich dann, wenn sich keine Verschmelzungen ergeben). Man kann sogar zeigen, dass M_L die geringste Größe hat:

2.5.8 Satz Sei M ein DEA für eine reguläre Sprache L, und sei M_L der aus M konstruierte Äquivalenzklassenautomat. Dann gilt:

- Kein L akzeptierender DEA besitzt weniger Zustände als M_L, d.h. M_L ist ein *Minimalautomat* für L.
- Genauer gesagt handelt es sich um *den* Minimalautomat, d.h. jeder weitere DEA N für L mit genauso wenig Zuständen wie M_L ist — abgesehen von evtl. unterschiedlichen Bezeichnungen für die einzelnen Zustände — zu M_L identisch. Die Zustandsgraphen sehen demnach also gleich aus, d.h. N und M_L sind zueinander *isomorph*. Man muss sich dann lediglich noch überlegen, welche Zustände aus den verschiedenen Automaten sich wechselseitig entsprechen.

Um diese wichtigen Eigenschaften von M_L zu zeigen, müssen wir uns zunächst ein einfaches, aber wichtiges Argument herleiten.

2.5.9 Lemma (*Schubfachprinzip*) Seien $A_1, A_2, \ldots, A_\ell$ und $B_1, B_2, \ldots, B_k$ zwei *Zerlegungen* einer beliebigen Menge C, d.h. es gilt

$$A_1 \dot\cup A_2 \dot\cup \ldots \dot\cup A_\ell = C \qquad \text{und} \qquad B_1 \dot\cup B_2 \dot\cup \ldots \dot\cup B_k = C \ .$$

Erinnerung: „$\dot\cup$“ (vgl. Def. 1.2.1 auf Seite 13) bezeichnet die disjunkte Vereinigung, d.h. für je zwei Mengen A_i und A_j mit $i \neq j$ gilt $A_i \cap A_j = \emptyset$. Jedes Element von C ist also (nur) in genau einer A– sowie in genau einer B–Menge enthalten. Ferner sollen die folgenden Restriktionen gelten:

- Jede B–Menge enthält zumindest ein Element, d.h. für alle $1 \leq j \leq k$ gilt $B_j \neq \emptyset$.
- Sind je zwei Elemente x, y in einer Menge A_i enthalten ($1 \leq i \leq \ell$), so sind sie auch gemeinsam in einer bestimmten B_j–Menge enthalten ($1 \leq j \leq k$), d.h. es gilt:

$$\forall 1 \leq i \leq \ell \colon \forall x, y \in A_i \colon \exists 1 \leq j \leq k \colon \ x, y \in B_j \ .$$

Dann gelten die folgenden drei Eigenschaften:

- Es gilt $\ell \geq k$.
- Es existiert eine Funktion

$$\sigma : \{1, \ldots, \ell\} \longrightarrow \{1, \ldots, k\}$$

  mit $A_i \subseteq B_{\sigma(i)}$ für alle $1 \leq i \leq \ell$.
- Für alle $j \in \{1, \ldots, k\}$ existiert ein $i \in \{1, \ldots, \ell\}$ mit $\sigma(i) = j$.
- Im Spezialfall $\ell = k$ gibt es für jedes $j \in \{1, \ldots, \ell\}$ sogar immer nur genau ein $i \in \{1, \ldots, \ell\}$ mit $\sigma(i) = j$, und es gilt $A_i = B_{\sigma(i)}$ für alle $1 \leq i \leq \ell$.

Beweis: Jede A–Menge A_i ist Teilmenge einer passend gewählten B–Menge, d.h. es gilt $A_i \subseteq B_j$ für ein geeignet gewähltes j. Denn falls sich der Inhalt von A_i auf mindestens zwei B–Mengen verteilen würde (etwa $A_i \cap B_j \neq \emptyset$ und $A_i \cap B_{j'} \neq \emptyset$ für passend gewählte Indizes $j \neq j'$), so würden zwei Elemente $x \in A_i \cap B_j$ und $y \in A_i \cap B_{j'}$ existieren, die der zweiten Restriktion widersprechen würden. Folglich existiert eine Abbildung

$$\sigma : \{1, \ldots, \ell\} \longrightarrow \{1, \ldots, k\} \ ,$$

die jedem A_i die passende Obermenge B_j zuordnet, so dass also $A_i \subseteq B_{\sigma(i)}$ für alle $1 \leq i \leq \ell$ gilt. Außerdem gibt es für jedes $j \in \{1, \ldots, k\}$ (mindestens) ein $i \in \{1, \ldots, \ell\}$ mit $\sigma(i) = j$. Um dies zu sehen, betrachte man eine beliebige Menge B_j. Gemäß der ersten Restriktion ist B_j nicht leer, enthält also mindestens ein Element x. Da die B–Mengen eine Zerlegung von C darstellen, kommt x also nur in B_j vor. Ebenso kommt x nur in genau einer A–Menge A_i vor. Dann muss aber $\sigma(i) = j$ gelten, ansonsten wäre wegen dem Element $x \in A_i$ die Bedingung $A_i \subseteq B_{\sigma(i)}$ verletzt.

Die Funktion σ besitzt maximal ℓ verschiedene Funktionswerte, da im Definitionsbereich nur ℓ viele Indizes zum Abbilden zur Verfügung stehen. Folglich muss $\ell \geq k$ gelten, denn ansonsten könnte σ nicht alle k verschiedenen Indizes im Bildbereich abdecken. Aus dem

gleichen Grund können im Fall $\ell = k$ keine zwei verschiedenen Mengen A_i und $A_{i'}$ mit $\sigma(i) = \sigma(i')$ existieren. Denn dann würde es zusammen mit dem Abbild der restlichen $\ell - 2$ Indizes maximal $1 + (\ell - 2) = \ell - 1 = k - 1 < k$ verschiedene Funktionswerte geben, die nicht alle Indizes im Bildbereich abdecken können. Folglich gibt es im Fall $\ell = k$ für jedes $j \in \{1, \ldots, \ell\}$ immer nur genau ein $i \in \{1, \ldots, \ell\}$ mit $\sigma(i) = j$. Es gibt demnach für jede B–Menge B_j genau eine A–Menge A_i mit $A_i \subseteq B_{\sigma(i)}$. Angenommen, für ein i und $j := \sigma(i)$ wäre die Teilmengenbeziehung $A_i \subseteq B_j$ echt. Sei in diesem Fall $x \in B_j \setminus A_i$ eines der restlichen Elemente. Da die A–Mengen eine Zerlegung von C darstellen und $x \notin A_i$ gilt, muss x in irgendeiner anderen Menge $A_{i'}$ enthalten sein. Wegen $x \in B_j$ muss dann $\sigma(i') = j$ gelten, was wegen $\sigma(i) = \sigma(i') = j$ der soeben hergeleiteten Tatsache widerspricht, dass für jedes $j \in \{1, \ldots, \ell\}$ immer nur genau ein $i \in \{1, \ldots, \ell\}$ mit $\sigma(i) = j$ existiert. Also gilt $A_i = B_{\sigma(i)}$ für alle $1 \leq i \leq \ell$. Damit ist alles gezeigt. □

Nun können wir den Beweis von Satz 2.5.8 angehen:

Beweis: Der Äquivalenzklassenautomat $M_L = (Z_L, \Sigma, \delta_L, z_L, E_L)$ besitzt eine bestimmte Anzahl von Zuständen, die wir mit k bezeichnen. Nach einer Umbenennung aller Zustände können wir also annehmen, dass $Z_L = \{z'_1, \ldots, z'_k\}$ gilt. Wir wollen zeigen, dass M_L höchstens so viele Zustände wie jeder andere DEA N für die Sprache L besitzt. Ist also $N := (Z_N, \Sigma, \delta_N, z_N, E_N)$ ein solcher weiterer DEA mit $L(N) = L$ und der Zustandsmenge $\{z_1, \ldots, z_\ell\}$, so müssen wir $\ell = |Z_N| \geq |Z_L| = k$ nachweisen. Hierfür wenden wir das Schubfachprinzip an.

Wir betrachten sowohl für N als auch für M_L diejenigen Wörter aus Σ^*, mit denen man bestimmte Zustände $z_i \in Z_N$ bzw. $z'_j \in Z_L$ erreichen kann. Formell betrachten wir also die Mengen

$$A_i := \{x \mid \hat{\delta}_N(z_N, x) = z_i\} \qquad \text{bzw.} \qquad B_j := \{x \mid \hat{\delta}_L(z_L, x) = z'_j\} \ .$$

Mit Hilfe dieser Mengen kann man Σ^* zerlegen. Es gilt nämlich

$$\Sigma^* = \bigcup_{z_i \in Z_N} A_i \qquad \text{bzw.} \qquad \Sigma^* = \bigcup_{z'_j \in Z_L} B_j \ .$$

Beachten Sie, dass es sich wirklich jeweils um Zerlegungen handelt, da man in DEAs mit einem Wort nicht mehrere unterschiedliche Zustände erreichen kann. Die A_i– bzw. B_j–Mengen sind also paarweise disjunkt. Da wir zudem von DEAs ohne überflüssige Zustände ausgehen, ist jeder Zustand mit zumindest einem Wort erreichbar und somit jede der obigen Wortmengen nicht leer. Damit ist insbesondere die erste Restriktion des Schubfachprinzips erfüllt.

Für den Nachweis der zweiten Restriktion sei nun zunächst $M := (Z_M, \Sigma, \delta_M, z_M, E_M)$ der Ausgangsautomat für die Konstruktion von M_L. Wir zeigen, dass für einen beliebigen Zustand $z_i \in Z_N$ und beliebige Wörter $v, w \in A_i$ stets

$$\forall x \in \Sigma^* \colon \ (\hat{\delta}_M(\hat{\delta}_M(z_M, v), x) \in E \ \Leftrightarrow \ \hat{\delta}_M(\hat{\delta}_M(z_M, w), x) \in E)$$

gilt. Für beliebige Wörter $x \in \Sigma^*$ ergibt sich nämlich

$$\begin{aligned}
& \hat{\delta}_M(\hat{\delta}_M(z_M, v), x) \in E_M \iff \hat{\delta}_M(z_M, vx) \in E_M \\
\iff\ & vx \in L \iff \hat{\delta}_N(z_N, vx) \in E_N \iff \hat{\delta}_N(\hat{\delta}_N(z_N, v), x) \in E_N \\
\iff\ & \hat{\delta}_N(z_i, x) \in E_N \iff \hat{\delta}_N(\hat{\delta}_N(z_N, w), x) \in E_N \iff \hat{\delta}_N(z_N, wx) \in E_N \\
\iff\ & wx \in L \iff \hat{\delta}_M(z_M, wx) \in E_M \iff \hat{\delta}_M(\hat{\delta}_M(z_M, w), x) \in E_M \ .
\end{aligned}$$

Nun ist die gezeigte Bedingung

$$\forall x \in \Sigma^*\colon (\hat{\delta}_M(\hat{\delta}_M(z_M, v), x) \in E \Leftrightarrow \hat{\delta}_M(\hat{\delta}_M(z_M, w), x) \in E)$$

aber gleichbedeutend mit

$$\hat{\delta}_M(z_M, v) \equiv_M \hat{\delta}_M(z_M, w) \ ,$$

d.h. die beiden Zustände $\hat{\delta}_M(z_M, v)$ und $\hat{\delta}_M(z_M, w)$ liegen in der gleichen Äquivalenzklasse bzgl. $\equiv_M$. Im Äquivalenzklassenautomat M_L erreichen die beiden Wörter v und w also den gleichen Zustand $[\hat{\delta}(z_M, v)]_M = [\hat{\delta}(z_M, w)]_M$. Die zweite Restriktion des Schubfachprinzips ist demnach auch erfüllt, denn für zwei Wörter $v, w \in A_i$ findet man stets eine Menge B_j mit $v, w \in B_j$ (indem man sich nämlich den Index j des Zustands $z_j = [\hat{\delta}(z_M, v)]_M \in Z_L$ anschaut).

Nach dem Schubfachprinzip folgt nun direkt $|Z_N| = \ell \geq k = |Z_L|$. Jeder frei wählbare DEA N mit $L(N) = L$ hat also mindestens soviele Zustände wie M_L. Damit ist die Minimalität von M_L gezeigt.

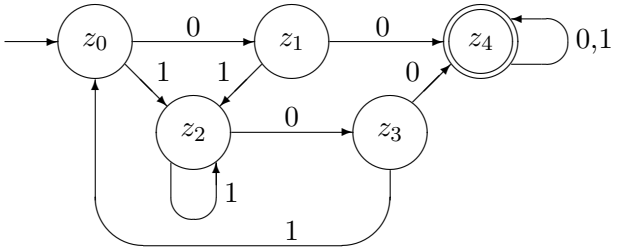
Für die Eindeutigkeit von M_L setzen wir mit unserer Beweisführung fort. Sei dazu wieder $N = (Z_N, \Sigma, \delta_N, z_N, E_N)$ ein beliebiger DEA, der diesmal jedoch genauso wenig Zustände wie M_L besitzt, d.h. es soll nun zusätzlich $\ell = |Z_N| = |Z_L| = k$ gelten. Nach dem Schubfachprinzip ist die Abbildung σ in diesem Fall eine eindeutige Zuordnung, d.h. jeder Zustand $z_i \in Z_N$ entspricht genau einem Zustand $z'_j \in Z_L$, und jeder Wortmenge A_i wird eindeutig eine identische Wortmenge B_j zugeordnet. Die Menge an Wörtern, mit denen man in N den Zustand z_i erreicht, ist also identisch zu der Menge an Wörtern, mit denen man in M_L den Zustand z'_j erreicht. Offenbar müssen also nach einer entsprechenden Umbenennung aller Zustände (jeweils von z_i nach z'_j) die Zustandsgraphen beider Automaten gleich sein. □

Wir haben noch nicht geklärt, wie man für einen beliebigen DEA $M = (Z, \Sigma, \delta, z_0, E)$ (wie immer mit totaler Übergangsfunktion δ und ohne unerreichbare Zustände) seine $\equiv_M$–äquivalenten Zustände berechnen kann. Dies holen wir nun nach und benutzen dafür den folgenden Algorithmus:

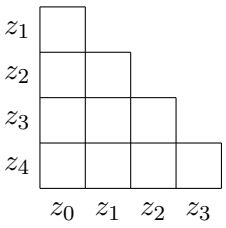
1. Stelle eine Tabelle aller Zustandspaare $\{z, z'\}$ mit $z \neq z'$ von M auf. (Wir bezeichnen hier ausnahmsweise 2–elementige Teilmengen von Z als Paare, obwohl sie nicht geordnet sind.) In dieser Tabelle werden nachfolgend alle nicht zueinander äquivalenten Zustandspaare markiert.
2. Markiere alle Paare $\{z, z'\}$, für die $z \in E$ und $z' \notin E$ (oder umgekehrt) gilt. Denn diese Paare sind gemäß Lemma 2.5.5 ganz sicher nicht miteinander äquivalent.

3. Teste für jedes noch unmarkierte Paar $\{z, z'\}$, ob es ein $a \in \Sigma$ gibt, so dass das Paar $\{\delta(z, a), \delta(z', a)\}$ bereits markiert ist. Wenn ja, dann markiere auch $\{z, z'\}$. Denn wenn $\delta(z, a)$ und $\delta(z', a)$ nicht äquivalent sind, dann trifft dies gemäß Lemma 2.5.4 auch auf z und z' zu.
4. Wiederhole den dritten Schritt, bis keine Änderung mehr eintritt.
5. Alle unmarkierten Paare in der Tabelle sind äquivalent und können zusammengefasst werden.

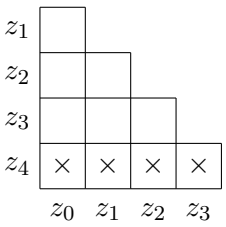
2.5.10 Beispiel Im DEA M aus Beispiel 2.5.6 sind alle Zustände von z_0 aus erreichbar, und alle Übergänge sind definiert, so dass der Algorithmus angewendet werden kann:



Schritt 1: Bei den hier vorhandenen fünf Zuständen z_0, z_1, z_2 ,z_3 und z_4 ergibt sich folgendes Layout für eine Tabelle, in der jedes Zustandspaar $\{z, z'\}$ mit $z \neq z'$ genau einmal vorkommt:

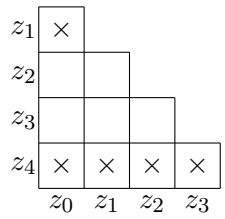
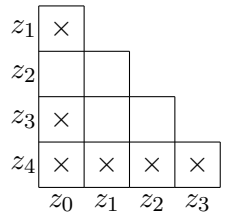
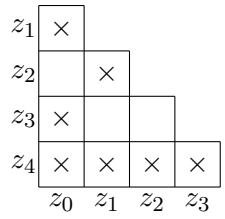
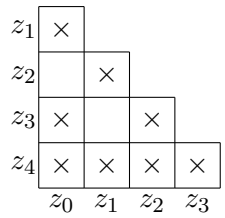


Schritt 2: Wir kreuzen alle Paare an, deren eine Komponente ein Endzustand und deren andere Komponente ein Nicht–Endzustand ist. Dies sind hier also alle Paare, die z_4 enthalten:



Schritt 3: Für jedes noch unmarkierte Paar $\{z_i, z_j\}$ mit $i < j$ und jedes $a \in \Sigma$ wird getestet, ob $\{\delta(z_i, a), \delta(z_j, a)\}$ bereits markiert ist. Wenn ja, dann wird auch $\{z_i, z_j\}$ markiert. Hier gilt $\Sigma = \{0, 1\}$, d.h. wir führen für jedes Zustandspaar den Test z.B. zuerst für $a = 0$ und anschließend für $a = 1$ durch. Falls aufgrund des ersten Tests schon eine Markierung notwendig wird, kann auf den zweiten Test natürlich verzichtet werden.

Die Tests verlaufen wie folgt:

$\{z_i, z_j\}$	a	$\{\delta(z_i, a), \delta(z_j, a)\}$	$\leftarrow$ bereits markiert?	Aktion	Aktuelle Tabelle
$\{z_0, z_1\}$	0	$\{z_1, z_4\}$	ja	markiere $\{z_0, z_1\}$	
$\{z_0, z_2\}$	0	$\{z_1, z_3\}$	nein	—	unverändert
$\{z_0, z_2\}$	1	$\{z_2, z_2\}$	nein	—	unverändert
$\{z_0, z_3\}$	0	$\{z_1, z_4\}$	ja	markiere $\{z_0, z_3\}$	
$\{z_1, z_2\}$	0	$\{z_4, z_3\}$	ja	markiere $\{z_1, z_2\}$	
$\{z_1, z_3\}$	0	$\{z_4, z_4\}$	nein	—	unverändert
$\{z_1, z_3\}$	1	$\{z_2, z_0\}$	nein	—	unverändert
$\{z_2, z_3\}$	0	$\{z_3, z_4\}$	ja	markiere $\{z_2, z_3\}$	

Schritt 4: Bei einer Wiederholung der im letzten Schritt vollzogenen Tests treten keine neuen Markierungen auf.

Schritt 5: Das Verfahren endet an dieser Stelle. Alle Paare, die jetzt noch unmarkiert dastehen (hier also $\{z_0, z_2\}$ und $\{z_1, z_3\}$) sind äquivalent und können für den Minimalautomat miteinander verschmolzen werden.

Durch die Verschmelzung von $\{z_0, z_2\}$ sowie $\{z_1, z_3\}$ in jeweils einen Zustand ergeben sich die in Beispiel 2.5.6 behaupteten Äquivalenzklassen bzgl. $\equiv_M$ (und damit die dort genannten Zustände des Minimalautomaten).

Ein Minimalautomat besitzt gemäß Konstruktion immer eine totale Übergangsfunktion. Folglich kann der Minimalautomat auch einen „toten“ Zustand enthalten, also einen Zustand, aus dem man keinen Endzustand mehr erreichen kann. Da wir im Allgemeinen nicht auf totale Übergangsfunktionen angewiesen sind, kann man manchmal (scheinbar verwirrenderweise) einen Minimalautomat durch das Entfernen eines toten Zustands noch weiter verkleinern. In Beispiel 2.5.6 ist dies allerdings nicht der Fall.

Mit einer „geschickten“ Implementierung lässt sich die Minimierung eines Automaten in $O(|\Sigma| \cdot n \log n)$ viel Zeit durchführen, wobei n die Anzahl der Zustände bezeichnet (siehe [4], Seiten 139–146). Der hier vorgestellte Algorithmus erfüllt diese Zeitvorgabe nicht, da schon allein das Hinschreiben der leeren Tabelle

$$1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2} = \Omega(n^2)$$

viel Zeit benötigt (zum Beweis der Summenformel verweisen wir auf das Beispiel in Anhang A). Für kleinere Automaten ist der Algorithmus jedoch völlig ausreichend.

Zum Ende dieses Abschnitts soll noch ein weiteres Kriterium für die Nichtregularität einer Sprache hergeleitet werden. Anders als das Pumping–Lemma kann man es jedoch auch einsetzen, um die Regularität einer Sprache zu belegen. Den Anfang macht auch hier die Definition einer Äquivalenzrelation.

2.5.11 Definition Sei $L \subseteq \Sigma^*$ und $x, y \in \Sigma^*$. Wir definieren eine Relation R_L wie folgt:

$$x \, R_L \, y \;:\Longleftrightarrow\; \forall z \in \Sigma^* \colon (xz \in L \;\Leftrightarrow\; yz \in L) \;.$$

Zwei Wörter x und y stehen also in Relation zueinander, wenn sie sich beim Anfügen von beliebigen Wörtern $z \in \Sigma^*$ bzgl. der Mitgliedschaft in L gleich verhalten. Wegen der Möglichkeit $z = \varepsilon$ gilt insbesondere $x \in L \;\Leftrightarrow\; y \in L$.

2.5.12 Lemma R_L ist eine Äquivalenzrelation.

Beweis: Wie in Lemma 2.5.2 können wir mit dem Beispiel 1.5.9 auf Seite 21 argumentieren. Dort wurde gezeigt, dass für beliebige Mengen A und B sowie für ein beliebiges Prädikat $\varphi : A \times B \longrightarrow \{\mathrm{W}, \mathrm{F}\}$ die Relation R über A mit

$$xRy \;:\Longleftrightarrow\; \forall z \in B \colon (\varphi(x, z) \;\Leftrightarrow\; \varphi(y, z))$$

eine Äquivalenzrelation ist. Diesmal ergibt sich die Behauptung also durch den Spezialfall $A := \Sigma^*$, $B := \Sigma^*$ und $\varphi(x, z) := xz \in L$. □

Σ^* zerfällt unter R_L also in Äquivalenzklassen. Mit $[x]_L$ sei die zu $x \in \Sigma^*$ gehörige R_L–Äquivalenzklasse bezeichnet, also $[x]_L := \{y \in \Sigma^* \mid x \, R_L \, y\}$. Wir wissen bereits, dass im Fall $x \, R_L \, y$ entweder beide Wörter in L liegen müssen oder keines von beiden. Folglich gilt auch für alle $x \in \Sigma^*$, dass die zugehörige Äquivalenzklasse $[x]_L$ entweder eine Teilmenge von L oder von $\Sigma^* \setminus L$ ist.

Mit $\mathit{Index}(R_L) := |\{[x]_L \mid x \in \Sigma^*\}|$ wird die Anzahl der Äquivalenzklassen von R_L bezeichnet.

2.5.13 Lemma Sei $x, y \in \Sigma^*$. Ist $x \, R_L \, y$, so gilt auch $xw \, R_L \, yw$ für alle $w \in \Sigma^*$, d.h. R_L ist *rechtsinvariant*.

Beweis: $x \, R_L \, y$ bedeutet, dass für alle Wörter $z \in \Sigma^*$ stets $xz \in L \Leftrightarrow yz \in L$ gilt. Insbesondere gilt dies für Wörter der Form $z = wz'$, wobei z' frei wählbar ist, d.h. wir haben

$$\forall z' \in \Sigma^* \colon (xwz' \in L \Leftrightarrow ywz' \in L) \ .$$

Also gilt $xw \, R_L \, yw$. □

2.5.14 Satz Ist eine Sprache $L \subseteq \Sigma^*$ regulär, so ist der Index von R_L endlich.

Beweis: Sei L regulär und $M = (Z, \Sigma, \delta, z_0, E)$ ein deterministischer Automat mit $L = L(M)$. Dann gilt für alle Wörter $x, y \in \Sigma^*$:

$$\begin{aligned}
& x \, R_L \, y \\
\iff \quad & \forall w \in \Sigma^* \colon (xw \in L \Leftrightarrow yw \in L) \\
\iff \quad & \forall w \in \Sigma^* \colon (\hat{\delta}(z_0, xw) \in E \Leftrightarrow \hat{\delta}(z_0, yw) \in E) \\
\iff \quad & \forall w \in \Sigma^* \colon (\hat{\delta}(\hat{\delta}(z_0, x), w) \in E \Leftrightarrow \hat{\delta}(\hat{\delta}(z_0, y), w) \in E) \\
\iff \quad & \hat{\delta}(z_0, x) \equiv_M \hat{\delta}(z_0, y) \ .
\end{aligned}$$

Falls also zwei Wörter x und y bzgl. R_L in unterschiedlichen Äquivalenzklassen liegen, so sind auch $\hat{\delta}(z_0, x)$ und $\hat{\delta}(z_0, y)$ bzgl. $\equiv_M$ in unterschiedlichen Äquivalenzklassen enthalten. R_L und $\equiv_M$ besitzen somit gleich viele Äquivalenzklassen, nämlich genau so viele, wie der Minimalautomat für L Zustände besitzt. Insbesondere sind das nur endlich viele. □

Satz 2.5.14 wird wie das Pumping–Lemma häufig in seiner kontrapositorischen Form angewendet. Zum Nachweis der Nicht–Regularität einer Sprache L genügt es zu zeigen, dass R_L einen unendlichen Index hat. Man braucht dazu natürlich nicht alle R_L–Äquivalenzklassen zu bestimmen. Vielmehr genügt es, unendlich viele R_L–inäquivalente Worte anzugeben.

2.5.15 Beispiel Wir beweisen noch einmal, dass $L := \{a^n b^n \mid n \geq 1\}$ nicht regulär ist. Für $i, j \in \mathbb{N}$ mit $i \neq j$ sind a^i und a^j nämlich nicht R_L–äquivalent, denn für $z = b^i$ gilt $a^i b^i \in L$, aber $a^j b^i \notin L$. Da es unendlich viele Paare i, j mit $i \neq j$ gibt, hat man unendlich viele Repräsentanten von verschiedenen Äquivalenzklassen. Also gilt $\mathit{Index}(R_L) = \infty$, d.h. L ist nicht regulär.

2.5.16 Beispiel Wir beweisen noch einmal, dass $L := \{0^m \mid m \text{ ist eine Quadratzahl}\}$ nicht regulär ist. Für $i, j \in \mathbb{N}$ mit $i \neq j$ sind 0^i und 0^j nämlich nicht R_L–äquivalent. Um dies zu sehen, nehmen wir o.B.d.A. $i < j$ an (sonst vertauschen wir in den nachfolgenden Ausführungen die Rollen von i und j). Unter Zuhilfenahme der Wortverlängerung

$$z := 0^{j^2-j}$$

ergibt sich nun:

- $0^j z = 0^j 0^{j^2-j} = 0^{j^2}$ ist ein Wort aus L.
- $0^i z = 0^i 0^{j^2-j} = 0^{j^2-j+i}$ ist kein Wort aus L, denn seine Länge $j^2-(j-i)$ ist wegen $j > i \geq 1$ und

  $$(j-1)^2 = j^2 - 2j + 1 < j^2 - j + 1 \leq j^2 - j + i = j^2 - (j-i) < j^2$$

  echt zwischen zwei benachbarten Quadratzahlen eingeschlossen und kann deshalb selbst keine Quadratzahl sein.

Somit muss jedes der unendlich vielen Wörter der Form 0^i in einer anderen Äquivalenzklasse liegen, von denen es folglich unendlich viele geben muss. Also gilt $\mathit{Index}(R_L) = \infty$, d.h. L ist nicht regulär.

2.5.17 Beispiel Wir beweisen noch einmal, dass $L := \{0^p \mid p \text{ ist eine Primzahl}\}$ nicht regulär ist. Für zwei verschiedene natürliche Zahlen i und j sind 0^i und 0^j nämlich nicht R_L–äquivalent. Wir nehmen wieder o.B.d.A. $i < j$ an (sonst können wir wieder die Rollen von i und j im nachfolgenden Beweis vertauschen). Sei q eine Primzahl mit $q \geq j$. Die Primzahl q existiert, da es nach Satz 1.4.2 unendlich viele Primzahlen gibt. Sei weiter p die größte Primzahl mit

$$q \leq p \leq q! + 1 \ .$$

Die Primzahl p existiert ebenfalls, da zumindest q die geforderte Bedingung erfüllt. Nun wählen wir die Wortverlängerung $z := 0^{p-i}$ (man beachte $p \geq q \geq j > i$ nach der Wahl von q). Dann gilt $0^i z = 0^i 0^{p-i} = 0^p \in L$, da p eine Primzahl ist. Das andere verlängerte Wort

$$0^j z = 0^j 0^{p-i} = 0^{p+(j-i)}$$

ist jedoch kein Wort aus L, denn wie wir gleich sehen werden, ist $p + (j - i)$ keine Primzahl. p wurde nämlich als größte Primzahl mit $p \leq q! + 1$ gewählt, d.h. alle Zahlen

zwischen einschließlich $p+1$ und $q!+1$ können keine Primzahlen sein. Weiter sind auch alle folgenden Zahlen von $q!+2$ bis $q!+q$ keine Primzahlen, denn

$$q!+2 = 2 \cdot 3 \cdots (q-1) \cdot q + 2$$

ist durch 2 teilbar, analog ist $q!+3$ durch 3 teilbar, usw. Also sind alle Zahlen von $p+1$ bis einschließlich $q!+q$ keine Primzahlen. Wegen

$$p+1 \leq p+(j-i) \leq p+j-1 \leq p+q-1 \leq q!+1+q-1 = q!+q$$

liegt $p+(j-i)$ jedoch genau in diesem Bereich.

Also haben wir auch hier unendlich viele Wörter der Form 0^i, die paarweise R_L–inäquivalent sind und folglich in jeweils anderen Äquivalenzklassen liegen müssen. Also gibt es auch hier unendlich viele Äquivalenzklassen. Folglich gilt erneut $\mathit{Index}(R_L) = \infty$, d.h. L kann nicht regulär sein.

2.5.18 Beispiel Die unendliche Folge $F_0, F_1, F_2, \ldots$ der *Fibonacci–Zahlen* (vgl. Beispiel A.1.2 auf Seite 256) ist wie folgt definiert:

$$F_n := \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & \text{falls } n \geq 2 \end{cases}$$

Die Zahlenreihe beginnt also mit $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Die unäre Kodierung dieser Zahlen

$$L := \{0^{F_n} \mid n \in \mathbb{N}_0\} = \{\varepsilon, 0, 00, 000, 00000, 00000000, \ldots\}$$

ist ebenfalls nicht regulär. Dazu betrachten wir alle Wörter 0^{F_n} mit $n \geq 3$. Jedes solche Wort liegt in einer anderen Äquivalenzklasse, da je zwei Wörter 0^{F_m} und 0^{F_n} mit o.B.d.A. $m < n$ nicht R_L–äquivalent sein können. Denn die Wortverlängerung $z := 0^{F_{m-1}}$ ergibt zum einen

$$0^{F_m} z = 0^{F_m} 0^{F_{m-1}} = 0^{F_m + F_{m-1}} = 0^{F_{m+1}} \in L \ ,$$

zum anderen jedoch $0^{F_n} z = 0^{F_n} 0^{F_{m-1}} = 0^{F_n + F_{m-1}} \notin L$, was man leicht einsieht: wegen $3 \leq m < n$ ist nämlich $F_{m-1} \geq 1$ und $F_{m-1} < F_{n-1}$, so dass $F_n + F_{m-1}$ wegen

$$F_n < F_n + F_{m-1} < F_n + F_{n-1} = F_{n+1}$$

zwischen zwei benachbarten Fibonacci–Zahlen liegt und deshalb selbst keine Fibonacci–Zahl sein kann.

Anders als das Pumping–Lemma 2.4.1 kann man den Index von R_L auch dazu verwenden, um die Regularität einer Sprache nachzuweisen. Die Rückrichtung von Satz 2.5.14 gilt nämlich ebenfalls:

2.5.19 Satz Sei $L \subseteq \Sigma^*$ eine Sprache. Ist der Index von R_L endlich, so ist L regulär.

Beweis: Sei $k := \mathit{Index}(R_L) < \infty$. Σ^* zerfällt unter R_L in k Äquivalenzklassen, d.h. es existieren k Repräsentanten $x_1, x_2, \ldots, x_k \in \Sigma^*$ mit

$$\Sigma^* \;=\; [x_1]_L \,\dot\cup\, [x_2]_L \,\dot\cup\, \ldots \,\dot\cup\, [x_k]_L \;\;.$$

Wir definieren jetzt einen DEA $M = (Z, \Sigma, \delta, z_0, E)$, dessen Zustände gerade die R_L–Äquivalenzklassen sind (M wird deshalb ebenfalls als *Äquivalenzklassenautomat* bezeichnet):

- $Z = \{[x_1]_L, [x_2]_L, \ldots, [x_k]_L\}$
- $\delta([x]_L, a) := [xa]_L$
- $z_0 := [\varepsilon]_L$
- $E := \{[x]_L \,|\, x \in L\}$.

Wir machen uns diese Konstruktion zunächst an einem Beispiel klar und fahren dann mit dem Beweis fort. Für die Sprache $L := \{x \in \{0,1\}^* \,|\, x \text{ endet mit } 00\}$ ergeben sich die folgenden drei Äquivalenzklassen:

$$\begin{aligned}
[\varepsilon]_L &= \{x \in \{0,1\}^* \,|\, x \text{ endet nicht mit } 0\} \;\;, \\
[0]_L &= \{x \in \{0,1\}^* \,|\, x \text{ endet mit } 0, \text{ aber nicht mit } 00\} \;\;, \\
[00]_L &= \{x \in \{0,1\}^* \,|\, x \text{ endet mit } 00\} \;\;,
\end{aligned}$$

und es gilt

$$L \;=\; [00]_L$$

sowie

$$\Sigma^* \;=\; [\varepsilon]_L \,\dot\cup\, [0]_L \,\dot\cup\, [00]_L \;\;.$$

Der zugehörige Äquivalenzklassenautomat hat also drei Zustände, nämlich $[\varepsilon]_L$, $[0]_L$ und $[00]_L$, und wir erhalten:

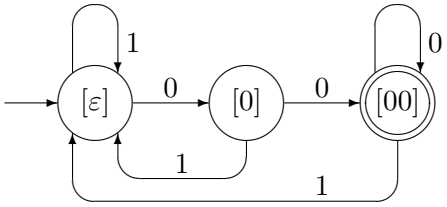
- $z_0 := [\varepsilon]_L$ ist der Startzustand,
- δ, gegeben durch $\delta([x]_L, a) := [xa]_L$, besitzt die Übergänge

$$\delta([\varepsilon]_L, 0) = [0]_L \;\;, \quad \delta([0]_L, 0) = [00]_L \;\;, \quad \delta([00]_L, 0) = [000]_L = [00]_L,$$

$$\delta([\varepsilon]_L, 1) = [1]_L = [\varepsilon]_L \;\;, \quad \delta([0]_L, 1) = [1]_L = [\varepsilon]_L \;\;, \quad \delta([00]_L, 1) = [001]_L = [\varepsilon]_L \;\;,$$

- die Menge E enthält nur einen Endzustand, nämlich $[00]_L$.

Der Äquivalenzklassenautomat entspricht also diesem Zustandsgraph:



Zur Fortführung des Beweises überprüfen wir jetzt zunächst wieder die *Wohldefiniertheit* von M: jeder Übergang $\delta([x]_L, a) := [xa]_L$ ist mit Hilfe eines Repräsentanten der Äquivalenzklasse $[x]_L$ (nämlich x selbst) festgelegt worden. Der Übergang muss aber von der konkreten Wahl des Repräsentanten unabhängig sein. Dies ist erfüllt, denn ein anderer Repräsentant $y \in [x]_L$ würde zwar formal zu der Klasse $[ya]_L$ führen, diese ist jedoch mit $[xa]_L$ identisch, da aus $x\, R_L\, y$ stets $xa\, R_L\, ya$ folgt (siehe Lemma 2.5.13). Die Wörter xa und ya gehören also zu der gleichen Klasse.

Ähnliches gilt für die Definition der Endzustandsmenge E. Falls es zwei Wörter $x \in L$ und $y \notin L$ mit $x\, R_L\, y$ (also $[x]_L = [y]_L$) geben würde, so würde wegen $x \in L$ zunächst $[x]_L \in E$ und wegen $y \notin L$ analog $[y]_L \notin E$ folgen. Dies wäre wegen $[x]_L = [y]_L$ ein Widerspruch. Wir haben uns aber bereits klar gemacht, dass es zwei solche Wörter nicht geben kann (siehe Def. 2.5.11).

Die Festlegung von δ impliziert direkt $\hat{\delta}([\varepsilon]_L, x) = [x]_L$ für alle $x \in \Sigma^*$. Zusammen mit der Wahl von z_0 und E ergibt sich so für beliebige $x \in \Sigma^*$:

$$x \in L(M) \iff \hat{\delta}(z_0, x) \in E \iff \hat{\delta}([\varepsilon]_L, x) \in E \iff [x]_L \in E \iff x \in L \ .$$

L wird also durch einen DEA beschrieben, d.h. L ist regulär. Da die Anzahl der Zustände von M mit der Anzahl der Äquivalenzklassen von R_L und somit $\equiv_M$ identisch ist, handelt es sich bei M sogar um den Minimalautomat für L. □

Die beiden Resultate 2.5.14 und 2.5.19 sind allgemein als Satz von Myhill–Nerode bekannt:

2.5.20 Satz (Myhill[1], Nerode[2]) Eine Sprache $L \subseteq \Sigma^*$ ist genau dann regulär, wenn der Index von R_L endlich ist. □

2.6. Eigenschaften regulärer Sprachen

Wir fassen jetzt zum Abschluss noch einige wichtige Eigenschaften der regulären Sprachen zusammen.

[1] John R. Myhill, †1987, 1966–1987 Professor an der State University of New York.
[2] Anil Nerode, US–amerikanischer Mathematiker, Professor für Mathematik an der Cornell University.

2.6.1 Satz Die Menge der regulären Sprachen über Σ ist unter

1. Vereinigung,
2. Schnitt,
3. Komplement,
4. Produkt (d.h. Konkatenation) und
5. Sternbildung (d.h. Bildung der *Kleeneschen Hülle*).

abgeschlossen, d.h. wenn zwei Sprachen L und L' regulär sind, dann ist auch $L \cup L'$, $L \cap L'$, $\overline{L}$, $L \cdot L'$ sowie L^* eine reguläre Sprache.

Beweis: Wegen

$$\alpha, \beta \in \mathit{Reg}(\Sigma) \implies (\alpha|\beta),\ \alpha\beta,\ (\alpha)^\star \in \mathit{Reg}(\Sigma)$$

sind die Aussagen 1., 4. und 5. direkte Folgerungen aus dem Satz 2.3.6 von Kleene auf Seite 60.

Die dritte Aussage beweist man wie folgt. Sei $L \subseteq \Sigma^*$ regulär und $M = (Z, \Sigma, \delta, z_0, E)$ ein DEA mit $L(M) = L$. O.B.d.A. sei δ wieder total. Dann ist $\overline{M} := (Z, \Sigma, \delta, z_0, Z \setminus E)$ ein DEA mit $L(\overline{M}) = \overline{L}$, denn jedes Wort, mit dem man bislang nicht in einen Endzustand gelangt war, wird jetzt akzeptiert und umgekehrt. Also gilt Aussage 3.

Aussage 2. folgt unmittelbar aus den Aussagen 1. und 3. und der DeMorgan'schen Regel $L \cap L' = \overline{\overline{L} \cup \overline{L'}}$. □

Viele Fragestellungen bezüglich einer regulären Sprache L sind zudem *entscheidbar*, d.h. es existieren Algorithmen, die in *endlicher* Zeit die jeweilige Fragestellung beantworten.

2.6.2 Satz Folgende Probleme bezüglich regulärer Sprachen über einem Alphabet Σ sind *entscheidbar*, d.h. man kann Computerprogramme z.B. in Java erstellen, welche die nachfolgenden Fragen beantworten (dabei seien die Sprachen durch reguläre Grammatiken G oder Ausdrücke α bzw. durch endliche Automaten M beschrieben):

a) Das *Wortproblem*: Ist $x \in L(G)$ bzw. $x \in \varphi(\alpha)$ bzw. $x \in L(M)$?

b) Das *Leerheitsproblem*: Ist $L(G) = \emptyset$ bzw. $\varphi(\alpha) = \emptyset$ bzw. $L(M) = \emptyset$?

c) Das *Endlichkeitsproblem*: Ist $|L(G)| < \infty$ bzw. $|\varphi(\alpha)| < \infty$ bzw. $|L(M)| < \infty$?

d) Das *Äquivalenzproblem* (d.h. die Frage nach der Gleichheit): Ist $L(G_1) = L(G_2)$ bzw. $\varphi(\alpha_1) = \varphi(\alpha_2)$ bzw. $L(M_1) = L(M_2)$?

Beweis: Es ist ausreichend, die Entscheidbarkeit jeweils nur für den Fall einer regulären Grammatik, eines regulären Ausdrucks oder eines endlichen Automaten nachzuweisen, da wir alle Darstellungsformen gegebenenfalls vorher konstruktiv ineinander überführen können. Wir argumentieren deshalb hier exemplarisch mit DEAs bzw. ihren Zustandsgraphen.

a) Ist ein DEA M gegeben, so kann man einfach die Verarbeitung eines Wortes $x \in \Sigma^*$ durch M nachvollziehen (und dies sogar sehr effizient, nämlich in linearer Zeit $O(|x|)$).

b) Sei im Folgenden n die Zahl aus dem Pumping–Lemma, d.h. die Anzahl der Zustände von M. Wir beweisen das folgende Kriterium:

$$L(M) \neq \emptyset \iff \exists x \in L(M)\colon |x| < n \ .$$

Die Rückrichtung ist klar, denn wenn ein Wort $x \in L(M)$ mit $|x| < n$ existiert, so kann $L(M)$ insbesondere nicht leer sein (die Eigenschaft $|x| < n$ ist hierfür unwichtig). Für die Hinrichtung sei $L(M)$ nicht leer, und x sei ein möglichst kurzes Wort aus $L(M)$. Würde nun $|x| \geq n$ gelten, so könnte man x gemäß dem Pumping–Lemma in $x = uvw$ mit $|v| \geq 1$ und $|uv| \leq n$ zerlegen, so dass speziell $uv^0w = uw$ ein Wort aus $L(M)$ sein müßte. Wegen $|v| \geq 1$ wäre dann aber uw kürzer als $x = uvw$, was ein Widerspruch zu Wahl von x wäre. Also muss $|x| < n$ gelten.

Der Entscheidungsalgorithmus für $L(M) \neq \emptyset$ testet also alle Wörter $x \in \Sigma^*$ mit $|x| < n$ auf Mitgliedschaft in $L(M)$. Alle diese endlich vielen Tests kann man wegen der Entscheidbarkeit des Wortproblems in endlicher Gesamtzeit durchführen.

c) Wir gehen ganz ähnlich wie gerade eben vor. Sei wieder n die Zahl aus dem Pumping–Lemma, d.h. wieder die Anzahl der Zustände von M. Wir zeigen:

$$|L(M)| = \infty \iff \exists x \in L(M)\colon n \leq |x| < 2n \ .$$

Sei zunächst $|L(M)| = \infty$. Unter allen Wörtern $y \in L(M)$ mit $|y| \geq n$ sei $x \in L(M)$ ein kürzestes. Wir nehmen nun $|x| \geq 2n$ an. Nach dem Pumping–Lemma ist x zerlegbar in $x = uvw$ mit $|v| \geq 1$ und $|uv| \leq n$, so dass auch $uv^0w = uw \in L(M)$ gilt. Wegen $|v| \geq 1$ gilt wieder $|x| = |uvw| > |uw|$, und wegen $|v| \leq |uv| \leq n$ erhalten wir zudem $|uw| = |x| - |v| \geq 2n - n = n$, d.h. uw ist ein Wort aus $L(M)$, das kürzer ist als x, aber dennoch mindestens n Symbole umfasst. Dies widerspricht der Wahl von x.

Sei nun umgekehrt $x \in L(M)$, $n \leq |x| < 2n$. Aufgrund des Pumping–Lemmas ist x zerlegbar in $x = uvw$ mit $|v| \geq 1$ und $uv^iw \in L(M)$ für alle $i \geq 0$. Also gilt $|L(M)| = \infty$.

Der Entscheidungsalgorithmus für $|L(M)| < \infty$ testet hier also alle Wörter $x \in \Sigma^*$ mit $n \leq |x| < 2n$ auf Mitgliedschaft in $L(M)$. Auch hier sind diese endlich vielen Tests in endlicher Gesamtzeit entscheidbar.

d) Man konstruiere und vergleiche die (nach Satz 2.5.8 eindeutigen) zugehörigen Minimalautomaten. Mit diesem Trick lässt sich auch die Aussage b) elegant beweisen: der eindeutige (totale) Minimalautomat für die leere Sprache besteht nämlich nur aus einem Startzustand, der alle Zeichen wieder auf sich selbst überführt (Endzustände gibt es keine). Sollte also nach der Minimierung eines DEAs für eine reguläre Sprache genau dieser Minimalautomat übrigbleiben, so ist die Sprache leer. □

3

Kontextfreie Sprachen

3.1. Grundlagen

Wir haben uns bereits im zweiten Kapitel ein wenig mit kontextfreien Sprachen beschäftigt. Kontextfreie Sprachen sind wichtig für geklammerte Sprachstrukturen, insbesondere in Programmiersprachen. Wir haben z.B. schon gesehen (siehe Beispiel 2.1.10 auf S. 37), dass die kontextfreie Grammatik $G = (V, \Sigma, P, S)$ mit $V = \{S\}$, $\Sigma = \{(,), +, -, *, /, a\}$, und

$$P = \{S \to a \,|\, (S + S) \,|\, (S - S) \,|\, (S * S) \,|\, (S/S)\}$$

genau alle korrekt geklammerten Ausdrücke mit der Programmvariablen a erzeugt (siehe Beispiel 2.1.1 auf Seite 33). Mit regulären Sprachen gelingt dies nicht, denn wir wissen bereits, dass ein endlicher Automat z.B. die öffnenden und schließenden Klammern in einem Ausdruck nicht mitzählen kann. Er kann deshalb nicht sicherstellen, dass deren Anzahl jeweils gleich ist.

Um sich von der „Leistungsfähigkeit“ von kontextfreien Grammatiken zu überzeugen, werden wir die obige Beispielgrammatik noch ein wenig erweitern.

3.1.1 Beispiel Wir benennen in den obigen Regeln die Variable S in E (für engl. „*expression*“) um, d.h. es gilt nun

$$P = \{E \to a \,|\, (E + E) \,|\, (E - E) \,|\, (E * E) \,|\, (E/E)\} \ .$$

Anschließend fügen wir die folgenden Symbole zu Σ hinzu:

$$=, <, >, \leq, \geq$$

Aus einer neuen Variablen C (für engl. „*comparison*“) können wir dann mit den zusätzlichen Regeln

$$C \to E == E \,|\, E < E \,|\, E > E \,|\, E \leq E \,|\, E \geq E$$

Vergleiche wie z.B. $(a + a) < (a * a)$ oder $a \geq (a * a)/(a + (a/a))$ ableiten.

Die neue Startvariable S soll nun Anweisungen erzeugen können (passenderweise kann man S auch als Abkürzung für engl. „*statement*“ deuten). Wir ergänzen dazu das Terminalalphabet Σ nochmals um diese Symbole:

$$\{, \}, \texttt{do}, \texttt{for}, \texttt{if}, \texttt{then}, \texttt{to}, \texttt{;}$$

(Es handelt sich hierbei wirklich um einzelne Symbole, und nicht etwa z.B. bei `then` um die Konkatenation von vier Terminalen!). Aus den Regeln

$$S \to a = E; \,|\, SS \,|\, \{S\} \,|\, \texttt{if}\ C\ \texttt{then}\ S \,|\, \texttt{for}\ a = E\ \texttt{to}\ E\ \texttt{do}\ S$$

lassen sich dann schon „richtige“ Programme ableiten, z.B. (auf mehrere Zeilen aufgeteilt und passend eingerückt):

$a = (a + a);$
`for` $a = (a + a)$ `to` $(a * a)$ `do` $\{$
  `if` $(a + a) < (a * a)$ `then` $\{$
    $a = (a + a);$
    $a = a - (a * a);$
  $\}$
$\}$

Mit der gleichen Methode und weiteren Regeln kann man die komplette Syntax moderner Programmiersprachen wie z.B. Java zusammen stellen. Wenn also in einer Entwicklungsumgebung überprüft werden soll, ob der Text in einem Editor ein gültiges Java–Programm darstellt, so muss im Endeffekt getestet werden, ob eine zugehörige kontextfreie Grammatik für Java–Programme diesen Text erzeugen kann. Es ist also erneut das Wortproblem zu lösen ($x \in L$?), diesmal jedoch für kontextfreie Sprachen. Wir werden darauf im übernächsten Abschnitt zurück kommen.

Einer Ableitung eines Wortes x mit einer kontextfreien Grammatik kann man einen *Syntaxbaum* (oder auch *Ableitungsbaum*) zuordnen. Dieser zeigt graphisch die Struktur der Ableitung auf. Wir verzichten dabei auf eine formale Definition (siehe z.B. [13], S. 88) und betrachten stattdessen das folgende Beispiel.

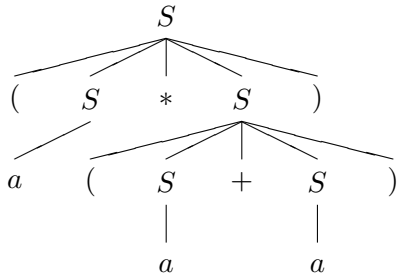
3.1.2 Beispiel Wir beziehen uns erneut auf die Grammatik $G = (V, \Sigma, P, S)$ aus dem Beispiel 2.1.10 auf S. 37, d.h. es gilt $V = \{S\}$, $\Sigma = \{(,), +, -, *, /, a\}$, und

$$P = \{S \to a \,|\, (S + S) \,|\, (S - S) \,|\, (S * S) \,|\, (S/S)\} \ .$$

Man betrachte nun die Ableitung

$$S \Rightarrow (S * S) \Rightarrow (S * (S + S)) \Rightarrow (a * (S + S)) \Rightarrow (a * (a + S)) \Rightarrow (a * (a + a))$$

Zu dieser Ableitung lässt sich folgender Syntaxbaum erstellen:



Jeder Ableitung kann eindeutig ein Ableitungsbaum zugeordnet werden. Umgekehrt ist dies jedoch nicht richtig, denn im obigen Syntaxbaum ist z.B. nicht ersichtlich, ob zuerst das erste vorkommende S in $(S * S)$ zu dem Terminal a abgeleitet wurde, oder ob zuerst das zweite S durch $(S + S)$ ersetzt wurde. Die Ableitung

$$S \Rightarrow (S * S) \Rightarrow (a * S) \Rightarrow (a * (S + S)) \Rightarrow (a * (a + S)) \Rightarrow (a * (a + a)) \ .$$

würde also den gleichen Ableitungsbaum erzeugen. Durch die Festlegung, in jeder Satzform immer die am weitesten links vorkommende Variable zu ersetzen, wird dieses Problem jedoch beseitigt. Man spricht dann auch von einer *Linksableitung*. Die zuletzt genannte Ableitung ist also eine Linksableitung, ebenso die folgende:

$$\begin{aligned} S \Rightarrow (S * S) \Rightarrow ((S + S) * S) \Rightarrow ((a + S) * S) \Rightarrow ((a + a) * S) \\ \Rightarrow ((a + a) * (S + S)) \Rightarrow ((a + a) * (a + S)) \Rightarrow ((a + a) * (a + a)) \ . \end{aligned}$$

3.1.3 Definition Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ heißt *eindeutig*, wenn es zu jedem $A \in V$ und $\alpha \in (V \cup \Sigma)^+$ mit $A \Rightarrow_G^* \alpha$ genau einen Ableitungsbaum (bzw. genau eine Linksableitung) von A nach α gibt. Sonst heißt G *mehrdeutig*.

3.1.4 Beispiel Sei $G = (V, \Sigma, P, S)$ mit $V = \{S\}$, $\Sigma = \{a\}$, und

$$P = \{S \to a \,|\, aa \,|\, aS\} \ .$$

Offenbar gilt $L(G) = \{a\}^+$. G ist mehrdeutig, denn für das Wort aa existieren z.B. zwei verschiedene Ableitungsbäume, die zu zwei unterschiedlichen Linksableitungen korrespondieren, nämlich zum einen einfach

$$S \Rightarrow aa$$

und zum anderen

$$S \Rightarrow aS \Rightarrow aa \ .$$

Man kann die Mehrdeutigkeit von G leicht entfernen, indem man die überflüssige Regel $S \to aa$ eleminiert. So einfach wie hier gelingt dies jedoch nicht immer.

3.1.5 Definition Eine kontextfreie Sprache L heißt *inhärent mehrdeutig*, wenn jede kontextfreie Grammatik G mit $L = L(G)$ mehrdeutig ist.

3.1.6 Beispiel Die Sprache

$$L := \{a^i b^j c^k \mid i, j, k \in \mathbb{N} \wedge (i = j \vee j = k)\}$$

ist zwar kontextfrei, aber inhärent mehrdeutig. Der Beweis hierfür ist zwar relativ einfach, aber auch recht lang, so dass wir aus Zeitgründen darauf verzichten. Interessierten Lesern sei die vorzügliche Darstellung in [13] ab Seite 106 empfohlen.

Bei mehrdeutigen Grammatiken kann man nicht zurückverfolgen, wie ein abgeleiteter Satz genau zustande gekommen ist. (Diesen unerwünschten Effekt kann man auch im täglichen Sprachgebrauch beobachten, z.B. bei dem Satz „Der Junge berührte das Mädchen mit dem Handschuh“.). Eindeutige kontextfreie Grammatiken sind für die Informatik daher von großem Interesse.

In der Literatur werden kontextfreie Grammatiken zuweilen in einer kompakten Notation aufgeschreiben, der sog. *Backus*[1]*–Naur*[2]*–Form* (*BNF*). Wir selbst werden diese Notation aber nicht benutzen.

3.2. Kellerautomaten

Unser erstes Ziel ist jetzt, analog zu den endlichen Automaten für reguläre Sprachen eine weitere „Maschinenform“ zu finden, die in der Lage ist, kontextfreie Sprachen zu erkennen. Rein intuitiv ist es offensichtlich, dass diese Maschine in der Lage sein muss, anders als die endlichen Automaten sich Gegebenheiten nicht nur anhand von Zuständen zu merken.

Sei zum Beispiel $L = \{({}^i a)^i \mid i \geq 1\}$ die kontextfreie Sprache aller vollständig und korrekt geklammerten Ausdrücke mit einmaligem Vorkommen der Variablen a, also z.B. (a) oder $((((a))))$. Ein endlicher Automat ist mit dem Erkennen von L überfordert, da er sich anhand seiner endlich vielen Zustände nur eine bestimmte Maximalanzahl von offenen Klammern merken kann. Die Anzahl der Klammern, die geöffnet werden dürfen, ist hier jedoch nach oben hin nicht beschränkt.

Bei dem neuen Maschinenmodell handelt es sich um den sog. *Kellerautomat*. Kellerautomaten erweitern NEAs so, dass genau die kontextfreien Sprachen erkannt werden können.

Beim Kellerautomaten wird der NEA um einen *Kellerspeicher* ergänzt, auf den nach dem *LIFO–Prinzip* (last–in–first–out) zugegriffen werden kann, also ein *Stapel*, *Keller*

[1] John W. Backus, *1924 Philadelphia, †2007 Ashland, Oregon, US–amerikanischer Informatiker, leitender Entwickler der ersten Programmiersprache *Fortran*.

[2] Peter Naur, *1928 Frederiksberg bei Kopenhagen, dänischer Informatiker, 1969–1998 Professor für Infomatik in Kopenhagen.

oder (engl.) *Stack*. Bei einem solchen Kellerspeicher ist immer nur das oberste Element zum Lesen oder für eine Veränderung verfügbar. Um an darunter liegende Elemente heranzukommen, müssen zunächst alle darüber liegenden Elemente entfernt werden.

Aktionen des Kellerautomaten hängen ab von

- dem aktuellen Zustand,
- dem aktuell gelesenen Eingabezeichen, und
- dem obersten Kellerzeichen.

Pro Rechenschritt ist beim Kellerautomaten

- eine Zustandsänderung sowie
- eine Änderung des Kellers

möglich.

3.2.1 Definition Ein *nichtdeterministischer Kellerautomat* (*NKA*) M wird beschrieben durch ein 6–Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$, wobei

- Z eine endliche Zustandsmenge,
- Σ ein (wie immer endliches) Eingabealphabet,
- Γ ein (endliches) Kelleralphabet,
- $z_0 \in Z$ den Startzustand,
- $\# \in \Gamma$ das unterste Kellerzeichen, sowie
- $\delta : Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}_e(Z \times \Gamma^*)$ die Zustandsübergangsfunktion ist, wobei $\mathcal{P}_e(X)$ die Menge der endlichen Teilmengen von einer Menge X bezeichnet.

Die auf den ersten Blick recht kryptisch wirkende Übergangsfunktion arbeitet wie folgt:

1. Fall: $(z', B_1 B_2 \dots B_k) \in \delta(z, a, A)$

Interpretation: Befindet sich M im Zustand z und wird das Eingabezeichen a gelesen, während zugleich A oberstes Kellerzeichen ist, kann M in einem Schritt in den Zustand z' übergehen und das oberste Kellerzeichen A durch $B_1 B_2 \dots B_k$ ersetzen, und zwar so, dass dann B_1 das oberste Kellerzeichen ist. Ist $k = 0$, so wird A ohne Ersatz aus dem Stapel entfernt. Ist $k = 2$ und $B_1 B_2 = BA$, so wird im Endeffekt nur ein weiteres Zeichen (nämlich das B) auf den Stapel gelegt.

2. Fall: $(z', B_1 B_2 \dots B_k) \in \delta(z, \varepsilon, A)$

Interpretation: Hier handelt es sich um einen sogenannten *spontanen Übergang*, der nicht von einem Eingabezeichen gesteuert wird. M kann (ohne eine Eingabe zu lesen) in den neuen Zustand z' übergehen und das oberste Kellerzeichen A durch $B_1 B_2 \dots B_k$ ersetzen.

Wie bei einem NEA ist die Übergangsfunktion eines NKA total, d.h. für alle möglichen Argumente definiert. Allerdings gibt man bei der Beschreibung eines NKA üblicherweise nur die „wirklichen“ Übergänge an, für die der Funktionswert also nicht die leere Menge

ist.

Der NKA terminiert bei einem leeren Keller, da dann kein weiterer Übergang mehr möglich ist. (Denn im Gegensatz zur Eingabe *muss* immer ein Zeichen vom Keller gelesen werden.) Ist zudem zu diesem Zeitpunkt die Eingabe bereits komplett gelesen, so wird sie von dem NKA akzeptiert. Ansonsten verwirft der NKA.

3.2.2 Definition Eine *Konfiguration* (d.h. eine Art „Momentaufnahme“) eines NKA $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ ist gegeben durch ein Tripel $k = (z, x, w) \in Z \times \Sigma^* \times \Gamma^*$, wobei

- z den aktuellen Zustand,
- x den noch zu lesenden Teil des Eingabewortes und
- w den aktuellen Kellerinhalt

darstellt. (Auf den bereits verarbeiteten Teil des Eingabewortes kann nicht mehr zugegriffen werden. Deshalb ist er für die weitere Verarbeitung ohne Bedeutung und wird daher auch nicht in einer Konfiguration festgehalten.)

Auf der Menge $Z \times \Sigma^* \times \Gamma^*$ aller Konfigurationen definieren wir eine Relation $\vdash_M$ (sprich: „geht über nach“), so dass $k \vdash_M k'$ genau dann gilt, falls k' aus k durch einen Rechenschritt hervorgehen kann. Es gilt also

$$(z, a_1 \dots a_n, A_1 \dots A_m) \vdash_M (z', a_2 \dots a_n, B_1 \dots B_k A_2 \dots A_m)$$

falls $(z', B_1 \dots B_k)$ in $\delta(z, a_1, A_1)$ enthalten ist. Ebenso gilt

$$(z, a_1 \dots a_n, A_1 \dots A_m) \vdash_M (z', a_1 \dots a_n, B_1 \dots B_k A_2 \dots A_m)$$

falls $(z', B_1 \dots B_k)$ in $\delta(z, \varepsilon, A_1)$ enthalten ist. Ferner sei $\vdash_M^*$ die *reflexiv transitive Hülle* von $\vdash_M$, d.h. $k \vdash_M^* k'$ gilt genau dann, wenn M in endlich vielen Schritten von k nach k' übergehen kann. Wenn der NKA M aus dem Zusammenhang her klar ist, schreiben wir auch einfach $\vdash$ und $\vdash^*$ statt $\vdash_M$ und $\vdash_M^*$.

3.2.3 Definition Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ ein NKA. Die von M akzeptierte Sprache ist

$$N(M) := \{x \in \Sigma^* \mid \exists z \in Z\colon (z_0, x, \#) \vdash_M^* (z, \varepsilon, \varepsilon)\} \ .$$

Eine Konfiguration der Form $(z_0, x, \#)$ wird auch als *Startkonfiguration* (von dem Wort $x \in \Sigma^*$) bezeichnet.

Es sei hier bemerkt, dass unsere Definition der NKAs keine Endzustände verwendet. In der Literatur findet man häufig aber auch NKAs, die analog zu NEAs eine Menge $E \subseteq Z$ von Endzuständen auszeichnen. Hier wird dann verlangt, dass nach Verarbeitung aller Eingabesymbole ein Endzustand eingenommen werden muss. Der Keller muss hingegen nicht unbedingt leer sein. Die von einem solchen NKA M akzeptierte Sprache lautet demnach

$$L(M) := \{x \in \Sigma^* \mid \exists z \in E\colon \exists \gamma \in \Gamma^*\colon (z_0, x, \#) \vdash_M^* (z, \varepsilon, \gamma)\} \ .$$

Man kann aber zeigen, dass beide Konzepte exakt die gleiche Sprachklasse charakterisieren. Der interessierte Leser kann den Beweis z.B. unter [14] im Kapitel 6.2 nachlesen.

Bevor wir unseren ersten NKA angeben, vereinbaren wir noch folgende Vereinfachung der Schreibweise:

$$zaA \to z'\gamma \ :\Longleftrightarrow \ (z', \gamma) \in \delta(z, a, A) \ .$$

Diese Notation gilt auch für spontane Übergänge:

$$z\varepsilon A \to z'\gamma \ :\Longleftrightarrow \ (z', \gamma) \in \delta(z, \varepsilon, A) \ .$$

3.2.4 Beispiel Sei $L := \{a_1 a_2 \dots a_n \$ a_n \dots a_2 a_1 \mid a_i \in \{a, b\}, n \geq 1\}$. Beispielsweise sind $abb\$bba$, $bab\$bab$ und $a\$a$ in L enthalten, nicht aber $ab\$ab$ oder $baab$. Wir entwerfen schrittweise einen passenden NKA $M = (\{z_0, z_1\}, \{a, b, \$\}, \{\#, A, B\}, \delta, z_0, \#)$, der L akzeptiert.

Anfangs befindet sich M im Startzustand z_0. Bis zum Auftreten des Trennzeichens \$ erzeugt M für jedes a ein A, das auf dem Keller abgelegt wird. Ebenso wird für jedes b ein B auf dem Keller abgelegt. Ferner wird sichergestellt, dass sich das Startzeichen # des Kellers immer ganz oben befindet (es „schwimmt“ sozusagen wie eine Schaumstoffkugel auf den gespeicherten A's und B's):

$$z_0 a\# \to z_1 \# A, \quad z_0 b\# \to z_1 \# B, \quad z_1 a\# \to z_1 \# A, \quad z_1 b\# \to z_1 \# B$$

Falls M z.B. das Wort $abb\$bba$ verarbeitet, so ergeben sich anfangs folgende Konfigurationsübergänge:

$$(z_0, abb\$bba, \#) \vdash (z_1, bb\$bba, \#A) \vdash (z_1, b\$bba, \#BA) \vdash (z_1, \$bba, \#BBA)$$

Unmittelbar vor der Bearbeitung des Trennzeichens \$ besteht der Kellerinhalt also im Wesentlichen aus dem Teilwort $a_1 \dots a_n$, aber „groß geschrieben“ und in umgekehrter Reihenfolge. Beachten Sie, dass beim Lesen des ersten Zeichens ein Zustandswechsel von z_0 nach z_1 stattfindet. Dies erzwingt, dass mindestens ein a oder b vor dem Trennzeichen \$ verarbeitet werden muss. Der folgende Übergang ist nämlich nur für z_1 definiert (zum Hintergrund dazu gleich mehr):

$$z_1\$\# \to z_1\varepsilon$$

Durch diesen Übergang kann M also das \$–Zeichen von der Eingabe und gleichzeitig das obere #–Zeichen von dem Keller entfernen. Für unser Beispielwort $abb\$bba$ ergibt sich also als nächster Übergang

$$\dots \vdash (z_1, \$bba, \#BBA) \vdash (z_1, bba, BBA)$$

Nun wird der Kellerinhalt mit der restlichen Eingabe verglichen und schrittweise wieder abgebaut. M akzeptiert, falls zum Schluss (d.h. nach dem Lesen der gesamten Eingabe) der gesamte Keller leer ist:

$$z_1 aA \to z_1\varepsilon, \quad z_1 bB \to z_1\varepsilon$$

Das Wort *abb$bba* wird demnach insgesamt mit diesen Konfigurationsübergängen akzeptiert:

$$(z_0, abb\$bba, \#) \vdash (z_1, bb\$bba, \#A) \vdash (z_1, b\$bba, \#BA) \vdash (z_1, \$bba, \#BBA)$$
$$\vdash (z_1, bba, BBA) \vdash (z_1, ba, BA) \vdash (z_1, a, A) \vdash (z_1, \varepsilon, \varepsilon)$$

Wie bereits erwähnt muss M anfangs einen Zustandswechsel von z_0 nach z_1 durchführen und deshalb zumindest ein a oder b verarbeiten. Also wird M wie gewünscht das Wort $\$ \notin L$ nicht akzeptieren (ohne den Zustandswechsel wäre dies möglich gewesen).

M ist von einer bestimmten Form, die in der nächsten Definition genauer beschrieben wird.

3.2.5 Definition Ein *deterministischer Kellerautomat* (*DKA*) M ist ein nichtdeterministischer Kellerautomat $(Z, \Sigma, \Gamma, \delta, z_0, \#)$, der für jede Konfiguration nur höchstens eine Nachfolgekonfiguration besitzt. Dies wird durch zwei Beschränkungen erzwungen:

- Für alle $z \in Z$, $a \in \Sigma \cup \{\varepsilon\}$ und $A \in \Gamma$ enthält $\delta(z, a, A)$ höchstens ein Element.
- Wenn $\delta(z, a, A)$ für ein $z \in Z$, $a \in \Sigma$ und $A \in \Gamma$ nicht leer ist, so gilt $\delta(z, \varepsilon, A) = \emptyset$.

Für $L = \{a_1 a_2 \dots a_n \$ a_n \dots a_2 a_1 \mid a_i \in \{a, b\}, n \geq 1\}$ existiert also ein DKA M mit $N(M) = L$ (nämlich derjenige aus Beispiel 3.2.4). Eine Variante von L, nämlich

$$L' := \{a_1 a_2 \dots a_n a_n \dots a_2 a_1 \mid a_i \in \{a, b\}, n \geq 1\} \ ,$$

ist allerdings nicht mehr durch einen DKA realisierbar, obwohl es leicht ist, einen entsprechenden NKA anzugeben. Man braucht dazu lediglich im vorherigen Beispiel 3.2.4 die Regel

$$z_1\$\# \to z_1\varepsilon$$

durch

$$z_1\varepsilon\# \to z_1\varepsilon$$

zu ersetzen. Die neue Regel sorgt dafür, dass das oberste Kellersymbol # ersatzlos entfernt werden kann, ohne gleichzeitig das Trennsymbol $ von der Eingabe lesen zu müssen. Folglich erhält man einen NKA M' mit $N(M') = L'$. Allerdings ist M' jetzt nicht mehr deterministisch, da die beiden folgenden Übergänge die zweite Restriktion verletzen:

$$z_1 a\# \to z_1\#A, \quad z_1\varepsilon\# \to z_1\varepsilon \ .$$

Wenn M' also in einer konkreten Konfiguration das Zeichen a in der Eingabe und das #–Symbol auf dem Keller vorfindet, so *kann* der Automat das a verarbeiten und unter das # ein neues A auf dem Keller schieben — er muss aber nicht. Stattdessen kann M' auch das Eingabezeichen ignorieren und einfach das #–Symbol entfernen. Eine entsprechende Wahlfreiheit besteht auch für die beiden Übergänge

$$z_1 b\# \to z_1\#B, \quad z_1\varepsilon\# \to z_1\varepsilon \ .$$

Der Nichtdeterminismus wird benötigt, um die Wortmitte zu erraten. Man kann zeigen, dass L' von keinem DKA akzeptiert wird. Der Nichtdeterminismus führt hier also (im Gegensatz zu den endlichen Automaten) zu einer echten Erweiterung der Sprachklasse. Wir werden nachfolgend sehen, dass NKAs genau die kontextfreien Sprachen erkennen können. Umgekehrt bedeutet dies, dass die von DKAs erkannten Sprachen eine echte Teilmenge hiervon bilden. Die Gesamtheit aller DKAs, die mit Endzuständen akzeptieren, erkennen genau die Klasse der sog. *deterministisch kontextfreien Sprachen* (die Endzustände sind hierbei erforderlich, denn die Äquivalenz der Akzeptanz mit Endzuständen bzw. leerem Keller gilt bei DKAs nicht). Programmiersprachen wie z.B. C++ oder Java sind aber genau so konzipiert, dass sie sich deterministisch kontextfrei analysieren lassen.

Allgemein kommen beim *Compilerbau* sog. $LL(k)$– und $LR(k)$–*Parser* zum Einsatz. Hierbei kandelt es sich um eine Klasse von bestimmten DKAs, die nach gewissen Konstruktionsvorschriften erstellt werden. Diese DKAs können bereits alle deterministisch kontextfreien Sprachen erkennen. Wir können aus zeitlichen Gründen eine Vertiefung hier nicht weiter verfolgen. Interessierte Leser finden weiterführendes Material z.B. in [13] ab Seite 270 sowie eine leicht verständliche Zusammenfassung in [18].

3.2.6 Satz Für jede kontextfreie Sprache $L \subseteq \Sigma^*$ existiert ein NKA M mit $L = N(M)$.

Beweis: Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik für L. Wir geben einen NKA M an, der die Produktionen von P in einem Keller simuliert. Als Kelleralphabet wird $\Gamma := V \cup \Sigma$ und als unterstes Kellersymbol die Startvariable S gewählt. Außerdem hat M nur einen einzigen Zustand z, d.h. M ist von der Form $(\{z\}, \Sigma, V \cup \Sigma, \delta, z, S)$. Bzgl. der Übergangsfunktion δ definieren wir für alle $A \in V$ und $a \in \Sigma$:

$$\delta(z, \varepsilon, A) := \{(z, \alpha) \,|\, A \to \alpha \in P\}$$

sowie

$$\delta(z, a, a) := \{(z, \varepsilon)\} \ .$$

Wir machen uns nun an einem Beispiel klar, dass diese Konstruktion immer die von G generierte Sprache akzeptiert. Wir betrachten dazu nochmals die Sprache

$$L := \{a_1 a_2 \ldots a_n \$ a_n \ldots a_2 a_1 \,|\, a_i \in \{a, b\}, n \geq 1\} \ .$$

L wird von einer kontextfreien Grammatik G mit den Produktionen

$$S \to a\$a \,|\, b\$b \,|\, aSa \,|\, bSb$$

erzeugt. Der zugehörige NKA $M = (\{z\}, \{a, b, \$\}, \{a, b, \$, S\}, \delta, z, S)$ besitzt gemäß den obigen Vorschriften die Übergänge

$$\delta(z, \varepsilon, S) = \{(z, a\$a), (z, b\$b), (z, aSa), (z, bSb)\}$$

sowie

$$\delta(z, a, a) = \{(z, \varepsilon)\} \ , \quad \delta(z, b, b) = \{(z, \varepsilon)\} \quad \text{und} \quad \delta(z, \$, \$) = \{(z, \varepsilon)\} \ .$$

M sieht völlig anders aus als der NKA aus dem vorherigen Beispiel 3.2.4, obwohl beide die gleiche Sprache akzeptieren. Der neue NKA ist in der Lage, jede Linksableitung von G zu simulieren. Wenn wir z.B. das Wort $abb\$bba \in L(G)$ und die zugehörige Linksableitung

$$S \Rightarrow_G aSa \Rightarrow_G abSba \Rightarrow_G abb\$bba$$

betrachten, so sehen wir, dass der NKA M mit diesen Konfigurationsübergängen das Wort akzeptiert:

$$\begin{aligned}
&\vdash (z, abb\$bba, S) && \\
&\vdash (z, abb\$bba, aSa) && // \text{ benutzter Übergang: } z\varepsilon S \to zaSa \\
&\vdash (z, bb\$bba, Sa) && // \text{ benutzter Übergang: } zaa \to z\varepsilon \\
&\vdash (z, bb\$bba, bSba) && // \text{ benutzter Übergang: } z\varepsilon S \to zbSb \\
&\vdash (z, b\$bba, Sba) && // \text{ benutzter Übergang: } zbb \to z\varepsilon \\
&\vdash (z, b\$bba, b\$bba) && // \text{ benutzter Übergang: } z\varepsilon S \to zb\$b \\
&\vdash (z, \$bba, \$bba) && // \text{ benutzter Übergang: } zbb \to z\varepsilon \\
&\vdash (z, bba, bba) && // \text{ benutzter Übergang: } z\$\$ \to z\varepsilon \\
&\vdash (z, ba, ba) && // \text{ benutzter Übergang: } zbb \to z\varepsilon \\
&\vdash (z, a, a) && // \text{ benutzter Übergang: } zbb \to z\varepsilon \\
&\vdash (z, \varepsilon, \varepsilon)\ . && // \text{ benutzter Übergang: } zaa \to z\varepsilon
\end{aligned}$$

Wir erkennen nun das Arbeitsprinzip dieses NKAs. Immer wenn das oberste Kellerzeichen eine Variable von G ist, kann eine der für diese Variable existierenden Regeln angewendet werden, d.h. die Variable wird durch die zugehörige rechte Seite ersetzt. Ist das oberste Kellerzeichen ein Terminalsymbol und stimmt dieses mit dem aktuellen Eingabezeichen überein, so kann man beide Symbole „synchron verbrauchen“. Da man mit der Startvariable S am Kellerboden beginnt, bildet man also eine Linksableitung für das zu verarbeitende Eingabewort nach. Nur wenn am Ende gleichzeitig der Kellerinhalt und die komplette Eingabe verbraucht worden sind, akzeptiert der NKA gemäß unseren Richtlinien. Es ist klar, dass somit der NKA M genau die von G generierte Sprache akzeptiert, also $N(M) = L(G)$ gilt. □

Die Umkehrung von Satz 3.2.6 gilt ebenfalls:

3.2.7 Satz Aus jedem NKA lässt sich konstruktiv eine passende kontextfreie Grammatik gewinnen, die die vom NKA erkannte Sprache erzeugt (abgesehen von dem evtl. enthaltenen leeren Wort). Insbesondere können NKAs also nur kontextfreie Sprachen erkennen.

Beweis: Der Beweis ist von der Idee her einfach, aber relativ lang und etwas technisch, so dass wir auf eine genaue Ausführung hier verzichten (siehe z.B. [25] ab Seite 121). □

3.2.8 Korollar Zu jedem NKA M existiert ein NKA M' mit nur einem Zustand und $N(M) = N(M')$.

Beweis: Aus M kann man nach Satz 3.2.7 zunächst eine gleichwertige Grammatik G und daraus wiederum nach Satz 3.2.6 einen entsprechenden NKA M' erstellen. Gemäß der vorgestellten Konstruktion besitzt M' wie gewünscht nur einen Zustand. □

3.3. Der CYK–Algorithmus

Wir kommen jetzt, wie bereits angekündigt, auf das Wortproblem zurück. Gegeben sei also eine kontextfreie Grammatik G sowie ein Wort $x \in \Sigma^*$. Um zu prüfen, ob x aus G ableitbar ist, könnte man prinzipiell nach Satz 3.2.6 einen passenden NKA M konstruieren und anschließend testen, ob M das Wort x abzeptiert. Diesen Weg sollte man jedoch nur beschreiten, wenn man sogar einen DKA angeben kann (dies sind z.B. die bereits erwähnten *LR(k)–Parser*). In diesem Fall lässt sich nämlich zeigen, dass der Test wie bei einem DEA in nur $O(|x|)$ viel Zeit (also in optimaler linearer Komplexität) erledigt werden kann. Falls wir jedoch nur einen NKA M angeben können, so ist die Simulation von M im Allgemeinen mit exponentiell hohem Aufwand verbunden. Selbst wenn man z.B. bei jedem Schritt ein Zeichen von x liest und dabei nur je zwei nichtdeterministische Übergänge ausprobieren muss, endet man bereits mit einem Aufwand von $\Omega(2^{|x|})$.

Wir müssen uns also für den allgemeinen Fall nach einem schnelleren Verfahren umsehen. Der sogenannte *CYK–Algorithmus* ist ein solches Verfahren und erreicht eine Laufzeit von $O(|x|^3)$. Dafür muss die Grammatik jedoch vorab passend aufbereitet und in die sogenannte *Chomsky–Normalform* überführt werden.

3.3.1 Definition Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ liegt in *Chomsky–Normalform* (*CNF*) vor, falls alle Regeln wie folgt aussehen ($A, B, C \in V$, $a \in \Sigma$):

- $A \to BC$
- $A \to a$

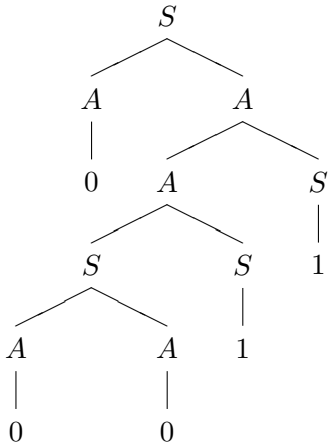
Beispielsweise ist die Grammatik $G = (\{S, A\}, \{0, 1\}, P, S)$ mit den Regeln

$$P = \{S \to AA \,|\, 1,\ A \to AS \,|\, SS \,|\, 0\}$$

in CNF.

Ein Ableitungsbaum einer CNF–Grammatik ist im Wesentlichen ein *Binärbaum*. Die meisten *inneren Knoten* (also Knoten mit Nachfolgern) haben also genau zwei Nachfolger. Beide Nachfolger sind stets Variablen, also selbst wieder innere Knoten. Wenn ein innerer Knoten nur einen Nachfolger aufweist, so stellt der Nachfolger ein Blatt dar, bei dem es sich um ein Terminalzeichen handelt.

Ein Ableitungsbaum für das Wort 00011 mit der oben erwähnten CNF–Grammatik G sieht beispielsweise wie folgt aus:



Um zu zeigen, dass man jede Grammatik G in CNF überführen kann, benötigen wir das folgende Lemma.

3.3.2 Lemma Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. Sei weiter $A \to B$ eine Produktion (eine sog. *Kettenregel*), und seien $B \to \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_r$ alle B–Produktionen in P. Geht nun $G' = (V, \Sigma, P', S)$ durch Hinzufügen der Produktionen

$$A \to \beta_1\,,\ \ A \to \beta_2\,,\ \ \ldots\,,\ \ A \to \beta_r$$

sowie durch Entfernen der Produktion $A \to B$ aus G hervor, so gilt $L(G) = L(G')$.

Beweis: Offenbar gilt $L(G') \subseteq L(G)$, denn ein Ableitungsschritt $A \Rightarrow_{G'} \beta_i$ kann in G durch $A \Rightarrow_G B \Rightarrow_G \beta_i$ simuliert werden. Falls umgekehrt $A \Rightarrow_G B$ in einer G–Ableitung benutzt wird, so muss die Variable B in einem späteren Schritt durch Benutzung einer Produktion der Form $B \to \beta_i$ ersetzt werden. Diese beiden Schritte können in G' durch den einen Schritt $A \Rightarrow_{G'} \beta_i$ zusammengefasst werden. □

3.3.3 Satz Zu jeder kontextfreien Sprache L existiert eine Grammatik G in CNF mit $L(G) = L$.

Beweis: Sei G eine kontextfreie Grammatik für L. Wir formen G schrittweise um, bis wir die gewünschte Darstellung in CNF erhalten. Als erstes kümmern wir uns um die Entfernung von Kettenregeln der Form $A \to B$ (wobei A und B Variablen sind).

- Wir entfernen zunächst alle *Zyklen*: falls $B_1, B_2, \ldots, B_k$ Variablen sind und Produktionen $B_1 \to B_2$, $B_2 \to B_3$, $\ldots$, $B_k \to B_1$ existieren, so sind offenbar alle Variablen gleichwertig. Wir ersetzen deshalb alle Vorkommen von $B_1, B_2, \ldots, B_k$ durch *eine* Variable. Offenbar bleibt die erzeugte Sprache von dieser Änderung unberührt.

- Sei n die Anzahl der verbliebenen Variablen. Wir benennen diese in $A_1, \ldots, A_n$ um und wählen die Umbenennung so, dass aus $A_i \to A_j \in P$ stets $i < j$ folgt (dies ist nach der vorangehenden Eliminierung aller Zyklen möglich).
- Wir gehen nun von hinten nach vorne vor und eliminieren für $k = n-1, n-2, \ldots, 1$ alle Regeln der Form $A_k \to A_{k'}$ $(k < k')$. Es seien dazu $A_{k'} \to \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_r$ alle Produktionen mit linker Seite $A_{k'}$. Dann ersetzt man die Regel $A_k \to A_{k'}$ durch $A_k \to \beta_1 \,|\, \beta_2 \,|\, \ldots \,|\, \beta_r$. Gemäß Lemma 3.3.2 hat dies auf die erzeugte Sprache keinen Einfluss, und es werden nach und nach alle Kettenregeln entfernt.

Ohne Kettenregeln sehen alle Produktionen der Grammatik $G = (V, \Sigma, P, S)$ wie folgt aus:

- $A \to a \;,\quad a \in \Sigma$
- $A \to \alpha \;,\quad \alpha \in (V \cup \Sigma)^+,\ |\alpha| \geq 2$

Für jedes $a \in \Sigma$ füge man nun eine neue Variable X_a zu V hinzu, sowie zu P die Regel $X_a \to a$. Ferner ersetzen wir jedes $a \in \Sigma$ auf der rechten Seite einer Produktion durch X_a (es sei denn, die Regel weist bereits die Form $A \to a$ auf). Nun haben alle Regeln eine der folgenden Formen:

- $A \to a \;,\quad a \in \Sigma$
- $A \to B_1 B_2 \ldots B_k \;,\quad k \geq 2$

mit geeigneten $A, B_i \in V$, $a \in \Sigma$. Die Regeln $A \to B_1 B_2 \ldots B_k$ mit $k > 2$ sind bislang noch nicht in CNF. Für jede solche Regel werden weitere Variablen $C_2, C_3, \ldots, C_{k-1}$ eingeführt; anschließend wird die Regel durch $k-1$ Produktionen

$$\begin{array}{rcl} A & \to & B_1 C_2 \\ C_2 & \to & B_2 C_3 \\ C_3 & \to & B_3 C_4 \\ & \vdots & \\ C_{k-2} & \to & B_{k-2} C_{k-1} \\ C_{k-1} & \to & B_{k-1} B_k \end{array}$$

ersetzt. Es ist leicht zu sehen, dass sich dabei die erzeugte Sprache nicht ändert. □

3.3.4 Beispiel Sei $G = (\{S, A, B, C, D, E\}, \{a, b, c\}, P, S)$ mit

$$P = \{S \to AB,\ A \to ab \,|\, aAB,\ B \to c \,|\, C,\ C \to D \,|\, E,\ D \to c \,|\, B,\ E \to c \,|\, cD\} \;.$$

Dann ist G zwar kontextfrei, aber (noch) nicht in CNF. Wir arbeiten systematisch die einzelnen Umformungsschritte aus dem Beweis von Satz 3.3.3 ab und beginnen mit der Entfernung aller Kettenregeln.

Zunächst fällt uns der Zyklus $B \to C$, $C \to D$ und $D \to B$ auf. Wir ersetzen deshalb alle Vorkommen von B, C und D durch *eine* Variable, z.B. durch B selbst. Wir erhalten

also

$$P = \{S \to AB,\ A \to ab \,|\, aAB,\ B \to c \,|\, B,\ B \to B \,|\, E,\ B \to c \,|\, B,\ E \to c \,|\, cB\}$$

und können die doppelte Regel für $B \to c$ sowie die offenbar überflüssige Produktion $B \to B$ weglassen. Zusammengefasst ergibt sich somit die Produktionsmenge

$$P = \{S \to AB,\ A \to ab \,|\, aAB,\ B \to c \,|\, E,\ E \to c \,|\, cB\} \ .$$

Nun gibt es nur noch eine Kettenregel, nämlich $B \to E$. Gemäß unserer Konstruktionsvorschrift ersetzen wir diese Regel durch $B \to c$ und $B \to cB$, da aus E nur die beiden rechten Seiten c und cB abgeleitet werden können. Die schon vorhandene Regel $B \to c$ brauchen wir natürlich nicht noch einmal aufzunehmen. Danach ist die Grammatik kettenregelfrei:

$$P = \{S \to AB,\ A \to ab \,|\, aAB,\ B \to c \,|\, cB,\ E \to c \,|\, cB\} \ .$$

Natürlich können wir nun auch noch alle Regeln für die Variable E weglassen, da E auf keiner rechten Seite einer Produktion mehr auftaucht:

$$P = \{S \to AB,\ A \to ab \,|\, aAB,\ B \to c \,|\, cB\} \ .$$

Jetzt kann die zu G äquivalente CNF–Version $G' = (V', \{a, b, c\}, P', S)$ erstellt werden:

- Konstruktion von V': $\quad V' = V \cup \{X_a, X_b, X_c\}$.
- Konstruktion von P': $\quad P' = P \cup \{X_a \to a,\ X_b \to b,\ X_c \to c\}$.
- Nun werden alle rechten Seiten von Regeln der Art $A \to \alpha$ mit $|\alpha| \geq 2$ so modifiziert, dass schließlich rechts nur noch Worte über V' stehen. Danach gilt:

  $$P' = \{S \to AB,\ A \to X_aX_b \,|\, X_aAB,\ B \to c \,|\, X_cB,\ X_a \to a,\ X_b \to b,\ X_c \to c\}$$

- Es gibt jetzt noch eine Regel der Form $A \to B_1B_2 \ldots B_k$ mit $k > 2$, nämlich $A \to X_aAB$. Diese wird durch die beiden Regeln $A \to X_aC$ und $C \to AB$ ersetzt.

Als Endresultat ergibt sich also die CNF–Grammatik

$$G' = (\{S, A, B, C, X_a, X_b, X_c\}, \{a, b, c\}, P', S)$$

mit

$$P' = \{S \to AB,\ A \to X_aX_b \,|\, X_aC,\ B \to c \,|\, X_cB,\ C \to AB,\ X_a \to a,\ X_b \to b,\ X_c \to c\}$$

als Produktionsmenge.

Die Chomsky–Normalform einer Grammatik ist Ausgangspunkt für den bereits erwähnten *CYK–Algorithmus* zur Lösung des Wortproblems („CYK“ aufgrund der drei Erfinder Cocke[1], Younger und Kasami[2]). Die Idee ist recht einfach:

[1] John Cocke, *1925 Charlotte, North Carolina, †2002, Valhalla, New York, US–amerikanischer Informatiker, der als Vater der modernen *RISC*–Rechnerarchitekturen gilt.

[2] Tadao Kasami, *1930 Kobe, Japan, †2007, japanischer Informatiker, Professor für Ingenieur– und Informationswissenschaften in Osaka und Hiroshima.

Ein Wort $x = a$ der Länge 1 kann aus einer Variablen A nur bei Existenz einer Produktion $A \to a$ abgeleitet werden. Ist dagegen $x = x_1 \dots x_n$ mit $n \geq 2$, so ist eine Ableitung $A \Rightarrow_G^* x$ genau dann möglich, falls eine Regel $A \to BC$ sowie eine bestimmte Grenze k mit $1 \leq k < n$ existiert, so dass $B \Rightarrow_G^* x_1 \dots x_k$ und $C \Rightarrow_G^* x_{k+1} \dots x_n$ gilt. Das Problem, ob sich x aus A ableiten lässt, wird damit auf zwei „kleinere“ Probleme (wegen der abnehmenden Wortlänge) reduziert. Der Algorithmus ist somit ein Paradebeispiel für die sog. *dynamische Programmierung*.

Der CYK–Algorithmus bestimmt für alle $1 \leq i \leq j \leq n$ die folgenden Mengen T_i^j:

$$T_i^j := \{X \in V \mid X \Rightarrow_G^* x_i \dots x_j\} \ .$$

Aus den obigen Erläuterungen ergibt sich dann direkt:

a) Für alle $1 \leq i \leq n$ gilt: $T_i^i = \{A \mid A \to x_i \in P\}$.

b) Für alle $1 \leq i < j \leq n$ gilt:

$$T_i^j = \{A \mid \exists k\colon\ i \leq k < j \ \wedge\ \exists B \in T_i^k\colon\ \exists C \in T_{k+1}^j\colon\ (A \to BC) \in P\} \ .$$

c) $x \in L(G) \iff S \in T_1^n$

Unter Verwendung dieser Eigenschaften berechnet der folgende Algorithmus CYK–Test nacheinander für $d = 0, 1, \dots, n-1$ alle T_i^j mit $j - i = d$.

3.3.5 Beispiel Betrachten Sie die CNF–Grammatik $G = (\{S, A, B, C\}, \{0, 1\}, P, S)$ mit folgenden Produktionen:

$$P = \{S \to AB \mid BC,\ A \to BA \mid 0,\ B \to CC \mid 1,\ C \to AB \mid 0\} \ .$$

Wir wollen wissen, ob $x = 110100 \in L(G)$ gilt. Die Länge dieses Wortes beträgt $n = 6$. Dementsprechend werden in den ersten drei Zeilen des CYK–Tests zunächst die Mengen $T_1^1, \dots, T_6^6$ bestimmt. Hierfür sind nur die Produktionen wichtig, die ein Terminalsymbol erzeugen, also

$$A \to 0 \ , \qquad B \to 1 \ , \qquad C \to 0 \ .$$

Es gilt z.B. $T_6^6 = \{A, C\}$, da man sowohl aus A als auch aus C das Terminalzeichen $x_6 = 0$ ableiten kann. Für die anderen Mengen gilt:

$$T_1^1 = \{B\} \ , \quad T_2^2 = \{B\} \ , \quad T_3^3 = \{A, C\} \ , \quad T_4^4 = \{B\} \ , \quad T_5^5 = \{A, C\} \ .$$

Die ermittelten T_i^i–Mengen sind Ausgangspunkt für eine dreiecksförmige Tabelle. Diese ist für die Indexwerte

$$i = 1, \dots, n = 6$$

und

$$d = 0, \dots, n - 1 = 5$$

mit der Nebenbedingung $i + d \leq n = 6$ ausgelegt. In jedem Feld mit den Koordinaten i und d wird nun der Inhalt der Menge T_i^{i+d} eingetragen.

Die oberste Zeile für $d = 0$ ist uns bereits bekannt:

```
Algorithmus CYK-Test

1    n := |x|; // x = x_1 x_2 ... x_n
2    for i = 1 to n do {
3       T_i^i := {A | A → x_i ∈ P};
4    }
5    for d = 1 to n − 1 do {
6       for i = 1 to n − d do {
7          T_i^{i+d} := ∅;
8          for k = i to i + d − 1 do {
9             T_i^{i+d} := T_i^{i+d} ∪ {A | ∃B ∈ T_i^k: ∃C ∈ T_{k+1}^{i+d}: (A → BC) ∈ P};
10         }
11      }
12   }
13   if S ∈ T_1^n then {
14      Output("x ∈ L(G)");
15   } else {
16      Output("x ∉ L(G)");
17   }
```

$i \rightarrow$	1	2	3	4	5	6
$d = 0$	B	B	A, C	B	A, C	A, C
$d = 1$						
$d = 2$						
$d = 3$						
$d = 4$						
$d = 5$						

Nun werden nach und nach alle Zeilen $d = 1, 2, \ldots, n-1 = 5$ ausgefüllt. Hierfür sind jetzt die anderen Produktionen entscheidend, also diejenigen, die aus einem Nichtterminal zwei erzeugen:

$$S \rightarrow AB \,|\, BC \;, \qquad A \rightarrow BA \;, \qquad B \rightarrow CC \;, \qquad C \rightarrow AB \;.$$

Es soll nun demonstriert werden, wie man jeweils den Inhalt eines Feldes ermitteln kann.

In der nachfolgenden Tabelle wurden mit dieser Technik bereits die nächsten beiden Zeilen berechnet. Wir sind jetzt an dem Inhalt des angekreuzten Feldes für $T_i^{i+d} = T_1^4$ interessiert:

$i \rightarrow$	1	2	3	4	5	6
$d = 0$	B	B	A, C	B	A, C	A, C
$d = 1$	$\emptyset$	A, S	C, S	A, S	B	
$d = 2$	A	C, S	B	$\emptyset$		
$d = 3$	$\times$					
$d = 4$						
$d = 5$						

Gemäß der „Formel“ für T_1^4 gilt:

$$
\begin{aligned}
& T_1^4 \\
= \; & \{A \,|\, \exists k\colon\, 1 \leq k < 4 \,\wedge\, \exists B \in T_1^k\colon\, \exists C \in T_{k+1}^4\colon\, (A \rightarrow BC) \in P\} \\
& \\
& = \; \{A \,|\, \exists B \in T_1^1\colon\, \exists C \in T_2^4\colon\, (A \rightarrow BC) \in P\} \\
& \cup \; \{A \,|\, \exists B \in T_1^2\colon\, \exists C \in T_3^4\colon\, (A \rightarrow BC) \in P\} \\
& \cup \; \{A \,|\, \exists B \in T_1^3\colon\, \exists C \in T_4^4\colon\, (A \rightarrow BC) \in P\} \;\; .
\end{aligned}
$$

Wir betrachten nun zunächst die erste Teilmenge

$$\{A \,|\, \exists B \in T_1^1\colon\, \exists C \in T_2^4\colon\, (A \rightarrow BC) \in P\} \;\; .$$

Es gilt $T_1^1 = \{B\}$ und $T_2^4 = \{C, S\}$. Wegen $B \in T_1^1$ und $C \in T_2^4$ sowie $(S \rightarrow BC) \in P$ gehört also S zu der ersten Teilmenge und damit auch später zu T_1^4. Die zweite Teilmenge

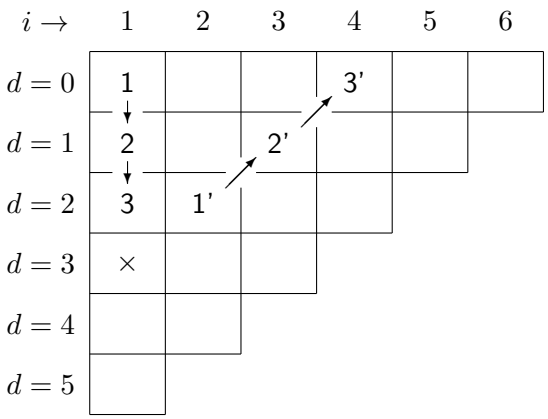
$$\{A \,|\, \exists B \in T_1^2\colon\, \exists C \in T_3^4\colon\, (A \rightarrow BC) \in P\}$$

bleibt dagegen leer, denn wegen $T_1^2 = \emptyset$ ist es unmöglich, irgendeine Variable aus dieser Menge zu wählen. Die dritte Teilmenge

$$\{A \,|\, \exists B \in T_1^3\colon\, \exists C \in T_4^4\colon\, (A \rightarrow BC) \in P\}$$

enthält schließlich wegen $T_1^3 = \{A\}$ und $T_4^4 = \{B\}$ sowie den Regeln $C \rightarrow AB$ und $S \rightarrow AB$ die beiden Variablen C und S. Also kommt noch S zu der Menge T_1^4 hinzu, die damit komplett ist. Insgesamt gilt also $T_1^4 = \{C, S\}$.

Bzgl. der jeweils benötigten „Vorgängermengen“ erkennt man folgendes Schema:



Für $k = 1, 2, 3$ enthalten die Felder mit den Zahlen k und k' die Mengen T_1^k und T_{k+1}^4, die jeweils zur Bildung einer zugehörigen Teilmenge von T_1^4 betrachtet werden müssen. Man kann sich allgemein zur Bestimmung der passenden Felder zwei Spielsteine vorstellen, wobei der erste anfangs ganz nach oben in die Spalte des zu bestimmenden Feldes gelegt wird. Der zweite Stein wird dagegen unmittelbar rechts oberhalb des zu bestimmenden Feldes platziert. Nun liegen die Steine auf dem ersten zugehörigen Feldpaar. Alle anderen Paare ergeben sich, indem man den ersten Stein jeweils um ein Feld nach unten und den zweiten Stein diagonal um ein Feld nach rechts oben verschiebt (solange, bis der erste Stein auf das zu bestimmende Feld trifft und der zweite Stein aus der Tabelle herausfällt).

Vollständig ausgefüllt führt unser Beispiel zu dieser Tabelle:

$i \to$	1	2	3	4	5	6
$d = 0$	B	B	A, C	B	A, C	A, C
$d = 1$	$\emptyset$	A, S	C, S	A, S	B	
$d = 2$	A	C, S	B	$\emptyset$		
$d = 3$	C, S	B	A, S			
$d = 4$	B	A, S				
$d = 5$	A, S					

Das unterste Feld entspricht der entscheidenden Menge T_1^6 (bzw. allgemein der Menge T_1^n). In diesem Beispiel enthält T_1^6 die Startvariable S, d.h. 110100 ist aus G ableitbar.

3.3.6 Satz Die Zeitkomplexität des CYK–Tests beträgt $O(n^3)$, wobei n die Wortlänge von x bezeichnet.

Beweis: Bei jeder der vier `for`–Schleifen bewegt sich der zugehörige Index zwischen 1 und n. Folglich wird die äußere der drei ineinander geschachtelten `for`–Schleifen in den Zeilen 5–12 höchstens $O(n)$–mal, die mittlere höchstens $O(n^2)$–mal, und die innerste höchstens $O(n^3)$–mal durchlaufen. Auch die erste `for`–Schleife wird nur n–mal ausgeführt. Die eigentlichen Anweisungen in den restlichen Zeilen sind von ihrer Implementierung her nur von der Grammatikgröße (z.B. der Anzahl der Variablen) abhängig. Diese Größe ist unabhängig von der Eingabelänge des Wortes und somit hier eine Konstante. Also beträgt die Gesamtkomplexität $O(n) + O(n^2) + O(n^3) = O(n^3)$. □

Neben der CNF existieren noch weitere Normalformen für kontextfreie Grammatiken:

3.3.7 Definition Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ liegt in *Greibach*[1]*–Normalform* (*GNF*) vor, wenn alle Regeln von der Gestalt $A \to aB_1B_2 \dots B_k$ mit $k \geq 0$ sind. Hierbei sind wieder $A, B_i \in V$ sowie $a \in \Sigma$.

Die GNF ist eine natürliche Verallgemeinerung von regulären Grammatiken (dort gilt zusätzlich die Restriktion $k = 0$ oder $k = 1$). Insbesondere wird bei jedem Ableitungschritt ein Terminalsymbol erzeugt.

3.3.8 Satz Zu jeder kontextfreien Sprache L gibt es eine Grammatik G in GNF mit $L(G) = L$.

Beweis: Mit ähnlichen Methoden wie zu Anfang dieses Abschnitts und wie beim Beweis von Satz 3.3.3 kann man schrittweise eine passende Grammatik für L in GNF bringen. Siehe dazu [13] ab Seite 101. □

3.4. Das Pumping–Lemma für kontextfreie Sprachen

Im folgenden werden wir das Pumping–Lemma für reguläre Sprachen auf kontextfreie Sprachen erweitern. Analog zum Pumping–Lemma für reguläre Sprachen ist das Pumping–Lemma für kontextfreie Sprachen das Haupthilfsmittel zum Nachweis, dass eine Sprache nicht kontextfrei ist. Das Pumping–Lemma ist wieder nur ein *notwendiges* Kriterium für die Kontextfreiheit einer Sprache. Es kann deshalb sein, dass eine Sprache nicht kontextfrei ist, das Pumping–Lemma jedoch dennoch gilt.

[1] Sheila A. Greibach, *1939 New York, US–amerikanische Informatikerin, seit 1970 Professorin für Infomatik in Los Angeles.

3.4.1 Satz (*Pumping–Lemma für kontextfreie Sprachen*) Ist L kontextfrei, so existiert ein $n \in \mathbb{N}$ mit den folgenden Eigenschaften:

Ist $z \in L$ mit $|z| \geq n$, so ist z zerlegbar in $z = uvwxy$ mit

1) $|vx| \geq 1$
2) $|vwx| \leq n$
3) $\forall i \geq 0\colon\ uv^iwx^iy \in L$

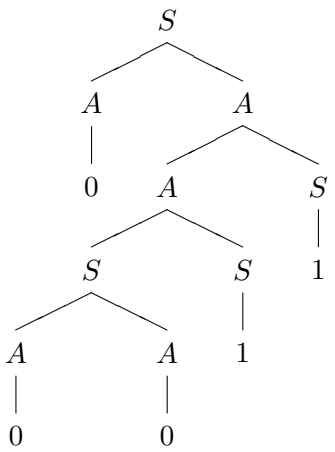
Wenn also L kontextfrei ist, so gilt formal die folgende Aussage:

$$\exists n \in \mathbb{N}\colon\ \forall z \in L\colon\ (|z| \geq n) \Longrightarrow (\exists u, v, w, x, y\colon\ z = uvwxy\ \wedge\ 1.\ \wedge\ 2.\ \wedge\ 3.)$$

Beweis: Sei G eine kontextfreie Grammatik, die o.B.d.A in CNF vorliegt. Wir wissen bereits, dass jede Ableitung eines Wortes $z \in L(G)$ als Syntaxbaum die Form eines Binärbaums hat und betrachten dazu nochmals die Grammatik $G = (\{S, A\}, \{0, 1\}, P, S)$ aus der Definition 3.3.1 mit den Regeln

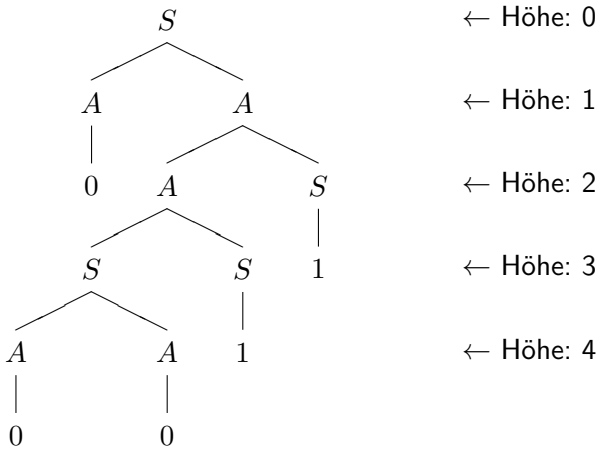
$$P = \{S \to AA \,|\, 1,\ A \to AS \,|\, SS \,|\, 0\}\ .$$

Für das Wort $z = 00011$ ergab sich dieser Ableitungsbaum:



Am „unteren Rand“ eines solchen Baums hängen die $|z|$ vielen Terminalzeichen als $|z|$ viele einzelne *Blätter* (zur Erinnerung: Blätter waren Knoten ohne Nachfolger).

Wir ordnen jetzt jedem inneren Knoten des Ableitungsbaums eine *Höhe* zu, nämlich die Länge des eindeutigen Pfads von dem Knoten bis zum *Wurzelknoten* an der oberen Spitze des Baums (mit dem Startsymbol S). Für unsere Beispielableitung ergeben sich beispielsweise folgende Höhenangaben (die Höhe 5 wurde nicht markiert, da dort keine inneren Knoten mehr stehen):



Als *Höhe* eines solchen Baums bezeichnen wir die maximale Höhe eines inneren Knotens. Der obige Baum hat also die Höhe 4.

Natürlich gibt es nur einen Knoten der Höhe null, nämlich den Wurzelknoten selbst. Weiter gibt es höchstens

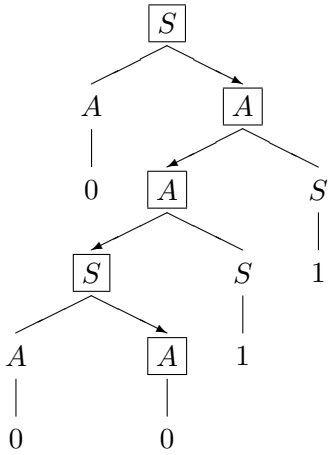
- ... zwei Knoten der Höhe eins,
- ... vier Knoten der Höhe zwei,
- ... usw.,

also allgemein höchstens 2^k Knoten der Höhe k. Ein Baum der Höhe k kann demnach maximal 2^k viele Blätter haben (also höchstens ein 2^k langes Wort ableiten). Diese Maximallänge erreicht man, indem man jede Variable bei einem Knoten der Höhe $h < k$ auf zwei weitere Variablen bei Knoten der Höhe $h+1$ ableitet und an jeden Knoten der Höhe k ein Terminalzeichen anhängt. Der obige Baum ist in diesem Sinne „verschwenderisch“ und erzeugt trotz seine Höhe von vier nur ein Wort der Länge 5, obwohl eine Länge von $2^4 = 16$ möglich gewesen wäre.

Man kann auch umgekehrt argumentieren, dass jeder CNF–Ableitungsbaum für ein mehr als 2^{k-1} langes Wort mindestens die Höhe k besitzen muss. Wenn nun $k := |V|$ die Anzahl der Variablen in G ist und wir $n := 2^k$ und damit $n > 2^{k-1}$ wählen, muss es im Ableitungsbaum eines beliebigen Wortes $z \in L(G)$ mit $|z| \geq n$ demnach mindestens einen inneren Knoten mit Mindesthöhe $k = |V|$ geben. In unserem Beispiel gilt $k = 2$ und $2^2 = 4$, und das abgeleitete Wort 00011 der Länge $5 > 4$ erzwingt die Existenz von zumindest einem Knoten der Höhe zwei (tatsächlich gibt es zwei Knoten der Höhe zwei, zwei weitere der Höhe drei, und sogar zwei der Höhe vier).

Man betrachte nun einen Pfad, der von dem Wurzelknoten bis zu einem inneren Knoten mit maximaler Höhe (also mindestens $|V|$) führt. Er beginnt also immer mit dem Startsymbol S, und für den Endpunkt haben wir in unserem Beispiel die Wahl zwischem dem linken und rechten unteren A. Wir entscheiden uns z.B. für das rechte Vorkommen. Der zugehörige Weg ist mit Pfeilen versehen, und die Variablen auf diesem Weg sind

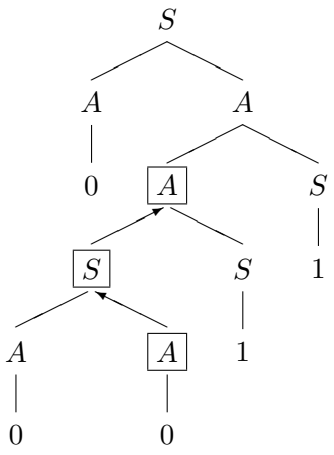
eingerahmt:



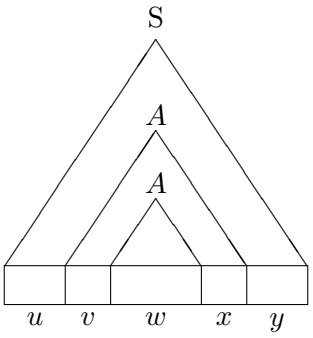
Wichtig ist nun die folgende Beobachtung. Da der Pfad zu einem Knoten mit Mindesthöhe k verläuft, befinden sich auf dem Pfad mindestens $k + 1$ Variablen. Da G aber nur $k = |V|$ verschiedene Variablen besitzt, muss mindestens eine der Variablen doppelt vorkommen (auf unserem Beispielpfad kommen sogar beide Variablen doppelt vor).

Wir wollen ein solches Doppelvorkommen nun möglichst weit unten im Baum entdecken. Wir arbeiten uns daher auf dem Pfad vom unteren Ende aus in umgekehrter Richtung wieder nach oben vor und merken uns dabei alle Variablen, die auf dem Pfad liegen. Treffen wir eine Variable zum zweiten Mal an, so ist das zweite Vorkommen höchstens k Schritte von dem unteren Ende entfernt, denn bis dahin wurden mindestens $k + 1$ Variablen betrachtet, so dass eine Variable doppelt vorkommen muss.

Hier also würden wir beim Zurückgehen auf halber Höhe stoppen, da dann das A zum zweiten Mal auftaucht:



Man betrachte nun die Teilbäume, die unter den beiden A's hängen. Sie induzieren eine Zerlegung des Wortes $z = 00011$. Aus dem unteren A wird die mittlere Null abgeleitet, aus dem oberen A das Wort 001, und aus der Startvariablen S das gesamte Wort 00011. Im Allgemeinen ergeben sich damit die folgende Bruchstücke u, v, w, x, y, die zusammengesetzt das Wort $z = uvwxy$ ergeben:



Es gilt also $S \Rightarrow^* uAy$, $A \Rightarrow^* vAx$ sowie $A \Rightarrow^* w$. In unserem Beispiel ist $u = 0$, $v = 0$, $w = 0$, $x = 1$ und $y = 1$.

Das obere A steht in der „$A \to BC$"–Regeln–Umgebung, d.h. es korrespondiert zu einer Regel $A \to BC$. Nur eine der beiden Variablen B und C ist in den betrachteten Pfad involviert, die andere hat mit dem aus dem unteren A erzeugten Wort w nichts zu tun. Auf Grund der CNF–Restriktionen muss diese Variable auf wenigstens ein Terminalsymbol abgeleitet werden. Dieses Terminal befindet sich am Ende dann entweder im Teilwort v oder x. Daraus folgt $|vx| \geq 1$, womit die erste Bedingung des Pumping–Lemmas erfüllt ist.

Als nächstes stellen wir fest, dass der unter dem oberen A hängende Teilbaum gemäß unserer Konstruktion höchstens die Höhe k besitzt. Denn zum einen kann der von uns betrachtete Pfad bis zum doppelten Auftreten der Variable A nicht länger als k sein, und zum anderen haben wir mit einem Knoten auf maximaler Hähe gestartet, d.h. die evtl. links und/oder rechts vorhandenen Nachbarpfade in dem Teilbaum können nicht länger sein.

Wir haben bereits am Anfang des Beweises gesehen, dass ein Baum der Höhe k nur die Ableitung eines höchstens $2^k = n$ langen Wortes darstellen kann. Also erfüllt das vom oberen A letztendlich abgeleitete Teilwort vwx die Längenbeschränkung $|vwx| \leq n$. Damit gilt auch die zweite Bedingung des Pumping–Lemmas.

Aus den hergeleiteten Ableitungen

$$S \Rightarrow^* uAy \ , \quad A \Rightarrow^* vAx \qquad \text{und} \qquad A \Rightarrow^* w$$

können wir neben der „Originalableitung"

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy = uv^1wx^1y = z$$

auch andere Ableitungen zusammensetzen. Zum Beispiel können wir den inneren Teil weglassen, der zu der Ableitung $A \Rightarrow^* vAx$ gehört. Dann erhalten wir

$$S \Rightarrow^* uAy \Rightarrow^* uwy = uv^0wx^0y \ .$$

Oder wir verdoppeln den inneren Teil:

$$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* uvvwxxy = uv^2wx^2y \ .$$

Natürlich können wir den inneren Teil auch noch häufiger wiederholen. Insgesamt lassen sich also für alle $i \in \mathbb{N}_0$ Wörter der Form $uv^iwx^iy \in L$ erzeugen. Damit gilt auch die dritte Bedingung, und der Satz ist vollständig bewiesen. □

Mit dem Pumping–Lemma ist es z.B. möglich, die Nicht–Kontextfreiheit einer Sprache zu zeigen, die wir schon am Anfang des zweiten Kapitels betrachtet hatten (siehe Beispiele 2.1.12 auf Seite 38). Den dort versprochenen Beweis können wir jetzt nachholen.

3.4.2 Beispiel Sei $L := \{a^mb^mc^m \mid m \geq 1\}$. Angenommen, L wäre kontextfrei. Dann würde das Pumping–Lemma gelten. Also würde ein $n \in \mathbb{N}$ existieren, so dass jedes Wort $z = a^mb^mc^m \in L$ mit $|z| \geq n$ in eine Aufteilung $z = uvwxy$ zerlegbar wäre, die die Bedingungen 1. – 3. des Pumping–Lemmas erfüllt. Wir wählen nun $z := a^nb^nc^n$. Dann ist z wegen $|z| = 3n \geq n$ lang genug. Wegen 2. ($|vwx| \leq n$) kann vx nicht aus a's, b's *und* c's bestehen (hierfür müsste vwx mindestens die Länge $n + 2$ haben). Wegen 3. ($\forall i \geq 0 : uv^iwx^iy \in L$) müsste (mit $i = 0$) $uv^0wx^0y = uwy \in L$ gelten. Durch das Fehlen von vx wird wegen 1. ($|vx| \geq 1$) das ursprüngliche Wort kürzer, aber es werden nicht gleichzeitig a's, b's *und* c's entfernt. Also kann das restliche Wort uwy nicht von der Form $a^mb^mc^m$ sein. Das Pumping–Lemma gilt also nicht. Folglich ist L nicht kontextfrei.

Eine weitere interessante Anwendung des Pumping–Lemmas zeigt, dass über unären (also einelementigen) Alphabeten die Klassen der kontextfreien und regulären Sprachen zusammenfallen:

3.4.3 Satz Jede kontextfreie Sprache L über einem unären Alphabet Σ ist regulär.

Beweis: Σ enthält nach Voraussetzung nur ein einziges Zeichen, welches wir hier z.B. mit \$ bezeichnen. Alle Wörter aus L bestehen dann also nur aus verschieden langen \$–Sequenzen. Da L zudem kontextfrei ist, gilt ferner das Pumping–Lemma für kontextfreie Sprachen. Bezeichnet nun n die Pumping–Lemma–Zahl, so setzen wir

$$q := n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

und teilen L so in zwei Mengen $L^{<q}$ und $L^{\geq q}$ auf, dass $L^{<q}$ alle Wörter mit einer Länge kleiner als q und $L^{\geq q}$ analog alle Wörter mit einer Länge größer oder gleich q enthält:

$$L^{<q} := \{w \mid w \in L \wedge |w| < q\} \qquad L^{\geq q} := \{w \mid w \in L \wedge |w| \geq q\}$$

Da es überhaupt nur endlich viele Wörter mit einer Länge kleiner als q gibt, muss $L^{<q}$ als Teilmenge davon ebenfalls endlich und somit regulär sein. Wenn wir also nachweisen, dass auch $L^{\geq q}$ regulär ist, so ist wegen der Abgeschlossenheit regulärer Sprachen unter Vereinigung auch $L^{<q} \cup L^{\geq q}$ und damit L selbst regulär, womit der Satz bewiesen wäre. Wir kümmern uns also im Folgenden nur noch um den Nachweis der Regularität von $L^{\geq q}$.

Als erstes behaupten wir, dass für jedes Wort $z \in L^{\geq q}$ stets auch das um q \$–Zeichen verlängerte Wort $z\$^q$ in $L^{\geq q}$ enthalten ist. Denn wegen $z \in L^{\geq q}$ gilt $|z| \geq q = n! \geq n$ und $z \in L$, d.h. auf z kann das Pumping–Lemma für kontextfreie Sprachen angewendet werden. Also existiert eine Zerlegung $z = uvwxy$, so dass $|vx| \geq 1$ und $|vwx| \leq n$ gilt sowie uv^iwx^iy ein Wort der Sprache L für alle $i \in \mathbb{N}_0$ ist. Wir wählen nun speziell die Zahl

$$i := \frac{q}{|vx|} + 1 \ .$$

Der Bruch geht dabei immer auf, denn es gilt $|vx| \geq 1$ und wegen $|vwx| \leq n$ auch $|vx| \leq n$, d.h. die Länge von vx ist auf jeden Fall ein Teiler von

$$q = n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \ .$$

Für die Länge des gepumpten Wortes $uv^iwx^iy \in L$ gilt dann aber

$$\begin{aligned} |uv^iwx^iy| = |uwy| + i \cdot |vx| = |uwy| + \Big(\frac{q}{|vx|} + 1\Big) \cdot |vx| = |uwy| + \frac{q}{|vx|}|vx| + |vx| \\ = |uwy| + q + |vx| = |uvwxy| + q = |z| + q \ , \end{aligned}$$

d.h. das gepumpte Wort ist nicht nur in L, sondern wegen $|z| + q \geq q + q \geq q$ sogar in $L^{\geq q}$ enthalten. Außerdem hat das Wort $z\$^q$ ebenfalls die Länge $|z| + q$ und besteht wie uv^iwx^iy nur aus \$–Zeichen, d.h. beide Wörter müssen gleich sein.

Also ist mit z auch immer $z\$^q$ ein Wort aus $L^{\geq q}$. Wenn wir auf das verlängerte Wort $z\$^q$ die gleichen Argumente nochmal anwenden, sehen wir, dass auch $z\$^q\$^q = z\$^{2q}$, $z\$^{2q}\$^q = z\$^{3q}$, usw. alles Wörter aus $L^{\geq q}$ sind.

Wir fassen nun die „Wortverlängerungen“ $\$^q$, $\$^{2q}$, $\$^{3q}$, ... in einer Sprache namens D_q zusammen:

$$D_q := \{\$^q, \$^{2q}, \$^{3q}, \dots\} \ .$$

Für alle $z \in L^{\geq q}$ und $x \in D_q$ gilt also $zx \in L^{\geq q}$. Oder anders formuliert: für alle $z \in L^{\geq q}$ ist $\{z\} \cdot D_q$ stets eine Teilmenge von $L^{\geq q}$.

Nun stellen wir uns alle Wörter aus $L^{\geq q}$ der Länge nach aufsteigend sortiert vor und markieren in Gedanken jedes Wort mit einer roten oder schwarzen Markierung nach dem folgenden Prinzip. Beginnend mit dem kürzesten Wort prüfen wir, ob das aktuelle Wort z schon einen Farbmarkierung trägt. Falls ja, gehen wir zum nächsten Wort über. Falls nicht (dies ist gleich zu Beginn beim ersten Wort natürlich immer der Fall), so markieren wir es rot und alle verlängerten Wörter aus $\{z\} \cdot D_q$ schwarz (diese Wörter müssen ja nach unseren bisherigen Erkenntnissen auch immer in der Wortreihe mit auftauchen). Dann

beginnen wir von vorn und suchen wieder ein erstes kürzestes Wort, welches noch keine Markierung trägt. Dieses färben wir wieder rot und alle verlängerten Wörter schwarz. Wir wiederholen diesen Prozess, bis jedes Wort eingefärbt ist.

Die geschilderte Einfärbungstechnik hat folgende Eigenschaften:

- Bei jeder Suche nach einem nächsten rot einzufärbenden Wort gelangen wir hinter alle bislang rot eingefärbten Wörter, denn ansonsten hätten wir in einem vorherigen Schritt ein noch unmarkiertes Wort übersehen.
- Wenn also ein Wort soeben rot eingefärbt wird, so befinden sich dahinter nur schwarze oder noch unmarkierte Wörter. Somit kann es nicht passieren, dass bei der anschließenden schwarzen Einfärbung der verlängerten Wörter bereits rot markierte Wörter nochmals umlackiert werden müssen. (Man kann sogar zeigen, dass auch die schwarzen Wörter nur genau 1x eingefärbt werden, aber dies ist hier unwesentlich.)

Die Sprache aller rot markierten Wörter fassen wir nun in einer Menge R_q zusammen. Die Sprache $R_q \cdot D_q$ enthält dann genau alle schwarz markierten Wörter, und da jedes Wort entweder rot oder schwarz markiert wurde, gilt

$$L^{\geq q} = R_q \cup R_q \cdot D_q \ .$$

Nun zeigen wir noch, dass R_q nur höchstens q Wörter enthält. Dazu betrachten wir die Längen aller Wörter aus R_q und bestimmen die zugehörigen Reste bei einer ganzzahligen Division durch q. Es kann keine zwei Wörter z und z' aus R_q geben, deren Wortlängen die gleichen Reste hinterlassen, denn ansonsten würden sich ihre Wortlängen nur um ein Vielfaches von q unterscheiden. Es würde also $z = z'x$ bzw. $z' = zx$ für ein passend gewähltes „Verlängerungswort“ $x \in D_q$ gelten, und nach der obigen Konstruktion wäre dann eines des beides Wörter schwarz markiert worden. Da es aber nur die q verschiedenen Reste $0, 1, \ldots, q-1$ gilt, muss wie behauptet $|R_q| \leq q$ gelten.

Aber dann ist R_q endlich und somit regulär. Auch D_q ist regulär, denn D_q lässt sich z.B. durch den regulären Ausdruck

$$(\underbrace{\$\$\ldots\$}_{q\text{–mal}})^+$$

beschreiben. Weil zudem die regulären Sprachen unter Vereinigung und Konkatenation abgeschlossen sind, muss folglich

$$L^{\geq q} = R_q \cup R_q \cdot D_q$$

ebenfalls regulär sein, und damit ist alles gezeigt. □

3.4.4 Korollar Die beiden folgenden Sprachen sind nicht kontextfrei:

a) $L := \{0^m \mid m \text{ ist eine Quadratzahl}\}$

b) $L := \{0^p \mid p \text{ ist eine Primzahl}\}$

Beweis: Beide Sprachen werden über dem einelementigen Alphabet $\Sigma = \{0\}$ gebildet. Uns ist bereits bekannt, dass sie nicht regulär sind (siehe Beispiel 2.4.5 und 2.4.6). Nach Satz 3.4.3 können sie somit auch nicht kontextfrei sein (wären sie kontextfrei, so wären sie auch regulär, ein Widerspruch). □

3.4.5 Satz Die Menge der kontextfreien Sprachen über Σ ist unter

1. Vereinigung,
2. Produkt (d.h. Konkatenation) und
3. der positiven Hülle (d.h. der *Kleeneschen Hülle* ohne das leere Wort)

abgeschlossen, nicht jedoch unter

4. Schnitt und
5. Komplement.

Beweis: Seien $G_1 = (V_1, \Sigma, P_1, S_1)$ und $G_2 = (V_2, \Sigma, P_2, S_2)$ kontextfreie Grammatiken mit o.B.d.A. $V_1 \cap V_2 = \emptyset$. Sei ferner S eine neue Variable, d.h. $S \notin V_1 \cup V_2$.

1. Vereinigung: $G := (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 \,|\, S_2\}, S)$ ist kontextfrei und erzeugt $L(G_1) \cup L(G_2)$.
2. Produktbildung: $G := (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$ ist kontextfrei und erzeugt $L(G_1) \cdot L(G_2)$.
3. Positive Hüllenbildung: $G := (V_1 \cup \{S\}, \Sigma, P_1 \cup \{S \to SS \,|\, S_1\}, S)$ ist kontextfrei und erzeugt $L(G_1)^+$.
4. Schnitt: Sei $L_1 := \{a^i b^j c^j \,|\, i, j \geq 1\}$ und $L_2 := \{a^i b^i c^j \,|\, i, j \geq 1\}$. L_1 ist kontextfrei, denn es gilt z.B. $L_1 = L(G_1)$ mit $G_1 = (\{S_1, A, B\}, \{a, b, c\}, P_1, S_1)$ und den Produktionen

   $$P_1 := \{S_1 \to AB,\ A \to a \,|\, aA,\ B \to bc \,|\, bBc\} \ .$$

   Ganz ähnlich lässt sich auch die Kontextfreiheit von L_2 beweisen. Der Schnitt $L_1 \cap L_2 = \{a^i b^i c^i \,|\, i \geq 1\}$ ist aber, wie bereits in Beispiel 3.4.2 gezeigt, nicht kontextfrei.
5. Komplement: Wegen $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ wären die kontextfreien Sprachen sonst unter Schnitt abgeschlossen, was wie soeben gesehen nicht der Fall ist. □

Ohne Beweis (siehe z.B. [13], Abschnitt 10.2) halten wir noch fest:

3.4.6 Satz Die deterministisch kontextfreien Sprachen sind unter Komplementbildung konstruktiv abgeschlossen, d.h. aus jedem DKA M (mit Akzeptanz durch Endzustände) kann man einen DKA $\overline{M}$ konstruieren, der genau die komplementäre Sprache $\overline{L(M)}$ erkennt. □

4

Berechenbarkeit

Wir wenden uns jetzt dem zweiten großen Bereich der theoretischen Informatik zu, nämlich der *Berechenbarkeitstheorie*. Wir haben bereits zu Anfang des zweiten Kapitels erwähnt, dass die heutigen Computer, Notebooks, etc. genauso leistungsfähig sind wie endliche Automaten und insbesondere nur die regulären Sprachen erkennen können. Der Grund hierfür sind die zwar großen, aber eben nach wie vor nur endlich großen Speicherkapazitäten solcher Geräte. Nun spielen allerdings für die meisten Probleme solche Speicherbeschränkungen kaum mehr ein Rolle. Die meisten Problemstellungen benötigen nur einen Bruchteil des Hauptspeichers moderner Computer. Wenn wir also davon ausgehen, dass ein Computer in der Praxis über potentiell unendlich viel Speicher verfügt — wie groß ist dessen „Berechnungskraft“ dann?

Wir werden in diesem Kapitel zunächst wieder ein neues — und nach wie vor recht einfaches — Automatenmodell einführen, die sogenannte *Turingmaschine*. Diese Maschinen entsprechen von ihrer Mächtigkeit her den Typ 0–Grammatiken bzw. (bei Vorliegen einer gewissen Restriktion) den Typ 1–Grammatiken. Interessanterweise sind Turingmaschinen aber auch gleichmächtig zu gängigen Computermodellen, wenn man wie oben erwähnt die Speichergrenzen außer Kraft setzt. Alles, was man z.B. mit Java–Programmen ausdrücken kann, ist auch mit Turingmaschinen berechenbar. Und anders herum: alles was man mit Turingmaschinen *nicht* berechnen kann, ist auch mit Java–Programmen nicht zu realisieren — und da gibt es so einige interessante Probleme, die wir im weiteren Verlauf untersuchen werden. Zunächst muss aber geklärt werden, was man unter dem Stichwort *Berechenbarkeit* überhaupt versteht.

4.1. Ein Berechenbarkeitsbegriff

Rein intuitiv könnte man sagen, irgendeine mathematische Funktion ist *berechenbar*, wenn man ein passendes Programm (z.B. in Java) schreiben könnte, welches die gewünschten Funktionswerte ermittelt. Wir werden uns im Folgenden zunächst auf die

Berechenbarkeitsfrage von (eventuell partiellen) Funktionen $f : \mathbb{N}_0^k \longrightarrow \mathbb{N}_0$ beschränken. Wie immer bedeutet „partiell“ hier wieder, dass nicht unbedingt für alle Eingaben $(n_1, n_2, \ldots, n_k) \in \mathbb{N}_0^k$ ein Funktionswert vorliegen muss. Eine solche Funktion ist also *berechenbar*, wenn es z.B. ein Java–Programm (oder allgemeiner einen *Algorithmus*) gibt, der f repräsentiert, d.h. zu einer Eingabe $(n_1, n_2, \ldots, n_k) \in \mathbb{N}_0^k$ soll das Programm bzw. der Algorithmus nach endlich vielen Schritten mit der Ausgabe $f(n_1, n_2, \ldots, n_k)$ stoppen, sofern ein Ergebnis existiert. Bei undefinierten Funktionswerten soll der Algorithmus nicht stoppen (also z.B. in eine Endlosschleife übergehen). Umgekehrt berechnet jedes Java–Programm eine solche (eventuell partielle) Funktion, sofern es k Zahlen aus $\mathbb{N}_0$ als Argumente/Parameter entgegennimmt und eine Zahl aus $\mathbb{N}_0$ zurückliefert. Falls das Programm für bestimmte Argumente nicht mehr anhält, ist die repräsentierte Funktion an dieser Stelle undefiniert. Den „Funktionswert“ *undefiniert* symbolisieren wir durch das Zeichen $\uparrow$.

Algorithmen werden stets durch *endliche* Programmtexte beschrieben.

4.1.1 Beispiel Funktionen wie e^n, $\sqrt{n}$, $\log n$, usw. sind alle berechenbar, da wir leicht z.B. Java–Programme angeben können, die die entsprechenden Funktionen repräsentieren. Die berechneten Werte müssen dabei in einer vorab festgelegten Weise gerundet werden, um formal einer Funktion mit Bildbereich $\mathbb{N}_0$ zu genügen. Falls man an den Nachkommastellen interessiert ist, kann man z.B. vereinbaren, die Ergebnisse zuerst mit 1.000.000 zu multiplizieren und erst dann eine Rundung vorzunehmen.

4.1.2 Beispiel Die folgende Java–Methode berechnet die *total undefinierte* Funktion $\Omega : \mathbb{N}_0 \to \mathbb{N}_0$ mit $\Omega(n) = \uparrow$ für alle $n \in \mathbb{N}_0$:

```
public static int allesUndefiniert(int n) {
   while (true) {  // Endlosschleife
   }
}
```

Beachten Sie, dass streng genommen der Programmcode nicht korrekt ist. Man kann nämlich keine beliebig große Zahlen an die Methode übergeben, sondern nur maximal die Zahl `Integer.MAX_VALUE`, die etwas über zwei Milliarden groß ist. Man kann diese Ungenauigkeit aber beheben, z.B. mit der Klasse `java.math.BigInteger`.

4.1.3 Beispiel Wir betrachten die Funktion

$$f(n) := \begin{cases} 1 & \text{falls } n \text{ einen Anfang der Ziffernfolge von } \pi \text{ darstellt} \\ 0 & \text{sonst} \end{cases}$$

Beispiele für Funktionswerte von f sind $f(3) = 1$, $f(31415) = 1$ sowie $f(2) = 0$ und $f(3142) = 0$. Die Funktion f ist berechenbar, da es Approximationsalgorithmen für π gibt, die *ziffernweise* gegen π konvergieren.

4.1.4 Beispiel Wir betrachten die Funktion

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

Es ist nicht bekannt, ob diese Funktion berechenbar ist, da keine allgemeinen Resultate über das Vorkommen einer beliebigen Zahlenfolge in der Dezimaldarstellung von π existieren. Wäre die Dezimaldarstellung von π z.B. so „zufällig“, dass jede mögliche Ziffernfolge irgendwo mindestens einmal vorkommt, dann wäre auch g berechenbar (dann wäre nämlich $g(n) = 1$ für alle $n \in \mathbb{N}_0$).

4.1.5 Beispiel Wir betrachten erneut die Ziffernfolge in der Dezimaldarstellung von π. Die Funktion

$$h(n) := \begin{cases} 1 & \text{falls die Ziffer 5 in } \pi \text{ irgendwo } n\text{–mal hintereinander auftaucht} \\ 0 & \text{sonst} \end{cases}$$

sieht auf den ersten Blick der Funktion g recht ähnlich. Allerdings ist diese Funktion berechenbar. Es sind nämlich nur zwei Fälle möglich:

a) Es gibt in der Dezimaldarstellung von π 5er–Ketten beliebiger Länge. Dann ist h sicherlich berechenbar, denn dann gilt $h(n) = 1$ für alle $n \in \mathbb{N}_0$. Das zugehörige Programm muss also stets einfach nur eine 1 ausgeben.

b) Es gibt in der Dezimaldarstellung von π nur 5er–Ketten bis zu einer bestimmten Länge $\ell \in \mathbb{N}$. Auch dann ist h berechenbar, denn nun gilt

$$h(n) := \begin{cases} 1 & \text{falls } n \leq \ell \\ 0 & \text{sonst} \end{cases}$$

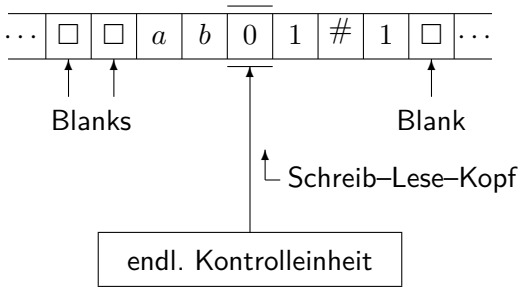
Man braucht also einfach nur zu testen, ob das übergebene n größer als ℓ ist oder nicht.

Zu beiden Fällen kann man also problemlos ein einfaches Programm schreiben — aber wir wissen nicht, welches Programm das richtige ist. Der vorliegende Berechenbarkeitsbegriff fordert lediglich die *Existenz* eines Algorithmus. Eine passende Konstruktionsvorschrift wird nicht verlangt. In diesem Beispiel ist es nicht möglich, eine solche anzugeben, da wir nicht wissen, ob 1. oder 2. zutrifft. Selbst wenn 2. zuträfe, wüssten wir nicht, wie groß ℓ ist. Da aber garantiert einer der beiden Fälle 1. oder 2. zutrifft, ist die geforderte Existenz eines Algorithmus und somit die Berechenbarkeit von h gezeigt.

4.2. Turingmaschinen

Eine *Turingmaschine* ist das für Typ 0–Sprachen passende Automatenmodell, und eine Variante davon, der sog. *linear beschränkte Automat*, kann genau die Typ 1–Sprachen

erkennen. Genauso wie ein endlicher Automat besitzt auch die Turingmaschine verschiedene Zustände und eine Übergangsfunktion. Darüber hinaus besitzen Turingmaschinen aber noch ein potentiell unendlich langes Band, das in einzelne Speicherfelder unterteilt ist. Es wird zuweilen auch als *Turingband* bezeichnet. Auf dem Band bewegt sich ein Schreib–Lese–Kopf hin und her, mit dem das gerade aktuelle Feld gelesen und sein Inhalt verändert werden kann. Jedes Speicherfeld enthält dabei immer ein Zeichen aus einem vorgegebenen *Bandalphabet* Γ. „Leere“ Felder gibt es also nicht. Auf allen bislang noch unbenutzten Feldern ist allerdings immer ein besonderes Zeichen gespeichert, das sog. *Blank*. Es gehört nicht zu dem sog. *Eingabealphabet* $\Sigma \subset \Gamma$, über dem die Eingabewörter gebildet werden. Ansonsten ist das Blank ein normales Zeichen, d.h. es kann beispielsweise wie jedes andere Zeichen auch von der Turingmaschine auf das Band geschrieben werden und somit auch mitten im momentanen Band–Arbeitsbereich auftreten.



4.2.1 Definition Eine (deterministische) *Turingmaschine*[1] (*TM*) ist formal gesehen ein 7–Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$. Dabei ist

- Z eine endliche *Zustandsmenge*,
- $z_0 \in Z$ der *Startzustand*,
- $E \subseteq Z$ die Menge der *Endzustände*,
- Γ das *Bandalphabet* (oder auch *Arbeitsalphabet*) mit $Z \cap \Gamma = \emptyset$,
- $\Sigma \subset \Gamma$ das *Eingabealphabet*,
- $\square \in \Gamma \setminus \Sigma$ das *Blank* und
- $\delta : (Z \setminus E) \times \Gamma \longrightarrow Z \times \Gamma \times \{L, N, R\}$ die *Übergangsfunktion*.

Beachten Sie, dass δ auf der Endzustandsmenge E undefiniert ist. Sobald also ein Endzustand erreicht wird, hält eine Turingmaschine stets an, da es keine weiteren Übergänge gibt (in der Literatur existieren aber auch abweichende Definitionen).

[1] Alan M. Turing, *1912 London, †1954 Wilmslow, Vereinigtes Königreich, britischer Logiker und Mathematiker. Nach ihm ist der *Turing–Award* benannt, der inoffizielle „Nobelpreis“ für Informatik.

Um die deterministische Übergangsfunktion zu betonen, wird eine solche TM häufig auch als *DTM* bezeichnet. Bei einer *nichtdeterministischen Turingmaschine* (*NTM*) kann dagegen (ähnlich wie bei NEAs und NKAs) für einen Übergang unter mehreren Möglichkeiten ausgewählt werden. Die Übergangsfunktion liegt bei NTMs daher in der Form

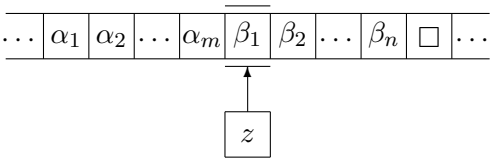
$$\delta : (Z \setminus E) \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{L, N, R\})$$

vor.

Ein Zustandsübergang $\delta(z, a) = (z', b, x)$ bzw. $\delta(z, a) \ni (z', b, x)$ hat folgende Bedeutung. Wenn sich M im Zustand z befindet und unter dem Schreib–Lese–Kopf das Zeichen a auf dem Band steht, so geht M im nächsten Rechenschritt in den Zustand z' über, schreibt das Zeichen b an die durch den Schreib–Lese–Kopf gekennzeichnete Stelle und führt dann entsprechend x mit dem Schreib–Lese–Kopf eine Bewegung um eine Stelle nach links (L), nach rechts (R) oder aber gar keine Bewegung (N für „neutral“) aus.

4.2.2 Definition Eine *Konfiguration* einer TM M ist ein Wort $k \in \Gamma^* Z \Gamma^*$. Interpretiert wird eine solche „Momentaufnahme“ $\alpha z \beta$ (mit $\alpha, \beta \in \Gamma^*$ und $z \in Z$) wie folgt:

- Ein Teil des unendlich langen Bands ist mit $\alpha\beta$ beschriftet. Alle anderen Felder sind mit Blanks belegt.
- Der momentane Zustand der TM M ist z, und der Schreib–Lese–Kopf steht auf dem ersten Zeichen von β.



Anfangs befindet sich nur die Eingabe $x \in \Sigma^*$ auf dem Band, und der Schreib–Lese–Kopf steht auf dem ersten Zeichen von x. Der aktuelle Zustand ist ferner z_0. Diese Situation wird durch die *Startkonfiguration* $z_0 x$ beschrieben.

4.2.3 Definition Auf der Menge aller möglichen Konfigurationen $\Gamma^* Z \Gamma^*$ wird eine Relation $\vdash_M$ (sprich: „hat als (mögliche) Folgekonfiguration“) definiert, so dass $k \vdash_M k'$ genau dann gilt, falls k' aus k durch einen Rechenschritt hervorgehen kann. Sei dazu $k := a_1 \ldots a_m z b_1 \ldots b_n$. Im Fall $\delta(z, b_1) = (z', c, N)$ gilt dann

$$a_1 \ldots a_m z b_1 \ldots b_n \vdash_M a_1 \ldots a_m z' c b_2 \ldots b_n \qquad (m \geq 0, n \geq 1) \ .$$

Im Fall $\delta(z, b_1) = (z', c, R)$ erhalten wir

$$a_1 \ldots a_m z b_1 \ldots b_n \vdash_M a_1 \ldots a_m c z' b_2 \ldots b_n \qquad (m \geq 0, n \geq 1) \ .$$

Und im Fall $\delta(z, b_1) = (z', c, L)$ gilt

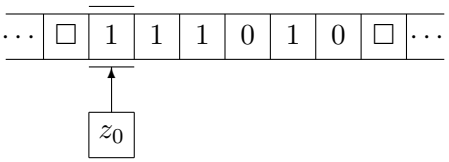
$$a_1 \dots a_m z b_1 \dots b_n \vdash_M a_1 \dots a_{m-1} z' a_m c b_2 \dots b_n \qquad (m \geq 1, n \geq 1) \ .$$

Man beachte dabei, dass sich die Nebenbedingungen bzgl. m und n durch explizite Angaben von Blanks immer erfüllen lassen. Es gilt nämlich $\alpha z \beta = \Box \alpha z \beta = \alpha z \beta \Box$. Bei Bedarf sind die Konfigurationsbeschreibungen also einfach um einige Blanks zu verlängern.

Wir schreiben wieder $k \vdash_M^* k'$, wenn k' in endlich vielen Schritten von k aus erreicht werden kann, d.h. $\vdash_M^*$ ist wieder die *reflexiv transitive Hülle* von $\vdash_M$. Ebenso lassen wir den Index M weg, wenn die zugehörige TM aus dem Zusammenhang her klar ist.

4.2.4 Beispiel Die folgende TM M erwartet als Eingabe eine korrekt kodierte Binärzahl $x \in \{0,1\}^+$ und addiert 1 hinzu:

$$M = (\underbrace{\{z_0, z_1, z_2, z_e\}}_{Z}, \underbrace{\{0,1\}}_{\Sigma}, \underbrace{\{0,1,\Box\}}_{\Gamma}, \delta, z_0, \Box, \underbrace{\{z_e\}}_{E})$$



Das „Programm“ der TM M gliedert sich in drei Phasen. In der ersten Phase läuft der Schreib–Lese–Kopf ohne Änderung des Bandes bis zum rechtesten Zeichen der Eingabe, das er als *least significant bit* (*LSB*) betrachtet. In der zweiten Phase wird von rechts nach links vorgehend eine 1 addiert, wobei Überträge zu berücksichtigen sind. In der dritten Phase ist wieder das linke Ende der Eingabe erreicht. Der Kopf steht dann unter dem *most significant bit* (*MSB*). Der letzte Übertrag wird berücksichtigt und der Endzustand eingenommen.

Phase 1: M bewegt ihren Kopf zum rechten Ende der Eingabe ohne Veränderung des Bandes:

$$\begin{aligned} \delta(z_0, 0) &:= (z_0, 0, R) \\ \delta(z_0, 1) &:= (z_0, 1, R) \end{aligned}$$

Bei Erreichen des ersten Blanks wird der Kopf um eine Position zurück unter das LSB bewegt:

$$\delta(z_0, \Box) := (z_1, \Box, L)$$

Phase 2: Von rechts nach links vorgehend wird eine 1 addiert. Sobald das Ergebnis komplett ist, wird Zustand z_2 eingenommen und der Kopf links von dem MSB positioniert:

$$\begin{aligned}\delta(z_1,0) &:= (z_2,1,L)\\ \delta(z_1,1) &:= (z_1,0,L)\\ \delta(z_2,0) &:= (z_2,0,L)\\ \delta(z_2,1) &:= (z_2,1,L)\end{aligned}$$

Phase 3: Das linke Ende der Eingabe ist erreicht. Der letzte Übertrag ist zu berücksichtigen, und danach ist der Endzustand so einzunehmen, dass der Schreib–Lese–Kopf auf dem MSB steht:

$$\begin{aligned}\delta(z_1,\Box) &:= (z_e,1,N)\\ \delta(z_2,\Box) &:= (z_e,\Box,R)\end{aligned}$$

Wir starten einen Testlauf mit der Eingabe $bin(5) = 101$. M arbeitet wie folgt:

Phase 1:

$$z_0101 \vdash_M 1z_001 \vdash_M 10z_01 \vdash_M 101z_0 = 101z_0\Box \vdash_M 10z_11$$

Phase 2:

$$\vdash_M 1z_100 \vdash_M z_2110 \vdash_M z_2\Box110$$

Phase 3:

$$\vdash_M \Box z_e110 = z_e110\ .$$

Der Übersichtlichkeit halber fasst man die Übergänge einer Turingmaschine auch häufig in einer *Turingtafel* zusammen. Hier würde sich z.B. folgende Tabelle ergeben:

	0	1	$\Box$
z_0	$(z_0,0,R)$	$(z_0,1,R)$	$(z_1,\Box,L)$
z_1	$(z_2,1,L)$	$(z_1,0,L)$	$(z_e,1,N)$
z_2	$(z_2,0,L)$	$(z_2,1,L)$	$(z_e,\Box,R)$
z_e	–	–	–

4.2.5 Definition Die von einer TM $M = (Z,\Sigma,\Gamma,\delta,z_0,\Box,E)$ *akzeptierte Sprache* wird durch

$$L(M) := \{x \in \Sigma^* \mid \exists \alpha,\beta \in \Gamma^*\colon\ \exists z \in E\colon\ z_0x \vdash_M^* \alpha z\beta\}$$

festgelegt. Es wird also lediglich verlangt, dass M irgendwann einen Endzustand erreicht.

Wir betrachten nun eine spezielle Unterklasse der Turingmaschinen. Es handelt sich dabei um die eingangs erwähnten *linear beschränkten Automaten* (*LBAs*), die den Teil des Bandes, auf dem die Eingabe steht, niemals verlassen.

4.2.6 Definition Eine (evtl. nichtdeterministische) TM $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ heißt *linear beschränkt*, wenn für alle $x \in \Sigma^*$ und alle Konfigurationen $\alpha z \beta$ mit

$$z_0 x \vdash_M^* \alpha z \beta$$

stets $|\alpha\beta| \leq |x|$ gilt.

4.2.7 Beispiel Die Turingmaschine aus dem Beispiel 4.2.4 ist kein LBA, da z.B. die binäre Addition von 1 zur binären Zahl 111 das Ergebnis 1000 erzeugt, so dass spätestens die Endkonfiguration mehr Platz als die Startkonfiguration belegt. Eine Turingmaschine, die die binäre Subtraktion von 1 auf positiven Zahlen realisiert, würde sich dagegen als LBA implementieren lassen, da das Ergebnis immer gleich viele oder weniger Stellen als die Ausgangszahl hat.

4.2.8 Satz LBAs charakterisieren die Typ 1–Sprachen, d.h. eine Sprache L wird genau dann von einem LBA akzeptiert, wenn sie kontextsensitiv ist. Ferner sind die durch allgemeine Turingmaschinen (egal ob deterministisch oder nichtdeterministisch) akzeptierten Sprachen genau die Typ 0–Sprachen.

Beweis: Der Satz lässt sich durch gegenseitige Simulationen beweisen, also prinzipiell genauso wie die Äquivalenz von endlichen Automaten und regulären Grammatiken (siehe Sätze 2.2.11 und 2.2.12). Aus zeitlichen Gründen können wir hier aber nicht weiter darauf eingehen. Interessierten Lesern sei die Ausführung z.B. in [23] ab Seite 84 empfohlen. □

Neben der Äquivalenz zu Typ 0–Sprachen interessiert uns vor allem die "Berechnungskraft" von Turingmaschinen. Diese kann man nämlich nicht nur zum Akzeptieren von Sprachen einsetzen, sondern auch zum Berechnen von Funktionen $f : \mathbb{N}_0^k \longrightarrow \mathbb{N}_0$.

4.2.9 Definition Eine (eventuell partielle) Funktion $f : \mathbb{N}_0^k \longrightarrow \mathbb{N}_0$ heißt *Turing–berechenbar*, wenn es eine DTM $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ mit $\Sigma = \{0, 1, \#\}$ gibt, so dass für alle $n_1, n_2, \ldots, n_k \in \mathbb{N}_0$ stets

$$\exists z_e \in E\colon\ z_0 bin(n_1)\#bin(n_2)\#\ldots\#bin(n_k) \vdash_M^* z_e bin(f(n_1, n_2, \ldots, n_k))$$

gilt. Hierbei bezeichnet $bin(n)$ wieder die Binärdarstellung einer Zahl $n \in \mathbb{N}_0$ (siehe Beispiel 2.1.3). (Falls M eine einstellige Funktion $f : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ berechnet, muss # nicht unbedingt im Eingabealphabet enthalten sein.) Ist $f(n_1, \ldots, n_k)$ undefiniert, so soll M entweder endlos laufen oder in einen undefinierten Übergang geraten.

Turingmaschinen können aber auch direkt Eingabewörter aus Σ^* verarbeiten, wobei Σ dann beliebig ist:

4.2.10 Definition Eine (eventuell partielle) Funktion $g : \Sigma^* \longrightarrow \Sigma^*$ bzw. $h : \Sigma^* \longrightarrow \mathbb{N}_0$ heißt *Turing–berechenbar*, wenn es eine DTM M gibt, so dass für alle $x \in \Sigma^*$ stets

$$\exists z_e \in E\colon\ z_0 x \vdash_M^* z_e g(x) \qquad \text{bzw.} \qquad \exists z_e \in E\colon\ z_0 x \vdash_M^* z_e bin(h(x))$$

gilt. Auch hier soll M endlos laufen oder in einem undefinierten Übergang enden, sofern $g(x)$ bzw. $h(x)$ undefiniert ist.

4.2.11 Beispiel Die Nachfolgefunktion $f : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ mit $f(n) = n + 1$ ist Turing–berechenbar (siehe Beispiel 4.2.4).

4.2.12 Beispiel Die total undefinierte Funktion Ω ist Turing–berechenbar. Sie wird z.B. durch eine DTM $M = (\{z_0\}, \{0, 1\}, \{0, 1, \Box\}, \delta, z_0, \Box, \emptyset)$ mit einer beliebigen Übergangsfunktion δ repräsentiert. Da es keine Endzustände gibt, kann niemals eine Ausgabe erzeugt werden.

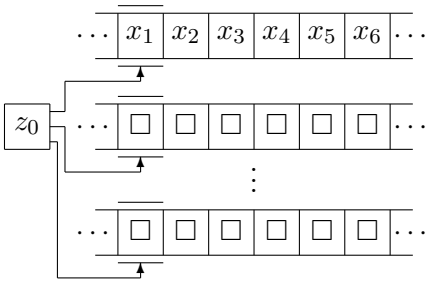
4.3. Makro–Programmierung von TMs

Ähnlich wie bei Kellerautomaten wird die „exakte“ Ausführung einer Turingmaschine mit genauen Angaben zu ihrer Übergangsfunktion schnell unübersichtlich. Wir überlegen uns deshalb in diesem Abschnitt, wie man die Arbeitsweise von Turingmaschinen ähnlich wie in Programmiersprachen beschreiben kann. Wir besorgen uns als erstes ein komfortableres Automatenmodell.

4.3.1 Definition Sei $k \geq 1$. Eine *k–Band Turingmaschine* kann auf k Bändern unabhängig voneinander mittels k Schreib–Lese–Köpfen operieren, d.h. die Übergangsfunktion δ ist jetzt eine Funktion

$$\delta : (Z \setminus E) \times \Gamma^k \longrightarrow Z \times \Gamma^k \times \{L, N, R\}^k \ .$$

Entsprechend verallgemeinert sich der Begriff einer Konfiguration, und die Übergangsrelationen $\vdash_M$ und $\vdash_M^*$ werden passend modifiziert. Bzgl. der von M akzeptierten Sprache $T(M)$ legen wir fest, dass die Eingabewörter immer auf dem ersten Band bereitgestellt werden sollen.



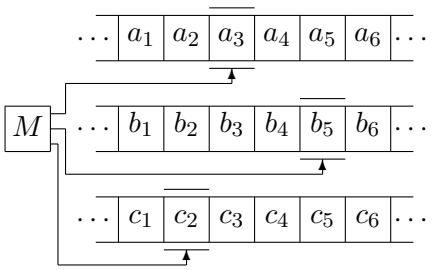
Auf dem ersten Band wird auch gegebenenfalls das Berechnungsergebnis hinterlegt, sofern M eine Funktion repräsentiert.

Durch die Verallgemeinerung gewinnt man nichts an prinzipieller Rechenkraft hinzu, sondern nur an Effizienz. Es gilt nämlich das folgende Resultat.

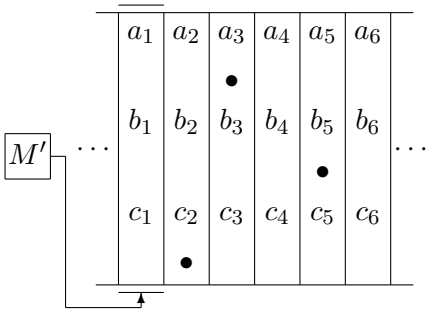
4.3.2 Satz Zu jeder k–Band Turingmaschine M gibt es eine 1–Band Turingmaschine M', die dieselbe Funktion berechnet (bzw. die gleiche Sprache akzeptiert).

Beweis: Sei Σ das Eingabealphabet sowie Γ das Bandalphabet von M. Wir simulieren im Folgenden M durch eine 1–Band TM. Die Idee hierbei ist, die k Bänder von M durch ein k–spuriges Band in M' zu ersetzen.

Sei z.B. die folgende Konfiguration von M momentan aktuell:



Wir „kleben“ jetzt die k Bänder untereinander zu einem Band mit k Spuren zusammen. Mit nur einem Schreib–Lese–Kopf werden an der gegenwärtigen Position immer alle k untereinanderstehenden Symbole der k Bänder gleichzeitig gelesen und geschrieben. Insbesondere kann M' seinen Kopf nicht in unterschiedliche Richtungen auf den verschiedenen Spuren wandern lassen. M' muss sich deshalb merken, wo in den einzelnen Spuren die simulierten Schreib–Lese–Köpfe von M stehen. Hierfür verwendet M' nochmals k Spuren, die jeweils fast nur aus Blanks bestehen. An genau einer Stelle einer solchen *Kopfspur* befindet sich jedoch ein spezielles Zeichen $\bullet$, welches die Position des Schreib–Lese–Kopfes von M auf der zugehörigen Bandspur markiert. Die obige Situation simuliert M' also wie folgt (hierbei sind auf den „Kopfspuren“ im Folgenden zur besseren Übersichtlichkeit nur die Kopfpositionen $\bullet$ angegeben; die Leerzeichen $\square$ wurden weggelassen):



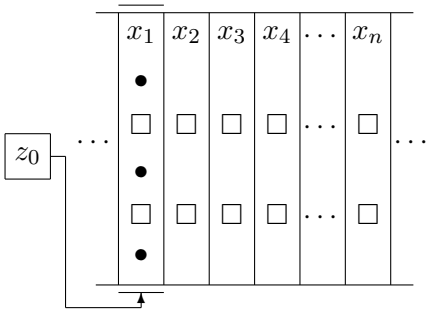
Das Konzept mehrerer *Spuren* auf einem Band wird durch ein stark vergrößertes Arbeitsalphabet bewerkstelligt. Sei

$$\Gamma' := \Gamma \cup (\Gamma \times \{\bullet, \square\})^k$$

das Arbeitsalphabet von M'. Jedes Symbol

$$a = (a_1, p_1, a_2, p_2, \ldots, a_k, p_k) \in (\Gamma \times \{\bullet, \square\})^k$$

stellt den Inhalt der k Band – und Kopfspuren an einer Querschnittsposition dar. Das ursprüngliche Bandalphabet Γ wird als Teilmenge von Γ' benötigt, da anfangs die normale Startkonfiguration einer Eingabe $x_1 x_2 \ldots x_n \in \Sigma^*$ möglich sein muss. M' ersetzt diese durch die simulierte Startkonfiguration von M:



Jeder Schritt von M wird durch M' wie folgt simuliert. M' startet mit seinem einzigen Schreib–Lese–Kopf links von allen $\bullet$–Markierungen und bewegt sich nach rechts, bis alle $\bullet$–Markierungen überschritten wurden. Dabei merkt sich M' jeweils die darüber befindlichen Symbole der entsprechenden Spur. Genau diese Symbole würde M beim nächsten Schritt von den k Bändern lesen. Die verschiedenen Symbole kann M' mit Hilfe (sehr vieler) verschiedener Zustände zwischenspeichern. Es handelt sich jedoch nach wie vor nur um endlich viele Zustände, denn für die k Symbole, die jeweils aus dem endlichen Alphabet Γ stammen, gibt es nur $|\Gamma|^k < \infty$ viele Kombinationsmöglichkeiten. Weiterhin kennt M' den aktuellen Zustand von M. Damit weiß M', wie M die gelesenen Symbole abändern würde, weiterhin, welche Kopfbewegungen auszuführen sind und wie der Folgezustand von M aussieht. Indem M' jetzt umgekehrt von rechts nach links die direkten Umgebungen aller $\bullet$–Markierungen entsprechend der Übergangsfunktion δ von M ändert (inklusive der $\bullet$–Markierungen selbst), ist ein Schritt von M simuliert. Sobald M in einen Endzustand übergeht, werden die Hilfsspuren wieder gelöscht, um die gewonnene Ausgabe allein auf das Band zu schreiben.

Die Simulation ist auch unter Verwendung von nur k Spuren möglich, indem die simulierten k Köpfe unmittelbar *vor* dem jeweils aktuellen Symbol auf die entsprechende Spur geschrieben werden. □

Als nächstes wollen wir einige für Programmiersprachen typische Konzepte durch k–Band Turingmaschinen simulieren. Hierdurch gewinnen wir vereinfachende (Makro–)

Schreibweisen für Turingmaschinen, so dass wir nach und nach von der mühsamen und unübersichtlichen Angabe von detaillierten Übergangsfunktionen abstrahieren können.

Simuliert eine k–Band Turingmaschine eine 1–Band TM $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ auf ihrem i–ten Band, so bezeichnen wir diese k–Band Turingmaschine als $\mathit{Embed}_k(i, M)$. Technisch realisiert man diese Turingmaschine wie folgt.

Für $1 \leq i \leq k$ ist $\mathit{Embed}_k(i, M) := (Z, \Sigma, \Gamma, \delta', z_0, \Box, E)$ eine k–Band TM, deren Übergangsfunktion δ' die ursprünglichen Übergänge auf dem i–ten Band nachvollzieht und alle anderen Bandinhalte unverändert lässt. Sei also $\delta(z, a) = (z', b, x)$ mit $x \in \{L, N, R\}$ ein Übergang von M. Dann existieren in $\mathit{Embed}_k(i, M)$ Übergänge der Form

$$\begin{aligned} &\delta'(z, a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_k) \\ &:= (z', a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_k, N, \ldots, N, x, N, \ldots, N) \end{aligned}$$

für alle $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k \in \Gamma$, wobei das x hinter dem $(i-1)$–ten N steht.

Ist die Bandanzahl k aus dem Zusammenhang klar oder momentan unwichtig, schreiben wir statt $\mathit{Embed}_k(i, M)$ auch verkürzend $\mathit{Embed}(i, M)$.

Analog kann man für $\ell \leq k$ durch $\mathit{Embed}_k(i_1, \ldots, i_\ell, M)$ die Einbettung einer ℓ–Band Turingmaschine M in die Bänder $i_1, \ldots, i_\ell$ einer k–Band Turingmaschine definieren.

4.3.3 Definition Mit $<\mathsf{Band}\,{+}{+}>$ bezeichnen wir die „+1“–*Addiermaschine* aus dem Beispiel 4.2.4. Für das i–te Band definieren wir $<\mathsf{Band}_i\,{+}{+}> := \mathit{Embed}(i, <\mathsf{Band}\,{+}{+}>)$. Genauso gibt es auch TMs $<\mathsf{Band}\,{-}{-}>$ bzw. $<\mathsf{Band}_i\,{-}{-}>$, die analog die Zahl 1 von dem derzeitigen Bandinhalt subtrahieren. Um Unterläufe zu vermeiden, kann man vorab vereinbaren, dass keine Operation durchzuführen ist, wenn der Bandinhalt bereits gleich 0 ist (wir werden nachher sehen, wie man dies „elegant“ testen kann).

Wir wollen ferner zwei verschiedene Turingmaschinen miteinander *verknüpfen*, d.h. sie hintereinander ausführen lassen, wobei die zweite TM das Ergebnis der ersten als Eingabe verwendet. Seien dazu $M_1 = (Z_1, \Sigma, \Gamma_1, \delta_1, z_1, \Box, E_1)$ und $M_2 = (Z_2, \Sigma, \Gamma_2, \delta_2, z_2, \Box, E_2)$ mit o.B.d.A. $Z_1 \cap Z_2 = \emptyset$ zwei 1–Band TMs. Für $t \in E_1$ konstruieren wir eine Turingmaschine $M_1 \xrightarrow{t} M_2 := (Z_1 \dot\cup Z_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, z_1, \Box, E_2)$ mit

$$\delta(z, a) := \begin{cases} \delta_1(z, a) & \text{falls } z \in Z_1 \setminus \{t\} \\ (z_2, a, N) & \text{falls } z = t \\ \delta_2(z, a) & \text{falls } z \in Z_2 \end{cases}$$

Ganz ähnlich kann man auch k–Band TMs miteinander verknüpfen. Die Verknüpfung funktioniert sogar dann, wenn die beteiligten TMs eine unterschiedliche Anzahl von Bändern haben. Hierzu bettet man zunächst die TM mit der kleineren Anzahl von Bändern in eine solche ein, die genauso viele Bänder wie die andere TM besitzt. Anschließend kann die eigentliche Verknüpfung erfolgen.

Sollen alle Endzustände von M_1 mit dem Startzustand von M_2 identifiziert werden, so schreibt man einfach $M_1 \longrightarrow M_2$.

Durch mehrfache Anwendung der „$\longrightarrow$“–Notation kann man jetzt komplexere TMs zusammensetzen. Oft ist es dabei übersichtlicher, die Start– und Endpunkte einer solchen TM explizit zu kennzeichnen. Hierzu benutzen wir die TMs $<\mathsf{Start}>$ und $<\mathsf{Stopp}>$, die beide „nichts“ machen. Man kann sie also z.B. als k–Band TMs $(\{s\}, \Sigma, \Gamma, \delta, s, \Box, \{s\})$ angeben, wobei δ überall undefiniert ist. Der Startzustand s ist gleichzeitig auch Endzustand. Eine solche TM reicht also ihre Eingabe ohne Änderung als Ausgabe weiter, ohne jemals einen Schritt auszuführen.

4.3.4 Beispiel Die Verknüpfung

$$<\mathsf{Start}> \longrightarrow <\mathsf{Band{+}{+}}> \longrightarrow <\mathsf{Band{+}{+}}> \longrightarrow <\mathsf{Band{+}{+}}> \longrightarrow <\mathsf{Stopp}>$$

liefert eine TM, die die Zahl drei zum Bandinhalt addiert. Sie wirkt also ähnlich wie eine Java–Anweisung `x = x + 3`, wobei `x` eine Variable vom Typ `int` ist. Natürlich kann man eine solche TM auch wieder auf einem bestimmten Band einer mehrbändigen TM einbetten.

Wir führen jetzt weitere Konzepte ein, die höheren Sprachkonstrukten ähneln.

Eine Wertzuweisung wie `variable = 0` kann z.B. ebenfalls leicht durch eine Turingmaschine $<\mathsf{Band} := 0>$ durchgeführt werden. Diese TM $(\{z_0, z_1\}, \Sigma, \Gamma, \delta, z_0, \Box, \{z_1\})$ überschreibt zunächst den aktuell genutzten Bandbereich mit Blanks. Wenn man davon ausgeht, dass anfangs der Schreib–Lese–Kopf am linken Rand des benutzten Bereichs steht (unter dem ersten Symbol) und keine Blanks in dem Bereich selbst vorkommen, reichen für den Löschvorgang Übergänge der Form

$$\delta(z_0, a) = (z_0, \Box, R)$$

für alle $a \neq \Box$ aus. Anschließend legt die TM die binäre Kodierung der Zahl 0 (also eine einzelne 0 selbst) auf dem Band ab:

$$\delta(z_0, \Box) = (z_1, 0, N)$$

Entsprechend initialisiert $<\mathsf{Band}_i := 0> := \mathit{Embed}(i, <\mathsf{Band} := 0>)$ den Inhalt des i–ten Bandes einer mehrbändigen TM mit der Zahl 0.

Umgekehrt lässt sich z.B. auch abfragen, ob auf einem Band die korrekte Kodierung der Zahl 0 steht. Dies leistet eine Turingmaschine $<\mathsf{Band} = 0?>$ mit der Zustandsmenge

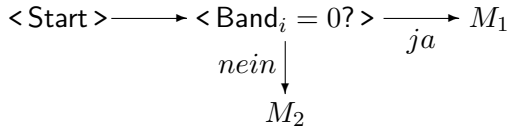
$$Z := \{z_0, z_1, ja, nein\} \ ,$$

wobei z_0 der Startzustand sowie ja und $nein$ die beiden Endzustände sind. Die TM arbeitet mit diesen Übergängen:

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

Mittels $<\mathsf{Band}_i = 0?> := \mathit{Embed}(i, <\mathsf{Band} = 0?>)$ lässt sich die Abfrage auch auf einer mehrbändigen TM durchführen.

Eine Abfrage mit der TM $<\mathsf{Band} = 0?>$ kann dazu ausgenutzt werden, in Abhängigkeit des Ergebnisses Verzweigungen vorzunehmen. Soll beispielsweise bei einem positiven Ergebnis eine TM M_1 ausgeführt werden und bei einem negativen Ergebnis eine andere TM M_2, so kann man dies mit den schon eingeführten Konzepten wie folgt bewerkstelligen:



Oft besteht auch Bedarf daran, den Inhalt eines Bandes zu duplizieren. Wir können eine entsprechende *Kopiermaschine* $<\mathsf{Band}_1 := \mathsf{Band}_2>$ durch eine zweibändige Turingmaschine $(\{z_0, z_1, z_2\}, \Sigma, \Gamma, \delta, z_0, \Box, \{z_2\})$ wie folgt konstruieren. Wir nehmen wieder an, dass sich die Schreib–Lese–Köpfe der Maschine zu Beginn des Kopiervorgangs an den jeweils linken Enden der benutzten Bereiche befinden und Blanks innerhalb der Bereiche nicht vorkommen. Folglich kopieren Übergänge der Form

$$\delta(z_0, a, b) = (z_0, b, b, R, R)$$

für $b \neq \Box$ den Inhalt vom zweiten auf das erste Band. Sobald auf dem zweiten Band das erste Blank auftritt, ist der eigentliche Kopiervorgang schon beendet. Allerdings ist evtl. der erste Bereich umfangreicher als der zweite, so dass noch weitere Zeichen zu löschen sind. Die Maschine bewegt sich dann zunächst an das Ende des zu löschenden Bereichs, d.h. sie verfügt über Übergänge der Form

$$\delta(z_0, a, \Box) = (z_0, a, \Box, R, R)$$

für alle $a \neq \Box$. Wenn diese Aufgabe erledigt ist, die Kopiermaschine also auch auf dem ersten Band auf ein Blank trifft, kehrt sie um und wandert mit dem Kopf von rechts nach links zurück:

$$\delta(z_0, \Box, \Box) = (z_1, \Box, \Box, L, L)$$

Jetzt wird der Rest vom ersten Band gelöscht und der schon kopierte Bereich in Ruhe gelassen, d.h. es gibt für alle $a \neq \Box$ Übergänge der Form

$$\delta(z_1, a, \Box) = (z_1, \Box, \Box, L, L) \qquad \text{und} \qquad \delta(z_1, a, a) = (z_1, a, a, L, L) \ .$$

Wieder am linken Rand angekommen (erkennbar anhand von zwei Blanks auf beiden Bändern) stoppt die Maschine und hält:

$$\delta(z_1, \Box, \Box) = (z_2, \Box, \Box, R, R) \ .$$

Die Maschine $<\mathsf{Band}_i := \mathsf{Band}_j> := \mathit{Embed}(i, j, <\mathsf{Band}_1 := \mathsf{Band}_2>)$ kopiert analog bei einer mehrbändigen TM den Inhalt von Band j nach Band i.

4.4. WHILE–Berechenbarkeit

Ziel dieses Abschnitts ist es zu zeigen, dass die Mächtigkeit von Turingmaschinen mit der Mächtigkeit üblicher Programmiersprachen wie z.B. Java übereinstimmt. Der Begriff *Mächtigkeit* bezieht sich dabei jeweils auf die innerhalb eines gegebenen Modells berechenbaren Funktionen. Für den Beweis führen wir zunächst eine elementare Programmiersprache ein, mit der sich sogenannte WHILE–Programme schreiben lassen. WHILE–Programme sind aus der Sicht der darin verwendeten Konzepte gängigen Programmiersprachen sehr ähnlich. Es handelt sich sozusagen um eine (scheinbar) stark abgespeckte Version von Java. Später werden wir jedoch sehen, dass die wenigen elementaren Anweisungen bereits ausreichen, um die gleichen Funktionen wie in Java berechnen zu können. Ferner sind WHILE–Programme gleichmächtig zu Turingmaschinen, d.h. im Endeffekt ist dann wie behauptet jede Java–berechenbare Funktion auch Turing–berechenbar und umgekehrt.

4.4.1 Definition WHILE–Programme sind wie folgt aufgebaut:

- Es gibt nur Variablen mit den Namen $\mathtt{x}_0$, $\mathtt{x}_1$, $\mathtt{x}_2$, ... Jede Variable kann nur Zahlen aus $\mathbb{N}_0$ speichern (dafür aber beliebig große), und es wird auch nur im Bereich der natürlichen Zahlen (inklusive der Null) gerechnet.
- In den Programmen kommen nur die Konstanten 0, 1, 2, ... vor.
- Alle Anweisungen sind von der Form $\mathtt{x}_i$:= $\mathtt{x}_j$ + c; oder $\mathtt{x}_i$:= $\mathtt{x}_j$ - c;, wobei c eine der obigen Konstanten ist. Die Wertzuweisung $\mathtt{x}_i$:= $\mathtt{x}_j$ + c weist der Variablen $\mathtt{x}_i$ den um $c \in \mathbb{N}_0$ vergrößerten Wert der Variablen $\mathtt{x}_j$ zu. Bei der Anweisung $\mathtt{x}_i$:= $\mathtt{x}_j$ - c wird die *modifizierte Subtraktion* verwendet, d.h. falls c größer als der Inhalt von $\mathtt{x}_j$ ist, so wird $\mathtt{x}_i$ auf 0 gesetzt, ansonsten wird der Variablen $\mathtt{x}_i$ der um c verringerte Wert der Variablen $\mathtt{x}_j$ zugewiesen. Somit wird verhindert, dass man in den Bereich der negativen Zahlen abrutscht.
- Ansonsten existieren nur noch die Schlüsselwörter WHILE, DO und END. Mit ihnen lassen sich Schleifen der Form

```
...
WHILE (xi > 0) DO
   ...
END;
...
```

  programmieren. Sie funktionieren genauso wie in Java, d.h. der Schleifenrumpf wird so oft hintereinander ausgeführt, bis der Wert der Variablen $\mathtt{x}_i$ gleich 0 ist (beachten Sie, dass die Variablen keine negativen Zahlen speichern können). Dabei können zwei extreme Situationen auftreten. Zum einen kann es sein, dass die Variable $\mathtt{x}_i$ schon zu Beginn den Wert 0 enthält. In diesem Fall wird der Schleifenrumpf kein einziges Mal ausgeführt. Zum anderen könnte es sein, das $\mathtt{x}_i$ anfangs einen Wert größer als 0 besitzt und niemals den Wert 0 zugewiesen bekommt. In diesem

Fall wird die Schleife unendlich oft ausgeführt, und das Programm hält nicht mehr an. Insbesondere werden dann auch die nachfolgenden Programmzeilen nie mehr durchlaufen.

- Ein `WHILE`–Programm startet mit der ersten Programmzeile.
- Ein `WHILE`–Programm stoppt, falls es keine weitere auszufühende Programmzeile mehr gibt (d.h. die nächste auszuführende Programmzeile wäre diejenige hinter dem Ende des Programmtextes).

4.4.2 Beispiel Ein einfaches `WHILE`–Programm ist z.B. das Folgende (wir werden gleich klären, was es genau „macht“):

```
x0 := x1 + 0;
WHILE (x2 > 0) DO
   x0 := x0 + 1;
   x2 := x2 - 1;
END;
```

Es berechnet wie alle `WHILE`–Programme eine bestimmte Funktion $\mathbb{N}_0^k \longrightarrow \mathbb{N}_0$. Hierfür muss man die Start– und Endbedingungen für den Programmablauf festlegen:

4.4.3 Definition Eine (evtl. partielle) Funktion $f : \mathbb{N}_0^k \longrightarrow \mathbb{N}_0$ heißt `WHILE`–*berechenbar*, wenn es ein `WHILE`–Programm gibt, welches für alle $n_1, n_2, \ldots, n_k \in \mathbb{N}_0$ den folgenden Anforderungen genügt:

- Das Programm wird mit den Werten $n_1, n_2, \ldots, n_k$ in den Variablen $\mathtt{x_1}$, $\mathtt{x_2}$, $\ldots$, $\mathtt{x_k}$ gestartet.
- Alle anderen Variablen $\mathtt{x_0}$, $\mathtt{x_{k+1}}$, $\mathtt{x_{k+2}}$, $\mathtt{x_{k+3}}$, $\ldots$ haben den Anfangswert 0.
- Das Programm stoppt mit dem Wert $f(n_1, n_2, \ldots, n_k)$ in der Variablen $\mathtt{x_0}$, sofern f auf den Argumenten $n_1, n_2, \ldots, n_k$ definiert ist.
- Im Fall $f(n_1, n_2, \ldots, n_k) = \uparrow$ hält das Programm nicht an.

4.4.4 Beispiel Eine `WHILE`–berechenbare Funktion ist beispielsweise die Addierfunktion $add : \mathbb{N}_0^2 \longrightarrow \mathbb{N}_0$, welche ein Paar $(n_1, n_2) \in \mathbb{N}_0^2$ auf den Wert $add(n_1, n_2) := n_1 + n_2$ abbildet. Denn das im Beispiel 4.4.2 genannte `WHILE`–Programm

```
x0 := x1 + 0;
WHILE (x2 > 0) DO
   x0 := x0 + 1;
   x2 := x2 - 1;
END;
```

leistet genau das Gewünschte. Beachten Sie, dass sich initial die Summanden n_1 und n_2 gemäß den obigen Konventionen in den Variablen x_1 und x_2 befinden.

Allgemein kann man leicht einen Programmausschnitt angeben, der die Additionsanweisung $x_i := x_j + x_k$ simuliert. Wir können daher in Zukunft so tun, als wäre diese Anweisung direkt als Befehl in WHILE–Programmen verfügbar. Ebenso sei $x_i := x_j$ von nun an eine Kurzform für $x_i := x_j + 0$ und weiter $x_i := c$ eine Kurzform für

```
WHILE (xi > 0) DO
   xi := xi - 1;
END;
xi := xi + c;
```

4.4.5 Beispiel Die Multiplikationsfunktion ist ebenfalls WHILE–berechenber:

```
WHILE (x2 > 0) DO
   x0 := x0 + x1;
   x2 := x2 - 1;
END;
```

Hierbei haben wir die oben erwähnte Anweisung $x_i := x_j + x_k$ als Makro verwendet. Analog können wir $x_i := x_j * x_k$ von nun an als weiteres Makro einsetzen.

Bedingte Anweisungen sind ebenfalls möglich. Das folgende Programm führt die mit „. . .“ angedeuteten Zeilen nur dann aus, wenn x_i mit dem Wert 0 belegt ist (da nur genau dann die erste WHILE–Schleife übersprungen und somit die zweite nicht übersprungen wird):

```
xj := xi;
xk := 1;
WHILE (xj > 0) DO
   xj := 0;
   xk := 0;
END;
WHILE (xk > 0) DO
   xk := 0;
   ...
END;
```

(Dabei seien x_j und x_k zwei bislang unbenutzte Variablen.) Das Programm simuliert damit die Abfrage

```
IF (x_i = 0) THEN
   ...
ENDIF;
```

Ganz ähnlich können wir uns auch für beliebige $c \in \mathbb{N}$ bedingte Anweisungen der Form

```
IF (x_i = c) THEN
   ...
ENDIF;
```

beschaffen. Weiter sind Vergleiche zwischen zwei Variablen, also z.B.

```
IF (x_i = x_j) THEN
   ...
ENDIF;
```

genauso möglich wie komplexere „`IF THEN ELSE`“–Konstrukte.

4.4.6 Beispiel Die ganzzahlige Division $div : \mathbb{N}_0^2 \longrightarrow \mathbb{N}_0$ mit $div(n_1, n_2) := \lfloor n_1/n_2 \rfloor$ (dabei bedeutet $\lfloor \cdots \rfloor$: „bis zur nächsten kleineren ganzen Zahl abrunden“) ist für alle Argumente (n_1, n_2) mit $n_2 = 0$ nicht definiert. Das folgende `WHILE`–Programm fängt diese Fälle mit einer `IF`–Abfrage ab und führt gegebenenfalls eine Endlosschleife aus (beachten Sie, dass die ab Zeile 7 beginnende `WHILE`–Schleife im Fall $n_1 = n_2 = 0$ mit dem falschen Ergebnis 0 anhalten würde). Also entspricht das Programm genau den Berechenbarkeitsrichtlinien (siehe Def. 4.4.3). Die Divisionsfunktion ist demnach ebenfalls `WHILE`–berechenbar.

```
IF (x_2 = 0) THEN
   x_2 := 1;
   WHILE (x_2 > 0) DO
      x_2 := 1;
   END;
ENDIF;
WHILE (x_1 > 0) DO
   IF (x_1 >= x_2) THEN
      x_0 := x_0 + 1;
   ENDIF;
   x_1 := x_1 - x_2;
END;
```

Würde man beide `IF`–Anweisungen auf die konkreten elementaren Anweisungen zurückführen, entstünde schon ein recht langes Programm. Nur am Rande sei bemerkt, dass

auch das folgende wesentlich kürzere Programm die ganzzahlige Divisionsfunktion realisiert und zudem ganz ohne IF–Anweisungen auskommt — es ist aber vielleicht nicht ganz so naheliegend wie die erste Version:

```
x1 := x1 + 1;
x1 := x1 - x2;
WHILE (x1 > 0) DO
   x1 := x1 - x2;
   x0 := x0 + 1;
END;
```

Wie auch immer, da die ganzzahlige Division WHILE–berechenbar ist, können wir von nun an auch Ausdrücke der Form `xi := xj / xk` zulassen. Somit sind wieder weitere Funktionen WHILE–berechenbar, z.B. die Modulo–Funktion, die den Rest bei einer ganzzahligen Division ermittelt:

```
x0 := x1 / x2;
x0 := x0 * x2;
x0 := x1 - x0;
```

Ausdrücke wie `xi := xj mod xk` sind deshalb ebenfalls möglich und werden sich noch als nützlich erweisen.

Wir werden jetzt zeigen, dass die Berechenbarkeitsbegriffe von WHILE–Programmen und Turingmaschinen gleichwertig sind. Den Anfang dazu macht der folgende Satz.

4.4.7 Satz Jede WHILE–berechenbare Funktion f ist auch Turing–berechenbar.

Beweis: Sei $f : \mathbb{N}_0^k \longrightarrow \mathbb{N}_0$ beliebig. Da f WHILE–berechenbar ist, existiert ein passendes WHILE–Programm P, welches f berechnet. Im Programmtext von P kommen nur endlich viele Variablen vor, etwa $\mathtt{x}_0$, $\mathtt{x}_1$, ..., $\mathtt{x}_r$. Wir konstruieren jetzt eine $(r+1)$–bändige DTM M, die genau die Arbeitsweise von P nachvollzieht. Der Inhalt des i–ten Bands entspricht dabei dem Inhalt der Variable $\mathtt{x}_i$, wobei wir diesmal die Bänder von 0 bis r durchnummerieren.

Zunächst zerlegt M den Inhalt des ersten Bands (mit dem Index 0, das also die Eingabe $bin(n_1)\#bin(n_2)\#\ldots\#bin(n_k)$ enthält) in seine Bestandteile $bin(n_1)$, $bin(n_2)$, usw. Dazu kopiert M beginnend vom links das erste Argument $bin(n_1)$ auf das zweite Band (mit dem Index 1, welches also der Variable $\mathtt{x}_1$ entspricht), bis es auf das Trennzeichen # stösst. Anschließend kopiert M das nächste Argument $bin(n_2)$ auf das dritte Band (mit dem Index 2, welches $\mathtt{x}_2$ entspricht), bis es auf das nächste Trennzeichen # trifft, usw. Das letzte Argument $bin(n_k)$ landet auf Band k und wird bis zum Auftreten des ersten Blanks kopiert. Die Initialisierungsphase endet damit, dass M das erste Band löscht und dort genauso wie auf den Bändern $k+1, k+2, \ldots, r$ eine einzelne Null ablegt (womit

die binäre Kodierung der Zahl 0 simuliert wird). Danach enthalten alle „Bandvariablen“ die gleichen Inhalte wie P unmittelbar vor dem Start.

Das eigentliche Programm P kann nun leicht 1:1 in eine weitere $(r+1)$–bändige DTM umgesetzt werden, die man anschließend nach der oben skizzierten „Initialisierungs“–DTM M mittels der „$\longrightarrow$“–Verknüpfung zur Ausführung bringt. Alle dazu benötigten Makro–Turingmaschinen sind uns bereits bekannt:

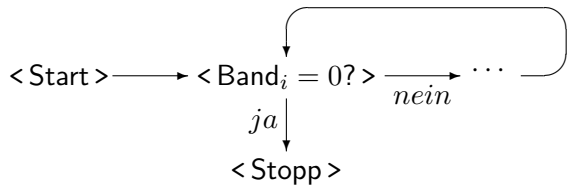
- Eine Wertzuweisung `x`$_i$ `:=` `x`$_j$ `+` c entspricht der zusammengesetzten Turingmaschine

$$<\mathsf{Band}_i := \mathsf{Band}_j> \longrightarrow \underbrace{<\mathsf{Band}_i\,{+}{+}> \longrightarrow <\mathsf{Band}_i\,{+}{+}> \longrightarrow \cdots \longrightarrow <\mathsf{Band}_i\,{+}{+}>}_{c\text{–mal}} .$$

- Analog simuliert man `x`$_i$ `:=` `x`$_j$ `-` c durch die Turingmaschine

$$<\mathsf{Band}_i := \mathsf{Band}_j> \longrightarrow \underbrace{<\mathsf{Band}_i\,{-}{-}> \longrightarrow <\mathsf{Band}_i\,{-}{-}> \longrightarrow \cdots \longrightarrow <\mathsf{Band}_i\,{-}{-}>}_{c\text{–mal}} .$$

- Mehrere simulierte Anweisungen und Programmstücke kann man mittels „$\longrightarrow$“ verknüpfen und damit wie im Original hintereinander ausführen lassen.
- Selbst eine `WHILE`–Schleife der Form `WHILE (x`$_i$ `> 0) DO ... END` können wir problemlos nachbilden:



Da das Original–Programm P am Ende das Funktionsergebnis $f(n_1, n_2, \ldots, n_k)$ in `x`$_0$ ablegt, befindet sich entsprechend bei M das Ergebnis $bin(f(n_1, n_2, \ldots, n_k))$ am Ende auf dem ersten Band (mit dem Index 0) und damit bereits an der richtigen Stelle. M braucht also nichts weiter zu tun, als genau dann in einen Endzustand überzugehen, wenn das simulierte Programm P stoppt.

Damit ist es gelungen, eine zum Programm P äquivalente Turingmaschine anzugeben, die demnach genau die gleiche Funktion f berechnet. Folglich ist f auch Turing–berechenbar. □

Die Umkehrung von Satz 4.4.7 gilt ebenfalls:

4.4.8 Satz Jede Turing–berechenbare Funktion f ist auch `WHILE`–berechenbar.

Beweis: Sei $f : \mathbb{N}_0^k \longrightarrow \mathbb{N}_0$ eine beliebige Funktion und $M = (Z, \Sigma, \Gamma, \delta, z_1, \Box, E)$ eine passende einbändige DTM, die f berechnet. (Wir werden in Kürze sehen, warum

hier ausnahmsweise der Startzustand z_1 und nicht z_0 ist.) Das Bandalphabet Γ umfasst neben dem Blank $\Box$ zumindest noch die Symbole 0, 1 und #, denn diese Symbole werden benötigt, um die k–stellige Eingabekodierung

$$bin(n_1)\#bin(n_2)\#\ldots\#bin(n_k)$$

auf dem Band darzustellen. (Streng genommen wird das Zeichen # im Fall $k = 1$ nicht benötigt, aber wir können es — sollte es fehlen — trotzdem einfach zum Bandalphabet hinzufügen, auch wenn es dann durch M nicht benutzt wird.) Evtl. enthält Γ auch noch weitere Symbole, und es sei $b := |\Gamma|$ die Gesamtanzahl aller Bandsymbole von M. Wir können den Inhalt der endliche Menge $\Gamma = \{a_0, a_1, \ldots, a_{b-1}\}$ dann so durchnummerieren, dass $a_0 = \Box$, $a_1 = 0$, $a_2 = 1$, und $a_3 = \#$ gilt (die Zuordnung der restlichen Symbole spielt keine Rolle).

Weiter können wir annehmen, dass M nur einen Endzustand besitzt. Denn bei mehreren Endzuständen kann man einen neuen Endzustand z einführen sowie Übergänge von den bisherigen Endzuständen nach z definieren, wobei die neuen Übergänge das aktuelle Symbol über dem Schreib–Lese–Kopf nicht verändern sollen. Die so modifizierte Turingmaschine ist nach wie vor deterministisch, da gemäß unserer Konvention keiner der bisherigen Endzustände zuvor einen Übergang besaß. Natürlich wird durch die beschriebene Änderung das Berechnungsverhalten von M nicht geändert.

Schließlich verlangen wir noch, dass der Startzustand nicht mit dem Endzustand identisch ist (eine solche TM würde immer sofort halten und dabei die Funktion $f : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ mit $f(n) := n$ berechnen). Auch dies kann durch die oben beschriebene Hinzunahme eines neuen Endzustands erzwungen werden, da sich der neue Endzustand dann natürlich von dem schon vorher vorhandenen Startzustand unterscheidet. Mit diesen Annahmen können wir alle Zustände $z_0, z_1, \ldots, z_\ell$ aus Z so durchnummerieren, dass der Startzustand z_1 sowie der Endzustand z_0 ist. Es wird in Kürze klar werden, warum wir auf z_0 als Endzustand bestehen.

Wir konzipieren jetzt ein Programm mit $k+8$ Variablen $\mathtt{x_0}$, $\mathtt{x_1}$, …, $\mathtt{x_{k+7}}$, welches genau den Ablauf von M simuliert. Die Idee ist dabei, jede Konfiguration $\alpha z_r \beta \in \Gamma^* Z \Gamma^*$ von M in den drei Variablen $\mathtt{x_{k+1}}$, $\mathtt{x_{k+2}}$ und $\mathtt{x_{k+3}}$ festzuhalten. Dies geschieht wie folgt:

- Den Index r des aktuellen Zustands z_r speichern wir in der Variablen $\mathtt{x_{k+1}}$. Um uns dies leichter merken zu können, bezeichnen wir diese Variable von nun an mit `zustand`.
- Der Inhalt $\alpha = a_{i_1} a_{i_2} \ldots a_{i_p}$ des Bandes links vom Schreib–Lese–Kopf wird durch eine Zahl in der Variablen $\mathtt{x_{k+2}}$ kodiert. Da Γ aus den Symbolen $a_0, a_1, \ldots, a_{b-1}$ besteht, liegen alle Indizes $i_1, i_2, \ldots, i_p$ zwischen 0 und $b-1$. Wir können die Indizes also als Ziffern einer Zahl zur Basis b auffassen, nämlich

  $$(i_1, i_2, \ldots, i_p)_b := \sum_{j=1}^{p} i_j b^{p-j} \quad .$$

  Lautet die aktuelle Konfiguration beispielsweise $0\#1\Box 0 z_4 1\Box\#0$, d.h. der links vom Kopf stehende Teil besteht aus $\alpha = 0\#1\Box 0$, so gilt wegen unserer Wahl $a_0 = \Box$,

$a_1 = 0$, $a_2 = 1$, und $a_3 = \#$ demnach $\alpha = a_1a_3a_2a_0a_1$. Also gilt $p = 5$ und $i_1 = 1$, $i_2 = 3$, $i_3 = 2$, $i_4 = 0$, und $i_5 = 1$. Für die kodierte Zahl ergibt sich somit der Wert

$$(i_1, i_2, \ldots, i_p)_b = (1, 3, 2, 0, 1)_b = 1 \cdot b^4 + 3 \cdot b^3 + 2 \cdot b^2 + 0 \cdot b^1 + 1 \cdot b^0 \ .$$

Falls nun z.B. $b = 5$ gilt (d.h. Γ enthält noch irgendein weiteres Symbol a_4, welches uns hier nicht weiter interessiert), so lautet die konkrete Zahl

$$(1, 3, 2, 0, 1)_5 = 1 \cdot 625 + 3 \cdot 125 + 2 \cdot 25 + 0 \cdot 5 + 1 \cdot 1 = 1051 \ .$$

Diese Art der Kodierung hat mehrere interessante Eigenschaften. Wegen $\square = a_0$ werden alle Blanks links von α auf dem Turingband implizit durch führende Nullen mitkodiert. Weiterhin lässt sich z.B. der Index i_p des unmittelbar links vom Kopf stehenden Symbols a_{i_p} leicht aus der Zahl $(i_1, i_2, \ldots, i_p)_b$ ermitteln. Hierzu braucht man lediglich den Wert

$$(i_1, i_2, \ldots, i_p)_b \bmod b$$

zu bestimmen. Auch Änderungen der Konfiguration können wir leicht nachvollziehen. Wenn sich z.B. der Kopf um eine Position nach links bewegt, so müssen wir statt $(i_1, i_2, \ldots, i_p)_b$ nun die Zahl $(i_1, i_2, \ldots, i_{p-1})_b$ kodieren, da das Zeichen a_{i_p} dann nicht mehr links vom Kopf, sondern unmittelbar darüber steht. Dies ist mit einer ganzzahligen Division ebenso einfach zu bewerkstelligen, denn es gilt die Beziehung

$$(i_1, i_2, \ldots, i_{p-1})_b = \lfloor (i_1, i_2, \ldots, i_p)_b / b \rfloor \ .$$

Die Variable `x_{k+2}` werden wir von jetzt an mit `links` bezeichnen.

- Den Inhalt $\beta = a_{j_0}a_{j_1} \ldots a_{j_q}$ des Bandes rechts vom Schreib–Lese–Kopf (inklusive des aktuellen Symbols) kodieren wir ganz ähnlich durch den Wert

$$(j_q, j_{q-1}, \ldots, j_0)_b \ = \ \sum_{i=0}^{q} j_i b^i \ .$$

Dieser Wert wird in der Variablen `x_{k+3}` gespeichert, die wir von jetzt an `rechts` nennen. Beachten Sie dabei die umgekehrte Kodierungsreihenfolge der Indizes. Der obige rechte Bandinhalt $\beta = 1\square\#0 = a_2a_0a_3a_1$ wird also durch den Wert

$$(1, 3, 0, 2)_5 = 1 \cdot 125 + 3 \cdot 25 + 0 \cdot 5 + 2 \cdot 1 = 202$$

kodiert. Durch die umgekehrte Reihenfolge wird sichergestellt, dass wir erneut leicht auf die Zeichen in der Umgebung des Kopfes zugreifen können. Der Index j_0 des aktuellen Symbols ergibt sich z.B. wieder durch den Ausdruck

$$(j_q, j_{q-1}, \ldots, j_0)_b \bmod b \ .$$

Die restlichen Variablen finden die folgende Verwendung:

- Die Variable `x_{k+4}` dient dazu, den konstanten Wert der Basis b zu speichern. Wir werden sie deshalb auch im Folgenden mit `b` bezeichnen.

- Die übrigen Variablen $\texttt{x}_{\texttt{k+5}}$, $\texttt{x}_{\texttt{k+6}}$ und $\texttt{x}_{\texttt{k+7}}$ speichern nur temporäre Werte und werden mit `temp1`, `temp2` und `temp3` angesprochen.

Das zu entwerfende Programm nimmt die Argumente $n_1, n_2, \ldots, n_k$ in den Variablen $\texttt{x}_1$, $\texttt{x}_2$, ..., $\texttt{x}_\texttt{k}$ entgegen. Als erstes muss daraus die entsprechende Startkonfiguration $z_1 bin(n_1)\#bin(n_2)\#\ldots\#bin(n_k)$ von M erzeugt werden. Betrachten Sie dazu das folgende Makro `SymbolLinksEinfügen(`i`)`, welches dem bislang simulierten Inhalt von `rechts` das Zeichen a_i voranstellt:

```
rechts := rechts * b;
rechts := rechts + i;
```

Beispielsweise wird die Zahl 202 (welche, wie oben gesehen, im Fall der Basis $b = 5$ einen rechten Bandinhalt $1\square\#0$ kodiert) durch einen Aufruf von `LinksEinfügen(1)` in die Zahl 1011 überführt. Diese Zahl entspricht dem Inhalt $01\square\#0$, also dem vorherigen Inhalt mit einer vorangestellten 0, da $a_1 = 0$ gilt.

Ein weiteres Makro `VariableLinksEinfügen(i)` erstellt eine binäre Kodierung von dem Inhalt der Variablen $\texttt{x}_\texttt{i}$ und stellt das Ergebnis dem bisherigen Inhalt von `rechts` voran:

```
IF (xi = 0) THEN
   SymbolLinksEinfügen(1);  // fügt eine 0 ein
ENDIF;
WHILE (xi > 0) DO
   temp1 := xi mod 2;
   temp1 := temp1 + 1;  // wandelt Ziffer in Symbolindex um
   SymbolLinksEinfügen(temp1);  // fügt die Ziffer (temp1 - 1) ein
   xi := xi / 2;
END;
```

Die `WHILE`–Schleife erzeugt die zu kodierenden Bits der Zahl von rechts nach links, d.h. bei der Zahl $13 = 1101_2$ wird zuerst eine 1, dann eine 0, dann eine 1, und zum Schluss nochmals eine 1 generiert. Weil diese Symbole von links an den bisherigen Inhalt von `rechts` vorangestellt werden, ergibt sich so der korrekte Code 1101 als neuer Präfix in der rechten Bandkodierung. Sinn der `IF`–Abfrage am Anfang ist es, ein einzelnes Symbol 0 zu erzeugen, falls die Zahl Null kodiert werden soll, denn die `WHILE`–Schleife wird in diesem Fall komplett übersprungen.

Nun ist es leicht, die gewünschte Startkonfiguration $z_1 bin(n_1)\#bin(n_2)\#\ldots\#bin(n_k)$ zu erzeugen, nämlich mit diesem Code:

```
zustand := 1;
links := 0;
rechts := 0;
```

```
VariableLinksEinfügen(k);
SymbolLinksEinfügen(3);  // fügt das Trennzeichen # ein
VariableLinksEinfügen(k-1);
SymbolLinksEinfügen(3);
VariableLinksEinfügen(k-2);
SymbolLinksEinfügen(3);
...
SymbolLinksEinfügen(3);
VariableLinksEinfügen(1);
```

Die nächsten drei Makros simulieren die Auswirkungen der einzelnen Übergänge von M.

Das erste Makro namens `SimuliereKopfNeutral(`r`,`i`)` bildet das Verhalten von M im Fall eines Übergangs $\delta(\dots) = (z_r, a_i, N)$ nach:

```
zustand := r;  // wechselt den Zustand
temp1 := rechts mod b;  // ermittelt das alte Zeichen über dem Kopf
rechts := rechts - temp1;  // ersetzt das Zeichen durch ein Blank
rechts := rechts + i;  // ersetzt das Blank durch das neue Zeichen
```

Im Fall eines Übergangs $\delta(\dots) = (z_r, a_i, R)$ wird der Kopf nach rechts bewegt. Das zugehörige Makro `SimuliereKopfRechts(`r`,`i`)` ist zu dem vorherigen recht ähnlich:

```
zustand := r;  // wechselt den Zustand
rechts := rechts / b;  // bewegt den Kopf nach rechts
links := links * b;  // fügt links vom Kopf ein neues Blank ein
links := links + i;  // ersetzt das Blank durch das neue Zeichen
```

Wird dagegen der Kopf wegen eines Übergangs $\delta(\dots) = (z_r, a_i, L)$ nach links bewegt, so ist das zugehörige Makro `SimuliereKopfLinks(`r`,`i`)` ein wenig komplizierter, weil ein Symbol separat von `links` nach `rechts` kopiert werden muss:

```
zustand := r;  // wechselt den Zustand
temp1 := rechts mod b;  // ermittelt das alte Zeichen über dem Kopf
rechts := rechts - temp1;  // ersetzt das Zeichen durch ein Blank
rechts := rechts + i;  // ersetzt das Blank durch das neue Zeichen
temp1 := links mod b;  // ermittelt das Zeichen links vom Kopf
links := links / b;  // bewegt den Kopf nach links
rechts := rechts * b;  // fügt über dem Kopf ein neues Blank ein
rechts := rechts + temp1;  // setzt das kopierte Zeichen ein
```

Nun können wir das Programm für die eigentliche Simulation von M angeben. Es besteht

im Wesentlichen aus einer WHILE–Schleife der Form

```
WHILE (zustand > 0) DO
   temp2 := zustand;
   temp3 := rechts mod b;  // ermittelt das aktuelle Symbol
   ...
END;
```

wobei in dem mit ... angedeuteten Bereich jeder mögliche Übergang

$$\delta(z_\ell, a_j) = (z_r, a_i, x)$$

mit $x \in \{L, N, R\}$ durch eine Codeblock der Form

```
IF (temp2 = ℓ) THEN
   IF (temp3 = j) THEN
      SimuliereKopfXXX(r,i);
   ENDIF;
ENDIF;
```

simuliert wird. Dabei ist SimuliereKopfXXX entsprechend durch SimuliereKopfLinks ($x = L$), SimuliereKopfNeutral ($x = N$) oder SimuliereKopfRechts ($x = R$) zu ersetzen. Sind in δ also z.B. 27 mögliche Übergänge definiert, so stehen 27 entsprechende Codeblöcke hintereinander in dem Rumpf der obigen WHILE–Schleife. Pro Schleifendurchlauf kommt nur bei genau einem dieser Blöcke das SimuliereKopfXXX–Makro zur Ausführung. Denn aufgrund der „disjunkten“ IF–Abfragen kann es immer nur einen zutreffenden Fall geben. Ist ein Übergang nicht definiert, so trifft überhaupt kein Fall zu, und das Programm bleibt in der Schleife hängen.

Die WHILE–Schleife wird verlassen, sobald der Zustand z_0 (also der Endzustand) erreicht wird. Jetzt ist auch der Grund für die Wahl von 0 als Index für den Endzustand ersichtlich, denn aufgrund der Syntax von WHILE–Schleifen kann eine Variable nur mit 0 verglichen werden.

Es verbleibt, den auf dem Band binär kodierten Funktionswert zurück in eine Zahl zu konvertieren. Da sich der Kopf unter dem ersten führenden Bit der kodierten Zahl befindet, müssen wir also nur die Variable rechts entsprechend auswerten. Falls z.B. rechts den Wert 287 enthält, so entspricht dies der Kodierung

$$287 = 2 \cdot 125 + 1 \cdot 25 + 2 \cdot 5 + 2 \cdot 1 = (2, 1, 2, 2)_5 \ ,$$

also der Zeichenfolge $a_2 a_2 a_1 a_2 = 1101$ und damit der Zahl 13. Die Umwandlung in die Zahl 13 lässt sich dabei besonders einfach durch das sogenannte *Horner–Schema*[1]

$$1101_2 = (((1 \cdot 2) + 1) \cdot 2 + 0) \cdot 2 + 1 = 13$$

[1] William G. Horner, *1786 Bristol, Vereinigtes Königreich, †1837 Bath, Vereinigtes Königreich, britischer Mathematiker.

bewerkstelligen. Allgemein führt deshalb der folgende Programmcode zum Ziel:

```
x0 := 0;
WHILE (rechts > 0) DO
   temp1 := rechts mod b;  // ermittelt das aktuelle Symbol
   temp1 := temp1 - 1;  // wandelt Symbolindex in Ziffer um
   rechts := rechts / b;  // bewegt den Kopf nach rechts
   x0 := x0 * 2;  // wendet Horner-Schema an
   x0 := x0 + temp1;
END;
```

Setzt man nun das Gesamtprogramm aus den skizzierten Teilstücken zusammen, so nimmt es die Anfangswerte $n_1, n_2, \ldots, n_k$ in den Variablen $\mathtt{x}_1$, $\mathtt{x}_2$, …, $\mathtt{x}_k$ entgegen und berechnet daraus in $\mathtt{x}_0$ den Funktionswert $f(n_1, n_2, \ldots, n_k)$. Falls $f(n_1, n_2, \ldots, n_k)$ nicht definiert ist, so hält die Turingmaschine M nicht an oder endet in einem undefinierten Übergang (vgl. Def. 4.2.9 auf S. 124). In beiden Fällen bleibt das WHILE–Programm in der Hauptschleife hängen und läuft deshalb unendlich lange. Dies steht mit dem WHILE–Berechenbarkeitsbegriff (vgl. Def. 4.4.3 auf S. 132) in Einklang. Also berechnet das Programm genau die Funktion f, und damit ist f WHILE–berechenbar. □

Wir sehen damit insgesamt, dass Turingmaschinen und WHILE–Programme genau die gleichen Funktionen berechnen können. Auch Java–Programme können keine anderen Funktionen beschreiben. Denn einerseits kann man sicherlich jedes WHILE–Programm durch ein passendes Java–Programm simulieren, da man die wenigen grundlegenden Befehle der WHILE-Programme direkt umsetzen kann. Andererseits haben wir schon gesehen, dass diese grundlegenden Befehle ausreichen, um nach und nach immer kompliziertere Funktionen aufzustellen. Wir können z.B. die Sinusfunktion durch ein WHILE–Programm berechnen, denn mit Hilfe der zugehörigen *Taylorreihe*[1]

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

kann man die Funktionswerte ausschließlich durch die Anwendung von Grundrechenarten ermitteln — diese stehen uns aber gemäß den Beispielen 4.4.4 ff. in WHILE–Programmen alle zur Verfügung. Auch Speicher in Form von Feldern können wir uns beliebig verschaffen. Die Simulation z.B. eines int–Feldes können wir wie im Beweis von Satz 4.4.8 durch die Kodierung einer (sehr großen) Zahl bewerkstelligen, wobei jede Ziffer den Inhalt einer Zelle des Feldes darstellt. Analysiert man alle sich durch Java bietenden Möglichkeiten weiter, so erkennt man, dass sich alle Konzepte auch durch WHILE–Programmkonstrukte beschreiben lassen.

Es hat in der Vergangenheit viele verschiedene Vorschläge gegeben, den „intuitiven“ Berechenbarkeitsbegriff formal zu fassen. Später stellte sich heraus, dass alle diese Ansätze

[1] Brook Taylor, *1685 Edmonton, Vereinigtes Königreich, †1731 London, britischer Mathematiker, ab 1712 Mitglied der Royal Society.

untereinander äquivalent sind. Dies führte zu der Überzeugung, dass man damit den intuitiven Berechenbarkeitsbegriff formal erfasst hat:

4.4.9 Satz (*Church[1]–Turing–These*) Die durch die formale Definition der

- Turing–Berechenbarkeit
- `WHILE`–Berechenbarkeit
- Java–Berechenbarkeit
- ...

erfassten Funktionen stimmen genau mit der Klasse der intuitiv berechenbaren Funktionen überein. Insbesondere braucht man nicht mehr zwischen „Java–Berechenbarkeit", „Turing–Berechenbarkeit" usw. zu unterscheiden. Es reicht aus, eine Funktion als berechenbar oder nicht berechenbar zu klassifizieren.

Beweis: Die Äquivalenz der genannten Berechenbarkeitsbegriffe haben wir uns in den Sätzen 4.4.7 und 4.4.8 sowie in der anschließenden Argumentation klar gemacht. Die „eigentliche" Church–Turing–These behauptet weiter die Übereinstimmung dieser formal berechenbaren Funktionen mit den intuitiv berechenbaren. Dies kann jedoch **nicht** bewiesen werden, denn „intuitiv berechenbar" ist keine formale Spezifikation eines Berechenbarkeitsbegriffs. □

4.5. Unentscheidbarkeit

Im letzten Abschnitt wurde gezeigt, dass moderne Computer genauso viel (oder pessimistischer ausgedrückt: genauso *wenig*) wie einfache Turingmaschinen berechnen können, ganz gleich, mit welcher noch so leistungsfähigen Hardware sie ausgestattet sind. Es ist deshalb nicht überraschend, dass bestimmte Probleme vollständig unlösbar sind. Auf diese kommen wir nun zu sprechen.

Jeder von Ihnen hat sicherlich schon einmal ein Programm in z.B. Java erstellt, welches aufgrund eines Programmfehlers „hängen blieb" und nicht mehr anhielt, z.B. weil man die Abbruchbedingung einer Programmschleife falsch formuliert hatte. Man könnte auf die Idee kommen, sich z.B. für seine Entwicklungsumgebung ein entsprechendes Analysetool zu installieren. Dieses soll das aktuelle Programm im Editor vorab daraufhin untersuchen, ob es nach dem Start wieder anhalten würde oder nicht.

Den Text im Editor (oder allgemeiner den Programmcode von beliebigen Klassen, Modulen usw.) kann man sich als einfache Zeichenketten vorstellen. Wenn man nämlich aus einem Java–Programm die an sich unwichtigen Zeilenumbrüche, Einrückungen usw.

[1] Alonzo Church, *1903 Washington D.C., †1995 Hudson, Ohio, US–amerikanischer Mathematiker und Logiker, 1929 Professor für Mathematik in Princeton und Los Angeles.

einmal herausnimmt, so bleibt von dem Quelltext nur eine solche (wenn auch lange) Zeichenkette übrig. Aus z.B.

```
public class Test {

   public static void main (String[] args) {
      int x=8;
   }
}
```

wird so

```
public class Test {public static void main (String[] args) {int x=8;}}
```

Ein passendes Analysetool könnte man sich dann in Form einer Java–Methode vorstellen, die ein zu untersuchendes Programm als `String`–Parameter entgegen nimmt:

```
public static boolean isHaltingProgram(String code) {
   ...
}
```

Für den Parameter

```
public class Test {public static void main (String[] args) {int x=8;}}
```

erwarten wir dann den Rückgabewert `true`, bei

```
public class Test {public static void main (String[] args) {for(;;);}}
```

dagegen den Wert `false`, und bei

```
Die Prinzessin warf die goldenen Kugel in einen Brunnen.
```

ebenfalls den Wert `false`, da es sich hier ja noch nicht einmal um ein Java–Programm handelt, sicherlich also auch nicht um ein anhaltendes Programm.

Kann eine solche Analysemethode programmiert werden? Die Antwort ist nein — und der Grund dafür ist das folgende Programm, welches auf den ersten Blick mit dem Problem scheinbar gar nichts zu tun hat:

```
1   public class ReproduceOwnProgramCode {
2
3      public static String returnOwnProgramCode() {
4         String x = "public class ReproduceOwnProgramCode {\n"
5            + "  \n"
6            + "  public static String returnOwnProgramCode() {\n"
7            + "    String x = \";\n"
8            + "    String y = x.substring(0, 106);\n"
9            + "    for (char c : x.toCharArray()) {\n"
10           + "      if (c == '\\\"' || c == '\\\\')\n"
11           + "        y += '\\\\';\n"
12           + "      if (c == '\\n')\n"
13           + "        y += \"\\\\n\\\"\\n      + \\\"\";\n"
14           + "      else y += c;\n"
15           + "    }\n"
16           + "    return y + x.substring(105, x.length());\n"
17           + "  }\n"
18           + "  \n"
19           + "  public static void main(String[] args) {\n"
20           + "    System.out.print(returnOwnProgramCode());\n"
21           + "  }\n"
22           + "}";
23        String y = x.substring(0, 106);
24        for (char c : x.toCharArray()) {
25           if (c == '\"' || c == '\\')
26              y += '\\';
27           if (c == '\n')
28              y += "\\n\"\n      + \"";
29           else y += c;
30        }
31        return y + x.substring(105, x.length());
32     }
33
34     public static void main(String[] args) {
35        System.out.print(returnOwnProgramCode());
36     }
37  }
```

Dieses Programm bringt das Kunststück zustande, *exakt seinen eigenen Programmcode* auf dem Bildschirm auszugeben.

Es funktioniert eigentlich ganz einfach:

- Die von der Methode `returnOwnProgramCode` zurückgegebene Zeichenkette setzt

sich aus drei Teilen zusammen, nämlich aus den ersten 106 Zeichen der Zeichenkette `x` (Zeile 23), dann aus bestimmten Zeichen, die in der `for`–Schleife (Zeile 24–30) angehängt werden, und schließlich aus den Suffix der Zeichenkette `x`, beginnend ab Position 105 (Zeile 31).

- Das Zeichen an der 105–ten Stelle ist das Anführungszeichen, welches innerhalb der Zeichenkette `x` in Zeile 7 des Programmcodes auftaucht (nachfolgend unterstrichen dargestellt):

```
+ "    String x = \";\n"
```

- Wir stellen uns nun für einen Moment vor, dass die `for`–Schleife übersprungen wird. Aus dem Inhalt der Zeichenkette `x` ist dann leicht abzulesen, dass die verbleibenden Zeilen 23 und 31 folgende Zeichenkette erzeugen würden (das Anführungszeichen an der Position 105 wird dabei doppelt ausgegeben):

```
public class ReproduceOwnProgramCode {

   public static String returnOwnProgramCode() {
      String x = "";
      String y = x.substring(0, 106);
      for (char c : x.toCharArray()) {
         if (c == '\"' || c == '\\')
            y += '\\';
         if (c == '\n')
            y += "\\n\"\n      + \"";
         else y += c;
      }
      return y + x.substring(105, x.length());
   }

   public static void main(String[] args) {
      System.out.print(returnOwnProgramCode());
   }
}
```

Es stimmt also soweit schon alles — nur die Zeichenkette `x` ist noch leer.

Um den noch fehlenden Inhalt der Zeichenkette `x` zu erzeugen, geht die `for`–Schleife nacheinander alle Zeichen von `x` durch und generiert daraus genau die Darstellung, die im Programmcode für die Kodierung der Zeichenkette benötigt wird. Jedem Anführungszeichen und jedem Backslash muss also gemäß den Java–Kodierrichtlinien zuerst ein Backslash voran gestellt werden (Zeilen 25, 26 und 29), und jeder Zeilenumbruch wird durch einen Backslash und ein „n“ kodiert. Außerdem folgt jedem Zeilenumbruch noch ein weiterer Teil der Zeichenkette von `x`, der im Programmcode entsprechend tief eingerückt und mit einem Pluszeichen sowie einem weiteren Anführungszeichen eingeleitet wird (Zeile 27 und 28).

Somit ist klar, dass das vollständige Programm genau seinen eigenen Programmcode erzeugt. Beispiele für solche sog. *Quines* lassen sich auch für andere gängige Programmiersprachen angeben [30]. Wir sehen aus dieser Konstruktion aber noch mehr:

4.5.1 Satz In jedem Programm können wir o.B.d.A. an jeder beliebigen Stelle eine Methode `returnOwnProgramCode` aufrufen, die genau den eigenen kompletten Programmcode zurück liefert.

Beweis: Angenommen, wir haben ein solches Programm erstellt, welches nach Belieben die besagte Methode `returnOwnProgramCode` verwendet und deshalb an diesen Stellen noch entsprechende Compiler–Fehler („unbekannte Methode“) verursacht. Dann ändern wir das Programm in vier Schritten wie folgt ab:

- Wir fügen in das Programm an beliebiger Position die folgende Methode ein:

```
public static String returnOwnProgramCode() {
   String x = "";
   String y = x.substring(0, 106);
   for (char c : x.toCharArray()) {
      if (c == '\"' || c == '\\')
         y += '\\';
      if (c == '\n')
         y += "\\n\"\n      + \"";
      else y += c;
   }
   return y + x.substring(105, x.length());
}
```

- Von dem dann im Editor stehenden Programm P erstellen wir (z.B. in einem anderen Editorfenster) eine Zeichenketten–Konstante `String x = "..."`, die genau den Programmcode P erzeugt (inkl. aller Zeilenumbrüche usw.).
- Mit dieser Zeichenkette ersetzen wir die Zeile `String x = "";` innerhalb der Methode `returnOwnProgramCode`.
- Abschließend passen wir die Zahlen `105` und `106` noch an die dann aktuelle Startposition des Anführungszeichens der Zeichenketten–Konstanten `x` an.

Genau so (mit einer elementaren `main`–Methode als einzigem weiteren Programmcode) ist auch das einführende Beispiel entstanden. □

Nun können wir die eingangs aufgeworfene Frage bzgl. der Existenz einer Analysemethode für das sog. *Halteproblem für Java* negativ beantworten:

4.5.2 Satz Das Halteproblem für Java ist *algorithmisch unentscheidbar*, d.h. man kann sich diese Frage nicht durch ein anderes Programm automatisiert beantworten lassen.

Beweis: Angenommen, eine solche Analysemethode würde existieren. Wir haben dann den folgenden Zusammenhang:

$$\texttt{isHaltingProgram(code)} = \texttt{true}$$
$$\iff \text{das Java–Programm } \texttt{code} \text{ hält an}$$

Durch Ergänzung mit der nachfolgenden `main`–Methode (die wiederum die Methode `returnOwnProgramCode` verwendet, was nach Satz 4.5.1 aber erlaubt ist) erhalten wir somit ein vollständiges Programm:

```
public static void main(String[] args) {
   if (isHaltingProgram(returnOwnProgramCode())) {
      for (;;);  // Endlosschleife
   }
}
```

Hält dieses Programm an? Falls ja, so kann es nicht in die Endlosschleife geraten sein, d.h. die ausgewertete `if`–Bedingung war falsch — was nicht sein kann, denn dann dürfte das Programm ja gerade nicht anhalten! Wenn das Programm aber nicht anhält, die Endlosscheife also ausgeführt wird, so müßte die ausgewertete `if`–Bedingung wahr gewesen sein — das Programm hätte also doch anhalten müssen. Das Programm hält also genau dann an, wenn es nicht anhält, was ein perfekter Widerspruch ist.

Was haben wir falsch gemacht? Eigentlich nichts — bis auf die Tatsache, dass wir von der Existenz der Routine `isHaltingProgram` ausgegangen sind. Eine derart „mächtige“ Methode kann also in Wirklichkeit nicht existieren. Damit ist der Beweis der Unentscheidbarkeit erbracht. □

Ein weiteres interessantes Problem wird durch den folgenden Quellcode motiviert:

```
public static void main(String[] args) {
   int i, j = 1;

   if (j == 1) i = 1;
   System.out.println(i);
}
```

Dieses Java–Programm ist leicht zu analysieren. Man könnte erwarten, dass es einfach eine 1 ausgibt. Tatsächlich kann man es aber gar nicht erst starten, denn der Java–Compiler bricht bei der Ausgabezeile `System.out.println(i)` mit der Fehlermeldung

The local variable `i` may not have been initialized.

ab. Es wird also vermutet, dass evtl. auf die Variable `i` lesend zugegriffen wird, noch bevor dort etwas gespeichert wurde. Wie man leicht sieht, ist diese Vermutung falsch, denn die Variable wird zwei Zeilen darüber korrekt initialisiert.

Allerdings steht der Java–Compiler hier auf verlorenem Posten, denn es gilt:

4.5.3 Satz Das sog. *Variableninitialisierungsproblem für Java* ist unentscheidbar, d.h. ein Computer kann nicht ermitteln, ob in einem Programm eine bestimmte Variable jemals initialisiert (d.h. mit einem ersten Inhalt beschrieben) wird. Es gibt also keine Analyseroutine der Form

```
public static boolean testVarInit(String code, String varName) {
   ...
}
```

welche genau dann `true` zurück gibt, wenn die lokale Variable mit dem Namen `varName` innerhalb des Programms mit dem Programmtext `code` irgendwann initialisiert wird. (Wir gehen dabei o.B.d.A. davon aus, dass jeder Variablenname nur einmal vergeben wird, was sich z.B. durch Umbenennungen leicht erreichen lässt.)

Beweis: Angenommen, eine solche Methode existiert. Dann können wir diese zusammen mit der folgenden `main`–Methode zu einem vollständigen Programm ergänzen:

```
public static void main(String[] args) {
   int test;
   if (!testVarInit(returnOwnProgramCode(), "test")) {
      test = 1;
   }
}
```

Dann stehen wir aber wieder vor einem interessanten Problem. Es gilt nämlich

die Variable `test` wird initialisiert
$\iff$ der Variablen `test` wird der Wert 1 zugewiesen
$\iff$ die `if`–Anweisung wurde ausgeführt
$\iff$ `!testVarInit(returnOwnProgramCode(), "test")` ist `true`
$\iff$ `testVarInit(returnOwnProgramCode(), "test")` ist `false`
$\iff$ die Variable `test` wird *nicht* initialisiert

Aus diesem Widerspruch erkennen wird, dass auch die Methode `testVarInit` nicht existieren kann. Also muss auch das Variableninitialisierungsproblem unentscheidbar sein. □

Wir betrachten die Problematik der Unentscheidbarkeit nun in einem allgemeineren Rahmen. Zunächst machen wir uns klar, dass die besprochenen zwei Probleme formal als Sprachen definiert werden können. Java–Programme (ohne Zeilenumbrüche usw.) kann man nämlich wie bereits gesehen als Zeichenketten (also Wörter) über einem passend

gewählten Alphabet Σ ansehen (welches also alle benötigten Buchstaben, Ziffern, Klammern, usw. enthält). Man kann dann alle stoppenden Programme des Halteproblems als Sprache über Σ auffassen:

$$HP_{\text{Java}} := \{code \mid code \text{ ist ein terminierendes Java–Programm}\} \ .$$

Das Wort

```
public class Test {public static void main (String[] args) {int x=8;}}
```

ist also in der Sprache HP_{Java} enthalten, nicht jedoch

```
public class Test {public static void main (String[] args) {for(;;);}}
```

oder

```
Die Prinzessin warf die goldenen Kugel in einen Brunnen.
```

Das Variableninitialisierungsproblem für Java kann man analog durch

$$INIT_{\text{Java}} := \{code\#var \mid \text{der Java–Quelltext } code \text{ initialisiert die Variable } var\} \ .$$

beschreiben. (Das „#“–Symbol dient hierbei als Trennzeichen zwischen dem Quellcode und dem Variablennamen, und es soll weder in *code* noch in *var* vorkommen.) Folglich ist z.B.

```
public class X {public static void main (String[] args) {int x=8;}}#x
```

ein Wort aus der Sprache $INIT_{\text{Java}}$, nicht jedoch

```
public class X {public static void main (String[] args) {int x;}}#x
```

4.5.4 Definition Eine Sprache $L \subseteq \Sigma^*$ heißt *entscheidbar* oder *rekursiv*, wenn die *charakteristische Funktion* (oder auch *Indikatorfunktion*) $\chi_L : \Sigma^* \longrightarrow \{0, 1\}$ mit

$$\chi_L(x) := \begin{cases} 1 & \text{falls } x \in L \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist. Die beiden Werte 1 und 0 haben hier die Bedeutung von „wahr“ und „falsch“, d.h. bei χ_L handelt es sich im Prinzip um ein Prädikat bzw. (programmtechnisch gesehen) um eine Java–Methode mit einem `boolean`–Rückgabewert.

Wir haben bereits gesehen, dass für das Halte– bzw. Variableninitialisierungsproblem die zugehörigen charakteristischen Funktionen `isHaltingProgram` bzw. `testVarInit` nicht existieren. Also sind die Sprachen HP_{Java} und $INIT_{\text{Java}}$ auch gemäß der letzten Definition unentscheidbar.

Wir werden als Nächstes zeigen, dass beide genannten Sprachen aber zumindest *semi–entscheidbar* sind — dies ist eine abgeschwächte Form der Entscheidbarkeit.

4.5.5 Definition Eine Sprache $L \subseteq \Sigma^*$ heißt *semi–entscheidbar*, wenn die „halbe“ Indikatorfunktion $\chi'_L : \Sigma^* \longrightarrow \{1\}$ mit

$$\chi'_L(x) := \begin{cases} 1 & \text{falls } x \in L \\ \uparrow & \text{sonst} \end{cases}$$

berechenbar ist. Der Funktionswert 1 ist an sich völlig unwichtig (man hätte auch 0 oder einen sonstigen Wert nehmen können). Wichtig ist nur, dass bei der Berechnung von $\chi'_L(x)$ das zugehörige Programm nur genau im Fall $x \in L$ anhält.

4.5.6 Satz Die Probleme HP_{Java} und $INIT_{\text{Java}}$ sind semi–entscheidbar.

Beweis: Jedes der genannten Probleme kann „teilweise“ gelöst werden, indem man ein konkret vorliegendes Java–Programm Schritt für Schritt (ähnlich wie in einem Debugger) ausführt. Bei dem Halteproblem wird die Simulation solange durchgeführt, bis das simulierte Programm anhält. In diesem Fall stoppt die Simulation, und als Ausgabe wird eine 1 zurückgegeben. Falls das Java–Programm nun nicht anhält, läuft die Simulation unendlich lange weiter, und der berechnete Funktionswert ist im Endeffekt undefiniert — auch dies ist dann genau richtig.

Im Fall der Sprache $INIT_{\text{Java}}$ gehen wir ähnlich vor, brechen die Simulation jedoch mit der Ausgabe 1 ab, sobald ein Schreibzugriff auf die angegebene Variable stattfindet. Es kann durchaus wieder passieren, dass die Simulation endlos lange läuft, wenn nämlich die Variableninitialisierung tatsächlich nie stattfindet. Auch hier ist dieses Verhalten dann korrekt. Sollte die Simulation ohne Variableninitialisierung enden, können wir absichtlich eine Endlosschleife betreten, um die Berechnung des korrekten Funktionswertes $\uparrow$ („undefiniert“) sicherzustellen.

In beiden Fällen wird also jeweils genau die halbe charakteristische Funktion berechnet. Beachten Sie, dass man vorab entscheiden kann, ob überhaupt ein simulationsfähiges Programm vorliegt. Falls nicht, kann man explizit eine Endlosschleife betreten. □

Der für die Praxis bedeutende Unterschied zwischen der halben und der „ganzen“ Indikatorfunktion besteht in dem folgenden Problem. Wenn eine bestimmte Methode nur die halbe Indikatorfunktion realisiert und angesetzt auf eine bestimmte Eingabe $x \in \Sigma^*$ nach fünf Tagen immer noch läuft, so weiß man — gar nichts. Vielleicht hält die Methode in Kürze mit der Ausgabe 1 an, vielleicht läuft sie aber auch unendlich lange weiter. Man weiß also nicht, ob man auf die Frage „Ist x in L enthalten?“ jemals eine Antwort bekommt. Leider kann man dies auch nicht computerunterstützt analysieren, denn dabei handelt es sich ja gerade um das Halteproblem — und das ist unentscheidbar.

Wir führen noch einen weiteren Begriff ein, der die Semi–Entscheidbarkeit einer Sprache anders klassifiziert:

4.5.7 Definition Eine Sprache $L \subseteq \Sigma^*$ heißt *rekursiv aufzählbar*, wenn man ein Programm (z.B. in Java) schreiben kann, welches nach und nach alle Wörter aus L auflistet.

Man sagt dann, dass die Sprache L „aufgezählt“ wird. Beachten Sie, dass manche (oder sogar alle) Wörter aus L auch durchaus mehrfach in der Aufzählung vorkommen dürfen. Wichtig ist allein, dass jedes Wort aus L irgendwann mindestens einmal auftritt.

4.5.8 Beispiel Für jedes Alphabet Σ ist Σ^* rekursiv aufzählbar.

Beweis: Angenommen, das Alphabet Σ umfasst genau neun Zeichen $a_1, a_2, \ldots, a_9$. In diesem Fall können wir jedem nichtleeren Wort eine natürliche Zahl zuordnen, indem wir jedes im Wort beteiligte Symbol a_i durch die Ziffer i ersetzen und das Ergebnis als eine Zahl interpretieren. Aus dem Wort $a_5a_1a_9a_8$ wird so die Zahl 5198. Natürlich ergeben unterschiedliche Wörter auch unterschiedliche Zahlen.

Unser L aufzählendes Programm geht nun in einer unendlich lang laufenden Schleife jede Zahl aus $\mathbb{N}_0$ durch und gibt für jede solche Zahl n ein Wort aus Σ^* aus. Sofern in den Ziffern von n keine Nullen vorkommen, drehen wir dazu den obigen Prozess einfach um und erhalten z.B. aus der Zahl 3716 das Wort $a_3a_7a_1a_6$. Einer Zahl mit einer Null als Ziffer ordnen wir stattdessen das leere Wort zu. Das leere Wort wird dadurch natürlich noch viel häufiger aufgezählt (z.B. erzeugen alle Zahlen zwischen 10000 und 11110 das leere Wort, da jedesmal irgendwo mindestens eine Null vorkommt), dies ist jedoch nicht verboten.

Nun enthält Σ nicht unbedingt genau neun, sondern allgemein k Symbole $a_1, a_2, \ldots, a_k$. Dann gehen wir ganz genauso vor, interpretieren aber jede Zahl n als eine Zahl zur Basis $k+1$. Jede Ziffer stellt dann einen Wert zwischen 0 und k dar, und wir können die obige Zuordnung zu den einzelnen Symbolen analog vornehmen. □

Wir werden in den folgenden Sätzen nun zeigen, dass die Begriffe der rekursiven Aufzählbarkeit und der Semi–Entscheidbarkeit zusammen fallen.

4.5.9 Satz Ist eine Sprache L rekursiv aufzählbar, so ist L auch semi–entscheidbar.

Beweis: Nach Voraussetzung gilt es ein L aufzählendes Programm. Jedes ausgegebene Wort wird mit x verglichen, und bei Gleichheit wird das Ergebnis 1 als Resultat zurückgegeben. Diese Konstruktion ist korrekt, denn falls x in L enthalten ist, wird irgendwann x vom Programm aufgezählt, und das Programm hält mit der Ausgabe 1 an. Falls umgekehrt $x \notin L$ gilt, wird x niemals unter den ausgegebenen Wörtern auftauchen, und das Programm läuft unendlich lange. Also berechnet unser Programm genau die halbe charakteristische Funktion χ'_L von L. □

4.5.10 Satz Ist L semi–entscheidbar, so gilt $L = L(M)$ für eine passend gewählte deterministische Turingmaschine M.

Beweis: L ist semi–entscheidbar, also ist die halbe charakteristische Funktion χ'_L berechenbar. Da alle Berechenbarkeitsbegriffe von Java–Programmen, Turingmaschinen usw. zusammenfallen (vgl. Satz 4.4.9 auf Seite 143), muss es also eine DTM M geben, die

χ'_L berechnet. Für alle Wörter $x \in L$ hält sie in einem Endzustand an (und zwar mit der Ausgabe 1 auf dem Band, aber dies ist nebensächlich), und für alle Wörter $x \notin L$ läuft sie unendlich lange oder gerät in einen undefinierten Übergang, d.h. solche Wörter werden verworfen. Also akzeptiert die DTM M genau die Sprache L. □

4.5.11 Satz Ist $L = L(M)$ für eine deterministische Turingmaschine M, so ist L rekursiv aufzählbar.

Beweis: Betrachten Sie zunächst den folgenden Codeausschnitt aus einem Java–Programm:

```
int x = 0;
int y = 0;
while (true) {
   if (y == 0) {
      y = x + 1;
      x = 0;
   } else {
      x = x + 1;
      y = y - 1;
   }
}
```

Das Programm läuft unendlich lange, und man sieht leicht, dass die beiden Variablen `x` und `y` als Paar (x, y) notiert nacheinander die folgenden Werte annehmen (jeweils vor dem nächsten Schleifendurchlauf):

$$(0,0),\ (0,1),\ (1,0),\ (0,2),\ (1,1),\ (2,0),\ (0,3),\ (1,2),\ (2,1),\ (3,0),\ (0,4),\ (1,3),\ \ldots$$

Genauer betrachtet durchläuft die Folge systematisch alle Paare (x, y), bei denen die Summe $x + y$ erst 0, dann 1, dann 2, usw. beträgt:

$$\underbrace{(0,0)}_{\Sigma=0},\ \underbrace{(0,1),\ (1,0)}_{\Sigma=1},\ \underbrace{(0,2),\ (1,1),\ (2,0)}_{\Sigma=2},\ \underbrace{(0,3),\ (1,2),\ (2,1),\ (3,0)}_{\Sigma=3},\ \underbrace{(0,4),\ (1,3),\ \ldots}_{\Sigma=4}$$

Offenbar wird so jedes Paar aus $\mathbb{N}_0 \times \mathbb{N}_0$ genau einmal erzeugt. Wir vernachlässigen an dieser Stelle, dass Variablen vom Typ `int` nur einen begrenzten Wertebereich haben (dies könnte man z.B. mit der bereits früher erwähnten Java–Klasse `java.math.BigInteger` beheben).

Nun ändern wir das Programm geringfügig ab und führen die Schleife nicht unendlich oft, sondern nur genau n–mal aus, wobei n eine als Parameter vorgegebene Zahl ist:

```
int berechneX(int n) {
   int x = 0;
   int y = 0;
   for (; n > 0; n = n - 1) {
      if (y == 0) {
         y = x + 1;
         x = 0;
      } else {
         x = x + 1;
         y = y - 1;
      }
   }
   return x;
}
```

Genauso kann man sich auch eine `berechneY(int n)`–Methode schreiben, die y statt x zurückgibt.

Mit diesen beiden Funktionen kann man sich jedes Paar der obigen Folge gezielt „herauspicken“ und z.B. aus der Zahl $n = 7$ das zugehörige siebte Paar $(1, 2)$ berechnen (das erste Paar $(0, 0)$ in der Folge wird durch die Eingabe $n = 0$ erzeugt).

Nun haben wir alle Hilfsmittel zusammen, um zu zeigen, dass L rekursiv aufzählbar ist.

Für jede Zahl $n = 0, 1, 2, 3, \ldots$ (in einer unendlichen `for`–Schleife) führen wir folgende Schritte aus:

- Zunächst wird das n–te Paar (x, y) aus der obigen Folge generiert.
- Im Fall $x = 0$ werden alle folgenden Schritte übersprungen, d.h. es geht gleich mit der nächstes Iteration der `for`–Schleife weiter.
- Das Programm aus dem Beispiel 4.5.8, welches ganz Σ^* rekursiv aufzählt, wird gestartet und die Ausgabe von x Wörtern abgewartet (wir zählen auch schon aufgetretene Wörter nochmals mit). Das dann zuletzt ausgegebene x–te Wort bezeichnen wir mit w. Beachten Sie, dass Σ^* unendlich viele Wörter enthält und deshalb auch unendlich viele Wörter ausgegeben werden. Nach einer gewissen endlichen Zeit wird also auch das x–te Wort w erzeugt, ganz gleich, wie groß x im konkreten Fall auch sein mag.
- Anschließend wird das Verhalten der Turingmaschine M auf der Eingabe w simuliert. Dies ist z.B. durch ein Java–Programm leicht zu realisieren (für das Turing–Band kann man z.B. eine `ArrayList` verwenden). Die Simulation wird jedoch nach spätestens y vielen Schritten abgebrochen.
- Wenn M angesetzt auf w bis dahin akzeptiert, wird das Wort w ausgegeben. Wenn nicht, so passiert nichts weiter, und in beiden Fällen geht es mit der nächsten Iteration der `for`–Schleife weiter.

Wir zeigen einige Eigenschaften dieser Prozedur:

- Jeder der fünf Einzelschritte benötigt nur endlich viel Zeit, d.h. das Programm führt nach und nach die obige `for`–Schleife wirklich für alle $n = 0, 1, 2, 3, \ldots$ durch.
- Es werden nur Wörter aus L ausgegeben, da jedes ausgegebene Wort w zuvor von der Turingmaschine M akzeptiert worden sein muss.
- Andererseits gibt es aber auch für jedes Wort $w \in L$ ein passendes $n \in \mathbb{N}_0$, so dass die n–te Iteration der `for`–Schleife das Wort w erzeugt. Um dies zu sehen, betrachte man ein bestimmtes Wort $w \in L$. Die DTM M akzeptiert gerade genau die Sprache L, also auch das Wort w, und zwar in einer gewissen Anzahl von Schritten $y \in \mathbb{N}_0$. Das Wort w über Σ liegt ferner in Σ^* und wird deshalb vom Programm aus dem Beispiel 4.5.8 irgendwann als ein x–tes Wort aufgezählt. Nun muss man nur noch nachschlagen, als welches n–te Paar das Paar (x, y) in der obigen Folge vorkommt. Es ist klar, dass dann die n–te Iteration der `for`–Schleife genau w berechnet, da die soeben aufgeführten Überlegungen in der umgekehrten Reihenfolge wieder abgearbeitet werden.

Folglich werden alle Wörter aus L nach und nach aufgezählt. Also ist L wie behauptet rekursiv aufzählbar. □

Die bisherigen Resultate lassen sich wie folgt zusammenfassen:

4.5.12 Korollar Für eine Sprache $L \subseteq \Sigma^*$ sind äquivalent:

a) L ist rekursiv aufzählbar.

b) L ist semi–entscheidbar.

c) L wird von einer DTM akzeptiert.

d) L wird von einer NTM akzeptiert.

e) L ist von Typ 0.

Beweis: Die Sätze 4.5.9, 4.5.10 und 4.5.11 implizieren die zyklische Beweiskette

$$\text{a)} \Rightarrow \text{b)} \Rightarrow \text{c)} \Rightarrow \text{a)} \ ,$$

d.h. aus jeder der ersten drei Aussagen kann man jede andere folgern, indem man „im Kreis herumwandert“. Also sind die ersten drei Aussagen alle miteinander äquivalent. Nach Satz 4.2.8 (Seite 124) sind auch die untersten drei Aussagen miteinander äquivalent, d.h. insgesamt sind alle Aussagen gleichwertig. □

Aus den Sätzen 4.5.2, 4.5.3 und 4.5.6 ist uns bekannt, dass die Sprachen HP_{Java} und $INIT_{\text{Java}}$ zwar unentscheidbar, aber zumindest semi–entscheidbar sind. Es gibt aber auch unentscheidbare Sprachen, die noch nicht einmal semi–entschieden werden können. Aus dem folgenden Satz lässt sich ein entsprechendes Kriterium ableiten.

4.5.13 Satz L ist genau dann entscheidbar, wenn L und $\overline{L}$ semi–entscheidbar sind.

Beweis: Angenommen, L ist entscheidbar, d.h. die Indikatorfunktion

$$\chi_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ 0 & \text{sonst} \end{cases}$$

ist berechenbar, z.B. durch eine Turingmaschine M. Das Turingprogramm wird nun leicht modifiziert. Sobald die Turingmaschine M in einen Endzustand wechselt, prüft sie nach, ob auf dem Band eine 0 als geplante Ausgabe steht. Sollte dies der Fall sein, geht sie in eine Endlosschleife über, ansonsten hält sie wie ursprünglich geplant an. Die so modifizierte Turingmaschine berechnet offenbar

$$\chi'_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ \uparrow & \text{sonst} \end{cases}$$

und damit die halbe Indikatorfunktion χ'_L.
Die andere halbe Indikatorfunktion $\chi'_{\overline{L}}$ lautet

$$\chi'_{\overline{L}}(x) = \begin{cases} 1 & \text{falls } x \notin L \\ \uparrow & \text{sonst} \end{cases}$$

Auch diese lässt sich aus der charakteristischen Funktion für L gewinnen, indem man bei geplanter Ausgabe 0 eine 1 ausgibt und bei geplanter Ausgabe 1 in eine Endlosschleife übergeht. Wenn also L entscheidbar ist, so sind L und $\overline{L}$ semi–entscheidbar.

Nun seien umgekehrt L und $\overline{L}$ semi–entscheidbar und damit rekursiv aufzählbar. Es seien P_L und $P_{\overline{L}}$ zwei L bzw. $\overline{L}$ aufzählende Programme. Dann kann man die Indikatorfunktion χ_L von L leicht wie folgt berechnen:

- Angesetzt auf ein Eingabewort $x \in \Sigma^*$ wird solange abwechselnd ein Arbeitsschritt von P_L und $P_{\overline{L}}$ simuliert, bis ein Wort von einem der beiden Programme erzeugt wird.
- Ein solches Wort wird mit der Eingabe x vergleichen. Bei Gleichheit stoppt das Programm und gibt eine 1 aus, falls es sich um eine Ausgabe von P_L handelte, ansonsten eine 0 (also bei Gleichheit mit einer Ausgabe von $P_{\overline{L}}$).
- Sollte das Wort x nicht ausgegeben worden sein, wird wieder mit der wechselseitigen Simulation der beiden Programme fortgefahren.

Diese Prozedur stoppt auf jeden Fall, da das Wort x entweder in L oder im Komplement $\overline{L}$ liegt, d.h. x wird irgendwann entweder von P_L oder von $P_{\overline{L}}$ aufgezählt. Somit ist klar, dass das skizzierte Programm die charakteristische Funktion χ_L korrekt berechnet. □

4.5.14 Korollar Ist eine Sprache zwar semi–entscheidbar, aber nicht rekursiv, so kann das Komplement der Sprache nicht semi–entscheidbar sein.

Beweis: Wäre auch das Komplement der Sprache semi–entscheidbar, so wäre die Ausgangssprache nach Satz 4.5.13 sogar rekursiv, was der Voraussetzung widerspricht. □

Damit ist klar:

4.5.15 Korollar Die komplementären Probleme $\overline{HP}_{\mathsf{Java}}$ und $\overline{INIT}_{\mathsf{Java}}$ lassen sich nicht rekursiv aufzählen.

Beweis: Nach den Sätzen 4.5.2, 4.5.3 und 4.5.6 sind die beiden Sprachen HP_{Java} und $INIT_{\mathsf{Java}}$ semi–entscheidbar, aber nicht rekursiv. □

Die Unentscheidbarkeit der Sprachen HP_{Java} und $INIT_{\mathsf{Java}}$ wurde „direkt“ mit Hilfe der Methode `returnOwnProgramCode` gezeigt. Es gibt jedoch auch noch eine andere Beweistechnik, die sich sogenannter *Reduktionen* bedient. Hierbei wird gezeigt, dass ein bestimmtes Problem mindestens so „schwer“ zu lösen ist wie ein anderes bekanntes Problem. Wenn nun das bekannte Problem bereits unentscheidbar ist, so muss das „schwerere“ Problem erst recht unentscheidbar sein.

4.5.16 Definition Es seien A und B zwei Sprachen über einem Alphabet Σ. Dann heißt A *auf* B *reduzierbar*, falls es eine totale und berechenbare Funktion $f : \Sigma^* \longrightarrow \Sigma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt:

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

Man schreibt dafür dann auch $A \leq B$.

Als ein Beispiel für eine Reduktion können wir festhalten:

4.5.17 Lemma Wir betrachten das *Halteproblem für deterministische Turingmaschinen*

$$HP_{\mathsf{DTM}} := \{M\#w \mid M \text{ ist die Beschreibung einer DTM, die das Wort } w \text{ akzeptiert}\} \ .$$

(Die Beschreibung einer DTM kann z.B. aus der ganz normalen Niederschrift des 7–Tupels $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ mit allen seinen Komponenten bestehen, sofern man das dazu benötigte Alphabet mit den zugehörigen Buchstaben, Indizes, usw. passend wählt.) Dann gilt für jede rekursiv aufzählbare Sprache L stets $L \leq HP_{\mathsf{DTM}}$.

Beweis: Sei $L \subseteq \Sigma^*$ rekursiv aufzählbar. Nach Korollar 4.5.12 wird L auch von einer DTM M akzeptiert. Die Frage

„Ist w in L enthalten?“

lässt sich demnach gleichwertig als

„Ist $M\#w$ in HP_{DTM} enthalten?“

formulieren. Folglich stellt die Abbildung f mit

$$f(w) := M\#w$$

eine Reduktion von L auf HP_{DTM} dar (das Wort w wird einfach an die Zeichenkette $M\#$ angehängt). □

Die Idee hinter Reduktionen wird durch das folgende Resultat klar:

4.5.18 Satz Es seien $A, B \subseteq \Sigma^*$ zwei Sprachen mit $A \leq B$. Ist dann B entscheidbar, so ist auch A entscheidbar.

Beweis: A ist auf B mittels einer Funktion $f : \Sigma^* \longrightarrow \Sigma^*$ reduzierbar. Um zu entscheiden, ob ein Wort $x \in \Sigma^*$ in A enthalten ist, berechnet man zunächst $f(x)$ und wendet das Ergebnis auf die charakteristische Funktion χ_B an, die nach Voraussetzung existiert. Dann gilt:

- Wenn x tatsächlich in A enthalten ist, so gilt aufgrund der Reduktionseigenschaft $f(x) \in B$, d.h. die Indikatorfunktion χ_B gibt für das Argument $f(x)$ den Wert 1 zurück.
- Falls umgekehrt $x \notin A$ gilt, so folgt analog $f(x) \notin B$ und somit $\chi_B(f(x)) = 0$.
- Die Hintereinanderausführung von f und χ_B ist eine totale Funktion, da dies auch auf f und χ_B zutrifft.

Also haben wir genau die charakteristische Funktion χ_A von A konstruiert. Somit ist auch A entscheidbar. □

Zur Anwendung kommt dieser Satz fast immer in seiner kontrapositorischen Form:

4.5.19 Korollar Lässt sich eine unentscheidbare Sprache A auf eine andere Sprache B reduzieren, so ist auch diese unentscheidbar.

Beweis: Wäre B entscheidbar, so würde daraus nach Satz 4.5.18 die Entscheidbarkeit von A folgen, was der Voraussetzung widerspricht. □

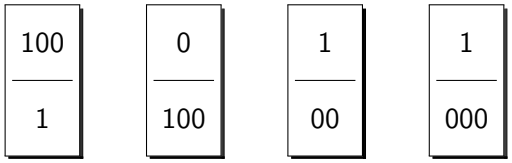
Man verwendet also das Wissen über die Unentscheidbarkeit eines Problems A zusammen mit einer Reduktion von A auf B, um die Unentscheidbarkeit von B zu zeigen.

4.5.20 Beispiel Das Halteproblem für deterministische Turingmaschinen HP_{DTM} aus Lemma 4.5.17 ist unentscheidbar. Zum Beweis nehme man einfach eine beliebige rekursiv aufzählbare, aber nicht rekursive Sprache L zur Hand, z.B. HP_{Java} oder $INIT_{\mathsf{Java}}$. Gemäß Lemma 4.5.17 gilt dann $L \leq HP_{\mathsf{DTM}}$, und aus Korollar 4.5.19 folgt die Behauptung. □

4.6. Das Postsche Korrespondenzproblem

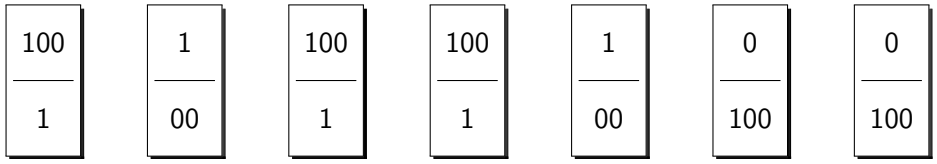
Als nächstes untersuchen wir ein interessantes kombinatorischen Problem, welches sich später negativ auf diverse Entscheidbarkeitsfragen bei kontextfreien Grammtiken auswirken wird.

Das Problem benutzt „Dominosteine“, auf denen jedoch keine Zahlen, sondern Paare von Wörtern aufgedruckt sind. Gilt z.B. $\Sigma = \{0, 1\}$, so wären diese Steine denkbar:



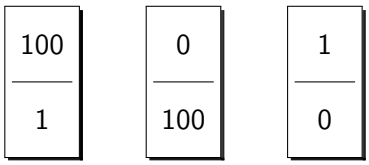
Ziel des *Postschen*[1] *Korrespondenzproblems* (engl. „*Post correspondence problem*“ oder kurz *PCP*) ist es, eine Folge von Dominosteinen so mit ihren langen Kanten aneinander zu legen, dass sich oben und unten in der Konkatenation die gleiche Zeichenkette ergibt. Jeder Stein darf dabei mehrfach verwendet werden (muss aber nicht). Es ist allerdings nicht erlaubt, einen Stein auf den Kopf zu drehen und damit ein für oben vorgesehenes Teilwort unten zu verwenden oder umgekehrt.

Mit den obigen Steinen ist z.B. diese Lösung möglich:



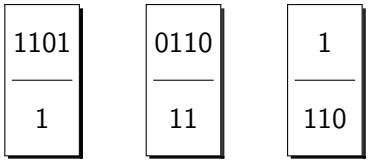
Oben und unten ergibt sich dabei die Zeichenkette 1001100100100. Der erste Stein kommt dabei dreimal, der zweite und dritte Stein jeweils zweimal, und der vierte Stein überhaupt nicht zum Einsatz.

Gibt es bei einem vorgegebenen Satz von Dominosteinen immer eine Lösung? Dies ist ein schwierigeres Problem, als es zunächst vielleicht den Anschein hat. Für die drei harmlos aussehenden Dominosteine



sind z.B. Lösungen möglich, aber die beiden kürzesten bestehen aus je 75 Steinen! Bei dem Satz

[1] Emil L. Post, *1897 Augustów, Polen, †1954 New York, polnisch–US–amerikanischer Mathematiker und Logiker, ab 1927 Professor für Mathematik an der Columbia University in New York.



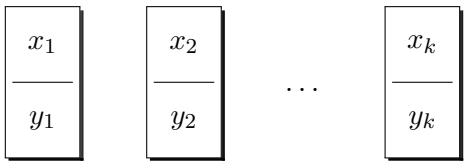
sind sogar schon 252 Steine für eine Lösung fällig. Die Situation ist jedoch noch schlimmer. Es ist nämlich hoffnungslos, mit einem Programm nach einer Lösung zu suchen:

4.6.1 Satz Das *PCP* ist unentscheidbar.

Um dies zu beweisen, benötigen wir zunächst einen hilfreichen Trick:

4.6.2 Lemma O.B.d.A. kann man bei einer konkreten *PCP*–Aufgabenstellung verlangen, dass die Lösung mit einem vorab bestimmten Stein beginnen soll. Gemeint ist hiermit, dass man zu jeder solchen „speziellen“ *PCP*–Instanz eine gleichwertige „gewöhnliche“ *PCP*–Instanz angeben kann, die ohne Nennung dieses Startsteines auskommt, den Beginn mit diesem aber implizit erzwingt.

Beweis: Angenommen, es liegen k Dominosteine vor, die mit irgendwelchen Wortpaaren $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ beschriftet sind:



Jede Lösung soll zudem mit dem ℓ–ten Stein beginnen, wobei die Zahl ℓ mit $1 \leq \ell \leq k$ vorgegeben ist. Dann kann man eine neue *PCP*–Instanz mit $k+2$ Steinen konstruieren, die ohne die Nennung dieser Zusatzbedingung auskommt, diese aber implizit erzwingt. Es seien dazu # und \$ zwei neue Symbole, die bislang in den Wörtern auf den Steinen nicht vorkommen. Das #–Symbol dient dabei als Trennzeichen zwischen den einzelnen Symbolen der Wörter auf den bisherigen Steinen. Im Einzelnen definieren wir für beliebige Wörter $w = a_1 a_2 \ldots a_n \in \Sigma^*$:

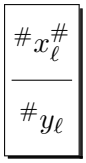
$$\begin{aligned} {}^{\#}w^{\#} &:= \#a_1\#a_2\#\ldots\#a_n\# \\ {}^{\#}w &:= \#a_1\#a_2\#\ldots\#a_n \\ w^{\#} &:= a_1\#a_2\#\ldots\#a_n\# \end{aligned}$$

Für $w = 1011$ gilt z.B.

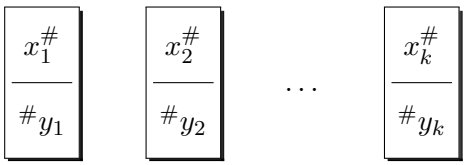
$${}^{\#}w^{\#} = \#1\#0\#1\#1\#\ , \quad {}^{\#}w = \#1\#0\#1\#1 \quad \text{und} \quad w^{\#} = 1\#0\#1\#1\#\ .$$

Nun können wir die Konstruktionsregeln für das neue *PCP*–Problem angeben:

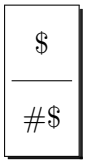
a) Für den ausgewiesenen Startstein mit dem Wortpaar (x_ℓ, y_ℓ) erzeugen wir einen neuen Stein der Form



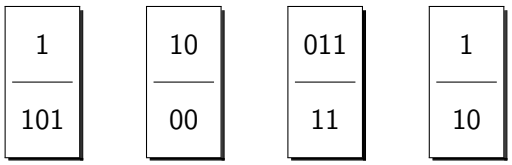
b) Für alle Steine der ursprünglichen *PCP*–Instanz (auch für den Startstein) erzeugen wir die k Steine



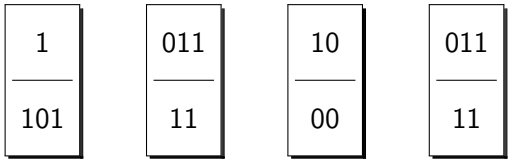
c) Schließlich existiert noch ein „Abschlussstein“, der später immer am Ende jeder Lösung zum Einsatz kommen wird:



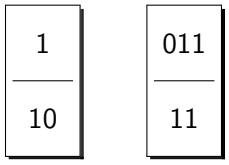
Betrachten Sie z.B. den folgenden Spielsatz mit $k = 4$ Steinen:



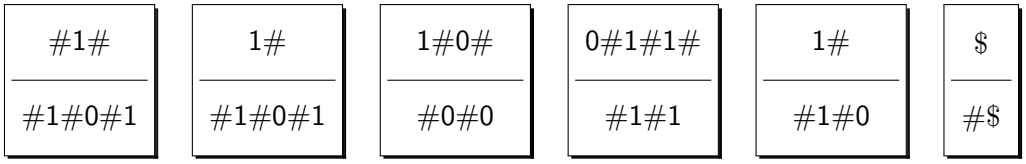
Man sieht recht einfach, dass z.B.



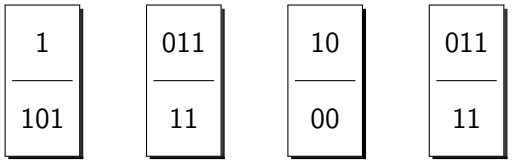
und



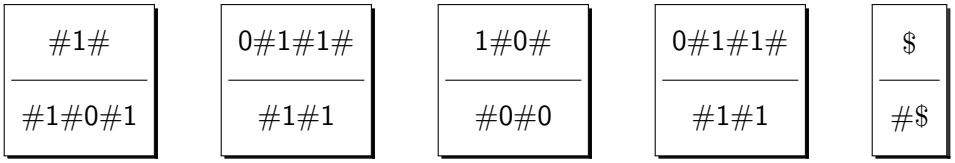
zwei mögliche Lösungen sind. Angenommen, wir wollen erzwingen, dass eine Lösung jetzt immer mit dem ersten Dominostein beginnen soll, so dass die zweite Lösung nicht mehr in Betracht kommt. Wir setzen also $\ell = 1$ und erzeugen mit Hilfe der obigen drei Regeln die folgenden $1 + 4 + 1 = 6$ neuen Dominosteine:



Zwischen beiden *PCP*–Instanzen besteht nun folgender Zusammenhang. Hat das ursprüngliche Problem eine Lösung, die mit dem ℓ–ten Stein beginnt, so kann man auch eine Lösung für das neue Problem angeben. Man braucht dazu nur den ersten ℓ–ten Stein gegen den neuen Startstein aus Regel a), sowie alle übrigen Steine durch die korrespondierenden neuen Steine aus Regel b) zu ersetzen. Abschließend muss noch am Ende der Schlussstein aus Regel c) angefügt werden. In unserem Beispiel ergibt sich so aus der alten Lösung



mit dem Wort 101110011 die neue Lösung



mit dem Wort #1#0#1#1#1#0#0#1#1#$. Allgemein verändert sich ein bisheriges Lösungswort w zu dem neuen Lösungswort ${}^{\#}w^{\#}\$$.

Ist umgekehrt das neue Problem lösbar, so muss als erster Stein immer der Startstein aus Regel a) verwendet werden, denn bei allen anderen Steinen stimmen die ersten Zeichen von beiden Wörtern nicht überein, d.h. mit solchen Steinen kann keine Lösung aufgebaut werden. Also hat man im neuen Problem im Endeffekt keine Wahl und muss immer mit dem Startstein aus Regel a) beginnen, obwohl dies nicht explizit gefordert wird. Man sieht nun leicht ein, dass ein Lösungswort für das neue Problem dann wieder von der Form ${}^{\#}w^{\#}\$$ sein muss, wobei w einem Lösungswort für das alte Problem entspricht. Da außerdem der Startstein aus Regel a) zu dem ℓ–ten Stein aus dem Ausgangsproblem korrespondiert, wird so demnach im alten Problem an erster Stelle der ℓ–te Stein erzwungen. □

Nun können wir den Beweis von Satz 4.6.1 angehen:

Beweis: Die Idee besteht darin, einer Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ und einem Wort $w \in \Sigma^*$ konstruktiv eine bestimmte Menge von Steintypen zuzuordnen, so dass das zugehörige *PCP* genau dann eine Lösung haben wird, wenn M angesetzt auf w anhält. Wir zeigen also die Reduktion $HP_{\mathsf{DTM}} \leq PCP$, so dass gemäß Korollar 4.5.19 die Unentscheidbarkeit des *PCP* folgt.

Die Konstruktionsvorschriften für die einzelnen Steintypen werden wir an einem einfachen Beispiel nachvollziehen. Sei dazu $M = (\{z_0, z_1, z_2\}, \{0, 1\}, \{0, 1, \Box\}, \delta, z_0, \Box, \{z_2\})$ eine DTM mit dieser Turingtafel:

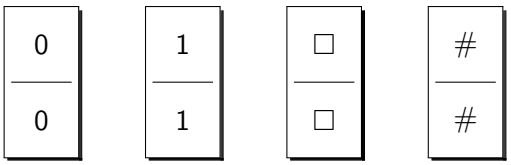
	0	1
z_0	$(z_0, 1, R)$	$(z_1, 0, N)$
z_1	$(z_1, 0, L)$	$(z_2, 0, N)$
z_2	$-$	$-$

Als Testwort wählen wir $w := 01$. M akzeptiert w mit der Konfigurationsfolge

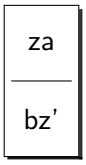
$$z_0 01 \vdash 1 z_0 1 \vdash 1 z_1 0 \vdash z_1 10 \vdash z_2 00 \ .$$

Wir definieren nun nach und nach die verschiedenen Steintypen. Die Wörter auf den Steinen werden über dem Alphabet $Z \cup \Gamma \cup \{\#\}$ gebildet, wobei # ein neues zusätzliches Zeichen ist. Das Alphabet wäre hier also $\{z_0, z_1, z_2, 0, 1, \Box, \#\}$, d.h. z.B. z_0 ist hier wirklich nur ein unzertrennliches Zeichen und besteht nicht etwa aus einem kleinen z, gefolgt von einer tiefergestellten 0.

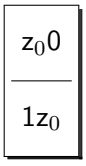
Für jedes Zeichen $a \in \Gamma \cup \{\#\}$ wird nun ein Steintyp angefertigt, der oben und unten nur aus diesem einen Zeichen besteht. Hier haben wir es also mit den vier Steinen



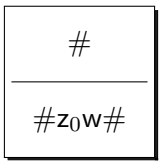
zu tun. Diese Steine werden wir nachher als „Kopierregeln“ bezeichnen. Weiterhin legen wir für jeden Übergang der Form $\delta(z, a) = (z', b, R)$, bei dem also der Kopf nach rechts bewegt wird, einen Steintyp



fest (eine sog. „Übergangsregel“). Hier im Beispiel gibt es nur einen solchen Übergang, nämlich $\delta(z_0, 0) = (z_0, 1, R)$. Demzufolge wird auch nur diese eine Übergangsregel erzeugt:

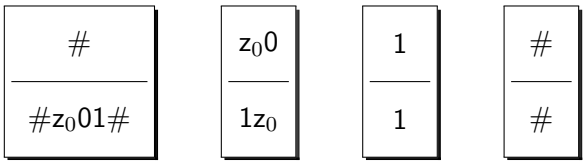


Nun legen wir noch einen Stein fest, der immer als erstes verwendet werden soll. Dies ist nach Lemma 4.6.2 ohne Einschränkung möglich. Der Startstein lautet:



Das Testwort w war hier als 01 gewählt, d.h. für dieses Beispiel ergibt sich

Man erkennt, dass das untere Wort die Startkonfiguration $z_0w = z_001$ enthält, während oben bislang „nichts“ steht (das führende Trennzeichen # ist an sich unwichtig und wird nur benutzt, um bei dem Startstein ein leeres oberes Wort zu vermeiden). Um also überhaupt zu einer Lösung zu kommen, müssen wir zunächst versuchen, die Startkonfiguration oben nachzubilden, um den unteren „Vorsprung“ aufzuholen. Dies ist mit den bislang skizzierten Steinen (und auch später) nur auf eine eindeutige Art und Weise möglich, nämlich wie folgt:



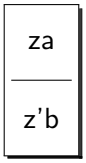
Der Vergleich von dem oberen mit dem unteren Wort ergibt:

$$\#z_001\#$$
$$\#z_001\#1z_01\#$$

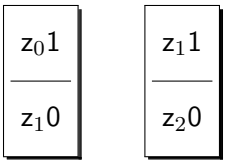
Wir haben nun zwar die Startkonfiguration oben erfolgreich nachgebildet, allerdings hat das untere Wort schon wieder einen Vorsprung herausgespielt. Wie man leicht sieht, handelt es sich dabei (von dem abschließenden Trennzeichen # abgesehen) genau um die erste Nachfolgekonfiguration $1z_01$.

Gemäß dieser Idee verläuft die weitere Konstruktion des Lösungswortes. Immer wenn man mit Hilfe der Kopier– und Übergangsregeln versucht, im oberen Wort den unteren Vorsprung aufzuholen, wird unten die nächste Nachfolgekonfiguration erzeugt, d.h. man „hinkt“ oben immer eine Konfiguration hinterher. Bevor wir dies am Beispiel weiter ausführen, müssen wir noch für die anderen Übergänge die passenden Steintypen festlegen.

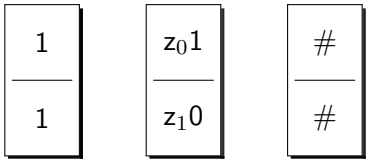
Für neutrale Übergänge $\delta(z, a) = (z', b, N)$ werden Steine der Form



verwendet. Die in der Turingmaschine M vorhandenen Übergänge $\delta(z_0, 1) = (z_1, 0, N)$ und $\delta(z_1, 1) = (z_2, 0, N)$ erzeugen also zwei weitere Übergangsregeln:



Jetzt lässt sich unser Beispiel fortsetzen. Die einzige Möglichkeit für eine Fortführung der bisherigen Lösung besteht in dem Anlegen der Steine

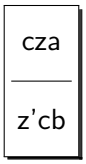


Hierdurch verlängern sich die bislang erzeugten Wörter zu

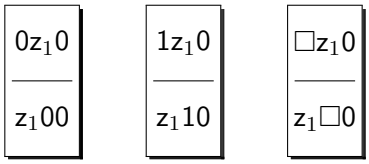
$$\#z_0 01\#1z_0 1\#$$
$$\#z_0 01\#1z_0 1\#1z_1 0\#$$

Das untere Wort liegt jetzt mit der zweiten Nachfolgekonfiguration $1z_1 0$ in Führung.

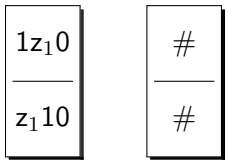
Nun fehlen noch die Steintypen für diejenigen Übergänge, bei denen der Kopf nach links bewegt wird. Hier müssen wir jetzt nicht einen, sondern $|\Gamma|$ viele Steine für jeden Übergang $\delta(z, a) = (z', b, L)$ einführen. Bei einer Kopfbewegung nach links wechselt nämlich das Symbol links vom Kopf auf die rechte Seite. Für jedes Symbol $c \in \Gamma$ definieren wir daher den Steintyp



In unserem Beispiel gibt es nur einen Übergang, der den Kopf nach links bewegt, nämlich $\delta(z_1, 0) = (z_1, 0, L)$. Aufgrund des Bandalphabets $\Gamma = \{0, 1, \square\}$ ergeben sich somit die drei neuen Steine



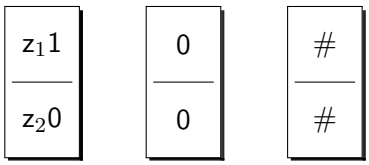
Unsere bislang konstruierte Lösung lässt sich so durch das Anlegen der Steine



zu

$$\#z_0 01\#1z_0 1\#1z_1 0\#$$
$$\#z_0 01\#1z_0 1\#1z_1 0\#z_1 10\#$$

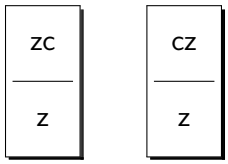
verlängern. Nun haben wir (fast) alle Steintypen zur Verfügung. Der abschließende Konfigurationsschritt $z_1 10 \vdash z_2 00$ lässt sich ohne weitere Steindefinitionen durch das Hinzufügen von



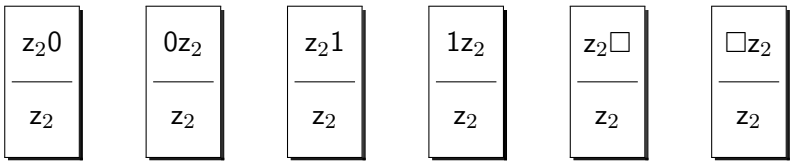
realisieren, und wir erhalten die Wörter

$$\#z_0 01\#1z_0 1\#1z_1 0\#z_1 10\#$$
$$\#z_0 01\#1z_0 1\#1z_1 0\#z_1 10\#z_2 00\#$$

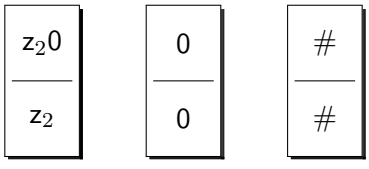
Jetzt gibt es keine weitere Anlegemöglichkeit mehr, denn für den Endzustand z_2 gibt es naturgemäß keine Übergänge und somit auch keine passenden Steintypen. Das macht aber nichts, denn für jeden Endzustand $z \in E$ und jedes Symbol $c \in \Gamma$ legen wir jetzt noch zwei „Löschregeln“ der Form



fest. In unserem Beispiel erhalten wir somit die Steine



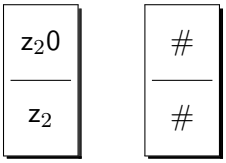
Die Idee hinter den Löschregeln wird klar, wenn wir unseren bisherigen Lösungsansatz durch die Steine



verlängern. Es ergeben sich dann die Wörter

$$\#z_0 01\#1z_0 1\#1z_1 0\#z_1 10\#z_2 00\#$$
$$\#z_0 01\#1z_0 1\#1z_1 0\#z_1 10\#z_2 00\#z_2 0\#$$

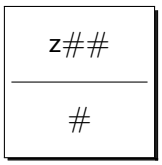
Wie man sieht, wurde im Wesentlichen die letzte Konfiguration kopiert, aber das Symbol 0 rechts neben dem Endzustand z_2 ist verschwunden. Durch das Anlegen von



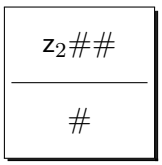
können wir diesen Vorgang nochmals wiederholen und erhalten damit die Wörter

$$\#z_0 01\#1z_0 1\#1z_1 0\#z_1 10\#z_2 00\#z_2 0\#$$
$$\#z_0 01\#1z_0 1\#1z_1 0\#z_1 10\#z_2 00\#z_2 0\#z_2\#$$

Bei jedem Kopiervorgang wird also ein Symbol entfernt und damit die Konfiguration um ein Zeichen kürzer. Mit den symmetrischen Löschregeln (bei denen das Symbol c links vom Endzustand z steht) könnten wir genauso gut alle Zeichen links von z entfernen, sofern hier welche vorhanden wären. Es bleibt also zum Schluss nur noch der Endzustand z allein übrig, gefolgt von dem Trennzeichen #. Als letzten Steintyp legen wir deshalb noch für jeden Endzustand $z \in E$ eine „Abschlussregel“ der Form



fest, hier also den Steintyp



Mit diesem Stein erhalten wir oben und unten das gleiche Lösungswort, nämlich

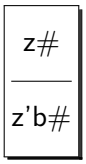
$$\#z_0 01\#1z_0 1\#1z_1 0\#z_1 10\#z_2 00\#z_2 0\#z_2\# \quad .$$

Wenn man sich genauer mit den genannten Steinen befasst, so sieht man leicht ein, dass man keine Möglichkeit hat, die vorgesehene Nachbildung der Konfigurationsübergänge zu umgehen. Die Lösung baut sich also weitestgehend eindeutig auf. Lediglich in der Löschphase kann man sich frei entscheiden, in welcher Reihenfolge man die Symbole links und rechts vom Endzustand entfernt. (Da es in unserem Beispiel nur Symbole rechts vom Endzustand gab, war hier auch diese Phase eindeutig.)

Zusammenfassend können wir also folgende Aussage treffen. Die Reduktionsfunktion nimmt die Beschreibung einer Turingmaschine M entgegen und erzeugt daraus wie oben gesehen den zugehörigen Startstein sowie alle genannten Kopier–, Übergangs–, Lösch– und Abschlussregeln. Wenn nun M angesetzt auf dem Wort w anhält, so gelingt nach dem vorgestellten Prinzip die Konstruktion eines gemeinsamen Wortes, d.h. das erzeugte *PCP* ist lösbar. Wenn jedoch M nicht anhält, so wird das obere Lösungswort immer dem unteren Lösungswort „hinterher hinken“, und man kommt nie zu einem Ende. Also ist das *PCP* in diesem Fall unlösbar. Insgesamt ist das konstruierte *PCP* also genau dann lösbar, wenn $M\#w \in HP_{\mathsf{DTM}}$ gilt. Somit ist die Reduktion korrekt und damit das *PCP* nach Korollar 4.5.19 unentscheidbar.

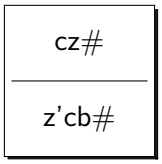
Zur Vervollständigung des Beweises müssen wir noch eine weitere Situation betrachten, die in unserem Beispiel nicht vorkam. Es kann nämlich passieren, dass die Turingmaschine M zwischendurch nach links oder rechts den bislang besuchten Bereich erweitert. In diesem Fall findet M stets Blanks auf dem Band vor, was durch einige noch fehlende Übergangsregeln simuliert werden muss.

Konkret interpretieren wir das rechts stehende Trennzeichen # immer als ein Blank (nämlich als dasjenige Blank, welches rechts an den bislang benutzten Bereich angrenzt). Folglich benötigen wir für jeden Übergang $\delta(z, \square) = (z', b, N)$ einen Steintyp

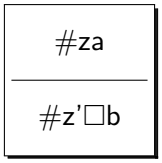


sowie analog für jeden Übergang $\delta(z, \square) = (z', b, R)$ einen Steintyp

Bei einer Bewegung nach links wird (wie bereits oben gesehen) das linke angrenzende Zeichen auf die rechte Seite kopiert, so dass wir für jeden Übergang $\delta(z, \square) = (z', b, L)$ wieder $|\Gamma|$ viele Steine benötigen, nämlich für jedes $c \in \Gamma$ einen Steintyp der Form

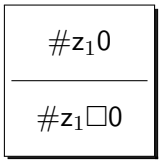


Für den linken Rand der aktuellen Konfiguration ist die Situation etwas einfacher. Hier benötigen wir für alle Übergänge $\delta(z, a) = (z', b, L)$ nur noch einen zusätzlichen Steintyp, nämlich



Dieser Stein simuliert die Bewegung des Schreib–Lese–Kopfes über den linken Rand des bislang benutzten Bereichs hinaus.

In unserem konkreten Fall gibt es keine Übergänge, die auf dem Blank □ definiert sind. Also erzwingt wegen $\delta(z_1, 0) = (z_1, 0, L)$ nur die letzte Übergangsregel einen weiteren Stein, nämlich



Dieser Stein wird hier zwar nicht gebraucht, aber die Reduktionsfunktion (die ja selbst keine Ahnung von den späteren konkreten Konfigurationsübergängen hat) würde ihn trotzdem erzeugen. □

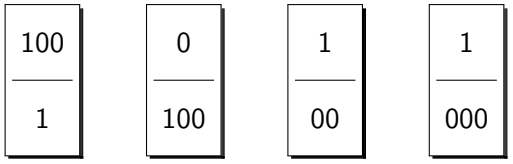
Zählt man in dem obigen Beispiel alle Steintypen zusammen, so kommt man auf immerhin 19 Stück, obwohl es sich um eine sehr einfache Turingmaschine gehandelt hat. (Zwei weitere Steine kommen noch hinzu, wenn man abschließend gemäß Lemma 4.6.2 das zugehörige „richtige“ *PCP* ohne Startstein konstruiert.) Bei größeren Übergangsfunktionen kommen sehr schnell viele weitere Steintypen zusammen. Man kann sich daher die Frage stellen, ob das *PCP* bereits dann unentscheidbar ist, wenn nur maximal k viele verschiedene Steintypen erlaubt sind, wobei k eine vorgegebene natürliche Zahl ist. Bis heute ist immerhin die Unentscheidbarkeit für $k \geq 7$ bewiesen (siehe [19]). Umgekehrt ist lediglich für $k = 1$ und $k = 2$ die Entscheidbarkeit des *PCP* bekannt. Für $3 \leq k \leq 6$ ist die Entscheidbarkeit des *PCP* noch ein Problem der aktuellen Forschung.

Die Unentscheidbarkeit des *PCP* zieht Konsequenzen bei wichtigen Problemen im Zusammenhang mit kontextfreien Grammatiken nach sich:

4.6.3 Satz Es ist unentscheidbar, ob zwei kontextfreie Grammatiken G_1 und G_2 wenigestens einen gemeinsamen Satz erzeugen, d.h. ob $L(G_1) \cap L(G_2) \neq \emptyset$ gilt (das sog. *Schnittproblem für kontextfreie Sprachen*).

Beweis: Wir zeigen, wie man aus einer konkreten *PCP*–Instanz zwei kontextfreie Grammatiken G_1 und G_2 konstruiert, deren Sprachen genau dann einen nichtleeren Schnitt haben werden, wenn für das *PCP*–Problem eine Lösung besteht.

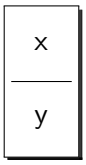
Wir betrachten dazu nochmals das Eingangsbeispiel zur *PCP*–Thematik:



Als Terminalalphabet Σ wählen wir zunächst die für die Wörter aus der *PCP*–Instanz benötigten Zeichen, hier also 0 und 1. Weiterhin führen wir für jeden Steintyp ein Sonderzeichen ein. Für den ersten Steintyp nennen wir es d_1, für den zweiten d_2, usw. In unserem Beispiel würde das Alphabet also

$$\Sigma := \{0, 1, d_1, d_2, d_3, d_4\}$$

lauten. Jede Grammatik besitzt nur die Startvariable S als einziges Nichtterminal, und auch der Aufbau der Produktionsmenge ist schnell erklärt. Für jeden Steintyp werden zwei Regeln erzeugt. Ist



der i–te Steintyp der *PCP*–Instanz, so werden die beiden Regeln

$$S \to xSd_i \,|\, xd_i$$

für die erste Grammatik sowie

$$S \to ySd_i \,|\, yd_i$$

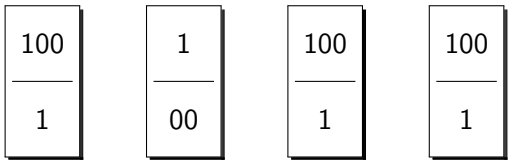
für die andere Grammatik zur Verfügung gestellt. In unserem Beispiel lautet die Produktionsmenge für $G_1 = (\{S\}, \Sigma, P, S)$ dann insgesamt

$$S \to 100Sd_1 \,|\, 100d_1 \,|\, 0Sd_2 \,|\, 0d_2 \,|\, 1Sd_3 \,|\, 1d_3 \,|\, 1Sd_4 \,|\, 1d_4$$

sowie für $G_2 = (\{S\}, \Sigma, P, S)$

$$S \to 1Sd_1 \,|\, 1d_1 \,|\, 100Sd_2 \,|\, 100d_2 \,|\, 00Sd_3 \,|\, 00d_3 \,|\, 000Sd_4 \,|\, 000d_4 \ .$$

Wir betrachten nun eine beliebige Kombination von Steinen der ursprünglichen *PCP*–Instanz:



Das resultierende obere Wort $100 \cdot 1 \cdot 100 \cdot 100 = 1001100100$ kann man durch die Regeln der ersten Grammatik G_1 entsprechend nachbilden:

$$S \Rightarrow 100Sd_1 \Rightarrow 1001Sd_3d_1 \Rightarrow 1001100Sd_1d_3d_1 \Rightarrow 1001100100d_1d_1d_3d_1 \ .$$

Links steht das obige Wort 1001100100, und die rechts vorhandenen Symbole $d_1d_1d_3d_1$ korrespondieren zu den benutzten Steinen, allerdings in umgekehrter Reihenfolge. Anhand der Symbole kann man also erkennen, dass die Steintypen

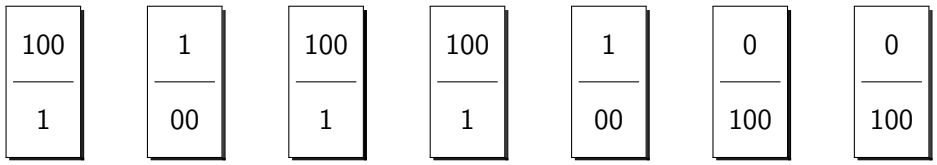
$$1, 3, 1, 1$$

nacheinander angelegt wurden.

Die zweite Grammatik G_2 ist analog in der Lage, das untere Wort $1 \cdot 00 \cdot 1 \cdot 1 = 10011$ abzuleiten:

$$S \Rightarrow 1Sd_1 \Rightarrow 100Sd_3d_1 \Rightarrow 1001Sd_1d_3d_1 \Rightarrow 10011d_1d_1d_3d_1 \ .$$

Besitzen beide Grammatiken einen gemeinsamen Satz? In diesem Fall ja, denn wir können die obige Steinreihe zu einer uns bereits bekannten Lösung mit den Steintypen $1, 3, 1, 1, 3, 2, 2$ ergänzen:



Sowohl oben als auch unten ergibt sich das Lösungswort 1001100100100, und wir können deshalb aus beiden Grammatiken den gleichen Satz $1001100100100d_2d_2d_3d_1d_1d_3d_1$ ableiten. Für G_1 gilt also

$$\begin{aligned} S &\Rightarrow_{G_1} 100Sd_1 \Rightarrow_{G_1} 1001Sd_3d_1 \Rightarrow_{G_1} 1001100Sd_1d_3d_1 \\ &\Rightarrow_{G_1} 1001100100Sd_1d_1d_3d_1 \Rightarrow_{G_1} 10011001001Sd_3d_1d_1d_3d_1 \\ &\Rightarrow_{G_1} 100110010010Sd_2d_3d_1d_1d_3d_1 \Rightarrow_{G_1} 1001100100100d_2d_2d_3d_1d_1d_3d_1 \end{aligned}$$

und für G_2 analog

$$\begin{aligned} S &\Rightarrow_{G_2} 1Sd_1 \Rightarrow_{G_2} 100Sd_3d_1 \Rightarrow_{G_2} 1001Sd_1d_3d_1 \\ &\Rightarrow_{G_2} 10011Sd_1d_1d_3d_1 \Rightarrow_{G_2} 1001100Sd_3d_1d_1d_3d_1 \\ &\Rightarrow_{G_2} 1001100100Sd_2d_3d_1d_1d_3d_1 \Rightarrow_{G_2} 1001100100100d_2d_2d_3d_1d_1d_3d_1 \ . \end{aligned}$$

Allgemein gilt: wenn eine *PCP*–Instanz lösbar ist, so besitzen die beiden zugehörigen Grammatiken einen entsprechenden gemeinsamen Satz. Umgekehrt setzt sich jeder aus G_1 und G_2 ableitbare Satz immer aus einem Präfix x mit Zeichen aus dem Steinalphabet sowie einem Suffix y mit d–Symbolen zusammen. Im Fall eines gemeinsamen Satzes müssen beide Präfixe und Suffixe jeweils identisch sein. Das identische Präfix x impliziert, dass in der zugehörigen *PCP*–Instanz sowohl oben als auch unten die identische Zeichenkette x erzeugt werden kann. Das identische Infix y besagt ferner, dass dabei beidesmal die gleichen Steintypen in der gleichen (umgekehrt aufgeführten) Reihenfolge zum Einsatz kommen. Also ist dann die *PCP*–Instanz insgesamt lösbar.

Wenn man also aus den Steintypen die beiden Grammatiken erzeugt (was sich durch ein entsprechendes Programm leicht automatisch erledigen lässt), so reduziert sich die Frage der Lösbarkeit der *PCP*–Instanz auf das Schnittproblem für zwei kontextfreien Sprachen. Diese Reduktion zeigt, dass das Schnittproblem unentscheidbar sein muss. □

4.6.4 Satz Es ist unentscheidbar, ob eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ alle Wörter aus Σ^* erzeugt (wir vernachlässigen an dieser Stelle wieder die Problematik mit dem leeren Wort ε, welches von einer kontextfreien Grammatiken eigentlich nicht erzeugt werden kann, vgl. die Diskussion dazu auf Seite 39).

Beweis: Wir zeigen zunächst, dass die in Satz 4.6.3 konstruierten Sprachen $L(G_1)$ und $L(G_2)$ sogar deterministisch kontextfrei sind. Ein passender DKA für $L(G_1)$ kann nämlich ein Wort wie z.B. $1001100100d_1d_1d_3d_1$ wie folgt verarbeiten. Zuerst liest er Zeichen für Zeichen ein und legt diese nacheinander auf dem Keller ab, wobei sich automatisch die Reihenfolge umkehrt. Sobald er auf das erste d–Zeichen trifft, geht der DKA in die nächste Phase über und verarbeitet jedes d–Symbol wie folgt. Falls er soeben das i–te Symbol d_i vorgefunden hat, erwartet er die entsprechende obere Zeichenkette des i–ten Steintyps in umgekehrter Reihenfolge auf dem Keller und baut diese in entsprechend vielen Einzelschritten wieder ab. Wenn schließlich das letzte d–Symbol verarbeitet wurde, muss auch gleichzeitig der Keller komplett leer sein. In diesem Fall akzeptiert der DKA, ansonsten nicht.

Ein solcher DKA für $L(G_1)$ kann konstruktiv aus G_1 gewonnen werden, und ebenso lässt sich daraus ein DKA für die komplementäre Sprache $\overline{L(G_1)}$ gewinnen (siehe Satz 3.4.6). Gleiches gilt für die Sprache $\overline{L(G_2)}$. Mit den erwähnten Konstruktionen aus den Sätzen 3.2.7 und 3.4.5 folgt daraus insgesamt, dass ein Programm aus einer *PCP*–Instanz heraus eine kontextfreie Grammatik G für die Sprache $\overline{L(G_1)} \cup \overline{L(G_2)}$ berechnen kann. Wegen

$$\overline{L(G_1)} \cup \overline{L(G_2)} = \Sigma^* \iff L(G_1) \cap L(G_2) = \emptyset$$

kann die Frage nach der Gleichheit $L(G) = \Sigma^*$ also nicht entscheidbar sein. □

4.6.5 Korollar Es seien L und L' zwei kontextfreie Sprachen über einem Alphabet Σ, die in Form von NKAs oder kontextfreien Grammatiken vorliegen. Weiterhin sei R eine reguläre Sprache über Σ, die erneut wahlweise durch einen endlichen Automat, einen regulären Ausdruck oder eine reguläre Grammatik beschrieben wird. Dann sind die folgenden Probleme unentscheidbar:

1. Ist $L = R$?
2. Ist $L \supseteq R$?
3. Ist $L = L'$?
4. Ist $L \supseteq L'$?

Beweis: Wir betrachten zunächst die Probleme 1 und 2 für den Fall, dass L durch eine kontextfreie Grammatik G sowie R durch einen regulären Ausdruck α gegeben ist. Die Fragen „Ist $L(G) = \varphi(\alpha)$?“ und „Ist $L(G) \supseteq \varphi(\alpha)$?“ können dann nicht entschieden werden. Ist nämlich $\Sigma = \{a_1, a_2, \ldots, a_n\}$, so wäre für α ein regulärer Ausdruck wie z.B.

$$(a_1 \,|\, a_2 \,|\, \ldots \,|\, a_n)^\star$$

möglich, der also ganz Σ^* erzeugt. Dann aber laufen beide Fragen auf das Problem „Ist $L(G) = \Sigma^*$?“ hinaus, welches nach Satz 4.6.4 unentscheidbar ist.

Für einen endlichen Automaten M können die beiden Fragen „Ist $L(G) = L(M)$?“ und „Ist $L(G) \supseteq L(M)$?“ ebenfalls nicht entschieden werden, denn andernfalls könnte man auch die ersten beiden Fragen entscheiden, indem man mit der uns bekannten Konstruktion (Satz 2.3.4 auf Seite 53) einen regulären Ausdruck α zunächst in einen gleichwertigen endlichen Automaten M umformt. Gleiches trifft zu, wenn die kontextfreie Sprache L durch einen NKA und / oder die reguläre Sprache R durch eine reguläre Grammtik gegeben ist. Also sind die ersten beiden Probleme bzgl. $L = R$ bzw. $L \supseteq R$ unentscheidbar — und die anderen beiden Probleme bzgl. $L = L'$ bzw. $L \supseteq L'$ auch, denn wir können alle Argumente analog wiederholen, indem wir für L' einen NKA bzw. eine kontextfreie Grammatik mit $L' = \Sigma^*$ angeben. □

5

Einführung in die Komplexitätstheorie

Zum Schluss unserer Einführung in die theoretischen Informatik werfen wir noch einen Blick auf die *Komplexitätstheorie*. Hierbei geht es in erster Linie darum, wieviel Zeit man genau für die Lösung von bestimmten Problemen benötigt. Manche Probleme lassen sich bekanntlich recht einfach berechnen, z.B. kann ein handelsüblicher Computer in wenigen Sekunden Milliarden (!) von Ganzzahlen zusammenaddieren. Andere Problemstellungen, wie z.B. das Problem des Handlungsreisenden aus der Einführung, sind scheinbar weitaus schwieriger zu handhaben und benötigen bislang schon bei kleinen Instanzen so viel Zeit, dass es praktisch unmöglich ist, optimale Lösungen zu berechnen.

Ein zweites Komplexitätsmaß ist der *Speicherplatz*, also die Menge an Speicher, die während der Laufzeit eines Verfahrens zumindest zeitweise benötigt wird. Wir werden im Rahmen dieses Kapitels auf die Platzkomplexität nicht eingehen, sondern dies erst später im Kapitel 6 nachholen (die Bücher in dem Literaturverzeichnis behandeln dieses Thema ebenfalls umfassend). Stattdessen werden wir zwei für die Praxis besonders bedeutende Komplexitätsklassen erläutern, eine daraus resultierende interessante Anwendung aus der *Kryptographie* (also der Verschlüsselungstechnik) untersuchen sowie das momentan bedeutendste Forschungsproblem in der theoretischen Informatik überhaupt besprechen.

Auf diesem Forschungsproblem basiert der oben erwähnte Aspekt, dass bestimmte an sich entscheidbare Fragestellungen als besonders „hartnäckig“ gelten. Für deren Berechnung wird also so viel Zeit benötigt, dass sie vom praktischen Standpunkt aus gesehen doch unentscheidbar sind (das Problem des Handlungsreisenden gehört nach heutigem Kenntnisstand mit dazu.) Wir werden zum Abschluss dieses Kapitels einige solche Entscheidungsfragen vorstellen und deren vermutlich vorliegende „Hartnäckigkeit“ beweisen.

5.1. Nichtdeterministische Komplexität

Theoretische Informatiker unterteilen algorithmische Problemstellungen in *Komplexitätsklassen*, die die benötigten Berechnungsressourcen berücksichtigen. Im Fall von Spei-

cherplatz kann man den Verbrauch z.B. in Bytes messen. Bzgl. der Zeit interessieren wir uns dagegen für die Anzahl der benötigten „elementaren Schritte“ von z.B. einem zugehörigen Java–Programm. Elementare Schritte sind dabei alle einfachen Programmzeilen wie z.B. Zuweisungen, `if`–Abfragen, Bildschirmausgaben von einzelnen Zeichen, usw., also alle Anweisungen, die in konstant viel Zeit ausgeführt werden können. Schleifen, Methodenaufrufe oder Ausgaben von ganzen Zeichenketten zählen also nicht dazu — hier muss genauer gezählt werden, wie oft jede Schleife durchgeführt wird, wieviele andere Anweisungen sich hinter einem Methodenaufruf verstecken und wie viele einzelne Zeichen insgesamt auf dem Bildschirm ausgegeben werden.

5.1.1 Beispiel Das Sortieren von n Zahlen kann leicht bewerkstelligt werden, z.B. durch den bekannten *Bubblesort*–Algorithmus:

```
void bubbleSort(int[] zahlenfeld) {
   for (int i = 0; i < zahlenfeld.length; i++) {
      for (int j = i+1; j < zahlenfeld.length; j++)
         if (zahlenfeld[i] > zahlenfeld[j]) {
            int temp = zahlenfeld[i];
            zahlenfeld[i] = zahlenfeld[j];
            zahlenfeld[j] = temp;
         }
      }
   }
}
```

Bei der Sortierung von n Zahlen wird die äußere Schleife n–mal ausgeführt, und jedesmal wird die innere Schleife einmal komplett ausgeführt. Diese benötigt beim ersten Mal $n-1$ Iterationen, beim zweiten Mal $n-2$ Iterationen, usw., also nie mehr als jeweils n Durchläufe. Der eigentliche Rumpf der beiden verschachtelten Schleifen läuft in konstanter Zeit ab (denn dieser besteht aus einer bzw. aus vier elementaren Anweisungen, je nachdem, ob die Bedingung in der `if`–Abfrage zutrifft oder nicht).

Wir nehmen an, dass n Zahlen in $O(n)$ viel Platz gespeichert werden können. Dies ist z.B. bei Ganzzahlen vom Typ `int` der Fall (hier werden vier Bytes pro Zahl belegt, also $4n = O(n)$ viele Bytes insgesamt). Der Algorithmus benötigt dann also $O(n^2)$ viel Schritte, und es gibt bekanntlich noch viel bessere Sortierverfahren wie z.B. *Heapsort*, die nur $O(n \log n)$ viel Zeit benötigen.

5.1.2 Beispiel Das Addieren zweier (langer) Zahlen mit jeweils n Stellen kostet offenbar $O(n)$ viele Schritte. Für das Multiplizieren mit dem üblichen Standardverfahren benötigt man dagegen $O(n^2)$ viele Schritte, denn bekanntlicherweise multipliziert man dabei jede der n Ziffern der einen Zahl mit allen n Ziffern der anderen Zahl, schreibt sich die Zwischenprodukte versetzt untereinander auf, und addiert sie anschließend noch zusam-

men. Für jedes der insgesamt n Zwischenprodukte werden $O(n)$ viele Schritte benötigt, insgesamt also $O(n^2)$ viele Schritte für alle Zwischenprodukte. Die Addition dieser n Zahlen mit jeweils $O(n)$ vielen Ziffern kostet dann nochmals $O(n^2)$ viel Zeit.

Auch für die Zahlenmultiplikation gibt es übrigens schnellere Verfahren. Der *Karatsuba*[1]–Algorithmus erreicht z.B. eine Komplexität von $O(n^{1,59})$, und es geht sogar noch wesentlich besser.

Man kann auch von der Komplexität einer entscheidbaren Sprache L sprechen.

5.1.3 Definition Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion. Eine Sprache $L \subseteq \Sigma^*$ liegt in DTIME($f(n)$), falls für jedes Wort $x \in \Sigma^*$ das Wortproblem „Ist $x \in L$?“ in $O(f(|x|))$ vielen Schritten entschieden werden kann, d.h. die Berechnung der charakteristischen Funktion $\chi_L(x)$ (siehe Def. 4.5.4) kostet höchstens $O(f(|x|))$ viel Zeit.

5.1.4 Beispiel Für jede kontextfreie Sprache L gilt $L \in \text{DTIME}(n^3)$, denn der CYK–Test, der im Endeffekt die charakteristische Funktion berechnet, ist gerade von kubischer Komplexität (siehe Satz 3.3.6 auf Seite 107). Für jede reguläre Sprache L gilt sogar $L \in \text{DTIME}(n)$, denn ein zugehöriger DEA entscheidet das Wortproblem für ein Wort $x \in \Sigma^*$ in linearer Zeit — pro Zustandswechsel wird ein Zeichen von x verbraucht, und deshalb liegt die Antwort nach spätestens $|x|$ Schritten vor.

5.1.5 Definition Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion. Eine formale Sprache $L \subseteq \Sigma^*$ liegt in NTIME($f(n)$), falls es eine Funktion $\hat{\chi}_L : \Sigma^* \times \Sigma^* \to \{0, 1\}$ gibt, die die folgenden Eigenschaften besitzt:

- Für jedes $x \in L$ existiert ein $y \in \Sigma^*$, so dass $\hat{\chi}_L(x, y) = 1$ gilt und dieser Funktionswert in $O(f(|x|))$ viel Zeit berechnet werden kann.
- Für jedes $x \notin L$ gibt es kein solches y, d.h. für alle $y \in \Sigma^*$ gilt $\hat{\chi}_L(x, y) = 0$.

Für jedes $x \in L$ gibt es also einen entscheidenden „Tipp“ oder „Beweis“ y, der die Berechnung der charakteristischen Funktion evtl. entscheidend vereinfacht. Man sagt auch, dass sich L in $O(f(|x|))$ viel Zeit *verifizieren* lässt. Ohne diesen Hinweis kann die Lösung des Wortproblems jedoch sehr zeitaufwendig sein.

5.1.6 Beispiel Betrachten Sie die Sprache aller *zusammengesetzten Zahlen* (also aller „Nicht–Primzahlen“)

$$L = \{1, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, \dots\}$$

über dem Alphabet $\Sigma = \{0, 1, 2, \dots, 9\}$. Um eine konkrete ungerade Zahl n auf Primalität zu testen, kann man z.B. unter allen ungeraden Zahlen $3, 5, \dots, n-2$ nach einem Teiler

[1] Anatolii A. Karatsuba, *1937 Grosny, Tschetschenische Republik, †2008 Moskau, russischer Mathematiker, Professor für Mathematik in Moskau.

von n suchen. Es reicht sogar aus, nur alle Zahlen zu überprüfen, die kleiner als $\sqrt{n}$ sind, denn wenn eine Zahl k mit $k \geq \sqrt{n}$ ein Teiler von n ist, so ist der Quotient n/k ebenfalls ein Teiler von n, und dieser muss dann kleiner als $\sqrt{n}$ sein, d.h. dieser wäre schon vorher überprüft worden. Probedivisionen mit den ersten

$$\frac{\sqrt{n}}{2} = O(\sqrt{n})$$

vielen ungeraden Zahlen reichen also aus.

Man könnte nun der Ansicht sein, dass L in DTIME($\sqrt{n}$) liegt — dies ist aber nicht der Fall. Als Eingabelänge geht nämlich nicht, wie schon im Beispiel 5.1.2 gesehen, die Zahl n selbst, sondern die Anzahl m ihrer Ziffern in die Komplexitätsbetrachtung ein. Eine Zahl mit m Ziffern hat mindestens den Wert 10^{m-1}, d.h. der oben skizzierte Algorithmus hat eine Zeitkomplexität von etwa

$$\sqrt{10^{m-1}} = \sqrt{\frac{10^m}{10}} = \frac{\sqrt{10^m}}{\sqrt{10}} = \frac{1}{\sqrt{10}}(10^m)^{\frac{1}{2}} = O(10^{\frac{m}{2}}) = O((10^{\frac{1}{2}})^m) \approx O(3,163^m) \ ,$$

d.h. die Laufzeit ist — gemessen an der Ziffernlänge m von n — exponentiell. Um zum Beispiel eine Zahl n mit 100 Ziffern auf Primalität zu testen (dies ist noch keine besonders große Zahl; in der Kryptographie kommen z.B. Primzahlen mit mehreren hundert Ziffern zum Einsatz), benötigt das obige Verfahren bis zu $3,163^{100} > 10^{50}$ viele Iterationen — kein Computer dieser Welt kann auch nur annähernd diese große Anzahl von Tests durchführen. (Zum Glück gibt es erheblich effizientere Verfahren zum Test einer Zahl auf *Primalität*, siehe z.B. [8].)

Für jede Zahl n existiert jedoch eine einfache Beweiszahl y, die die Zusammengesetztheit von n belegt — nämlich ein passender Teiler y von n. Dann muss man nur eine Testdivision durchführen, d.h. wir können die Funktion $\hat{\chi}_L$ ganz leicht wie folgt angeben:

$$\hat{\chi}_L(n,y) := \begin{cases} 1 & \text{falls } n \bmod y = 0 \ \wedge \ y > 1 \ \wedge \ y < n \\ 0 & \text{sonst} \end{cases}$$

Gemessen an den $m = O(\log n)$ vielen Ziffern von n benötigt die Berechnung der Testdivision n/y nur $O(m^2)$ viel Zeit (wie schon im Beispiel 5.1.2 erwähnt). Also liegt die Sprache aller zusammengesetzten Zahlen in NTIME(m^2).

5.1.7 Beispiel Im ersten Abschnitt des Einführungskapitels wurden die Grundlagen der Aussagenlogik besprochen. Eine interessante Frage ist, ob sich eine Aussage wie z.B.

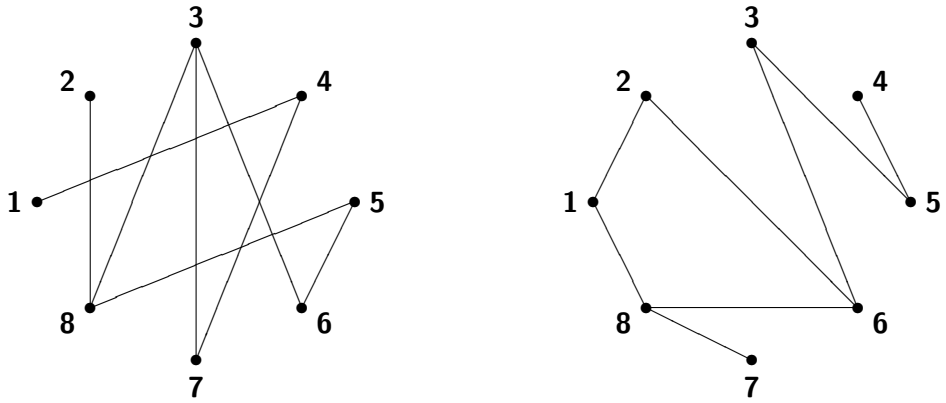
$$(\neg A \vee \neg C \vee D) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg B \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg B \vee D)$$

überhaupt *erfüllen* lässt, d.h. ob es eine bestimmte Belegung der booleschen Variablen A, B, C und D mit passenden Wahrheitswerten gibt, so dass die gesamte Aussage wahr wird. Natürlich kann man dazu alle Variablenbelegungen einfach durchprobieren, aber eine Aussage mit n Zeichen kann auch $O(n)$ viele Variablen enthalten, so dass größenordnungsmäßig $O(2^n)$ viele Tests durchgeführt werden müssten. Der naive Algorithmus

ist also von exponentieller Komplexität, und tatsächlich hat man bis heute keinen entscheidend besseren Algorithmus gefunden. In der Fachwelt ist die obige Sprache aller erfüllbaren aussagenlogischen Formeln als *Erfüllbarkeitsproblem* (oder kurz *SAT* für engl. „*satisfiability problem*“) bekannt.

Trotz der vermuteten hohen Komplexität liegt *SAT* aber in NTIME(n), denn ein Beweis für die Erfüllbarkeit einer logischen Aussage besteht einfach aus der Angabe einer passenden Variablenbelegung. Diese kann man in die Formel einsetzen und anschließend den Wahrheitsgehalt durch einfaches Ausrechnen in linearer Zeit überprüfen.

5.1.8 Beispiel Ein *ungerichteter Graph* G besteht aus einer endlichen Menge V von *Knoten* sowie einer endlichen Menge E von *Kanten*, die zwischen je zwei Knoten verlaufen. Die beiden folgenden Graphen besitzen z.B. je acht Knoten, die als schwarze Punkte dargestellt werden. Der Einfachheit halber wurden sie im Uhrzeigersinn durchnummeriert, d.h. die Knotenmenge ist in beiden Fällen $V := \{1, 2, 3, 4, 5, 6, 7, 8\}$. Die Kanten werden durch gerade Verbindungen symbolisiert:



Der linke Graph links besitzt also z.B. eine Kante zwischen den Knoten 3 und 8, nicht jedoch z.B. zwischen den Knoten 4 und 5. Man sagt dann auch, dass die Knoten 3 und 8 *benachbart* oder *adjazent* sind. Eine Kante aus der Menge E geben wir durch eine 2–elementige Menge an, d.h. wie soeben beschrieben gilt für den linken Graph $\{3, 8\} \in E$, aber $\{4, 5\} \notin E$. Wie bei Mengen üblich gibt es keine Ordnung, d.h. die Kante $\{3, 8\}$ ist mit der Kante $\{8, 3\}$ identisch. Aus diesem Grund ist der Graph ungerichtet (bei *gerichteten Graphen* werden die Kanten durch Pfeile dargestellt, die nur in der angegebenen Orientierung durchlaufen werden dürfen).

Stellen Sie sich nun vor, dass alle Kanten „Gummiseile“ sind und straff gespannt bleiben, wenn man die Knoten an neue Positionen verschiebt. Ist es dann möglich, nur mit Knotenverschiebungen aus dem linken Graph den rechten zu gewinnen? Etwas formaler ausgedrückt suchen wir also eine Zuordnungsfunktion $\sigma : \{1, 2, \ldots, 8\} \to \{1, 2, \ldots, 8\}$, so dass für jeden linken Knoten mit der Nummer i der Wert $\sigma(i)$ die entsprechende Position im rechten Graph angibt.

In unserem Beispiel gibt es eine solche Zuordnung, und es ist relativ einfach, sie zu finden.

Zunächst stellen wir fest, dass z.B. der linke Knoten 8 nicht mit dem rechten Knoten 5 übereinstimmen kann, denn der linke Knoten 8 besitzt drei Kanten, der rechte Knoten 5 jedoch nur zwei. Man spricht bei der Anzahl der Kanten auch vom *Grad* eines Knotens. Im rechten Graph gibt es nur zwei Knoten vom Grad drei, nämlich die Knoten 6 und 8, so dass einer davon dem linken Knoten 8 entsprechen muss. Der Knoten 6 kommt dafür jedoch nicht in Frage, denn dieser hat keinen Nachbarn vom Grad 1, der linke Knoten 8 aber schon. Also gilt $\sigma(8) = 8$, d.h. der Knoten 8 bleibt (zufälligerweise) an seinem Platz.

Die drei Nachbarn des linken Knotens 8 haben die Grade 1, 2 und 3. Aus den entsprechenden Graden der Nachbarn vom rechten Knoten 8 können wir also direkt die Zuordnungen $\sigma(2) = 7$, $\sigma(3) = 6$ und $\sigma(5) = 1$ ablesen. Unter den noch nicht zugeordneten Knoten weist links dann nur noch Knoten 1 den Grad 1 auf, d.h. dieser muss rechts dem Knoten 4 entsprechen, und es folgt $\sigma(1) = 4$. Genauso müssen sich die beiden zugehörigen Nachbarn 4 und 5 entsprechen, d.h. es gilt $\sigma(4) = 5$. Diese beiden Knoten haben wiederum ebenfalls noch einen weiteren Nachbarn, die sich ebenfalls entsprechen müssen, was $\sigma(7) = 3$ erzwingt. Nun bleiben links nur noch der Knoten 6 und rechts der Knoten 2 übrig, woraus die letzte Zuordnung $\sigma(6) = 2$ folgt. Eine kurze Kontrolle zeigt, dass für je zwei Knoten $i \neq j$ immer

$$i \text{ und } j \text{ sind links adjazent} \iff \sigma(i) \text{ und } \sigma(j) \text{ sind rechts adjazent}$$

gilt. Also sind im Endeffekt die beiden Graphen bis auf eine Umbenennung der Knoten identisch oder *isomorph*. Die Zuordnungsfunktion σ, die die *Isomorphie* belegt, können wir durch eine Wertetabelle

i	1	2	3	4	5	6	7	8
$\sigma(i)$	4	7	6	5	1	2	3	8

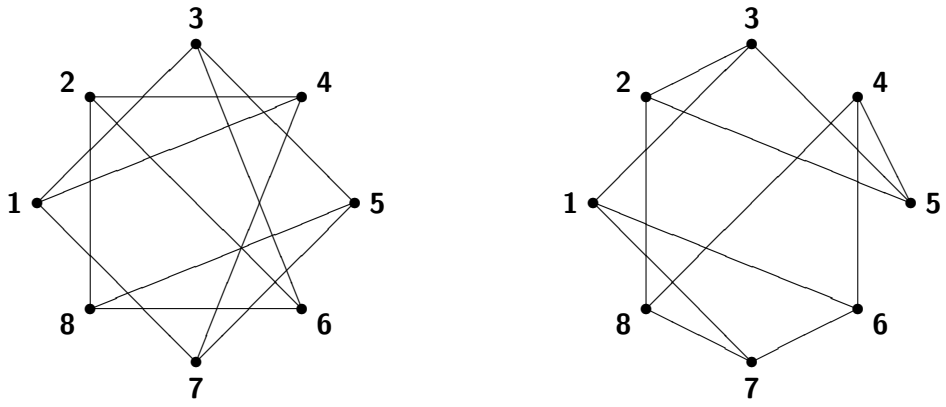
angeben. Sie bringt also die ersten n Zahlen (hier war $n = 8$) in eine andere Anordnung und heißt deshalb auch *Permutation*.

Das *Graphenisomorphieproblem* behandelt die Frage, ob zwei gegebene Graphen G_1 und G_2 zueinander isomorph sind. Dies lässt sich bei Vorliegen eines Beweises (natürlich in Form einer passenden Permutation σ) leicht verifizieren — man muss dazu lediglich einmal alle Kanten des eines Graphen durchgehen und für je zwei verbundene Knoten i und j prüfen, ob die entsprechenden Knoten $\sigma(i)$ und $\sigma(j)$ des anderen Graphen ebenfalls miteinander verbunden sind (und sonst keine). Dies gelingt mit einer geschickten Implementierung in Linearzeit (gemessen an der Größe des Graphen, bei der neben der Anzahl der Knoten auch die Anzahl der Kanten mit einher geht). Wenn die beiden Graphen also $O(n)$ viel Speicher belegen, so wird auch nur $O(n)$ viel Zeit für die Verifikation benötigt d.h. das Graphenisomorphieproblem liegt in NTIME(n).

Natürlich müssen zwei isomorphe Graphen immer die gleiche Anzahl von Knoten und Kanten haben — genauer gesagt muss für jedes $k \in \mathbb{N}_0$ die Anzahl der Knoten vom Grad k in beiden Graphen gleich sein. Genau dies haben wir ausgenutzt, um die Zuordnungen

in unserem Beispiel schnell zu finden. Wenn aber nicht genügend Nebenbedingungen abgeleitet werden können, kann es sehr schwer sein, eine passende Zuordnung zu finden — sofern es eine solche überhaupt gibt.

Betrachten Sie dazu die beiden folgenden Graphen:



Hier gibt es zunächst überhaupt keine Anhaltspunkte — die Anzahl der Knoten ist gleich, die Anzahl der Kanten auch, und jeder Knoten ist vom Grad 3. Jeder linke Knoten könnte also für jede rechte Position in Frage kommen. Natürlich könnte man nun einfach alle Permutationen durchspielen, aber davon gibt es sehr viele, denn für den ersten Knoten kommen acht Positionen in Frage, für den zweiten dann noch sieben, für den dritten sechs, usw., d.h. es gibt insgesamt

$$8 \cdot 7 \cdot \cdots \cdot 2 \cdot 1 = 40320$$

mögliche Zuordnungen. Unter diesen ist dann auch die gesuchte Permutation

i	1	2	3	4	5	6	7	8
$\sigma(i)$	7	3	8	1	4	2	6	5

.

Das Durchspielen aller 40320 Möglichkeiten ist für moderne Computer natürlich nur eine Angelegenheit von Millisekunden, aber die beiden Graphen sind auch sehr klein — bei zwei Graphen mit z.B. je 100 Knoten gibt es schon

$$100 \cdot 99 \cdot \cdots \cdot 2 \cdot 1 \approx 10^{158}$$

Möglichkeiten. (Bei dieser Zahl handelt es sich natürlich um die *Fakultät* 100!, die wir bereits in dem einführenden Kapitel kennengelernt haben, vgl. Seite 17.) Trotzdem wäre auch ein Graph mit 100 Knoten immer noch sehr klein (zumindest für Computerverhältnisse), denn seine Darstellung im Hauptspeicher würde nur wenige Kilobytes belegen. Im Verhältnis zum belegten Platz steigt die Zeitkomplexität also enorm an.

Bis heute hat man keinen wesentlich besseren Algorithmus als das Durchspielen aller Möglichkeiten gefunden. Dies kostet jedoch exponentiell viel Zeit — das Graphenisomorphieproblem scheint also sehr schwer zu sein, auch wenn es sich, wie bereits gesehen, leicht verifizieren lässt.

Das „N“ in dem Begriff NTIME stammt übrigens von „nichtdeterministisch“. Eine nichtdeterministische Turingmaschine kann nämlich z.B. eine Instanz x des Graphenisomorphieproblems schnell überprüfen, indem sie anfangs einen Beweis y „rät“, d.h. sie geht auf ihrem Band hinter die eigentliche Eingabe x nach rechts und schreibt mit Hilfe von nichtdeterministischen Übergängen zufällige Zeichen auf ihr Band, die das Wort y bilden. Anschließend berechnet sie „ganz normal“ (also deterministisch) die Funktion $\hat{\chi}_L(x, y)$. Analog steht das „D“ in DTIME für „deterministisch“ — hier sind keine nichtdeterminstischen Übergänge erlaubt, und eine entsprechende DTM muss (wie auch die von uns benutzten Java–Programme) ohne einen entsprechenden Hinweis y auskommen.

5.2. Zero–Knowledge–Beweise

Wir wollen unsere Erkenntnisse über den Unterschied zwischen „leicht berechenbar“ und „leicht verifizierbar“ für eine interessante Anwendung ausnutzen.

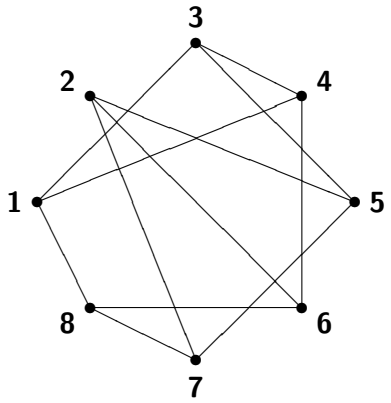
Eine gängige Methode zur Authentifizierung von Benutzern im Internet besteht darin, dass eine Zugangsnummer, eine Benutzerkennung, ein Passwort oder Ähnliches eingegeben werden muss — ein Geheimnis, welches nur dem richtigen Benutzer bekannt sein darf. In diesem Zusammenhang ist Ihnen sicherlich das sog. *Phishing*–Problem ein Begriff, welches im Internet weit verbreitet ist. Mittels gefälschter elektronischer Nachrichten wird dabei versucht, an genau diese sensiblen Daten wie Benutzernamen, Passwörter oder Zugangsnummern für z.B. das Online–Banking zu gelangen. Kriminelle Personen haben mit dieser Methode immer wieder Erfolg, weil die Mechanismen zur Authentifizierung die Preisgabe des Geheimnisses erfordern. Wenn also irgendwie das Geheimnis (z.B. beim Eintippen des Passworts) mitgeschnitten wird, so kann sich auch jemand anders als der vermeintliche Kunde ausgeben.

Ähnliche Probleme treten auch z.B. an Geldautomaten auf, bei denen fingierte Kartenleser und Kameras installiert wurden. Das Geheimnis (hier also die Geheimnummer der EC– bzw. Kreditkarte) wird vom arglosen Kunden bei einer Barabhebung eingegeben und dabei mitprotokolliert. Später kann dann mit einer Kopie der Karte und der illegal erworbenen Geheimnummer das entsprechende Konto abgeräumt werden.

Scheinbar kann man diese Kriminalität nur durch Appelle an die Benutzer eindämmen, damit diese vorsichtiger mit sensiblen Daten umgehen. Überraschenderweise ist dies aber falsch, denn es gibt für die Authentifizierungsproblematik auch einen anderen interessanten Ansatz, bei dem *kein Wissen übertragen wird*. Man kann also jemanden davon überzeugen, dass man ein bestimmtes Geheimnis kennt, ohne dieses Geheimnis zu verraten. Was auf den ersten Blick unmöglich erscheint, geht tatsächlich, nämlich mit sog. *Zero–Knowledge–Protokollen*.

In solchen *kryptographischen Protokollen* gibt es typischerweise zwei Personen, die wir hier Alice und Boris nennen. Alice kennt ein bestimmtes Geheimnis, und Boris möchte sich gerne davon überzeugen, dass er es wirklich mit Alice zu tun hat, sie also diejenige Person ist, die das Geheimnis kennt.

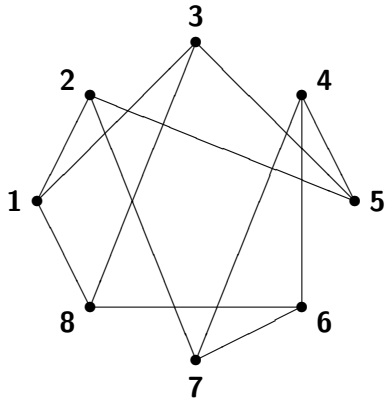
Das Geheimnis von Alice muss nicht nur für Menschen, sondern auch insbesondere für Computer schwer zu „knacken“ sein — sonst wäre es nicht wirklich ein Geheimnis. Hierfür bietet sich z.B. wieder das Graphenisomorphieproblem an. Alice erfindet dazu als erstes irgendeinen Graphen G_1 mit mehreren hundert Knoten (wir bleiben der Einfachheit halber hier bei kleinen Graphen):



Anschließend stellt Alice eine zufällige Permutation zusammen, z.B.

i	1	2	3	4	5	6	7	8
$\sigma(i)$	3	7	8	1	6	2	4	5

und wendet diese auf den vorher gewählten Graphen an, womit sie einen zweiten Graphen G_2 erhält:



Diese beiden Graphen werden veröffentlicht, und Alice ist als die Person bekannt, die die von ihr geheim gehaltene Permutation σ zwischen beiden Graphen kennt. Es handelt sich um ein gutes Geheimnis, denn es gibt — wie im letzten Abschnitt gesehen — selbst mit Hochleistungsrechnern keine Möglichkeit, die gesuchte Permutation zu finden (zumindest wenn man die Graphen etwas größer gewählt hat).

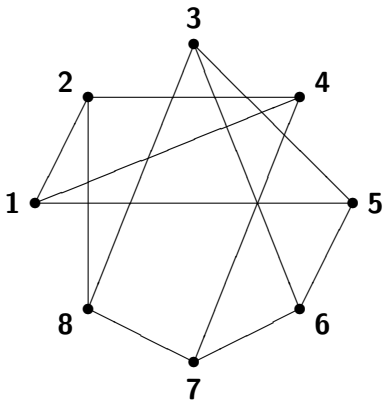
Nun möchte Alice Boris überzeugen, dass sie die „richtige“ Alice ist. Dazu könnte sie natürlich Boris einfach die Permutation σ verraten, und Boris könnte dann deren Kor-

rektheit leicht überprüfen. Aber danach wäre Boris in der Lage, sich ebenfalls als Alice auszugeben, da er dann ja das Geheimnis kennt. Deshalb wählt Alice einen anderen Weg, und einigt sich mit Boris auf den folgenden Ablauf.

Als erstes erfindet Alice nochmals irgendeine Permutation τ_1, z.B.

i	1	2	3	4	5	6	7	8
$\tau_1(i)$	6	2	5	3	1	8	4	7

und wendet diese Permutation auf den ersten Ausgangsgraph G_1 an, woraus sich ein dritter Graph H ergibt:



Dieser Graph ist aber nicht nur zu G_1, sondern auch zu dem anderen öffentlichen Graph G_2 isomorph. Die zugehörige Permutation τ_2 kann Alice leicht aus den Permutation τ_1 zwischen G_1 und H sowie aus der von ihr geheim gehaltenen Permutation σ zwischen G_1 und G_2 berechnen. Um dies zu sehen, geben wir hier nochmals die Permutation σ an:

i	1	2	3	4	5	6	7	8
$\sigma(i)$	3	7	8	1	6	2	4	5

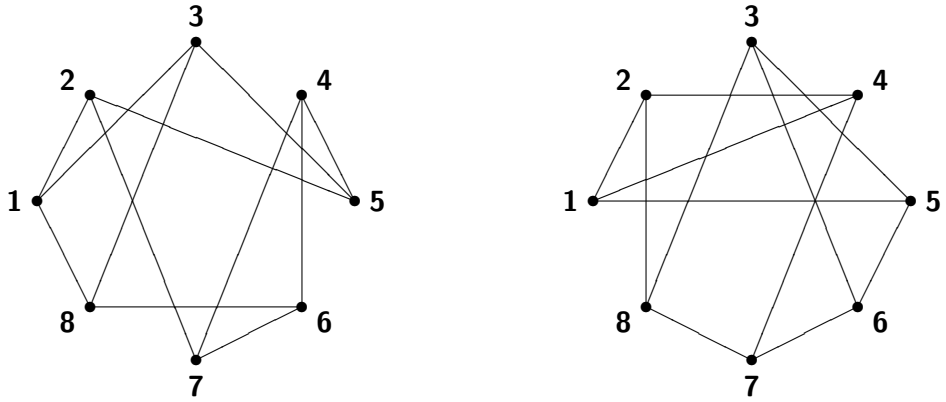
Wegen $\sigma(4) = 1$ entspricht der Knoten 1 von G_2 dem Knoten 4 von G_1, der wiederum wegen $\tau_1(4) = 3$ dem Knoten 3 von H entspricht. Genauso folgt aus $\sigma(6) = 2$ und $\tau_1(6) = 8$, dass der Knoten 2 von G_2 dem Knoten 6 von G_1 und damit dem Knoten 8 von H entspricht. Damit haben wir bereits die ersten beiden Zuordnungen der Permutation τ_2 gefunden:

i	1	2	3	4	5	6	7	8
$\tau_2(i)$	3	8	$*$	$*$	$*$	$*$	$*$	$*$

Nach dem gleichen Prinzip kann man auch die restlichen Einträge der Permutation τ_2 berechnen:

i	1	2	3	4	5	6	7	8
$\tau_2(i)$	3	8	6	4	7	1	2	5

Wenn man die beiden Graphen G_2 (links) und H (rechts) gegenüberstellt, kann man sich nochmals leicht von der Korrektheit der Permutation τ_2 überzeugen:



Aus der zufällig gewählten Permutation τ_1 hat Alice also den Graph H und die Permutation τ_2 berechnet. Genausogut hätte Alice eine Permutation τ_2 erfinden und daraus den Graph H und die passende Permutation τ_1 ermitteln können — dies ist für das folgende Verfahren ohne Belang.

Alice legt nun Boris den Graph H vor, der daraufhin genau eine von zwei Bitten äußern darf (die Wahl ist ihm überlassen), nämlich entweder

„Liebe Alice, nenne mir bitte die Permutation τ_1 zwischen G_1 und H.“

oder

„Liebe Alice, nenne mir bitte die Permutation τ_2 zwischen G_2 und H.“

Alice hat damit kein Problem — sie kennt beide Permutationen, beantwortet also artig die gestellte Frage von Boris, und Boris kann daraufhin ihre Antwort verifizieren. Alice und Boris wiederholen das genannte Protokoll noch mehrere Male (wir werden gleich noch sehen warum), d.h. Alice erfindet wieder eine neue Permutation τ_1, berechnet daraus einen neuen Graph H, usw. Beachten Sie, dass Boris jeweils nichts von der geheimen Permutation σ erfährt und er nur aufgrund der Kenntnis von τ_1 oder τ_2 auch nicht auf σ schließen kann. Trotzdem kann sich Boris sicher sein, dass wirklich die richtige Alice vor ihm steht, sofern jede Runde fehlerfrei abläuft. Dies kann man sich klarmachen, wenn man sich in die Situation eines Betrügers hineinversetzt.

Ein Betrüger — nennen wir ihn Carlo — hat keine Ahnung von der geheimen Permutation σ zwischen G_1 und G_2, und er kann sie auch nicht berechnen. Wenn nun Boris auf ihn zukommt, so kann Carlo natürlich trotzdem wie Alice eine Permutation τ_1 erfinden

und diese auf den öffentlich zugänglichen Graph G_1 anwenden. Allerdings kann er dann — anders als Alice — nicht die Permutation τ_2 zwischen G_2 und H bestimmen. Carlo lässt sich natürlich nichts anmerken und legt Boris den Graph H vor. Wenn Boris ihn nun nach der Permutation τ_1 zwischen G_1 und H fragt, so hat er Glück gehabt und kann die Frage beantworten. Sollte Boris sich jedoch nach der anderen Permuation τ_2 erkundigen, so muss Carlo passen, und der Schwindel fliegt auf.

Falls Carlo vermutet, dass Boris ihn nach der Permutation τ_2 zwischen G_2 und H befragen wird, so wird er seine frei wählbare Permutation natürlich auf G_2 anwenden, und kann, falls Boris dann wirklich nach τ_2 fragen sollte, ihm diese auch nennen. Allerdings kann Carlo nun die Permutation τ_1 nicht bestimmen. Sollte ihn Boris also wider seinen Erwartungen nach τ_1 befragen, muss Carlo erneut aufgeben.

Da Carlo nicht weiß, welche Frage ihm Boris stellen wird, stehen seine Chancen, eine Spielrunde heil zu überleben, bei 50%. Zwei Runden übersteht er nur mit einer Chance von $1:4$, drei Runden nur mit einer Chance von $1:8$, und allgemein n Runden nur mit einer Chance von $1:2^n$. Diese sind recht gering (siehe [2]):

Anzahl Runden	Die Betrugswahrscheinlichkeit ist kleiner als die Wahrscheinlichkeit ...
18	... dass Sie auf Ihrem nächsten Flug entführt werden.
20	... dass Sie im Zeitraum eines Jahres vom Blitz getroffen werden.
24	... dass Sie 6 Richtige im Lotto haben.
40	... dass Sie von einem Meteoriten erschlagen werden.

Boris kann sich also praktisch 100%ig sicher sein, dass die echte Alice vor ihm steht, und auch Alice selbst ist zufrieden. Denn Boris kann aus den kommunizierten Daten keine Rückschlüsse auf die geheime Permutation σ zwischen G_1 und G_2 ziehen, d.h. Alice bleibt am Ende die einzige Person, die das Geheimnis kennt. Selbst wenn irgendwelche kriminellen Personen den gesamten Dialog zwischen Alice und Boris abhören würden, kämen solche Personen nur zu dem gleichen Schluss, dass offenbar die „richtige“ Alice vor Boris steht. Das Geheimnis aber können sie nicht ermitteln.

5.3. Die $\mathcal{P} = \mathcal{NP}$ Frage oder „Wer wird Millionär?“

Wir haben in den beiden ersten Abschnitten gesehen, dass Probleme mit exponentieller Zeitkomplexität selbst für moderne Computer praktisch nicht zu lösen sind, da der Berechnungsaufwand schnell ins Unermessliche anwächst. Umgekehrt gelten Probleme mit polynomieller Zeitkomplexität, also mit einem Berechnungsaufwand von $O(n^k)$ für ein $k \in \mathbb{N}$, als (relativ) leicht lösbar. Man kann zwar argumentieren, dass ein Problem in z.B. DTIME(n^{100}) ebenfalls einen dramatisch steigenden Berechnungsaufwand besitzt, allerdings hat man in solchen Fällen bislang immer auch einen „guten“ Algorithmus mit einer viel niedrigeren Komplexität (also mit einem viel niedrigeren Polynomgrad) gefunden.

5.3.1 Definition Eine Sprache L ist in der Komplexitätsklasse $\mathcal{P}$ (engl. „*polynomial time*“) enthalten, wenn sie in der Klasse $\mathrm{DTIME}(n^k)$ für ein bestimmtes $k \in \mathbb{N}$ liegt. Also gilt

$$\mathcal{P} := \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(n^k) \ .$$

Probleme aus $\mathcal{P}$ lassen sich also schnell *berechnen*.

5.3.2 Beispiel Aus dem Beispiel 5.1.2 ist uns bekannt, dass Zahlen in polynomieller Zeit addiert und multipliziert werden können. Die zugehörigen Sprachen (für das Additionsproblem z.B. $L := \{(n, m, s) \,|\, n, m, s \in \mathbb{N}_0 \wedge n + m = s\}$) liegen also in $\mathcal{P}$. Genauso ist nach Beispiel 5.1.4 das Wortproblem für kontextfreie Sprachen in kubischer, also insbesondere in polynomieller Zeit lösbar. Also liegen auch alle kontextfreien Sprachen in $\mathcal{P}$.

5.3.3 Definition Eine Sprache L ist in der Komplexitätsklasse $\mathcal{NP}$ (engl. „*nondeterministic polynomial time*“) enthalten, wenn sie in der Klasse $\mathrm{NTIME}(n^k)$ für ein bestimmtes $k \in \mathbb{N}$ liegt. Also gilt analog

$$\mathcal{NP} := \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k) \ .$$

Probleme aus $\mathcal{NP}$ lassen sich also schnell durch entsprechende Hinweise *verifizieren*. Auch hier rührt der Name „nondeterministic“ daher, dass eine nichtdeterministische Turingmaschine einen entsprechenden Beweis selbst raten und anschließend verifizieren kann.

5.3.4 Beispiel Die Beispiele 5.1.6, 5.1.7 und 5.1.8 zeigen, dass das Graphenisomorphieproblem sowie die Sprachen der zusammengesetzten Zahlen und der erfüllbaren booleschen Formeln alle in $\mathcal{NP}$ liegen.

5.3.5 Satz Für jede beliebige Kompexitätsfunktion $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ gilt

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{NTIME}(f(n)) \ .$$

Beweis: Jedes Problem, welches sich in $f(n)$ Zeit berechnen lässt, lässt sich insbesondere auch in dieser Zeit verifizieren — den beigefügten Hinweis lässt man einfach unbenutzt liegen und berechnet die Lösung direkt, natürlich immer noch in $O(f(n))$ viel Zeit. Anders ausgedrückt: für ein Problem aus $\mathrm{DTIME}(f(n))$, dessen charakteristische Funktion $\chi_L(x)$ also in $O(f(|x|))$ viel Zeit für jede Instanz x berechenbar ist, können wir leicht die geforderte Funktion $\hat{\chi}_L(x, y)$ durch die Festlegung

$$\hat{\chi}_L(x, y) := \chi_L(x)$$

bereitstellen. Das Argument y mit dem Hinweis wird also nicht verwendet, da man die Lösung auch ohne diesen Hinweis schnell berechnen kann.

5.3.6 Korollar Es gilt $\mathcal{P} \subseteq \mathcal{NP}$.

Beweis: Dies folgt unmittelbar aus Satz 5.3.5 und der Beziehung

$$\mathcal{P} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(n^k) \subseteq \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k) = \mathcal{NP} \ .$$

□

Gilt auch die Umkehrung $\mathcal{P} \supseteq \mathcal{NP}$ und damit $\mathcal{P} = \mathcal{NP}$? Man könnte versucht sein direkt „nein“ zu sagen, da z.B. das Graphenisomorphieproblem (welches in $\mathcal{NP}$ liegt) scheinbar bei „schwierigen“ Instanzen das Ausprobieren aller Möglichkeiten verlangt. Dies ist aber keineswegs so offensichtlich, wie es zunächst scheint, denn es *könnte* vielleicht doch einen viel geschickteren Ansatz zur Lösung dieses Problems geben. Betrachten Sie hierfür nochmals die Sprache der zusammengesetzten Zahlen aus Beispiel 5.1.6. Auch hier könnte man der Ansicht sein, dass man für den Test auf Primalität notwendigerweise alle möglichen Teiler durchprobieren muss, was wie gezeigt mit exponentieller Komplexität einhergeht, also nicht in $\mathcal{P}$ liegt. Auch die Mehrheit der Experten auf diesem Gebiet war lange Zeit der Meinung, dass die Sprache der zusammengesetzten Zahlen von nichtpolynomieller deterministischer Komplexität ist. Es kam deshalb einer Sensation gleich, als im Jahr 2002 dann doch ein effizientes Verfahren aus $\mathcal{P}$ gefunden wurde. Der entsprechende Algorithmus wurde als *AKS–Primalitätstest* bekannt.

Es ist also keineswegs klar, ob es nicht auch für das Graphenisomorphieproblem und viele weitere Probleme aus $\mathcal{NP}$ effizientere Verfahren gibt.

Die $\mathcal{P} = \mathcal{NP}$ Frage ist seit ihrer Formulierung im Jahr 1971 das wichtigste ungelöste Problem in der theoretischen Informatik. Für die Lösung des $\mathcal{P} = \mathcal{NP}$ Problems hat das *Clay Mathematics Institute* eine Million US–Dollar als Preisgeld ausgelobt [7]. Das Problem hat auch für die Praxis große Bedeutung. Das im letzten Abschnitt gezeigte Zero–Knowledge–Verfahren wird z.B. völlig wertlos, wenn $\mathcal{P} = \mathcal{NP}$ gelten sollte und in diesem Zusammenhang ein effizienter Test auf Graphenisomorphie bekannt wird. Denn dann kann das Geheimnis (also die unbekannte Permutation zwischen den beiden bekannten Graphen) von jeder Person leicht selbst berechnet werden.

Im nächsten Abschnitt werden wir gute Argumente für die Annahme $\mathcal{P} \neq \mathcal{NP}$ angeben. Ein konkreter Beweis scheint jedoch sehr schwierig zu sein. Dies zeigen die großen Anstrengungen, die man bislang auf diesem Gebiet unternommen hat. Schon früh wurde bekannt, dass sich die $\mathcal{P} = \mathcal{NP}$ Frage mit vielen Beweistechniken nicht lösen lässt, die sich an anderer Stelle bewährt haben. Trotzdem ist vielleicht eine Lösung mit einem anderen (noch unbekannten) Argument ganz einfach. Vielleicht gelingt es ja gerade Ihnen, hier fündig (und Millionär) zu werden.

5.4. Hartnäckige Probleme

Unter allen Problemen aus $\mathcal{NP}$ gibt es bestimmte Probleme, für die ein effizienter Algorithmus $\mathcal{P} = \mathcal{NP}$ implizieren würde, d.h. dann wären sogar *alle* Probleme aus $\mathcal{NP}$ in

deterministischer Polynomialzeit lösbar. Umgekehrt bedeutet dies, dass sich im (von der Mehrheit der Experten vermuteten) Fall $\mathcal{P} \neq \mathcal{NP}$ diese Probleme nicht effizient lösen lassen, d.h. es handelt sich gewissermaßen um die „hartnäckigsten“ Probleme aus $\mathcal{NP}$.

Für eine formale Definition dieser Probleme benötigen wir zunächst eine Verschärfung des Reduktionsbegriffs aus dem vierten Kapitel.

5.4.1 Definition Es seien A und B zwei Sprachen über einem Alphabet Σ. Dann heißt A *auf B polynomiell reduzierbar*, falls es eine totale und in deterministischer Polynomialzeit berechenbare Funktion $f : \Sigma^* \longrightarrow \Sigma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt:

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

Man schreibt dafür dann auch $A \leq_p B$.

Im Vergleich zur „normalen“ Reduzierbarkeit (vgl. Def. 4.5.16 auf Seite 157) wird also zusätzlich verlangt, dass die Reduktionsfunktion f „schnell“ berechnet werden kann. Für f muss demnach ein Programm in z.B. Java sowie eine Zahl $k \in \mathbb{N}$ vorliegen, so dass das Programm gemessen an der Eingabegröße eines Wortes $x \in \Sigma^*$ nur $O(|x|^k)$ viel Zeit zur Berechnung von $f(x)$ benötigt.

Alle bislang verwendeten Reduktionen waren polynomielle Reduktionen. Betrachten Sie z.B. nochmals die Reduktion aus dem Beispiel 4.5.17 auf Seite 157:

$$f(w) := M\#w$$

Hier wird einfach ein Wort w durch einen neuen, aber immer gleichen Präfix (nämlich $M\#$) vorne ergänzt. Hat w die Länge n, so übersteigt der Aufwand zur Erstellung des Ergebnisses dann sicherlich nicht $O(c+n) = O(n)$, wobei c die konstante Länge von dem Wort $M\#$ ist. Die Reduktion arbeitet also sogar in Linearzeit.

Ähnlich war es beim Beweis (ab Seite 162) von Satz 4.6.1, bei dem die Beschreibung einer Turingmaschine auf einen Satz von *PCP*–Dominosteinen reduziert wurde. Beachten Sie, dass aus der Turingtafel heraus nur die Steintypen erzeugt werden müssen — danach ist das Reduktionsergebnis bereits fertig konstruiert. Die Länge des eventuellen *PCP*–Lösungswortes hängt später von der Laufzeit der umgewandelten Turingmaschine ab und könnte allein zum Hinschreiben sehr viel Zeit verbrauchen, aber das Lösungswort selbst gehört gar nicht zum Reduktionsergebnis — sondern eben nur die Steintypen, und diese lassen sich direkt aus der Turingtafel ablesen.

Zur Betonung des Unterschieds wird nun eine nichtpolynomielle Reduktion aufgeführt.

5.4.2 Beispiel Sei $\Sigma := \{0, 1, \ldots, 9\}$ das Alphabet, mit dem wir alle natürlichen Zahlen darstellen können. Das folgende Programm reduziert die Sprache der zusammengesetzten Zahlen aus Beispiel 5.1.6 auf die einelementige Sprache $\{1\}$:

```
int reduktionsFunktion(int n) {
   for (int i = 2; i < n; i++) {
      if (n % i == 0) {
         return 1;
      }
   }
   return 0;
}
```

Das Programm realisiert die naive Methode zum Test auf eine zusammengesetzte Zahl, d.h. es probiert alle möglichen Teiler durch. Die Reduktionsfunktion arbeitet natürlich korrekt, doch im Vergleich zur Eingabelänge der Zahl n wird dazu exponentiell viel Zeit benötigt — dies wurde bereits im Beispiel 5.1.6 aufgezeigt.

5.4.3 Definition Eine Sprache L heißt $\mathcal{NP}$*–hart* oder $\mathcal{NP}$*–schwer*, wenn für jede Sprache $A \in \mathcal{NP}$ stets $A \leq_p L$ gilt. Eine $\mathcal{NP}$–harte Sprache L heißt $\mathcal{NP}$*–vollständig*, wenn zusätzlich $L \in \mathcal{NP}$ liegt.

$\mathcal{NP}$–harte Probleme sind ein möglicher Ansatz zur Lösung der $\mathcal{P} = \mathcal{NP}$ Frage. Es gilt nämlich:

5.4.4 Satz Sei $L \subseteq \Sigma^*$ ein $\mathcal{NP}$–hartes Problem. Dann gilt

$$L \in \mathcal{P} \implies \mathcal{P} = \mathcal{NP} \ .$$

Für den Beweis von Satz 5.4.4 benötigen wir zwei vorbereitende Lemmata:

5.4.5 Lemma Die Relation $\leq_p$ ist transitiv.

Beweis: Erfüllen drei Sprachen $A, B, C \subseteq \Sigma^*$ die Bedingungen $A \leq_p B$ und $B \leq_p C$, so existieren zwei Reduktionen $f, g : \Sigma^* \longrightarrow \Sigma^*$ mit

$$x \in A \iff f(x) \in B \quad \text{und} \quad x \in B \iff g(x) \in C \ .$$

Beide Reduktionen benötigen nur polynomiell viel Zeit für die Berechnung ihrer Ergebnisse, d.h. es existieren zwei Zahlen $k, r \in \mathbb{N}$, so dass für alle $x \in \Sigma^*$ die Funktionswerte $f(x)$ und $g(x)$ in höchstens $O(|x|^k)$ bzw. $O(|x|^r)$ viel Zeit berechnet werden können. Ferner wird hierdurch die Abschätzung $|f(x)| = O(|x|^k)$ impliziert, denn allein zum Hinschreiben des Ergebnisses würde ein längerer Funktionswert auch mehr als $O(|x|^k)$ viel Zeit erfordern.

Dann aber gilt auch

$$x \in A \iff f(x) \in B \iff g(f(x)) \in C \ ,$$

d.h. die Hintereinanderausführung von f und g ist eine Reduktion von A auf C, und für die Berechnung von $g(f(x))$ wird höchstens

$$O(|x|^k) + O(|f(x)|^r) = O(|x|^k) + O((|x|^k)^r) = O(|x|^{k \cdot r})$$

viel Zeit benötigt. Also erfolgt auch die Reduktion von A auf C immer noch in Polynomialzeit. Aus $A \leq_p B$ und $B \leq_p C$ folgt also stets $A \leq_p C$. □

5.4.6 Lemma Sei $L \subseteq \Sigma^*$ eine Sprache mit o.B.d.A. $1 \in \Sigma$ (im Zweifelsfall kann das Symbol 1 zu Σ einfach hinzugefügt werden). Dann gilt

$$L \in \mathcal{P} \iff L \leq_p \{1\} \ .$$

Beweis: Wir müssen beide Richtungen der Äquivalenz zeigen:

„$\Longrightarrow$“: Nach Voraussetzung ist die charakteristische Funktion $\chi_L : \Sigma^* \longrightarrow \{0, 1\}$ mit

$$\chi_L(x) := \begin{cases} 1 & \text{falls } x \in L \\ 0 & \text{sonst} \end{cases}$$

in deterministischer Polynomialzeit berechenbar. Also gilt

$$x \in L \iff \chi_L(x) = 1 \iff \chi_L(x) \in \{1\} \ ,$$

d.h. als Reduktionsfunktion können wir einfach χ_L wählen und haben damit $L \leq_p \{1\}$ gezeigt.

„$\Longleftarrow$“: Umgekehrt gelte nun $L \leq_p \{1\}$, d.h. es existiert eine Reduktion $f : \Sigma^* \longrightarrow \Sigma^*$ mit

$$x \in L \iff f(x) \in \{1\}$$

für alle $x \in \Sigma^*$, die in deterministischer Polynomialzeit berechenbar ist. Also lässt sich auch die charakteristische Funktion χ_L schnell berechnen. Diese ermittelt im Wesentlichen nur den Wert $f(x)$ und gibt im Fall $f(x) = 1$ den Wert 1 sowie in allen anderen Fällen den Wert 0 zurück. □

Nun können wir Satz 5.4.4 leicht beweisen:

Beweis: Wir nehmen $L \in P$ an und weisen nach, dass dann jede Sprache $A \in \mathcal{NP}$ auch in $\mathcal{P}$ liegt, so dass $\mathcal{NP} \subseteq \mathcal{P}$ und somit $\mathcal{P} = \mathcal{NP}$ folgt. Sei also $A \in \mathcal{NP}$ beliebig. Wegen $L \in \mathcal{P}$ und Lemma 5.4.6 folgt zunächst $L \leq_p \{1\}$. Aufgrund der $\mathcal{NP}$–Härte von L gilt aber auch $A \leq_p L$, so dass sich wegen der Transitivität von $\leq_p$ (siehe Lemma 5.4.5) $A \leq_p \{1\}$ ergibt. Eine nochmalige Anwendung von Lemma 5.4.6 beweist somit wie gewünscht $A \in \mathcal{P}$. □

Wie fassen unsere bisherigen Erkenntnisse wie folgt zusammen:

5.4.7 Korollar Sei L ein $\mathcal{NP}$–vollständiges Problem. Dann sind äquivalent:

a) $\mathcal{P} = \mathcal{NP}$

b) L ist effizient lösbar (d.h. L liegt in $\mathcal{P}$)

c) alle $\mathcal{NP}$–vollständigen Probleme sind effizient lösbar

Beweis: Wir zeigen die zyklische Beweiskette a) ⇒ c) ⇒ b) ⇒ a):

„a) ⇒ c)“: Alle $\mathcal{NP}$–vollständigen Probleme sind in $\mathcal{NP}$ enthalten. Im Fall $\mathcal{P} = \mathcal{NP}$ liegen sie somit sogar in $\mathcal{P}$, d.h. sie müssen effizient lösbar sein.

„c) ⇒ b)“: Dies ist offensichtlich, denn wenn alle $\mathcal{NP}$–vollständigen Probleme effizient lösbar sind, dann insbesondere auch L selbst.

„b) ⇒ a)“: Jedes $\mathcal{NP}$–vollständige Problem ist immer auch $\mathcal{NP}$–hart. Also folgt die Aussage direkt aus Satz 5.4.4. □

Korollar 5.4.7 belegt, warum die Mehrheit der theoretischen Informatiker von der Ungleichheit $\mathcal{P} \neq \mathcal{NP}$ ausgeht und somit wahrscheinlich kein einziges $\mathcal{NP}$–vollständiges Problem effizient lösbar ist. Denn ansonsten wären auf einen Schlag *alle* $\mathcal{NP}$–vollständigen Probleme effizient lösbar, und darunter sind viele, bei denen man schon jahrzehntelang vergeblich nach effizienten Algorithmen gesucht hat. $\mathcal{NP}$–vollständige Probleme gelten deshalb als besonders „hartnäckig“.

Eine Übersicht über mehrere hundert $\mathcal{NP}$–vollständige Probleme sind in dem Buch von Garey und Johnson [10] aufgelistet. Viele von diesen sind für die Praxis relevant. Wir haben eines dieser Probleme sogar schon kennengelernt:

5.4.8 Satz *SAT* (das *Erfüllbarkeitsproblem* der Aussagenlogik, siehe Beispiel 5.1.7) ist $\mathcal{NP}$–vollständig.

Wir werden diese Aussage in einer noch schärferen Form beweisen. Das Erfüllbarkeitsproblem ist nämlich sogar schon dann $\mathcal{NP}$–vollständig, wenn die aussagenlogischen Formeln in der sogenannten *konjunktiven Normalform* vorliegen:

5.4.9 Definition Sind $A, B, C, \ldots$ boolesche Variablen, so werden sowohl sie selbst als auch ihre Negationen $\neg A, \neg B, \neg C, \ldots$ als *Literale* bezeichnet. Eine *Klausel* ist eine disjunktive Verknüpfung mehrerer Literale, also z.B. $\neg A \vee B \vee C \vee D$ oder $A \vee \neg C \vee C$ (wobei die zweite Klausel natürlich immer wahr ist). Werden schließlich mehrere Klauseln zunächst geklammert und anschließend durch Konjunktionen miteinander verknüpft, so erhält man eine Formel in *konjunktiver Normalform* oder kurz *KNF*.

5.4.10 Beispiel Die bereits im Beispiel 5.1.7 erwähnte Formel

$$(\neg A \vee \neg C \vee D) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg B \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg B \vee D)$$

lag in KNF vor.

5.4.11 Lemma Jede aussagenlogische Formel mit k Variablen kann in eine gleichwertige KNF–Formel mit höchstens 2^k Klauseln umgewandelt werden.

Beweis: Das Umwandlungsverfahren kann durch ein einfaches Beispiel demonstriert werden. Wir wandeln die Formel

$$(A \wedge (\neg B \vee C)) \vee \neg(A \vee \neg C) \vee (B \wedge C)$$

mit den drei Variablen A, B und C in KNF um. Dazu gehen wir alle $2^3 = 8$ möglichen Belegungen dieser Variablen durch und stellen eine Wahrheitstabelle auf, ganz so, wie wir es aus dem ersten Abschnitt 1.1 über die Aussagenlogik kennen gelernt haben:

A	B	C	Formelwert
W	W	W	W
W	W	F	F
W	F	W	W
W	F	F	W

A	B	C	Formelwert
F	W	W	W
F	W	F	F
F	F	W	W
F	F	F	F

Nun stellen wir eine Bedingung für jede Belegung zusammen, für die die obige Formel falsch ist. Eine solche Belegung ist z.B. $A = \mathrm{W}$, $B = \mathrm{W}$ und $C = \mathrm{F}$. Dies können wir durch die Bedingung $A \wedge B \wedge \neg C$ ausdrücken. Analog ergeben sich für die beiden anderen Fälle die Bedingungen $\neg A \wedge B \wedge \neg C$ und $\neg A \wedge \neg B \wedge \neg C$. Insgesamt gesehen ist die Formel also genau dann falsch, wenn einer dieser drei Fälle zutrifft, d.h. wenn

$$(A \wedge B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge \neg C)$$

wahr ist. Wenn wir demnach diesen Ausdruck zu

$$\neg\Big((A \wedge B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge \neg C)\Big)$$

negieren, ist er also genau zur Ausgangsformel äquivalent. Die Anwendung der Regeln von DeMorgan (siehe Seite 11) ergibt dann

$$\neg(A \wedge B \wedge \neg C) \wedge \neg(\neg A \wedge B \wedge \neg C) \wedge \neg(\neg A \wedge \neg B \wedge \neg C)$$

und weiter

$$(\neg A \vee \neg B \vee C) \wedge (A \vee \neg B \vee C) \wedge (A \vee B \vee C) \ ,$$

also genau den gesuchten Ausdruck in KNF. Jede Bedingung steuert dabei eine Klausel zum KNF–Ausdruck bei.

Im allgemeinen Fall fallen bei k Variablen höchstens 2^k Bedingungen an, so dass die Anzahl der Klauseln ebenfalls durch 2^k begrenzt ist. Natürlich kann im Allgemeinen die resultierende Formel noch vereinfacht werden, für unsere Zwecke reicht das bewiesene Resultat jedoch aus. □

5.4.12 Definition Die *Länge* eines booleschen Ausdrucks ist die Anzahl aller einzelnen Zeichen. Wenn man also ein entsprechendes Alphabet Σ definiert, in dem jedes einzelne verwendete Zeichen vorkommt, so kann man den Ausdruck als ein gewöhnliches Wort über Σ auffassen, und die Länge entspricht der normalen Wortlänge. Die Länge der KNF–Formel aus Beispiel 5.4.10

$$(\neg A \vee \neg C \vee D) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg B \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg B \vee D)$$

beträgt also z.B. 39, und das Alphabet wäre $\Sigma = \{A, B, C, D, \neg, \wedge, \vee, (,)\}$.

Eine genügend lange KNF–Formel kann auch entsprechend viele Variablen aufweisen, d.h. das zugrundeliegende Alphabet Σ müsste evtl. bei steigender Ausdruckslänge immer größer gewählt werden. Wenn wir aber das SAT–Problem als formale Sprache definieren wollen, so müssen wir uns auf irgendein endliches Alphabet festlegen. Dies ist für den Preis einer logarithmischen Formelverlängerung kein Problem.

5.4.13 Lemma Jede aussagenlogische Formel der Länge n kann gleichwertig als eine Formel der Länge $O(n \log n)$ über dem Alphabet $\{0, 1, \neg, \wedge, \vee, (,)\}$ ausgedrückt werden.

Beweis: Eine Formel der Länge n kann (grob abgeschätzt) höchstens n verschiedene Variablen $A, B, C, \ldots$ enthalten. Wir nummerieren nun alle Variablen mit den Zahlen $1, 2, 3, \ldots$ neu durch und ersetzen jedes Variablenvorkommen in der Formel durch die entsprechende Zahl in binärer Kodierung. Aus z.B.

$$(\neg A \vee \neg C \vee D) \wedge (\neg A \vee B \vee \neg C)$$

wird so

$$(\neg 1 \vee \neg 11 \vee 100) \wedge (\neg 1 \vee 10 \vee \neg 11) \ .$$

Die vorgenommenen Ersetzungen sind dabei $A = 1$, $B = 10$, $C = 11$ und $D = 100$.

Für jede Umkodierung sind nicht mehr als $O(\log n)$ viele Bits erforderlich, d.h. alle (höchstens n) Ersetzungen benötigen zusammen nicht mehr als $O(n \log n)$ viele Nullen und Einsen. Da der Rest der Formel unverändert bleibt, ist somit die Gesamtlänge der neuen Formel auf $O(n \log n) + O(n) = O(n \log n)$ begrenzt. □

Nun können wir den gewünschten Beweis angehen:

5.4.14 Satz (Cook[1]) Das *Erfüllbarkeitsproblem der Aussagenlogik*

$$SAT := \{F \mid F \text{ ist eine erfüllbare aussagenlogische Formel in KNF}\}$$

ist $\mathcal{NP}$–vollständig. (Jede Formel sei dabei mit der Technik aus Lemma 5.4.13 behandelt worden, so dass SAT formal eine Sprache über dem Alphabet $\{0, 1, \neg, \wedge, \vee, (,)\}$ darstellt.)

[1] Stephen A. Cook, *1939 Buffalo, US–amerikanischer Informatiker, Professor für Infomatik in Toronto.

Beweis: Wir haben uns bereits im Beispiel 5.1.7 klar gemacht, dass SAT in $\mathcal{NP}$ enthalten ist, denn mit einer passenden Variablenbelegung als Beweis kann man leicht den Wahrheitsgehalt jeder erfüllbaren Formel verifizieren.

Sei deshalb jetzt L eine beliebige Sprache aus $\mathcal{NP}$. Wir müssen zeigen, dass $L \leq_p SAT$ gilt.

In der Def. 5.3.3 der Klasse $\mathcal{NP}$ wurde dargelegt, dass für jede Sprache aus $\mathcal{NP}$ eine zugehörige NTM existiert, die die Sprache akzeptiert und zudem polynomial zeitbeschränkt ist. Ist $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ eine solche NTM für unsere konkrete Sprache L, so gibt es also eine bestimmte Polynomfunktion

$$p : \mathbb{N}_0 \longrightarrow \mathbb{N}_0 \ ,$$

so dass M zur Akzeptanz eines Wortes $x \in L$ höchstens $p(|x|)$ viel Zeit benötigt. Beachten Sie, dass M dann auch nur höchstens $p(|x|)$ Speicherfelder links oder rechts von der Ausgangsposition auf dem Turingband erreichen kann. Wir versehen diese Speicherfelder mit Indizes von $-p(n)$ bis $p(n)$. Das Startfeld mit dem ersten Zeichen x_1 eines Eingabewortes $x = x_1 x_2 \ldots x_n \in \Sigma^*$ hat dann den Index 0, x_2 steht im Feld mit dem Index 1, usw.

Wir werden in Abhängigkeit vom Eingabewort x den Aufbau einer KNF–Formel F_x skizzieren, die genau dann erfüllbar ist, wenn x aus L stammt. Es wird also

$$x \in L \iff F_x \in SAT$$

gelten. F_x wird dazu einen akzeptierenden Lauf von M simulieren.

In F_x werden die folgenden booleschen Variablen enthalten sein:

Variable	Bereiche der Indizes	Bedeutung
$\mathsf{state}_{t,z}$	$t = 0, 1, \ldots, p(n)$ $z \in Z$	$\mathsf{state}_{t,z}$ ist genau dann wahr, wenn M nach t Schritten im Zustand z ist.
$\mathsf{pos}_{t,i}$	$t = 0, 1, \ldots, p(n)$ $i = -p(n), \ldots, p(n)$	$\mathsf{pos}_{t,i}$ ist genau dann wahr, wenn sich der Schreib–Lese–Kopf von M nach t Schritten an Position i befindet.
$\mathsf{band}_{t,i,a}$	$t = 0, 1, \ldots, p(n)$ $i = -p(n), \ldots, p(n)$ $a \in \Gamma$	$\mathsf{band}_{t,i,a}$ ist genau dann wahr, wenn sich nach t Schritten auf Bandposition i das Zeichen a befindet.

Bevor wir klären, welche Bedingungen diese Variablen im Einzelnen erfüllen müssen, besprechen wir noch eine kleine Hilfsformel. Es wird nämlich häufiger vorkommen, dass

von bestimmten Variablen $x_1, \ldots, x_m$ immer nur genau eine wahr sein soll. Hierfür setzen wir den folgenden Ausdruck ein:

$$G(x_1, x_2, \ldots, x_m) \;:=\; \Big(\bigvee_{i=1}^{m} x_i \Big) \wedge \Big(\bigwedge_{i \neq j} \neg x_i \vee \neg x_j \Big) \;.$$

Für drei Variablen x_1, x_2, x_3 ergibt sich z.B.

$$G(x_1, x_2, x_3) \;=\; (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3) \;.$$

Für G gilt:

- Der vordere Teil drückt aus, dass mindestens eine der Variablen wahr sein muss. Der hintere Teil besagt dagegen, dass nicht gleichzeitig zwei Variablen wahr sein dürfen. Also folgt:

  $$G(x_1, x_2, \ldots, x_m) = \mathrm{W} \iff \text{es existiert genau ein } i \text{ mit } x_i = \mathrm{W} \;.$$

- Die Länge von G im vorderen Teil beträgt offenbar $O(m)$. Im hinteren Teil entstehen dagegen

  $$(m-1) + (m-2) + \cdots + 2 + 1 = \frac{m(m-1)}{2} = O(m^2)$$

  viele Klauseln mit je zwei Literalen (zum Beweis der Summenformel verweisen wir auf das Beispiel in Anhang A). Somit beträgt die Gesamtlänge von G bei m Variablen $O(m^2)$.

- G ist in KNF.

Wir stellen nun alle Bedingungen in KNF zusammen, die eine korrekte Simulation von M beschreiben. M besitze das Arbeitsalphabet $\Gamma = \{a_1, a_2, \ldots, a_\ell\}$ und die Zustandsmenge $Z = \{z_0, z_1, \ldots, z_k\}$, wobei wie immer z_0 der Startzustand sein soll.

- Zu jedem Zeitpunkt $0 \leq t \leq p(n)$ befindet sich M in genau einem Zustand:

  $$\bigwedge_{t=0}^{p(n)} G(\mathsf{state}_{t,z_0}, \ldots, \mathsf{state}_{t,z_k}) \;. \tag{1}$$

  Die Länge von (1) ist $(p(n)+1) \cdot O((k+1)^2) = O(p(n))$.

- Zu jedem Zeitpunkt $0 \leq t \leq p(n)$ befindet sich der Schreib–Lese–Kopf von M unter genau einem Zeichen auf dem Turingband:

  $$\bigwedge_{t=0}^{p(n)} G(\mathsf{pos}_{t,-p(n)}, \ldots, \mathsf{pos}_{t,p(n)}) \;. \tag{2}$$

  Die Länge von (2) ist $(p(n)+1) \cdot O((2p(n)+1)^2) = O(p^3(n))$.

- Zu jedem Zeitpunkt $0 \leq t \leq p(n)$ befindet sich an jeder Position $-p(n) \leq i \leq p(n)$ genau ein Zeichen auf dem Turingband:

$$\bigwedge_{t=0}^{p(n)} \bigwedge_{i=-p(n)}^{p(n)} G(\mathsf{band}_{t,i,a_1}, \ldots, \mathsf{band}_{t,i,a_\ell}) \ . \tag{3}$$

  Die Länge von (3) ist $(p(n)+1) \cdot (2p(n)+1) \cdot O(\ell^2) = O(p^2(n))$.

- Zum Zeitpunkt $t = 0$ befindet sich M in seiner Startkonfiguration, d.h. der aktuelle Zustand lautet z_0, auf dem Band steht die Eingabe $x = x_1 \ldots x_n$, davor und dahinter stehen nur Blanksymbole, und der Kopf befindet sich unter dem ersten Zeichen der Eingabe:

$$\mathsf{state}_{0,z_0} \wedge \mathsf{pos}_{0,0} \wedge \bigwedge_{i=0}^{n-1} \mathsf{band}_{0,i,x_{i+1}} \wedge \bigwedge_{i=-p(n)}^{-1} \mathsf{band}_{0,i,\square} \wedge \bigwedge_{i=n}^{p(n)} \mathsf{band}_{0,i,\square} \ . \tag{4}$$

  Die Länge von (4) ist $O(1+1+(2p(n)+1)) = O(p(n))$.

- Die nächste Formel beschreibt die Auswirkungen der einzelnen Übergänge durch diese Bedingung: wenn sich M zum Zeitpunkt t mit dem Schreib–Lese–Kopf an der i–ten Position aufhält, dabei auf dem Band das Zeichen a vorfindet und zudem momentan den Zustand z einnimmt, dann soll — gemäß einem der evtl. mehreren möglichen nichtdeterministischen Übergänge $(z', b, y) \in \delta(z, a)$ — zum Zeitpunkt $t+1$ nach z' gewechselt, auf dem Band das Symbol b eingetragen und der Kopf wie vorgeschrieben bewegt werden können. Die Kopfbewegungen L, N, und R werden hierbei in dem Index y durch die drei Zahlenwerte -1, 0 und 1 kodiert:

$$\mathsf{state}_{t,z} \wedge \mathsf{pos}_{t,i} \wedge \mathsf{band}_{t,i,a} \quad \Rightarrow \bigvee_{(z',b,y)\in\delta(z,a)} \mathsf{state}_{t+1,z'} \wedge \mathsf{band}_{t+1,i,b} \wedge \mathsf{pos}_{t+1,i+y} \ .$$

  Diesen booleschen Ausdruck bezeichnen wir mit $\delta_{t,i,a,z}$. Er ist zwar nicht direkt in KNF, aber er besitzt bei festem t, i, a und z nur höchstens

$$1+1+1+|Z|+|\Gamma|+3 = 3+(k+1)+\ell+3 = k+\ell+7$$

  viele Variablen. Nach Lemma 5.4.11 besteht er also auch nach einer Umformung in KNF nur aus höchstens

$$2^{k+\ell+7}$$

  vielen Klauseln, und jede der Klauseln besteht aus höchstens $k+\ell+7$ vielen Literalen. Die maximale Länge einer entsprechenden KNF–Formel für $\delta_{t,i,a,z}$ ist also

$$O(2^{k+\ell+7} \cdot (k+\ell+7)) = O(1)$$

  und damit konstant, da dieser Ausdruck nicht von der Eingabelänge n abhängt.

Die Formel $\delta_{t,i,a,z}$ muss nun für alle möglichen Kombinationen der vier Indizes t, i, a und z wahr sein, um die Korrektheit aller Bandveränderungen sicherzustellen. Wir stellen also die Bedingung

$$\bigwedge_{t=0}^{p(n)-1} \bigwedge_{i=-p(n)}^{p(n)} \bigwedge_{a\in\Gamma} \bigwedge_{z\in Z} \delta_{t,i,a,z} \tag{5}$$

auf. Ihre Länge beträgt $p(n) \cdot (2 \cdot p(n) + 1) \cdot \ell \cdot (k + 1) \cdot O(1) = O(p^2(n))$.

- Auf jeweils allen anderen Feldern darf nichts verändert werden, d.h. wenn zu einem Zeitpunkt t an einer bestimmten i–ten Position ein Symbol a eingetragen ist und sich der Schreib–Lese–Kopf dort nicht aufhält, so muss sich dieses Symbol auch noch zum darauffolgenden Zeitpunkt $t + 1$ dort wiederfinden. Wir verlangen also, dass jeweils

  $$\neg \mathsf{pos}_{t,i} \wedge \mathsf{band}_{t,i,a} \;\Rightarrow\; \mathsf{band}_{t+1,i,a}$$

  erfüllt ist, was zu

  $$\mathsf{pos}_{t,i} \vee \neg \mathsf{band}_{t,i,a} \vee \mathsf{band}_{t+1,i,a}$$

  umformuliert werden kann. Für alle Zeitpunkte t, alle Positionen i und alle Symbole a erhalten wir daraus die Bedingung

  $$\bigwedge_{t=0}^{p(n)-1} \bigwedge_{i=-p(n)}^{p(n)} \bigwedge_{a\in\Gamma} \mathsf{pos}_{t,i} \vee \neg \mathsf{band}_{t,i,a} \vee \mathsf{band}_{t+1,i,a} \;. \tag{6}$$

  Die Länge von (6) ist $p(n) \cdot (2 \cdot p(n) + 1) \cdot \ell \cdot O(1) = O(p^2(n))$.

- Schließlich müssen wir noch sicherstellen, dass M während der Verarbeitung in einen Endzustand gelangt. Dazu modifizieren wir ganz zu Anfang die Turingtafel von M. Für jeden Endzustand z und jedes Symbol $a \in \Gamma$ fügen wir einen Übergang $\delta(z, a) = \{(z, a, N)\}$ hinzu, d.h. M kann in jedem Endzustand mit neutralen Übergängen verharren, bis die komplette Zeit $p(n)$ verstrichen ist. Also reicht die folgende Testbedingung aus:

  $$\bigvee_{z\in E} \mathsf{state}_{p(n),z} \;. \tag{7}$$

  Die Länge von (7) ist $|E| \cdot O(1) = O(1)$.

Alle genannten Bedingung müssen gleichzeitig erfüllt sein, d.h. wir erstellen als Gesamtergebnis aus der Eingabe x die Formel

$$F_x \;:=\; (1) \wedge (2) \wedge (3) \wedge (4) \wedge (5) \wedge (6) \wedge (7) \;.$$

Diese Formel ist in KNF, und ihre Gesamtlänge beträgt

$$O\Big(p(n) + p^3(n) + p^2(n) + p(n) + p^2(n) + p^2(n) + 1\Big) = O(p^3(n)) \;.$$

Eine Reduktionsfunktion, die aus x die Formel F_x berechnet, ist z.B. in Java leicht zu realisieren. Die entsprechende Methode besteht nur aus diversen `for`–Schleifen, die 1:1 die obigen Formelbestandteile niederschreiben. Die Reduktion ist korrekt, denn im Fall $x \in L$ gibt es eine nichtdeterministische Berechnung mit höchstens $p(n)$ vielen Schritten, die zu einer akzeptierenden Konfiguration führt. Durch die erwähnte Ergänzung der Turingtafel bei der Teilformel (7) gibt es dann immer auch eine entsprechende Berechnung mit genau $p(n)$ vielen Schritten. Also kann man alle booleschen Variablen mit den passenden zugehörigen Werten (bezogen auf diese akzeptierende Rechnung) belegen und sieht damit, dass F_x erfüllbar ist.

Ist umgekehrt F_x durch eine gewisse Variablenbelegung erfüllbar, so müssen insbesondere alle Bedingungen (1)–(7) wahr sein. Hieraus kann man dann direkt einen akzeptierenden Lauf von M ablesen. Also gilt $x \in L$.

Die Reduktionsfunktion unterzieht die erstellte Formel gemäß Lemma 5.4.13 noch einer abschließenden Verkleinerung des zugrundeliegenden Alphabets. Dabei verlängert sich F_x von $O(p^3(n))$ auf

$$O(p^3(n)\log(p^3(n))) = O(p^3(n)\cdot 3\log(p(n))) = O(p^3(n)\log(p(n)))\ ,$$

also nur um einen logarithmischen Faktor. Somit hat F_x nach wie vor nur polynomielle Länge, und es gilt

$$x \in L \iff F_x \in SAT\ .$$

Also folgt $L \leq_P SAT$ für jede beliebige Sprache $L \in \mathcal{NP}$, d.h. der Beweis des Satzes von Cook ist abgeschlossen. □

Das SAT–Problem bleibt sogar dann $\mathcal{NP}$–vollständig, wenn man die maximale Anzahl der Literale pro Klausel stark einschränkt.

5.4.15 Definition Eine KNF–Formel liegt in *k–KNF* vor, wenn jede Klausel höchstens k Literale enthält.

5.4.16 Beispiel Die im Beispiel 5.4.10 erwähnte Formel

$$(\neg A \vee \neg C \vee D) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg B \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg B \vee D)$$

liegt in 3–KNF vor.

5.4.17 Satz Die Sprache

$$\textit{3–SAT} := \{F \mid F \text{ ist eine erfüllbare aussagenlogische Formel in 3–KNF}\}$$

ist $\mathcal{NP}$–vollständig.

Bevor wir den Beweis von Satz 5.4.17 angehen, besorgen wir uns zunächst ein einfacheres Kriterium für die $\mathcal{NP}$–Vollständigkeit einer Sprache. Es gilt nämlich:

5.4.18 Satz Sei L eine beliebige Sprache. Gilt dann $L' \leq_p L$ für eine $\mathcal{NP}$–harte Spache L', so ist auch L $\mathcal{NP}$–hart und somit im Fall $L \in \mathcal{NP}$ sogar $\mathcal{NP}$–vollständig.

Beweis: Sei $A \in \mathcal{NP}$ beliebig. Aufgrund der $\mathcal{NP}$–Härte von L' gilt $A \leq_p L'$, wegen $L' \leq_p L$ und der Transitivität von $\leq_p$ (Lemma 5.4.5) folglich sogar $A \leq_p L$. Also lässt sich jede Sprache $A \in \mathcal{NP}$ in deterministischer Polynomialzeit auf L reduzieren, d.h. L muss ebenfalls $\mathcal{NP}$–hart sein. □

Nun können wir den Beweis von Satz 5.4.17 angeben:

Beweis: Wir zeigen *3–SAT* $\in \mathcal{NP}$ sowie $SAT \leq_p$ *3–SAT*. Nach den Sätzen 5.4.14 und 5.4.18 ergibt sich daraus die $\mathcal{NP}$–Vollständigkeit von *3–SAT*.

Der Nachweis von *3–SAT* $\in \mathcal{NP}$ ist einfach, denn wie schon beim ursprünglichen SAT–Problem kann man einfach eine Variablenbelegung raten und anschließend den Wahrheitsgehalt der Formel verifizieren.

Für den Reduktion von SAT auf *3–SAT* betrachten wir KNF–Formeln der Form

$$c_1 \wedge c_2 \wedge \cdots \wedge c_t \ ,$$

wobei $c_1, c_2, \ldots, c_t$ die einzelnen Klauseln darstellen. Wir geben nun ein einfaches Verfahren an, wie man jede solche Formel nach 3–KNF umwandelt, ohne dabei ihre Erfüllbarkeit zu verändern. Die umgewandelte Formel wird also genau dann erfüllbar sein, wenn dies auch auf die Ausgangsformel zutrifft.

Wir müssen jede Klausel c_i umwandeln, die von der Form

$$c_i = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_s)$$

mit den Literalen $\ell_1, \ell_2, \ldots, \ell_s$ und $s > 3$ ist. Hierzu ersetzen wir eine solche Klausel durch eine 3–KNF–Formel und benutzen dazu $s - 3$ neue Variablen A_3, A_4, …, A_{s-1}:

$$(\ell_1 \vee \ell_2 \vee A_3) \wedge (\neg A_3 \vee \ell_3 \vee A_4) \wedge (\neg A_4 \vee \ell_4 \vee A_5) \wedge (\neg A_5 \vee \ell_5 \vee A_6) \wedge \ldots$$
$$\cdots \wedge (\neg A_{s-3} \vee \ell_{s-3} \vee A_{s-2}) \wedge (\neg A_{s-2} \vee \ell_{s-2} \vee A_{s-1}) \wedge (\neg A_{s-1} \vee \ell_{s-1} \vee \ell_s) \ .$$

Falls nun c_i durch eine bestimmte Belegung erfüllt wird, so kann man auch die zugehörige 3–KNF–Formel erfüllen, ohne die Wahrheitswerte der Literale $\ell_1, \ldots, \ell_s$ ändern zu müssen. Ist nämlich c_i wahr, so muss auch mindestens eines seiner Literale wahr sein. Sei also ℓ_j mit $1 \leq j \leq s$ ein solches wahres Literal. Für den angegebenen 3–KNF–Ausdruck setzen wir dann alle neuen Variablen A_k mit $k \leq j$ auf wahr sowie alle anderen Variablen A_k mit $k > j$ auf falsch. Dann sind alle Klauseln des neuen Ausdrucks erfüllt.

Sind umgekehrt alle neuen Klauseln erfüllt, so muss auch mindestens eines der ursprünglichen Literale ℓ_j mit $1 \leq j \leq s$ wahr sein. Denn angenommen nicht, so folgt aus der ersten erfüllten 3–KNF–Klausel $\ell_1 \vee \ell_2 \vee A_3$, dass A_3 wahr sein muss. Die zweite neue 3–KNF–Klausel $\neg A_3 \vee \ell_3 \vee A_4$ impliziert dann, dass auch A_4 wahr sein muss, und genauso müssen auch alle anderen neuen Variablen wahr sein. Dann aber ist spätestens die letzte 3–KNF–Klausel $\neg A_{s-1} \vee \ell_{s-1} \vee \ell_s$ nicht erfüllt, was ein Widerspruch ist.

Formt man also alle Klauseln mit mehr als drei Literalen wie angegeben um, so ist der resultierende Ausdruck genau dann erfüllbar, wenn dies auch auf die ursprüngliche Formel zutrifft. Offensichtlich kann diese Umformung mit einem einfachen Programm automatisiert in linearer Zeit (gemessen an der Formellänge) durchgeführt werden. Es gilt also $SAT \leq_p$ *3–SAT* und damit insgesamt die Behauptung. □

5.5. Weitere $\mathcal{NP}$–vollständige Probleme

Wie bereits erwähnt sind noch viele andere Probleme aus der Praxis $\mathcal{NP}$–vollständig. Wir besprechen einige interessante Beispiele und verweisen ansonsten nochmals auf das Buch von Garey und Johnson [10].

Unser erstes Beispiel betrifft einen Dieb, der nachts in ein Kaufhaus einbrechen und möglichst viele Gegenstände erbeuten will. Diese haben aber alle ein bestimmtes Gewicht, und der Dieb kann nur ein gewisses Gesamtgewicht tragen. Kann er unter allen Gegenständen eine Auswahl treffen, so dass er seine Transportkapazität genau ausreizt?

Die entsprechende Frage ist in der Fachwelt als *Rucksackproblem* oder *SUBSET–SUM* bekannt. Gegeben sei eine endliche Teilmenge $S \subset \mathbb{N}$ von natürlichen Zahlen (die Gewichte seien also der Einfachheit halber ganzzahlig gewählt) sowie eine weitere Zahl $m \in \mathbb{N}$, die das Gesamtgewicht modelliert. S darf abweichend von dem üblichen Mengenbegriff gleiche Zahlen auch mehrfach enthalten (d.h. es kann mehrere Gegenstände mit dem gleichen Gewicht geben). Kann man dann bestimmte Zahlen aus S so auswählen, dass deren Summe genau gleich m ist?

Im Fall

$$S = \{3, 6, 7, 11, 13, 21, 24, 28, 43, 48, 71, 72, 88\} \quad \text{und} \quad m = 90$$

wäre dies möglich (z.B. wegen $7 + 11 + 72 = 90$ oder auch $6 + 11 + 21 + 24 + 28 = 90$), im Fall

$$S = \{5, 9, 11, 16, 20, 21, 32, 37, 47, 68, 73, 78, 83\} \quad \text{und} \quad m = 91$$

aber nicht, was (scheinbar) nicht leicht zu entdecken ist.

Der Dieb sollte sich besser nicht zu lange an einer Optimierung versuchen.

5.5.1 Satz *SUBSET–SUM* ist $\mathcal{NP}$–vollständig.

Beweis: Natürlich gilt *SUBSET–SUM* $\in \mathcal{NP}$, da man eine passende Auswahl einfach raten und deren Summe verifizieren kann. Wir reduzieren *3–SAT* auf *SUBSET–SUM*, um den Beweis zu vervollständigen.

Hat eine 3–KNF–Formel n Variablen und k Klauseln, so erzeugen wir insgesamt $2(k+n)$ viele Zahlen mit jeweils $k + n$ vielen Ziffern, die zusammen die Menge S ausmachen.

In der 3–KNF Formel

$$(\neg A \vee \neg C \vee D) \wedge (A \vee \neg C) \wedge (\neg B \vee C \vee \neg D)$$

gibt es z.B. $n = 4$ Variablen sowie $k = 3$ Klauseln, so dass alle Zahlen $k + n = 7$ Ziffern haben werden. Diese werden wie folgt konstruiert:

- Wir wählen eine beliebige Variable aus und betrachten die erste Klausel genauer. Falls die Variable dort ohne Negation vorkommt, notieren wir eine 1, andernfalls (also wenn sie nur mit Negation oder gar nicht vorkommt) eine 0. Genauso verfahren wir mit den anderen Klauseln und notieren die entsprechenden Ziffern der Reihe nach dahinter. Für die Variable A würde sich in unserem Beispiel die Zahl 010 ergeben (wir lassen die führende Null der Anschaulichkeit halber stehen). Anschließend hängen wir an die so erzeugte Zahl noch einen „Positionscode“ an, der aus einer 1 und $n - 1$ Nullen besteht, hier also 1000. Die komplette Zahl für die Variable A wäre demnach 0101000.
- Wir verfahren genauso für die negierte Form der Variable und notieren erneut für jedes Vorkommen eine 1, andernfalls eine 0. Der Positionscode bleibt gleich. In unserem Beispiel erhalten wir für $\neg A$ die Zahl 1001000.
- Auch für die anderen Variablen erzeugen wir jeweils zwei Zahlen nach dem gleichen Muster, nur der Positionscode ändert sich. Die einzelne 1 wandert für jede weitere Variable um eine Stelle nach rechts, alle anderen Ziffern bleiben 0. Für die zweite Variable B würden wir also in unserem Beispiel den Code 0100 verwenden, für C den Code 0010 und für D 0001.

  Die resultierenden Zahlen (mit gegebenenfalls führenden Nullen) wären hier also:

$$\begin{array}{cccccccc} A: & 0101000 & B: & 0000100 & C: & 0010010 & D: & 1000001 \\ \neg A: & 1001000 & \neg B: & 0010100 & \neg C: & 1100010 & \neg D: & 0010001 \end{array}$$

  Addiert man diese „Variablenzahlen“ zusammen, so ergibt sich die Summe 3232222. Dabei geben die ersten $k = 3$ Ziffern an, wie viele Literale sich in den einzelnen Klauseln befinden — in der ersten und dritten Klausel sind dies drei, in der zweiten nur zwei. Die restlichen Ziffern müssen sich jeweils zu einer 2 aufaddieren, da nur je genau zwei Variablenzahlen dort eine 1 stehen haben.
- Die restlichen Zahlen mit $k + n$ vielen Ziffern nennen wir „Füllzahlen“. Die ersten beiden beginnen mit einer 1 und bestehen ansonsten nur aus Nullen. In unserem Beispiel wäre dies also zweimal die Zahl 1000000. Bei den nächsten beiden Füllzahlen ist die 1 um eine Position nach rechts verschoben, d.h. wir erhalten hier zweimal 0100000. Diesem Prinzip folgend erzeugen wir für jede Klausel zwei identische Füllzahlen, bei denen pro Klausel die 1 um eine Position weiter nach rechts verschoben wird. In unserem Beispiel würde also zusätzlich nur noch die 0010000 erzeugt, da es ja insgesamt nur drei Klauseln gibt.

  Es ist klar, dass die Summe aller Füllzahlen von der Form

$$\underbrace{22\ldots2}_{k\text{–mal}}\underbrace{00\ldots0}_{n\text{–mal}}$$

ist. Die Summe von allen bislang erzeugten Zahlen kann also

$$\underbrace{55\ldots5}_{k\text{–mal}}\underbrace{22\ldots2}_{n\text{–mal}}$$

nicht übersteigen. Offenbar kann es auch keine internen Überträge geben.

Damit ist die Erzeugung aller Zahlen für die Menge S abgeschlossen.

- Die Ergebniszahl m ist von der Form

$$\underbrace{33\ldots3}_{k\text{–mal}}\underbrace{11\ldots1}_{n\text{–mal}}\ ,$$

d.h. für das Beispiel ergibt sich die Zahl 3331111.

Wir müssen nun zeigen, dass eine Auswahl der erzeugten Zahlen sich genau dann zu m aufsummieren lässt, wenn die ursprüngliche Formel erfüllbar ist.

Sei also zunächst die Ausgangsformel erfüllbar. Dann gibt es eine entsprechende erfüllende Belegung, und wir können die entsprechenden „Variablenzahlen“ aufsummieren, die zu den zugehörigen wahren Literalen gehören. In unserem Beispiel würde z.B. die Belegung $A = \mathrm{F}$, $B = \mathrm{W}$, $C = \mathrm{F}$ und $D = \mathrm{F}$ die Formel erfüllen, und die zugehörigen Zahlen für $\neg A$, B, $\neg C$ und $\neg D$ summieren sich zu

$$1001000 + 0000100 + 1100010 + 0010001 = 2111111\ .$$

Gemäß der gewählten Konstruktion folgen für die Summe der genannten Zahlen diese Eigenschaften:

- Im hinteren Bereich der Positionscodes steuert jede Variablenzahl an einer anderen Position eine 1 zum Ergebnis bei, und es werden auch alle Positionen abgedeckt. Folglich besteht der hintere Bereich immer nur aus Einsen.
- Jede Ziffer im vorderen Bereich gibt die Anzahl der erfüllten Literale in der entsprechenden Klausel an. In dem Beispiel wird die erste Klausel durch zwei Literale erfüllt (nämlich durch $\neg A$ und $\neg C$), in der zweiten Klausel ist dagegen nur ein Literal wahr (nämlich $\neg C$). Ebenso wird die dritte Klausel nur durch ein Literal erfüllt (nämlich durch $\neg D$). Da jede Klausel höchstens drei Literale enthält und mindestens eines davon wahr sein muss, liegt jede Ziffer im vorderen Bereich zwischen 1 und 3.
- Also ist es kein Problem, durch Aufsummieren von geeigneten Füllzahlen jede Ziffer im vorderen Bereich noch um 1 oder 2 zu erhöhen und damit den Wert 3 zu erreichen. In unserem Beispiel würden wir noch einmal die Zahl 1000000 sowie je zweimal die Zahl 0100000 und 0010000 auswählen, so dass sich die gewünschte Gesamtsumme

$$2111111 + 1000000 + 0100000 + 0100000 + 0010000 + 0010000 = 3331111 = m$$

ergibt.

Also existiert immer eine Lösung für das konstruierte *SUBSET–SUM*–Problem, wenn die Ausgangsformel erfüllbar ist.

Sei jetzt umgekehrt das konstruierte *SUBSET–SUM*–Problem lösbar. Für die ausgewählten Zahlen aus der Menge S und deren Summe m gilt:

- Da m im hinteren Bereich nur Einsen enthält und es keine Überträge gibt, muss die Summe genau n Variablenzahlen enthalten, und zwar solche, die genau die n verschiedenen Positionscodes abdecken. Daraus folgt, dass für jede Variable genau eine ihrer beiden Variablenzahlen in die Summe mit einfließt. Diese wiederum korrespondieren zu den beiden Wahrheitswerten wahr und falsch. Man kann also aus den Summanden eine eindeutige Variablenbelegung ablesen.
- Im vorderen Bereich lassen sich die 3er–Ziffern der Zahl m nicht allein aus den Füllzahlen erzeugen (diese können wie gesehen an jeder Stelle nur maximal eine 2 erzeugen). Also muss jeweils zumindest eine Variablenzahl eine 1 mit zur Summe beisteuern. Dies bedeutet aber, dass das entspechende Literal in der Klausel und somit die Klausel insgesamt wahr sein muss.
- Wenn aber jede Klausel wahr ist, so gilt dies auch für die Ausgangsformel insgesamt, d.h. mit der abgelesenen Belegung ist die Formel erfüllbar.

Wenn also eine Lösung für das konstruierte *SUBSET–SUM*–Problem existiert, muss die Ausgangsformel erfüllbar gewesen sein.

Die Konstruktion der Zahlen kann offenbar durch ein kurzes Programm mit einigen einfachen Schleifenanweisungen erfolgen. Insgesamt sind $2(k + n)$ Zahlen mit jeweils $k + n$ vielen Ziffern zu erzeugen. Sowohl k als auch n sind gegen die Länge der Formel abschätzbar. Also ist sicherlich polynomiell viel Zeit für die Reduktion ausreichend. □

Mit einer ähnlichen Technik lässt sich die $\mathcal{NP}$–Vollständigkeit eines weiteren interessanten Problems beweisen. Nehmen Sie an, für ein umfangreiches Softwareprojekt sind verschiedene Teilaufgaben zu erledigen, z.B. ist die Benutzeroberfläche zu erstellen, der Algorithmus zu entwerfen, eine Schnittstelle zu programmieren, usw. Dem Projektleiter stehen verschiedene Softwareentwickler zur Verfügung, die unterschiedliche Kenntnisse haben. Jeder Informatiker kann eine ganz bestimmte Teilmenge der obigen Aufgaben erledigen, von den anderen Aufgaben hat er keine Ahnung. Natürlich sollen die Personalkosten möglichst gering bleiben. Ist es also möglich, eine geschickte Auswahl unter den Entwicklern zu treffen, so dass man mit einer vorgegebenen Anzahl von Personen auskommt? Eine Auswahl ist nur dann korrekt, wenn für jede Aufgabe mindestens eine qualifizierte Person zur Verfügung steht.

Die Aufgaben können wir der Reihe nach durchnummerieren, d.h. bei m Aufgaben handelt es sich um die Menge $M := \{1, 2, \ldots, m\}$, und der Kenntnisstand jedes Informatikers ist einfach eine Teilmenge davon. Wenn nun eine bestimmte Anzahl ℓ von Informatikern ausreichen soll, so suchen wir eine Auswahl von ℓ dieser Teilmengen, deren Vereinigung ganz M ergibt. Man spricht dann auch von einer *Überdeckung* von M. Das Problem ist entsprechend als *SET–COVER* bekannt.

5.5.2 Satz *SET–COVER* ist $\mathcal{NP}$–vollständig.

Beweis: Natürlich kann man einfach irgendeine Auswahl treffen und dann prüfen, ob die Überdeckungseigenschaft erfüllt ist. Dies beweist $SET\text{–}COVER \in \mathcal{NP}$. Außerdem gilt $3\text{–}SAT \leq_p SET\text{–}COVER$, wie wir nun zeigen werden.

Wir gehen wieder von einer 3–KNF–Formel mit n Variablen und k Klauseln aus. Dann setzen wir $m := k + n$, d.h. die zu überdeckende Menge $M = \{1, 2, \ldots, k + n\}$ besteht aus den ersten $k + n$ Zahlen. Im Fall der 3–KNF-Formel aus dem letzten Satz

$$(\neg A \vee \neg C \vee D) \wedge (A \vee \neg C) \wedge (\neg B \vee C \vee \neg D)$$

gibt es also wieder $n = 4$ Variablen sowie $k = 3$ Klauseln, so dass M aus der Menge $\{1, 2, \ldots, 7\}$ besteht.

Weiterhin definieren wir $2n$ Teilmengen $T_1^w, T_1^f, T_2^w, T_2^f, \ldots, T_n^w, T_n^f$ von M wie folgt:

- Wir beginnen wieder mit irgendeiner Variable und gehen alle k Klauseln nacheinander durch. Wenn die Variable in der i–ten Klausel ohne Negation vorkommt, so fügen wir die Zahl i zu T_1^w hinzu. Also enthält T_1^w am Ende die Indizes von genau denjenigen Klauseln, die erfüllt werden, wenn die Variable wahr ist.
- Abschließend fügen wir noch die Zahl $k+1$ als sog. „Auswahlwächter“ zu T_1^w hinzu. Wenn wir also in unserem Beispiel als erstes die Variable A betrachten, so erhalten wir die Menge $T_1^w = \{2, 4\}$.
- Analog gehen wir alle Klauseln für die negierte Variable durch und ergänzen die damit erzeugte Indexmenge wieder durch den Auswahlwächter $k + 1$. Die resultierende Menge heißt T_1^f und wäre dann hier $\{1, 4\}$.
- Die anderen Variablen erzeugen in der gleichen Weise die übrigen Mengen

  $$T_2^w, T_2^f, \ldots, T_n^w, T_n^f \ ,$$

  die allerdings mit den Auswahlwächtern $k + 2$, $k + 3$, usw. versehen werden.

Wenn wir in unserem Beispiel die Variablen in der Reihenfolge A, B, C und D durchlaufen, so erhalten wir:

T_1^w :	$\{2, 4\}$	T_2^w :	$\{5\}$	T_3^w :	$\{3, 6\}$	T_4^w :	$\{1, 7\}$
T_1^f :	$\{1, 4\}$	T_2^f :	$\{3, 5\}$	T_3^f :	$\{1, 2, 6\}$	T_4^f :	$\{3, 7\}$

Als Anzahl ℓ der zu verwendenden Teilmengen setzen wir abschließend die Variablenanzahl n fest. Hier wäre also die Frage, ob sich M mit vier Teilmengen überdecken lässt.

Angenommen, die Ausgangsformel liegt in $3\text{–}SAT$. Dann gibt es eine passende Belegung, die alle Klauseln erfüllt. Dem Wahrheitswert jeder Variable entsprechend wählen wir nun die jeweils zugehörige T^w– bzw. die T^f–Menge aus. In unserem Beispiel war z.B. $A =$ F, $B =$ W, $C =$ F und $D =$ F eine solche Belegung, und wir würden uns für die Mengen

$$T_1^f, \quad T_2^w, \quad T_3^f \quad \text{und} \quad T_4^f$$

entscheiden.

Jede Menge enthält (von den Auswahlwächtern abgesehen) die Indizes der zugehörigen erfüllten Klauseln. Also müssen alle Klauseln mit den Indizes von 1 bis k vertreten sein, da ja alle Klauseln wahr sind. Die ausgewählten Teilmengen enthalten auch alle Auswahlwächter mit den Indizes $k+1$ bis $k+n$, da jede Variable mit einer T^w– oder T^f–Menge vertreten ist. Also überdecken alle Teilmengen insgesamet den Zahlenbereich von 1 bis $k+n$ und damit die Menge M.

Wird umgekehrt $M = \{1, \ldots, k+n\}$ von n ausgewählten Teilmengen überdeckt, so müssen insbesondere alle Auswahlwächter $k+1, \ldots, k+n$ in den Teilmengen enthalten sein. Ein Auwahlwächter $k+i$ mit $1 \leq i \leq n$ ist aber nur in den beiden Teilmengen T_i^w und T_i^f vertreten. Also muss mindestens eine dieser beiden Teilmengen mit an der Überdeckung beteiligt sein. Weil dies für alle i mit $1 \leq i \leq n$ gilt und insgesamt nur n Teilmengen benutzt werden dürfen, kann aber von jeder T_i^w– bzw. die T_i^f–Menge nur genau eine zum Einsatz kommen. Also kann man implizit an den beteiligten Mengen eine zugehörige Variablenbelegung ablesen. Diese erfüllt bekanntlicherweise genau die Klauseln, deren Indizes in den entsprechenden Teilmengen vorkommen. Dort sind aber alle Indizes von 1 bis k vertreten, da ja $M = \{1, \ldots, k+n\}$ ganz überdeckt wird. Also werden alle Klauseln erfüllt, und die Formel muss insgesamt wahr sein.

Natürlich ist auch hier die Berechnung aller Zahlenmengen leicht durch ein entsprechendes Java–Programm zu bewerkstelligen, welches polynomiale Laufzeit besitzt. □

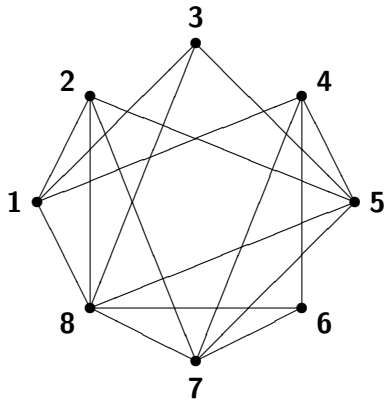
Für unser nächstes Problem nehmen wir an, dass ein Hochzeitspaar zu seiner Trauung viele Gäste einlädt. Es handelt sich um einen bunt gemischten Personenkreis von Verwandten, Arbeitskollegen, Freunden, usw., die sich untereinander entweder sehr gut oder gar nicht kennen. Das frisch vermählte Paar möchte für das Abendessen die Gäste möglichst geschickt an den einzelnen Tischen platzieren. Es sollen an jedem Tisch nur Personen zu sitzen, bei denen jeder jeden kennt. Kann das Hochzeitspaar solche „Cliquen“ einfach ermitteln?

Dieses Problem lässt sich wieder gut durch Graphen darstellen. Sei dazu G ein ungerichteter Graph, bei denen jeder Knoten einen Mensch und jede Kante zwischen zwei Knoten deren gute Bekanntschaft modelliert. Eine *k–Clique* ist eine Menge von k Knoten, die jeweils paarweise miteinander durch eine Kante verbunden sind (d.h. diese k Personen könnten dann an einem Tisch sitzen). Der Graph auf der nächsten Seite besitzt also z.B. eine 4–Clique (die Knoten 2, 5, 7 und 8), aber keine 5–Clique.

Ähnlich wie bei dem *SAT*–Problem können wir uns wieder eine passende Kodierung einfallen lassen, die jedem ungerichteten Graphen G eine eindeutige Zeichenkette über einem bestimmten Alphabet zuordnet. Wir werden dieses an sich unwichtige Detail im Folgenden aber nicht mehr erwähnen.

5.5.3 Satz Die Sprache

$$CLIQUE := \{(G,k) \mid G \text{ ist ein ungerichteter Graph und besitzt eine } k\text{-Clique}\}$$



ist $\mathcal{NP}$–vollständig (d.h. das Hochzeitspaar steht vor einer schwierigen Aufgabe).

Beweis: Offensichtlich gilt $CLIQUE \in \mathcal{NP}$ (einfach eine k–elementige Knotenmenge raten und die Clique–Eigenschaft verifizieren). Wir zeigen nun $SAT \leq_p CLIQUE$.

Sei $F = c_1 \wedge c_2 \wedge \cdots \wedge c_k$ ein boolescher Ausdruck in KNF mit k Klauseln. Die j–te Klausel c_j sei von der Form $\ell_{1,j} \vee \ell_{2,j} \vee \ell_{3,j} \vee \ldots$. Wir fassen nun jedes Literal $\ell_{i,j}$ als einen Knoten des zu konstruierenden Graphen G auf und verbinden jeweils zwei Knoten $\ell_{i,j}$ und $\ell_{i',j'}$ durch eine Kante, falls

- erstens $i \neq i'$ gilt (d.h. die beiden Literale müssen zu unterschiedlichen Klauseln gehören),
- zweitens $\ell_{i,j}$ ungleich $\neg\ell_{i',j'}$ ist (d.h. die beiden Literale dürfen sich nicht unmittelbar widersprechen).

Beispielsweise erzeugt die Formel

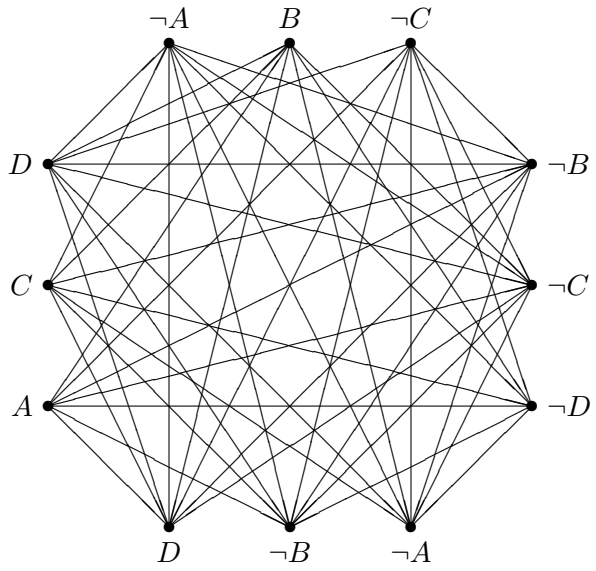
$$(A \vee C \vee D) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg B \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg B \vee D)$$

den auf der nächsten Seite dargestellten Graph.

Angenommen, F ist erfüllbar. Dann kann man aus jeder der k Klauseln ein wahres Literal auswählen; diese sind gemäß Konstruktion paarweise miteinander verbunden. Also enthält der konstruierte Graph eine k–Clique.

Enthält umgekehrt G eine k–Clique, so müssen aufgrund der ersten Konstruktionsbedingung alle zugehörigen Literale aus paarweise verschiedenen Klauseln stammen. Da überhaupt nur k Klauseln existieren, sind somit alle Klauseln vertreten. Aufgrund der zweiten Konstruktionsbedingung können sich die Literale paarweise nicht widersprechen. Also lassen sich alle Literale durch eine geeignete Belegung problemlos erfüllen, und diese Belegung erfüllt dann auch F.

Offenbar kann man die Reduktion mit einem einfachen Java–Programm durchführen, welches mit zwei verschachtelten Schleifen alle zu verbindenden Literal–Paare ausfindig macht. Also kann die Reduktion in quadratischer und damit polynomieller Laufzeit



realisiert werden. □

Gerechtes Teiles kann ebenfalls ein hartnäckiges Problem sein. Angenommen, ein Elternpaar hat für seine beiden Kinder ein großes Sparschwein angelegt und es auch regelmäßig mit unterschiedlichen Geldbeträgen gefüttert. Eines Tages soll das Schwein geschlachtet und das Vermögen unter den beiden Kindern aufgeteilt werden, so dass beide Kinder gleich bedacht werden. Ist dies immer möglich?

Formal haben wir es hier wieder mit einer endlichen Menge $S \subset \mathbb{N}$ zu tun, die die Werte der angesparten Geldmünzen usw. modelliert. Auch hier seien wieder gleiche Zahlen erlaubt. Ist es dann möglich, eine Teilmenge $T \subseteq S$ ausfindig zu machen, so dass

$$\sum_{x \in T} x = \sum_{x \in S \setminus T} x$$

gilt? Dies ist das sog. *PARTITION*–Problem.

Je nach Größe des Sparschweins stehen die Eltern vor einem harten Problem.

5.5.4 Satz *PARTITION* ist $\mathcal{NP}$–vollständig.

Beweis: Man kann leicht eine Teilmenge $T \subseteq S$ raten und dann prüfen, ob diese zufällig gerade die gewüschte Aufteilung darstellt. Folglich gilt $PARTITION \in \mathcal{NP}$. Für den Nachweis der $\mathcal{NP}$–Vollständigkeit reduzieren wir *SUBSET–SUM* auf *PARTITION*.

Die Reduktion ist einfach zu bewerkstelligen. Wenn die Zahlen $x_1, x_2, \ldots, x_k \in \mathbb{N}$ und die gewünschte Summe $m \in \mathbb{N}$ einer *SUBSET–SUM*–Instanz vorgegeben sind, so ermitteln

wir zunächst die Summe

$$s := \sum_{i=1}^{k} x_i$$

und erzeugen daraus dann die Menge

$$S := \{x_1, x_2, \ldots, x_k, s - m + 1, m + 1\} \ .$$

Im Fall der auf Seite 201 erwähnten *SUBSET–SUM*–Instanz

$$S = \{3, 6, 7, 11, 13, 21, 24, 28, 43, 48, 71, 72, 88\} \quad \text{mit} \quad m = 90$$

würden wir also $s = 435$ ermitteln und diese neue Menge S erzeugen:

$$S = \{3, 6, 7, 11, 13, 21, 24, 28, 43, 48, 71, 72, 88, 346, 91\} \ .$$

Wir behaupten, dass sich diese Menge genau dann in zwei gleichen Hälften partionieren lässt, wenn die ursprüngliche *SUBSET–SUM*–Instanz eine Lösung besitzt.

Angenommen, die *SUBSET–SUM*–Instanz hat eine Lösung, d.h. eine Auswahl T der x_i–Zahlen lässt sich auf m aufsummieren. Folglich lässt sich dann die Menge T' aller unbeteiligten Zahlen auf den Rest $s - m$ aufsummieren, und wenn man nun noch T um die Zahl $s - m + 1$ sowie T' um die Zahl $m + 1$ ergänzt, so ergeben beide Summen die Zahl $s + 1$, d.h. die Zahlen aus der Menge S lassen sich gerecht aufteilen. Also hat dann auch die *PARTITION*–Instanz eine Lösung.

Die obige *SUBSET–SUM*–Instanz hatte z.B. die Lösung $6 + 11 + 21 + 24 + 28 = 90$. Wir können daher die konstruierte Menge

$$S = \{3, 6, 7, 11, 13, 21, 24, 28, 43, 48, 71, 72, 88, 346, 91\}$$

wie geplant in die beiden gleichen Summen

$$6 + 11 + 21 + 24 + 28 + 346 = 436$$

und

$$3 + 7 + 13 + 43 + 48 + 71 + 72 + 88 + 91 = 436$$

zerlegen.

Die Summe aller Zahlen in S beträgt

$$x_1 + x_2 + \cdots + x_k + (s - m + 1) + (m + 1) = s + (s - m + 1) + (m + 1) = 2(s + 1) \ .$$

Wenn also umgekehrt eine Lösung für die erzeugte *PARTITION*–Instanz möglich ist, dann müssen beide Partitionen den Wert $s + 1$ haben. Wir betrachten jetzt die Zahlenmenge, in der die Zahl $s - m + 1$ liegt. Die Zahl $m + 1$ liegt dann in der anderen Partition, denn die Summe beider Zahlen wäre wegen

$$(s - m + 1) + (m + 1) = s + 2$$

schon zu groß. Die Partition mit der Zahl $s-m+1$ besteht also ansonsten nur noch aus bestimmten x_i–Zahlen, und ohne die Zahl $s-m+1$ hat deren Summe den Wert

$$(s+1)-(s-m+1)=m \ .$$

Also bilden diese Zahlen eine Lösung des anfänglichen *SUBSET–SUM*–Problems. □

Mit einem ähnlichen Problem haben Spediteure zu kämpfen. Angenommen, verschiedene Güter mit unterschiedlichen Gewichten sollen in mehrere LKWs verladen werden. Natürlich soll die Anzahl der beteiligten LKWs möglichst gering bleiben. Ist also die Beladung so möglich, dass eine bestimmte Anzahl von LKWs ausreicht und bei keinem LKW das zulässige Gesamtgewicht von z.B. 40t überschritten wird?

Für die Modellierung können wir erneut eine endliche Menge $S \subset \mathbb{N}$ von Gewichten betrachten, und auch hier dürfen diverse Gewichte mehrfach vorkommen. Ferner ist eine Anzahl m von Behältern mit einer gewissen Kapazität $k \in \mathbb{N}$ vorgegeben. Die Frage beim sog. *BIN–PACKING*–Problem ist dann, ob sich alle Zahlen aus S so auf die m Behälter verteilen lassen, das bei keinem die Summe der darin enthaltenen Zahlen größer als k ist.

5.5.5 Satz *BIN–PACKING* ist $\mathcal{NP}$–vollständig.

Beweis: *BIN–PACKING* ist in $\mathcal{NP}$ enthalten, da wir wie bei allen anderen besprochenen Problemen eine Lösung erst raten und dann überprüfen können. Die nachfolgende (überraschend einfache) Reduktion von *PARTITION* auf *BIN–PACKING* beweist deshalb die $\mathcal{NP}$–Vollständigkeit dieses Problems.

Ist eine Menge S für das *PARTITION*–Problem vorgegeben, so lassen wir diese unverändert und setzen $m := 2$ Behälter der Größe

$$k := \Big(\sum_{x\in S} x\Big)/2$$

fest. Es ist klar, dass dann die Frage nach einer Verteilung aller Zahlen auf diese beiden Behälter genau zu dem ursprünglichen Partitionierungsproblem äquivalent ist. □

Die $\mathcal{P}=\mathcal{NP}$ Frage ist wie erwähnt das wichtigste ungelöste Problem in der theoretischen Informatik. Bislang hat man leider nur viele Erkenntnisse gewonnen, wie man es *nicht* beweisen kann. Es gibt jedoch starke Indizien dafür, das vermutlich $\mathcal{P} \neq \mathcal{NP}$ gilt, was für alle $\mathcal{NP}$–vollständigen Probleme deren faktische Unlösbarkeit nach sich zieht. Wenn Sie in der Praxis ein solches Problem algorithmisch lösen müssen, sollten Sie nicht versuchen, einen effizienten Algorithmus dafür zu finden. Vielmehr müssen Sie bereit sein, einen Kompromiss einzugehen. Dazu gibt es verschiedene Möglichkeiten.

Es gibt *randomisierte Algorithmen*, deren Antwort vom Zufall abhängt und evtl. auch falsch sein kann. Die Fehlerwahrscheinlichkeit ist aber im Allgemeinen begrenzt, und

man kann häufig durch eine mehrfache Ausführung eines solchen Verfahrens die Fehlerwahrscheinlichkeit auf ein akzeptables Maß drücken. Das Zero–Knowledge–Protokoll aus dem Abschnitt 5.2 war ein Beispiel dafür.

Möglicherweise lässt sich eine Problemstellung auch *approximieren*. Man verzichtet also auf eine optimale Lösung, wenn man im Gegenzug eine Näherung schneller berechnen kann. Wenn Sie sich z.B. das *Problem des Handlungsreisenden* aus dem einführenden Text nochmals anschauen, so kann eine etwas längere Reise durchaus akzeptabel sein. In der Tat gibt es z.B. ein deterministisches Polynomialzeitverfahren (die sog. *Christofides–Heuristik*), welches eine Approximationsgüte von 1,5 garantiert, d.h. jede vorgeschlagene Rundreise ist höchstens 50% länger als die bestmögliche. Weitere Approximationsverfahren für andere $\mathcal{NP}$–vollständigen Probleme finden Sie z.B. in [8].

Unabhängig davon ist jedoch nicht ausgeschlossen, dass auch $\mathcal{P} = \mathcal{NP}$ gelten könnte. Wenn jemand ein effizientes Verfahren für ein $\mathcal{NP}$–vollständiges Problem angeben und damit $\mathcal{P} = \mathcal{NP}$ beweisen würde, hätte dies evtl. eine große (negative) Auswirkung auf die Sicherheit kryptographischer Verfahren. Doch möglicherweise wird auch nur die Existenz von einem solchen Algorithmus nachgewiesen, ohne ihn aber konkret angeben zu können. Es bleibt eine spannende Frage, wie das $\mathcal{P} = \mathcal{NP}$ Problem am Ende ausgeht.

6 Zeit– und Platzhierarchien

In diesem abschließenden Kapitel behandeln wir eine zentrale Frage:

Kann man mit mehr Zeit und / oder mehr Speicher auch mehr berechnen?

Selbstverständlich (würde man meinen) — oder doch nicht? Die Antwort ist gar nicht so einfach. Wir werden im Folgenden versuchen, eine Art Tabelle von Zeit– und Speicher–Komplexitätsklassen zu erstellen, so dass jede weiter unten bzw. hinten aufgeführte Komplexitätsklasse mehr Sprachen als die davorliegende Klasse enthält. Wir möchten also eine *Hierarchie* von Komplexitätsklassen nachweisen. Die jetzt aufkommende interessante Frage ist, wie klein man die zusätzlichen Ressourcen wählen kann, um die nächstgrößere Stufe zu erreichen. Wir möchten also möglichst *dichte* Hierarchien konstruieren. Dies erfordert recht komplizierte Beweistechniken. Es hat in der Vergangenheit viel Forschungsarbeit gekostet, um die intuitiv erwarteten dichten Hierarchien wirklich nachweisen zu können.

Die wichtigsten Ergebnisse sowie die dazugehörigen Beweisideen werden nachfolgend präsentiert. Anschließend werden wir eine Aussage besprechen, die den dichten Komplexitätsstufen scheinbar völlig widerspricht. Man kann nämlich zeigen, dass beliebig große Lücken in den Hierarchien existieren, d.h. ab einer bestimmten Stufe führt erheblich mehr Zeit oder Speicher (z.B. exponentiell viel mehr) zu keiner zusätzlich berechenbaren Sprache. Dieses Paradoxon werden wir aber auflösen können. Außerdem werden wir nachweisen, dass es Sprachen gibt, die man nicht optimal schnell (oder mit optimal wenig Speicher) berechnen kann. Für solche Sprachen findet man nämlich unendlich viele passende Algorithmen, und jeder solche Algorithmus kann durch einen alternativen Algorithmus überboten werden, der noch schneller (oder platzeffizienter) arbeitet. Für diesen findet man dann noch effiziente Verfahren, usw.

Dieses letzte Kapitel steigt vom Schwierigkeitsgrad her stetig an — die hier zum Einsatz kommenden Beweisideen sind manchmal nicht gleich auf den ersten Anhieb zu durchschauen. Nicht ohne Grund wurde dieser interessante, aber doch recht anspruchsvolle Lehrstoff bis zum Ende aufgehoben.

6.1. Komplexitätsklassen bei Turing–Maschinen

Wir beginnen mit einer Neudefinition von DTIME und NTIME. Der Grund dafür ist, dass wir für die Hierarchiesätze den Zeitaufwand ganz exakt erfassen müssen. Dies ist bei Turingmaschinen ganz einfach durch das Mitzählen der ausgeführten Schritte möglich. Gleiches gilt für die Speicherkomplexität (indem wir die Anzahl der benutzten Bandfelder erfassen), für die wir im Übrigen noch gar keine Komplexitätsklassen definiert hatten.

Wir machen uns zunächst klar, dass man den Speicherbedarf von Turingmaschinen um einen konstanten Faktor verringern kann. Nehmen Sie z.B. an, auf dem Turingband stehen in drei aufeinanderfolgenden Feldern die Symbole a, b und c, und der Kopf befindet sich momentan unter dem Feld mit dem a. Wenn die Turingmaschine sich nun z.B. im Zustand z_4 befindet und die drei Schritte

$$\begin{aligned} \delta(z_4, a) &:= (z_4, c, R) \\ \delta(z_4, b) &:= (z_5, a, R) \\ \delta(z_5, c) &:= (z_2, a, L) \end{aligned}$$

ausführt, so befindet sie sich danach im Zustand z_2. Außerdem hat sie das Wort abc mit caa ersetzt, und der Kopf befindet sich jetzt unter dem ersten a. Eine „speichereffizientere“ TM könnte nun auf einem vergrößerten Bandalphabet operieren, wobei jedes neue Zeichen einer Kombination von drei bisherigen Zeichen entspricht. Somit könnte sie immer drei Felder auf einmal verarbeiten und die Ersetzung von abc durch caa in einem Schritt durchführen. Natürlich muss sie sich zusätzlich merken, wo genau innerhalb eines solchen Kombinationszeichens der ursprüngliche Kopf stehen würde. Da es dafür aber nur drei Möglichkeiten gibt (er steht innerhalb des aktuellen Kombinationszeichens entweder unter dem linken, dem mittleren oder dem rechten Symbol), kann diese Information in einer entsprechend vergrößerten Zustandsmenge festgehalten werden.

Diese Idee können wir auf mehr als drei benachbarte Zeichen verallgemeinern und erhalten damit das folgende Ergebnis:

6.1.1 Satz Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion und $M = (Z, \Sigma, \Gamma, \delta, z_0)$ eine TM, die für Eingabewörter der Länge n nur höchstens $f(n)$ viele Felder beschreibt. Dann existiert für alle Zahlen c mit $0 < c < 1$ eine TM M', die die gleiche Sprache erkennt und in der gleichen Situation mit höchstens $cf(n)$ viel Speicher auskommt (im Fall $c = \frac{1}{2}$ also z.B. mit nur halb so vielen Feldern).

Beweis: Wir nehmen zunächst an, dass M einbändig ist. Wir runden die Zahl $\frac{1}{c}$ auf die nächste ganze Zahl m auf. M' entsteht aus M, indem wir jeweils m benachbarte Felder von M zu einem einzelnen Feld von M' zusammenfassen. Das Bandalphabet Γ von M geht so in das Bandalphabet $\Gamma' := \Gamma^m$ von M' über. Außerdem ersetzen wir die Zustandsmenge Z durch $Z \times \{1, 2, \ldots, m\}$. Jeder Zustand (z, i) enthält in seiner ersten Komponente den entsprechenden aktuellen Zustand von M sowie in seiner zweiten Komponente die aktuelle Position des Kopfes von M innerhalb des momentanen

Zeichens $(a_1, a_2, \ldots, a_m) \in \Gamma'$. Nun können wir durch Modifikation der Übergangsfunktion das Verhalten der ursprünglichen Turingmaschine M problemlos nachbilden. M' benötigt dafür aber nur noch höchstens $\frac{f(n)}{m}$ viele Felder, unter Berücksichtigung einer Aufrundung auf die nächste ganze Zahl also nicht mehr als $\frac{f(n)}{m} + 1$ Felder. Wegen $m \geq \frac{1}{c}$ (also $\frac{1}{m} \leq c$) ist der Speicherbedarf von M' somit auf $cf(n) + 1$ begrenzt. Durch eine nochmalige Erweiterung der Zustandsmenge kann sich M' eines ihrer komprimierten Zeichen in ihren Zuständen merken. Damit kann sie ein Feld einsparen und erreicht so die behauptete Platzschranke von höchstens $cf(n)$ vielen Feldern.

Die vorgestellte Kompressionstechnik lässt sich leicht auf eine mehrbändige TM übertragen (auf allen Bändern wird eine Komprimierung im Verhältnis $1 : m$ vorgenommen, M' merkt sich die Position von allen simulierten Köpfen von M gleichzeitig, usw.). □

Mit der gleichen Idee lässt sich auch der Zeitverbrauch einer Turingmaschine um einen konstanten Faktor beschleunigen:

6.1.2 Satz Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion mit $f(n) \geq n$ für alle $n \in \mathbb{N}_0$. Weiter sei $M = (Z, \Sigma, \Gamma, \delta, z_0)$ eine k–bändige TM mit $k \geq 2$, die für Eingabewörter der Länge n nur höchstens $f(n)$ viele Schritte durchführt. Dann existiert für alle Zahlen c mit $0 < c < 1$ eine TM M', die die gleiche Sprache erkennt und in der gleichen Situation mit höchstens $cf(n) + n + O(1)$ vielen Schritten auskommt. Hinter dem Term $O(1)$ verbirgt sich dabei eine kleine Konstante, deren genaue Größe ohne Belang ist.

Lassen Sie mich zunächst die beiden gegenüber Satz 6.1.1 zusätzlich aufgeführten Bedingungen kommentieren. Zum einen wird verlangt, dass für alle $n \in \mathbb{N}_0$ immer $f(n) \geq n$ gelten soll. Dies ist keine wirkliche Einschränkung, denn eine TM benötigt allein $n + 1$ Schritte, um ihr Eingabewort der Länge n sowie das Ende dieses Eingabeworts (anhand des ersten Blanks dahinter) zu erfassen.

Zum anderen werden mindestens zwei Bänder vorausgesetzt. Viele der späteren Resultate bzgl. der Zeitkomplexität gelten nicht für einbändige Turingmaschinen, weil man z.B. das Kopieren von Bandbereichen mit zwei Bändern deutlich schneller bewerkstelligen kann. Selbst bei der Analyse von Bandkomplexitäten werden wir in Kürze mindestens zwei Bänder voraussetzen — gedulden Sie sich einfach noch einen Augenblick.

Nun zurück zum Beweis von Satz 6.1.2:

Beweis: Wieder fassen wir je m Felder zu einem Feld zusammen (die Größe m werden wir in Abhängigkeit von c später noch genauer angeben). Nehmen Sie an, das Turingband von M' enthält bereits eine entsprechend komprimierte Version des aktuellen Bandinhalts der ursprünglichen Turingmaschine M. Nun könnte man erwarten, dass M' „automatisch“ m–mal schneller als M arbeitet. Unter ungünstigen Bedingungen ist dies jedoch nicht der Fall. Beispielsweise könnte sich der simulierte Kopf von M innerhalb des aktuellen komprimierten Symbols ganz rechts befinden. Wenn M nun den Kopf weiter nach rechts bewegt, befindet er sich (entsprechend simuliert) in dem rechts benachbarten komprimierten Bandsymbol auf dem ersten linken kodierten Zeichen. Wenn nun der Kopf

von M immer abwechselnd nach links und rechts oszilliert, muss auch M' jedesmal einen Schritt ausführen, und nichts wäre gewonnen. Also modifizieren wir die Arbeitsweise von M' ein wenig.

M' informiert sich jedes Mal zunächst über die benachbarten komprimierten Symbole links und rechts von dem aktuellen Symbol, indem sie ihren Kopf zuerst nach links, dann zweimal nach rechts und dann wieder einmal nach links bewegt. Dies führt sie auf allen Bändern gleichzeitig durch und kostet sie insgesamt vier Arbeitsschritte. Die beiden benachbarten Symbole von jedem Band merkt sich M' zusammen mit dem aktuell simulierten Zustand von M sowie den simulierten Kopfpositionen von M durch das Betreten eines entsprechenden Zustands. Die Zustandsmenge von M' umfasst folglich alle möglichen Kombinationen und ist entsprechend groß. Da es aber nur endlich viele solcher Kombinationen gibt, ist dies wie im Satz 6.1.1 kein prinzipielles Problem. Nun muss M mindestens $m + 1$ Schritte durchführen, um nach links oder rechts aus dem soeben eingelesenen Bereich (bestehend aus dem aktuellen Symbol von M' sowie den beiden eingelesenen Nachbarn) heraus zu laufen. Die entsprechenden simulierten Modifikationen der nächsten m Schritte von M spielen sich also mit Sicherheit innerhalb dieser drei komprimierten Symbole ab. Nur höchstens zwei dieser drei Symbole können sich in diesen m simulierten Schritten ändern, nämlich schlimmstenfalls das in der Mitte und evtl. das eine links oder rechts davon (aber nicht beide gleichzeitig). Also benötigt M' nur höchstens zwei Schritte, um die entsprechend modifizierten Symbole auf ihre Bänder zu schreiben und ihre Köpfe entsprechend umzupositionieren, sofern sich ein simulierter Kopf von M nun darin befindet.

Insgesamt kann M' in maximal sechs Schritten m Schritte von M simulieren. Weiterhin benötigt sie anfangs $n+1$ Schritte, um das Eingabewort mit seinem Ende zu erfassen und gleichzeitig auf einem anderen Band die zugehörige komprimierte Version zu erstellen. Beachten Sie, dass wir an dieser Stelle wirklich mindestens zwei Bänder benötigen. Da das komprimierte Wort (inkl. einem evtl. nur teilweise gefüllten Feld) nur höchstens $\frac{n}{m}+1$ viele Felder belegt, müssen wir mit dem Kopf einer nur einbändigen TM die anwachsende Distanz zwischen der Original– und der komprimierten Version zurücklegen. Am Ende sind dies etwa

$$n - (\frac{n}{m} + 1) \approx \frac{m-1}{m} n = \Omega(n)$$

viele Felder, die es für den Transport von einzelnen Zeichen zu überwinden gilt. M' kann sich in ihren endlich vielen Zuständen aber nur jeweils endlich viele der insgesamt

$$\frac{n}{m} + 1 = \Omega(n)$$

vielen komprimierten Zeichen merken und muss deshalb auch $\Omega(n)$–oft diese Distanz hin– und herpendeln. Am Ende führt dies zu einem quadratischen Mehraufwand. Mit zwei oder mehr Bändern ist dieser Vorgang dagegen mit nur linearem Aufwand möglich.

Während der Kodierung überschreibt M' die ursprüngliche Eingabe mit Blanks. Wenn sie die komprimierte Version des Eingabeworts erstellt hat, benötigt M' nochmals $\frac{n}{m} + O(1)$ Schritte, um das erstellte Wort auf das Band mit der mittlerweile gelöschten Eingabe

zurück zu kopieren, das linke Ende des komprimierten Wortes festzustellen und ihren Kopf unter das erste komprimierte Symbol zu stellen. Danach startet M' mit der oben beschriebenen Simulation.

Der gesamte Zeitaufwand beträgt also $n+1$ Schritte für die Kompression der Eingabe, $\frac{n}{m} + O(1)$ Schritte für die Fertigstellung der Anfangskonfiguration und schließlich (aufgerundet)

$$\frac{6}{m} f(n) + 1$$

viele Schritte für die eigentliche Simulation an sich. Dies sind

$$n + 1 + \frac{n}{m} + O(1) + \frac{6}{m} f(n) + 1 = \frac{1}{m} n + \frac{6}{m} f(n) + n + O(1)$$

Schritte insgesamt, wegen $n \leq f(n)$ also höchstens

$$\frac{7}{m} f(n) + n + O(1)$$

Schritte. Wenn wir folglich m größer als $\frac{7}{c}$ wählen, ist alles bewiesen. □

Satz 6.1.2 verwendet im Endeffekt die gleiche Technik, die auch bei aktuellen Computern zum Tragen kommt. Früher gab es 4–Bit–Prozessoren, dann 8–Bit–Prozessoren, später 16–Bit–Prozessoren, usw. Es werden also immer größere Informationen gleichzeitig in einer bestimmten Zeiteinheit verarbeitet.

Kann man auch zeit– und platzeffizient die Anzahl der Bänder einer TM auf nur ein Band reduzieren? Wir haben im Beweis des letzten Satzes bereits angedeutet, dass hierbei bzgl. der Zeit mit einem quadratischen Mehraufwand gerechnet werden muss. Man kann beweisen, dass bestimmte Sprachen diesen Mehraufwand auch tatsächlich erzwingen. Bzgl. der Bandkomplexität ist die Reduktion auf ein Band jedoch ohne Mehrkosten möglich:

6.1.3 Satz Zu jeder k–bändigen TM M existiert eine genauso speichereffiziente einbändige TM M', die die gleiche Sprache erkennt.

Beweis: Wir simulieren M durch eine einbändige TM M' mit der Technik aus Satz 4.3.2. Jedes von M' belegte Feld wird in mindestens einer Spur auch von der simulierten TM M benutzt. M belegt also mindestens genauso viele Felder wie M', d.h. M' ist mindestens ebenso speichereffizient wie M. □

Die linearen Beschleunigungssätze für Speicher und Zeit zeigen, dass es auf konstante Faktoren bei den Komplexitätsmaßen nicht ankommt. Folglich macht es auch bei der Verwendung von Turingmaschinen Sinn, Komplexitätsklassen mit Hilfe der O–Notation zu definieren:

6.1.4 Definition Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion mit $f = \Omega(n)$. Eine Sprache $L \subseteq \Sigma^*$ liegt in DTIME($f(n)$) bzw. NTIME($f(n)$), falls eine DTM bzw. NTM M mit $L(M) = L$ existiert, die für jedes Wort $x \in \Sigma^*$ höchstens $O(f(|x|))$ viele Schritte ausführt.

Ganz ähnlich legen wir nun die Speicherkomplexität für DTMs und NTMs fest. Wir wollen prinzipiell messen, wie viele Bandfelder während einer Berechnung maximal in Anspruch genommen werden. Dabei müssen wir aber die von einem Eingabewort anfangs belegten Felder ignorieren, da ansonsten eine Eingabe der Länge n immer eine Bandkomplexität von mindestens $\Omega(n)$ verursacht. Eine sublineare Bandkomplexität wäre somit nicht möglich.

Wir verwenden daher von nun an zumindest zweibändige Turingmaschinen, wobei das erste Band wie zuvor als *Eingabeband* dient, auf dem das Eingabewort x hinterlegt wird. Der Kopf auf dem Eingabeband darf sich nur noch maximal ein Feld links oder rechts von x aufhalten (anhand der dort befindlichen Blanks kann die TM den Anfang und das Ende von x erkennen). Für den Kopf auf dem Eingabeband gibt es also nur noch $|x| + 2$ mögliche Positionen. Weiter darf das Eingabeband nur noch gelesen, aber nicht mehr beschrieben werden. Somit können wir auf diesem Band auch kein evtl. Ausgabewort hinterlassen. Hierfür verwenden wir stattdessen irgendeines der anderen Bänder (die wir von nun an *Arbeitsbänder* nennen), z.B. das letzte. Nur die Arbeitsbänder werden zum Messen des Speicherverbrauchs herangezogen.

Bzgl. der Zeitkomplexität ist diese Modifikation nicht notwendig, sie wirkt sich aber auch nicht negativ aus. Der Einfachheit halber verwenden wir daher das modifizierte Modell auch für die Zeitkomplexität. Wenn wir zukünftig von einer k–bändigen TM sprechen, bezieht sich die Zahl k immer auf die Anzahl der Arbeitsbänder. Das Eingabeband kommt als zusätzliches Band noch mit dazu.

6.1.5 Definition Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion. Eine Sprache $L \subseteq \Sigma^*$ liegt in DSPACE($f(n)$) bzw. NSPACE($f(n)$), falls eine DTM bzw. NTM M mit $L(M) = L$ existiert, die für jedes Wort $x \in \Sigma^*$ zwischendurch höchstens $O(f(|x|))$ viele Felder (auf allen Arbeitsbändern insgesamt) belegt.

Beachten Sie, dass eine TM M für nichtakzeptierte Eingabewörter evtl. unendlich lange läuft, obwohl sie gleichzeitig eine vorgegebene Bandbeschränkung einhält. (Beispielsweise könnte M „stur“ an einer Position verharren.) M kann sich aber selbst eine zusätzliche Zeitbeschränkung der Ordnung $O(f)$ für eine bestimmte Funktion $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ auferlegen. Hierfür markiert M auf einem separaten Band zuerst $O(f(n))$ viele Felder und läuft dann während der eigentlichen Arbeit mit ihrem Extra–Kopf einmal den abgesteckten Bereich entlang. Sobald M das Ende des Bereichs erreicht, verwirft sie wegen Zeitüberschreitung das Eingabewort. Ganz ähnlich kann M sicherstellen, dass sie bestimmte Speichergrenzen beachtet. Hierzu markiert die TM vorab auf jedem Band den erlaubten Bereich mit bestimmten Symbolen und bricht die Berechnung ab, sobald ein Kopf den Bereich nach links oder rechts zu verlassen versucht. Dies alles funktioniert aber nur, falls die TM ihre Bänder für die Zeit– oder Speichermessung auch wirklich selbst markieren kann.

6.1.6 Definition Eine Funktion $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ heißt *konstruierbar*, wenn es eine Turingmaschine gibt, die für alle Eingaben der Länge n in $O(n + f(n))$ viel Zeit und $O(f(n))$ viel Platz eines ihrer Bänder mit $f(n)$ vielen Symbolen bedruckt. (Zum Messen von Speicher kann die TM anschließend den abgesteckten Bereich leicht auf alle anderen Arbeitsbänder kopieren.)

Beachten Sie, dass auch für sublineare Funktionen wie z.B. $\log(n)$ immer mindestens linear viel Zeit für deren Konstruktion zur Verfügung stehen muss. Wir erlauben also $O(n + f(n))$ und nicht nur $O(n)$ viel Zeit, da wie bereits erwähnt eine TM mindestens $n + 1 = \Omega(n)$ Schritte benötigt, um ein Eingabewort der Länge n mit seinem Ende komplett zu erfassen.

Alle „üblichen“ Funktionen (z.B. alle Polynome) sind konstruierbar. Dies können Sie in den Übungen selbst ausprobieren.

In der gängigen Literatur zur Komplexitätstheorie wird übrigens zwischen *zeitkonstruierbaren* und *platzkonstruierbaren* Funktionen unterschieden. Für unsere Zwecke reicht aber ein gemeinsamer Konstruktionsbegriff aus.

Wir haben nun vier Komplexitätsklassen festgelegt und somit eine solide Grundlage für das Messen des Zeit– und Speicheraufwands zur Lösung bestimmter Probleme geschaffen. Wie aber hängen diese Komplexitätsmaße miteinander zusammen? Kann man z.B. zeigen, dass bei einer bestimmten Limitierung des Speicherplatzes auch die Zeitkomplexität begrenzt sein muss? Dies werden wir im folgenden Abschnitt klären.

6.2. Beziehungen zwischen den Komplexitätsmaßen

Manche Abhängigkeiten zwischen den Komplexitätsmaßen DTIME, DSPACE, NTIME und NSPACE sind offensichtlich, andere lassen sich mit interessanten Konstruktionen zeigen. Die einfachen Beziehungen fasst der folgende Satz zusammen.

6.2.1 Satz Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion. Dann gilt:

a) $\text{DTIME}(f(n)) \subseteq \text{NTIME}(f(n))$

b) $\text{DSPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$

c) $\text{DTIME}(f(n))) \subseteq \text{DSPACE}(f(n))$

d) $\text{NTIME}(f(n)) \subseteq \text{NSPACE}(f(n))$

Beweis: Die Aussage a) haben wir bereits in Satz 5.3.5 bewiesen. Mit unserer auf Turingmaschinen basierenden Neudefinition von DTIME und NTIME können wir alternativ argumentieren, dass jede DTM auch eine NTM ist (die zwar von ihren nichtdeterministischen Fähigkeiten keinen Gebrauch macht, aber dies ist ja nicht verboten). Analog folgt Aussgabe b). Die beiden letzten Aussagen folgen aus der Tatsache, dass eine TM in $O(f(n))$ viel Zeit nicht mehr als $O(f(n))$ viel Speicher beschreiben kann. □

Für die nächsten (weniger offensichtlichen) Zusammenhänge benötigen wir den folgenden Hilfssatz.

6.2.2 Lemma Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion mit $f = \Omega(\log n)$. Sei L eine Sprache in DSPACE($f(n)$) bzw. NSPACE($f(n)$). Sei weiter $M = (Z, \Sigma, \Gamma, \delta, z_0)$ eine entsprechende DTM bzw. NTM mit $L(M) = L$, die diese Speichergrenze respektiert. Dann gibt es eine von M abhängige Konstante $d > 1$, so dass jede Berechnung für ein Wort $x \in \Sigma^*$ höchstens $O(d^{f(|x|)})$ verschiedene Konfigurationen durchläuft.

Beweis: Nach Voraussetzung benutzt M für die Akzeptanz eines Wortes der Länge n nicht mehr als $O(f(n))$ viele Felder, d.h. es gibt zwei Konstanten $c \in \mathbb{N}$ und $n_0 \in \mathbb{N}$, so dass für alle $n \geq n_0$ der konkrete Speicherbedarf auf $c \cdot f(n)$ Felder begrenzt ist. Jedes dieser Felder enthält ein Symbol aus Γ, d.h. es gibt insgesamt höchstens $|\Gamma|^{c \cdot f(n)}$ verschiedene Bandinhalte. Nun berücksichtigen wir zusätzlich die verschiedenen Kopfpositionen auf den Bändern der TM. Auf dem Eingabeband kann der Lesekopf nur $n+2$ verschiedene Positionen einnehmen, jeder Kopf auf einem Arbeitsband dagegen $c \cdot f(n)$ viele Positionen. Wenn k die Anzahl der Arbeitsbänder von M bezeichnet, sind also insgesamt $(n+2)(c \cdot f(n))^k$ verschiedene Kombinationen für die Kopfpositionen möglich. Kombiniert mit den verschiedenen Bandinhalten und den $|Z|$ möglichen Zuständen ergeben sich so insgesamt höchstens

$$|Z|(n+2)(c \cdot f(n))^k |\Gamma|^{c \cdot f(n)} = |Z|(n+2)c^k (f(n))^k (|\Gamma|^c)^{f(n)}$$

verschiedene Konfigurationen, die M im Fall $n \geq n_0$ während ihrer Arbeit einnehmen kann.

Nun gilt $f = \Omega(\log n)$, also existieren zwei Konstanten $c' > 0$ und $n_0' \in \mathbb{N}$, so dass für alle $n \geq n_0'$ die Ungleichung $f(n) \geq c' \cdot \log_2 n$ erfüllt ist. Daraus folgern wir

$$(\sqrt[c']{2})^{f(n)} \geq (\sqrt[c']{2})^{c' \cdot \log_2 n} = ((\sqrt[c']{2})^{c'})^{\log_2 n} = 2^{\log_2 n} = n \ .$$

Für alle $n \geq n_0'' := \max\{2, n_0, n_0'\}$ können wir deshalb die obige Anzahl aller möglichen Konfigurationen wie folgt abschätzen:

$$\ldots \ \leq \ |Z| \cdot 2n \cdot c^k (f(n))^k (|\Gamma|^c)^{f(n)} \ \leq \ |Z| \cdot 2(\sqrt[c']{2})^{f(n)} c^k (f(n))^k (|\Gamma|^c)^{f(n)} \ .$$

Weiter gilt für jede positive Zahl $m \geq 0$ sicherlich $m < 2^m$. Da wie gesehen

$$f(n) \geq c' \cdot \log_2 n \geq c' > 0$$

für alle $n \geq n_0'' \geq \max\{2, n_0'\}$ erfüllt ist, erhalten wir $f(n) < 2^{f(n)}$, also

$$(f(n))^k < (2^{f(n)})^k = 2^{k \cdot f(n)} = (2^k)^{f(n)}$$

und damit im Fall $n \geq n_0''$ als Abschätzung für die Anzahl aller Konfigurationen

$$\ldots \ \leq \ |Z| \cdot 2(\sqrt[c']{2})^{f(n)} c^k (2^k)^{f(n)} (|\Gamma|^c)^{f(n)} \ = \ \underbrace{2c^k |Z|}_{\text{konstant}} \cdot (\sqrt[c']{2} \cdot 2^k \cdot |\Gamma|^c)^{f(n)} \ = \ O(d^{f(n)})$$

mit der Wahl $d := \sqrt[c']{2} \cdot 2^k \cdot |\Gamma|^c$. □

Nun können wir unser nächstes Resultat beweisen:

6.2.3 Satz Sei L ein Sprache über einem Alphabet Σ und $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion. Dann gilt:

a) Ist $L \in \text{NTIME}(f(n))$, so gilt auch $L \in \text{DTIME}(c^{f(n)})$ für eine von L abhängige Konstante c.

b) Ist $L \in \text{NSPACE}(f(n))$ und $f = \Omega(\log n)$, so gilt ebenfalls $L \in \text{DTIME}(c^{f(n)})$ für eine von L abhängige Konstante c.

Beweis: Wir zeigen zunächst Aussage b). Sei $L \in \text{NSPACE}(f(n))$ mit $f = \Omega(\log n)$. Sei ferner M eine NTM für L, die mit $O(f(n))$ viel Speicher auskommt. Eine deterministische Turingmaschine M' kann wie folgt die Menge K aller möglichen Konfigurationen erstellen, die M bei der Verarbeitung eines Wortes $x \in \Sigma^*$ erreichen kann. Anfangs enthält K nur die Startkonfiguration z_0x. Dann durchläuft M' wiederholt alle Konfigurationen in K und überprüft, ob sich von einer dieser Konfigurationen eine Folgekonfiguration erreichen lässt, die noch nicht in K gespeichert ist. Ist dies der Fall, fügt sie diese Konfiguration zu K hinzu und startet die nächste Überprüfung von K. Sobald keine Änderung mehr eintritt, überprüft M', ob mindestens eine Konfiguration in K akzeptierend ist, woraufhin auch M' das Eingabewort x akzeptiert. Offenbar erkennt so M' die gleiche Sprache L wie M.

Wieviel Zeit benötigt M' für diesen Algorithmus? Wie geben im Folgenden eine recht großzügige Abschätzung an, die für unsere Zwecke jedoch ausreicht.

K enthält zu keinem Zeitpunkt mehr als $O(d^{f(|x|)})$ viele Konfigurationen, denn nach Lemma 6.2.2 sind mehr Konfigurationen gar nicht verfügbar. Somit wird M' die Menge K auch in höchstens $O(d^{f(|x|)})$ vielen Durchläufen konstruieren, da K bei jedem Durchlauf um mindestens eine Konfiguration anwächst. Also wird M' höchstens

$$O(d^{f(|x|)}) \cdot O(d^{f(|x|)}) = O((d^{f(|x|)}))^2 = O(d^{2 \cdot f(|x|)}) = O((d^2)^{f(|x|)})$$

mal versuchen, aus einer Konfiguration heraus eine Nachfolgekonfiguration zu bestimmen. Um eine Konfiguration zu speichern, verwenden wir die Technik aus Satz 4.3.2, d.h. wir kleben die Bandinhalte zu mehreren Spuren zusammen und markieren die passenden Kopfpositionen. Den aktuellen Zustand kodiert M durch eines von $|Z|$ neuen Zeichen in Γ und speichert ihn direkt hinter dem Bandinhalt einer Konfiguration, womit zugleich die Konfigurationen voneinander abgegrenzt werden. Eine Konfiguration besteht somit aus bis zu $O(f(|x|)+1) = O(f(|x|))$ vielen Feldern, und die gesamte Menge K kann mit

$$O(d^{f(|x|)}) \cdot O(f(|x|)) = O(d^{f(|x|)} \cdot f(|x|))$$

viel Platz gespeichert werden.

Für das Anfertigen einer Nachfolgekonfiguration läuft M' wie in Satz 4.3.2 die jeweilige Ausgangskonfiguration zweimal ab (was $O(f(|x|)$ viel Zeit kostet), erstellt das Ergebnis aber auf einem Extra–Band. Da es nur endlich viele nichtdeterministische Auswahlmöglichkeiten bei jedem Übergang gibt, wird es (für eine passend gewählte Konstante $k \in \mathbb{N}$) nie mehr als jeweils k solche Nachfolgekonfigurationen geben. Das Erstellen *aller* möglichen Nachfolgekonfiguration kostet also höchstens $O(k \cdot f(|x|)) = O(f(|x|))$ viel

Zeit. Jede ermittelte Nachfolgekonfiguration muss nun noch daraufhin überprüft werden, ob sie bereits in K vertreten ist. Da K bis zu $O(d^{f(|x|)})$ viele Einträge besitzt und jeder Vergleich aufgrund der Länge einer Konfiguration mit $O(f(x))$ viel Zeit zu Buche schlägt, kostet dies $O(d^{f(|x|)}) \cdot O(f(|x|)) = O(d^{f(|x|)} \cdot f(|x|))$ viel Zeit. Da dieser Aufwand wie bereits gesehen maximal $O((d^2)^{f(|x|)})$ oft anfällt, beträgt der Gesamtaufwand zur Konstruktion von K

$$O((d^2)^{f(|x|)}) \cdot O(d^{f(|x|)} \cdot f(|x|)) = O((d^3)^{f(|x|)} \cdot f(|x|)) \ ,$$

was sich wegen $f(|x|) < 2^{f(|x|)}$ für genügend lange Wörter x (siehe Beweis von Lemma 6.2.2) auf

$$O((d^3)^{f(|x|)} \cdot 2^{f(|x|)}) = O((2d^3)^{f(|x|)})$$

abschätzen lässt. Mit der Wahl $c := 2d^3$ ist somit Aussage b) bewiesen.

Für Zeitkomplexitätsklassen haben wir immer mindestens lineare Zeitschranken vorausgesetzt, da sich ansonsten die Eingabewörter nicht komplett lesen lassen. Wenn also $L \in \mathrm{NTIME}(f(n))$ erfüllt ist, so gilt sicherlich $f = \Omega(n)$, also auch $f = \Omega(\log n)$. Aussage a) folgt daher aus Aussage b) und Satz 6.2.1 d) wegen

$$L \in \mathrm{NTIME}(f(n)) \implies L \in \mathrm{NSPACE}(f(n)) \implies L \in \mathrm{DTIME}(c^{f(n)})$$

für eine wie in Aussage b) passend gewählte Konstante c. □

Die deterministische Simulation nichtdeterministischer Turingmaschinen ist also mit einem exponentiell teurem Zusatzaufwand an Zeit verbunden. Man könnte der Ansicht sein, dass dies auch für Speicher gilt, da die in Satz 6.2.3 vorgestellte Simulation auch exponentiell mehr Platz verlangt. Durch eine anders ausgerichtete Simulation kann man den Mehraufwand jedoch quadratisch beschränken. 1970 konnte Savitch folgendes Resultat beweisen [22]:

6.2.4 Satz Sei $f = \Omega(\log n)$ eine konstruierbare Funktion. Dann gilt

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{DSPACE}(f^2(n)) \ ,$$

wobei $f^2(n)$ eine abkürzende Notation für $f(n) \cdot f(n)$ ist. Es gilt also beispielsweise

$$\mathrm{NSPACE}(n^3) \subseteq \mathrm{DSPACE}(n^6)$$

oder auch

$$\mathrm{NSPACE}(\log n) \subseteq \mathrm{DSPACE}(\log^2 n) \ .$$

Beweis: Sei $L \in \mathrm{NSPACE}(f(n))$ und M eine NTM für L, die mit $O(f(n))$ viel Speicher auskommt. Nach Lemma 6.2.2 gibt es eine von M abhängige Konstante d, so dass M nur höchstens $O(d^{f(|x|)})$ verschiedene Konfigurationen bei der Verarbeitung eines Wortes $x \in \Sigma^*$ durchlaufen kann. Insbesondere macht eine Berechnung über mehr als $O(d^{f(|x|)})$

viele Schritte keinen Sinn, da sich dann Konfigurationen wiederholen müssen. Die NTM würde sich also in einer gleichen Situation wie zuvor befinden, und die entsprechende „Schleife“ könnte ohne Nebenwirkungen aus der Berechnung herausgeschnitten werden. Jedes Wort $x \in L$ kann also durch eine Berechnung mit höchstens $O(d^{f(|x|)})$ vielen Schritten erkannt werden. Es gibt demnach zwei Konstanten $c \in \mathbb{N}$ und $n_0 \in \mathbb{N}$, so dass M für alle Eingabewörter x der Länge $n \geq n_0$ höchstens $c \cdot d^{f(n)}$ Konfigurationen zu durchlaufen braucht. Formen wir diese Grenze zu

$$c \cdot d^{f(n)} = 2^{\log_2(c \cdot d^{f(n)})} = 2^{\log_2 d \cdot f(n) + \log_2 c}$$

um und runden wir weiter den Ausdruck $\log_2 d \cdot f(n) + \log_2 c$ zu einer ganzen Zahl t auf, so erhalten wir als obere Zeitschranke den Wert $2^{t \cdot f(n)}$, also eine Zweierpotenz. Für kürzere Wörter x mit $|x| < n_0$ gilt dies nicht unbedingt. Da es sich aber nur um endlich viele Ausnahmen handelt, die in der O–Notation später keine Rolle spielen werden, betrachten wir diesen Aspekt nicht weiter.

Die Idee ist nun einfach: Die simulierende DTM M' versucht, die in der Mitte liegende Konfiguration K zu finden, die nach der Hälfte dieser Schritte eingenommen wird. Raten kann M' diese Konfiguration nicht, da M nicht nichtdeterministisch ist. Sehr wohl kann sie aber systematisch alle Konfigurationen durchgehen, die mit $O(f(n))$ viel Speicher überhaupt möglich sind. Hierfür steckt M' einen entsprechenden Bandbereich ab (dies kann sie, da f konstruierbar ist) und iteriert über alle Wörter dieser Länge. Dabei werden zwar auch Wörter überprüft, die allein von ihrem Aufbau her gar keiner gültigen Konfiguration entsprechen können (z.B. weil nirgends ein aktueller Zustand von M kodiert ist), aber dies kostet nur unnötig Zeit, nicht jedoch Speicherplatz. Nur diesen möchten wir hier optimieren.

Woran kann M' erkennen, dass die aktuelle Konfiguation K eine passende ist, also in der Mitte einer zu ermittelnden Berechnung liegt? Hierfür muss sie zwei Bedingungen überprüfen:

- Ist K in höchstens halb so vielen Schritten (also mit einer maximal $2^{t \cdot f(n)-1}$ langen Berechnung) von der Startkonfiguration aus erreichbar?
- Ist umgekehrt von K aus mit einer ebenfalls maximal $2^{t \cdot f(n)-1}$ langen Berechnung eine akzeptierende Konfiguration erreichbar?

Wir können direkt zwei Eigenschaften beobachten:

- Wenn M' beide Tests nacheinander ausführt, kann sie den dafür benötigten Platz (den wir gleich noch genauer analysieren werden) zweimal verwenden.
- Jeder der beiden Tests lässt sich analog in zwei noch kleinere Tests aufteilen. Für den ersten Test muss M' beispielsweise nach einer Konfiguration K' suchen, so dass K' von der Startkonfiguration aus in höchstens $2^{t \cdot f(n)-2}$ vielen Schritten erreichbar ist. Gleiches gilt für die Erreichbarkeit von K ausgehend von K' aus. Wir erhalten also ein rekursives Testverfahren.
- Die Rekursion endet, wenn nur noch höchstens ein Schritt erlaubt ist. Dann müssen nämlich entweder beide Konfigurationen identisch sein (was 0 Schritten entspricht)

oder die eine Konfiguration muss eine direkte Nachfolgekonfiguration der anderen sein. Dies lässt sich anhand der Übergangsfunktion leicht überprüfen.

Die obigen Überlegungen führen zu dem folgenden Programm, welches (als Turingprogramm formuliert) von M' ausgeführt wird:

```
bool findeAkzeptierendeBerechnung(Eingabewort x) {
   Sei K_0 die Startkonfiguration von M für das Wort x;
   for jede akzeptierende Konfiguration K_e von M do {
      if test(K_0, K_e, t · f(n)) then {
         return true;
      }
   }
   return false;
}

/*
 * test(K_1, K_2, i) gibt genau dann true zurück, wenn M von
 * K_1 aus in höchstens 2^i vielen Schritten K_2 erreichen kann.
 */

bool test(Konfiguration K_1, Konfiguration K_2, int i) {
   if i = 0 then {
      if (K_1 = K_2) or (K_1 ⊢ K_2) then {
         return true;
      }
   } else {
      for jede Konfiguration K von M do {
         if test(K_1, K, i - 1) and test(K, K_2, i - 1) then {
            return true;
         }
      }
   }
   return false;
}
```

Wieviel Platz wird hierfür von M' benötigt? Jeder Aufruf der rekursiven Methode `test(`K_1`,` K_2`,` i`)` benötigt im Wesentlichen nur Speicher für die drei Parameter K_1, K_2 und i sowie für die Schleifenvariable K. Jede Konfiguration kommt mit $O(f(n))$ viel Speicher auf den Arbeitsbändern aus, da M nach Voraussetzung entsprechend speicherbegrenzt ist. Zusätzlich steht auf dem Eingabeband noch das n Zeichen lange Eingabewort x. Da dieses von M aber nicht geändert werden kann, braucht M' es in den Konfigurationen nicht mit zu speichern. Stattdessen muss sich M' nur die aktuelle Position des Kopfes auf dem Eingabeband merken. Den entsprechenden Zahlenwert von

0 (eine Position links von der Eingabe) bis $n+1$ (eine Position rechts von der Eingabe) kann M' binär kodiert in $O(\log n)$ vielen Feldern festhalten. Wegen $f = \Omega(\log n)$ kann also eine komplette Konfiguration in $O(\log n) + O(f(n)) = O(f(n))$ viel Platz gespeichert werden. Der Parameter i bekommt seinen größten Wert in Hauptprogramm zugewiesen und beträgt dann $t \cdot f(n)$, was nur $O(\log(t \cdot f(n))) = O(\log(f(n)))$ viele Bits benötigt. Also können alle Variablen innerhalb der rekursiven Funktion `test` mit $O(f(n))$ viel Platz gespeichert werden. Gleiches gilt für das Hauptprogramm mit seinen Konfigurationsvariablen K_0 und K_e.

Bei jedem rekursiven Aufruf sinkt der Zähler i um eins, und beim Stand $i = 0$ endet die Rekursion. Die maximale Rekursionstiefe beträgt also $t \cdot f(n) = O(f(n))$, d.h. die gesamte Speicherbelastung für das Festhalten aller lokalen Variablen und Parameter von allen ineinander verschachtelten `test`–Aufrufen übersteigt zu keinem Zeitpunkt

$$O(f(n)) \cdot O(f(n)) = O(f^2(n)) \ ,$$

was die behauptete Platzschranke beweist. □

6.2.5 Korollar Der Satz 6.2.4 von Savitch gilt sogar dann, wenn $f = \Omega(\log n)$ nicht konstruierbar ist.

Beweis: M muss für ein Eingabewort der Länge n den Wert $f(n)$ herausfinden, um die obige Simulation durchführen zu können. Bislang hat sie dafür einfach $f(n)$ konstruiert, was nun nicht mehr möglich ist. Aber M kann sich an den Wert $f(n)$ „herantasten“, indem sie im obigen Hauptprogramm nicht über jede akzeptierende Konfiguration iteriert, sondern stattdessen über jede Konfiguration der Länge m, wobei m aufsteigend die Werte $1, 2, 3, 4, \ldots$ durchläuft. Während der Berechnung lässt sie dabei nur Konfigurationen der Länge $m-1$ zu. Hierdurch testet M, ob M' bei Eingabe x überhaupt Konfigurationen der Länge m erreichen kann. Sobald sie merkt, dass keine erreichbare Konfiguration der Länge m existiert, weiß M, dass M' angesetzt auf x nicht mehr als $m-1$ Bandfelder benötigt. Nun hat sie fehlende Größe ermittelt und kann wie oben beschrieben den eigentlichen Test für x durchführen. □

Auch deterministische Zeit kann man platzeffizient simulieren und dabei immerhin einen logarithmischen Faktor einsparen. Die genaue Darstellung der Idee ist hier jedoch zu aufwendig. Wir notieren daher das folgende Resultat von Hopcroft, Paul und Valient aus dem Jahr 1977 ohne Beweis [15]:

6.2.6 Satz Sei f eine konstruierbare Funktion. Dann gilt

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n)/log(f(n))) \ .$$

Somit haben wir festgestellt, dass zwischen allen vier Komplexitätsklassen gewisse Beziehungen bestehen. Ist eine Sprache L in einer Klasse enthalten, so auch in jeder anderen Klasse, wobei wir natürlich gegebenenfalls die zugehörige Komplexitätsfunktion erheblich (z.B. exponentiell) aufstocken müssen.

6.3. Hierarchiesätze

Wir kommen nun für alle vier Komplexitätsmaße auf unsere eingehende Frage zu diesem Kapitel zurück:

Kann man mit mehr Zeit und / oder mehr Speicher auch mehr berechnen?

Wir werden diese Frage nacheinander für alle vier Komplexitätsmaße beantworten. Mit geeigneten Techniken werden wir dichte Hierarchien nachweisen. Nur bei der nichtdeterministischen Zeit werden wir leichte Abstriche machen müssen (dazu später mehr). Wir benötigen jedoch zuerst als Vorbereitung eine sog. *Aufzählung* aller Turingmaschinen. Dabei geht es um ein Verfahren, welches jeder Zeichenkette über $\{0,1\}$ eine konkrete Turingmaschine zuordnet.

Wir gehen als Erstes der Einfachheit halber davon aus, dass alle Turingmaschinen von nun an auf dem Eingabealphabet $\Sigma = \{0,1\}$ und dem Bandalphabet $\Gamma = \{0,1,\square\}$ arbeiten. Dies ist keine wirkliche Einschränkung, denn wenn eine Turingmaschine neben dem Blank z.B. noch acht weitere Bandsymbole hat, kann man jedes Symbol durch einen dreistelligen Binärcode darstellen, wobei dann drei Felder belegt werden. Man führt also eine Art *Banddekompression* durch (also sozusagen das Gegenteil von Satz 6.1.1) und erreicht damit das gewünschte binäre Bandalphabet. Die Wahl von $\Sigma = \{0,1\}$ wurde bewusst so gewählt, da die gleich vorgstellte Kodierung einer DTM ebenfalls in ein Wort über diesem Alphabet erfolgt und somit eine DTM eine andere (kodierte) DTM auf ihrem Band speichern und verarbeiten kann.

Ein konkretes Kodierungsverfahren (es sind viele verschiedene denkbar) z.B. für DTMs kann man nun am einfachsten an einem Beispiel mitverfolgen. Betrachten Sie dazu nochmals die einfache Turingmaschine auf Seite 163 aus dem Beweis des Satzes 4.6.1:

	0	1
z_0	$(z_0, 1, R)$	$(z_1, 0, N)$
z_1	$(z_1, 0, L)$	$(z_2, 0, N)$
z_2	$-$	$-$

Beachten Sie, dass die Kodierung einer solchen Turingtafel ausreichend ist. Die Zustandsmenge ist daraus nämlich klar erkennbar (hier also $\{z_0, z_1, z_2\}$), und das Eingabe– und Bandalphabet haben wir bereits auf $\{0,1\}$ und $\{0,1,\square\}$ festgelegt. Als Endzustände betrachten wir zudem einfach alle Zustände, die überhaupt keine Übergänge besitzen (hier also nur z_2). Turingmaschinen, die solche übergangslosen Zustände als Nicht–Endzustände mitführen, würden auch ohne solche Zustände auskommen und können daher außer Acht gelassen werden.

Unser Kodierungsverfahren arbeitet nun wie folgt:

Schritt 1: Wir ergänzen die Turingtafel so, dass alle Übergänge für alle drei Symbole 0, 1 und $\square$ angegeben werden (wenn es keine solchen Übergänge gibt, tragen wir „$-$“ ein). Hier fehlen beispielsweise noch alle Übergänge für das Blank:

	0	1	□
z_0	$(z_0, 1, R)$	$(z_1, 0, N)$	–
z_1	$(z_1, 0, L)$	$(z_2, 0, N)$	–
z_2	–	–	–

Schritt 2: Wenn wir vereinbaren, dass so wie hier in der ersten Spalte die Übergänge für die 0, in der zweiten Spalte die Übergänge für die 1 und in der dritten Spalte die Übergänge für das Blank eingetragen sind, so können wir die Spaltenüberschriften auch weglassen. Genauso lassen sich die Zeilenbeschriftungen einsparen, wenn wie wie üblich als Bezeichner für die Zustände $z_0, z_1, z_2, \ldots$ benutzen und in dieser Reihenfolge die Übergänge angeben:

$(z_0, 1, R)$	$(z_1, 0, N)$	–
$(z_1, 0, L)$	$(z_2, 0, N)$	–
–	–	–

Schritt 3: Nun kodieren wir in jedem Tripel (z_i, a, x) den Zustand z_i durch $i+1$ Einsen, d.h. z_2 wird z.B. durch 111 ersetzt. Ferner ersetzen wir das zu schreibende Zeichen a durch 00 (falls $a = 1$) oder durch 000 (falls $a = \square$). Im Fall $a = 0$ lassen wir die einzelne 0 stehen. Schließlich ersetzen wir die Kopfbewegung $x = L$ durch den Code 10, die Kopfbewegung $x = N$ durch 110 und die Kopfbewegung $x = R$ durch 1110. Einen undefinierten Übergang kodieren wir durch das Tripel $(1, 0000, 10)$, welcher durch den ansonsten nicht verfügbaren mittleren Code 0000 klar als solcher erkennbar ist. Daraus erhalten wir in unserem Beispiel:

$(1, 00, 1110)$	$(11, 0, 110)$	$(1, 0000, 10)$
$(11, 0, 10)$	$(111, 0, 110)$	$(1, 0000, 10)$
$(1, 0000, 10)$	$(1, 0000, 10)$	$(1, 0000, 10)$

Schritt 4: In jedem Tripel können wir nun auf die trennenden Kommata und die Klammern verzichten, denn die Zustandskodierung besteht nur aus Einsen und endet deshalb mit dem durch Nullen kodierten Zeichen a. Dessen Kodierung wiederum endet mit dem Beginn der Kodierung der Kopfbewegung, da diese immer mit einer Eins beginnt:

1001110	110110	1000010
11010	1110110	1000010
1000010	1000010	1000010

Schritt 5: Weiter beginnt jetzt jedes kodierte Tripel mit einer Einserfolge, wechselt intern auf eine Nullfolge, wieder zurück auf eine Einserfolge und endet mit einer abschließenden Null. Also können wir sogar alle kodierten Tripel dicht an dicht schreiben:

10011101101101000010
1101011101101000010
10000101000010100010

Schritt 6: Da es immer genau drei Übergänge pro Zeile gibt, können wir sogar alle Zeilen in eine zusammenfassen:

100111011011010000101101011101101000010100001010000101000010

Schritt 7: Die Kodierung einer DTM in ein eindeutiges Wort über dem binären Alphabet $\Sigma = \{0, 1\}$ ist damit im Prinzip abgeschlossen. Aus später ersichtlichen Gründen möchten wir die Kodierungen jedoch so wählen, dass jede Turingmaschine sogar *unendlich* viele korrekte Kodierungen besitzt. Dies können wir ganz einfach dadurch erreichen, dass wir zusätzliche Präfixe

$$\varepsilon, 0, 00, 000, 0000, \ldots$$

vor jeder Kodierung erlauben. Die obige DTM besitzt somit diese Kodierungen:

100111011011010000101101011101101000010100001010000101000010
010011101101101000010110101110110100001010000101000010100001010
001001110110110100001011010111011010000101000010100001010000101000010
000100111011011010000101101011101101000010100001010000101000010100001010
...

Da die eigentliche Kodierung immer mit einer 1 beginnt, können wir einen vorangestellten Präfix immer korrekt erkennen und bei einer Dekodierung ignorieren.

Schritt 8: Jede deterministische Turingmaschine besitzt nun unendlich viele Kodierungen. Umgekehrt muss aber nicht jedes Wort aus Σ^* einer korrekten Kodierung entsprechen. Beispielsweise endet jede korrekte Kodierung gemäß unserer Konstruktion mit einer 0, also kann z.B. 11111 nicht korrekt sein. Es ist aber in vielen Beweisen nützlich, wenn jedes Wort einer gültigen DTM entspricht. Dazu legen wir einfach fest, dass jede bislang ungültige Kodierung als eine ordnungsgemäße Kodierung der obigen Beispiel–DTM zu interpretieren ist. In der Praxis bedeutet dies, dass man sich an einer Dekodierung eines Wortes aus Σ^* gemäß den obigen Schritten in umgekehrter Rehenfolge versucht und, falls dies misslingt, ersatzweise die obige DTM als Ergebnis zurückgibt.

Schritt 9: Somit entspricht jetzt jedes Wort $x \in \Sigma^*$ genau einer bestimmten DTM, die wir mit M_x bezeichnen. Wir sortieren abschließend alle Wörter aus Σ^* nach ihrer Länge und ferner Wörter gleicher Länge nach ihrem binär kodierten Wert. Somit können wir alle DTMs der Reihe nach aufzählen:

$$M_\varepsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{000}, M_{001}, \ldots$$

Mit ganz ähnlichen Kodierungsverfahren können wir auch NTMs, mehrbändige TMs usw. aufzählen.

Die Aufzählung von Turingmaschinen erlaubt den Nachweis diverse Negativresultate durch sogenannte *Diagonalisierungsbeweise*. Beispielsweise hatten wir in Beispiel 4.5.20 nach langen Vorbereitungen die Existenz einer nicht rekursiv aufzählbaren Sprache hergeleitet. Mit Hilfe der Diagonalisierungstechnik ist dies ganz einfach.

6.3.1 Satz Es gibt eine nicht rekursiv aufzählbare Sprache.

Beweis: Sei $M_\varepsilon, M_0, M_1, M_{00}, \ldots$ eine Aufzählung aller DTMs. Wir werden zeigen, dass die Sprache

$$L := \{x \in \{0,1\}^* \mid M_x \text{ verwirft das Wort } x\}$$

von keiner DTM akzeptiert wird, was nach Korollar 4.5.12 mit der Behauptung identisch ist.

Angenommen, L wird von einer DTM M erkannt, d.h. es gilt $L = L(M)$. Dann taucht M irgendwo in der Aufzählung aller DTMs auf (tatsächlich sogar unendlich oft). Sei also $M = M_y$ für eine passende Kodierung $y \in \{0,1\}^*$. Doch dann gilt

$$y \in L \iff M_y \text{ verwirft } y \iff y \notin L(M_y) \iff y \notin L(M) \iff y \notin L \ ,$$

was ein Widerspruch ist. □

In den gleich vorgestellten Hierarchiesätzen vergleichen wir jeweils zwei Komplexitätsklassen, bei denen die eine weniger Zeit– bzw. Platzressourcen zur Verfügung stellt als die andere. Die Sätze 6.1.1 und 6.1.2 bzgl. der Platz– und Zeitkompression lehren uns, dass sich die Ressourcen um mehr als nur einen linearen Faktor unterscheiden müssen, wenn sich ein Unterschied ergeben soll. Um dies auszudrücken, benötigen wir in Ergänzung zu der bereits häufig verwendeten O– und Ω–Notation noch ein drittes *Landausches Symbol* (siehe Seite 31), nämlich die sog. o–Notation. Während $O(h)$ bekanntlich die Menge

$$O(h) = \{g : \mathbb{N}_0 \longrightarrow \mathbb{R} \mid \exists c \in \mathbb{N}\colon\ \exists n_0 \in \mathbb{N}\colon\ \forall n \geq n_0\colon\ g(n) \leq c \cdot h(n)\}$$

aller Funktionen bezeichnet, die *nicht schneller* als h wachsen, stellt $o(h)$ (sprich: „klein Oh von h“) die Menge aller Funktionen dar, die *asymptotisch langsamer* als h wachsen:

$$o(h) = \{g : \mathbb{N}_0 \longrightarrow \mathbb{R} \mid \forall c \in \mathbb{N}\colon\ \exists n_0 \in \mathbb{N}\colon\ \forall n \geq n_0\colon\ c \cdot g(n) \leq h(n)\} \ .$$

Der Unterschied zwischen $O(h)$ und $o(h)$ liegt darin, dass jeder noch so große Unterscheidungsfaktor c irgendwann eingeholt wird. Für jede Funktion $g = o(h)$ geht also der Bruch

$$\frac{g(n)}{h(n)}$$

für genügend große Argumente n gegen 0. Beispielweise gilt $n^2 = o(n^3)$. Das folgende Lemma beweist diesen Aspekt noch einmal mathematisch exakt.

6.3.2 Lemma Seien f und g zwei Funktionen mit $f = \Omega(\log n)$ und $f = o(g)$. Dann gilt

$$\forall c, d \in \mathbb{N}\colon\ \exists n_0 \in \mathbb{N}\colon\ \forall n \geq n_0\colon\ c \cdot f(n) + d \leq g(n)\ .$$

Beweis: Die Menge $o(g)$ entspricht gemäß Definition dieser Menge:

$$\{f : \mathbb{N}_0 \longrightarrow \mathbb{R} \,|\, \forall c' \in \mathbb{N}\colon\ \exists n_0' \in \mathbb{N}\colon\ \forall n \geq n_0'\colon\ c' \cdot f(n) \leq g(n)\}\ .$$

Speziell für $c' := c+d$ gibt es eine Zahl n_0, so dass für alle $n \geq n_0$ stets $(c+d)f(n) \leq g(n)$ gilt. Wegen $f = \Omega(\log n)$ müssen ferner bis auf endlich viele Ausnahmen alle Funktionswerte von f größer oder gleich 1 sein. Wir korrigieren gegebenenfalls die Zahl n_0 nach oben, so dass auch diese Bedingung für alle $n \geq n_0$ erfüllt ist. Mit dieser Wahl von n_0 erhalten wir wegen

$$c \cdot f(n) + d \leq c \cdot f(n) + d \cdot f(n) = (c + d) \cdot f(n) \leq g(n)$$

die Behauptung. □

Mit diesen Vorbereitungen können wir nun den ersten Hierarchiesatz beweisen.

6.3.3 Satz Seien f und g zwei Funktionen mit $f = \Omega(\log n)$ und $f = o(g)$. Sei ferner g konstruierbar. Dann gilt

$$\mathrm{DSPACE}(f(n)) \subset \mathrm{DSPACE}(g(n))\ .$$

Beweis: Natürlich gilt $\mathrm{DSPACE}(f(n)) \subseteq \mathrm{DSPACE}(g(n))$, da man mit mindestens genauso viel Speicher auch immer mindestens genauso viel berechnen kann. Warum aber ist die Teilmengenbeziehung echt? Wir werden nachweisen, dass es eine Sprache $L \in \mathrm{DSPACE}(g(n))$ gibt, die nicht in $\mathrm{DSPACE}(f(n))$ liegt. Die Beweisidee basiert auf der Konstruktion einer gewissen Simulations–DTM M.

M arbeitet wie vereinbart auf dem Eingabealphabet $\Sigma = \{0, 1\}$. Angesetzt auf ein Eingabewort $x \in \Sigma^*$ steckt M zunächst den Wert $g(|x|)$ auf allen ihren Bändern ab (dies kann sie, da g konstruierbar ist). Danach interpretiert M die Eingabe x als eine einbändige DTM M_x und simuliert anschließend M_x auf x, also auf der Kodierung von M_x. Nun kommt es auf den Simulationsverlauf an, wie sich M weiter verhält:

- Wenn während der Simulation der markierte Platz verlassen wird, so stoppt M und verwirft.
- Wenn M_x den markierten Bereich einhält und x akzeptiert, dann akzeptiert auch M_x.
- Wenn M_x den markierten Bereich einhält und auch anhält, aber x nicht akzeptiert, dann verwirft M_x.

- Wenn M_x den markierten Bereich einhält, aber nach spätestens $2^{g(|x|)}$ Schritten immer noch nicht angehalten hat, so stoppt M und verwirft. M weiß nämlich in diesem Fall, dass M_x nicht akzeptieren wird (zumindest wenn x genügend lang ist, dazu gleich mehr). M kann die genannte Zeit messen, indem sie auf einem Extra–Band $g(|x|)$ viele Nullen hinterlässt, diesen Inhalt als einen binäre Zähler interpretiert und dann bei jedem Simuationsschritt um 1 erhöht. Sobald der Zähler überläuft, ist die Zeit verbraucht.

Kurz gesagt bildet also M das Verhalten von M_x auf der Eingabe x nach, sofern der Platz dafür ausreichend ist.

Wir können aus M eine DTM $\overline{M}$ konstruieren, die genau die zu $L(M)$ komplementäre Sprache $L := \Sigma^* \setminus L(M)$ akzeptiert. $\overline{M}$ besteht im Prinzip aus dem gleichen Turingprogramm wie M, dreht aber am Ende das Akzeptanzverhalten von M um. Wenn also bislang ein Blank unter dem Kopf stand (M also nicht akzeptiert hätte), druckt sie noch ein Zeichen aus Σ auf das aktuelle Feld und akzeptiert damit doch. Umgekehrt druckt sie ein Blank, wenn dort vorher kein Blank stand.

Die von $\overline{M}$ akzeptierte Sprache L liegt in DSPACE($g(n)$), da sich M und damit $\overline{M}$ vor der eigentlichen Simulation eine Selbstbeschränkung von $k \cdot g(n) = O(g(n))$ vielen Feldern auferlegen (dabei sei k die Anzahl ihrer Arbeitsbänder) und bei einer Verletzung dieser Grenze die Simulation abbrechen.

Die Sprache L liegt jedoch nicht in DSPACE($f(n)$). Denn angenommen doch, dann sei M' eine passende $O(f(n))$–speicherbeschränkte DTM mit $L(M') = L$. Wir können M' gemäß Satz 6.1.3 als einbändig annehmen. Nach Lemma 6.2.2 gibt es eine von M' abhängige Konstante d, so dass M' nur höchstens $O(d^{f(|x|)})$ verschiedene Konfigurationen bei der Verarbeitung von x durchläuft. Von endlich vielen Ausnahmen abgesehen ist also für eine passend gewählte Konstante c die Anzahl der verschiedenen Konfigurationen durch

$$c \cdot d^{f(|x|)} = c \cdot 2^{\log_2 d \cdot f(|x|)} < 2^c \cdot 2^{\log_2 d \cdot f(|x|)} = 2^{\log_2 d \cdot f(|x|)+c}$$

begrenzt. Nach Lemma 6.3.2 gibt es eine natürliche Zahl $n_0 \in \mathbb{N}$, so dass für alle $n \geq n_0$ die Bedingung $\log_2 d \cdot f(|x|) + c \leq g(|x|)$ erfüllt ist. Wir können davon ausgehen, dass n_0 die erwähnten endlich vielen Ausnahmen mit abdeckt, ansonsten erhöhen wir n_0 entsprechend.

Welchen Speicherbedarf hätte eine Simulation von M' durch M? M' ist $O(f(n))$–speicherbeschränkt, also werden für genügend große Wörter $x \in \Sigma^*$ und eine passend gewählte Konstante c' nie mehr als $c' \cdot f(|x|)$ simulierte Felder benötigt. Da das Bandalphabet Γ von M allerdings größer als das Bandalphabet von M sein kann, kodieren wir jedes Zeichen aus Γ binär durch $\log_2 |\Gamma|$ (aufgerundet) viele Bits in entsprechend vielen Feldern von M. Einen Zustand von M_x kodieren wir ebenfalls in logarithmisch vielen Feldern. Bis auf endlich viele Ausnahmen (durch zu kurze Wörter x) beträgt der Speicherbedarf für die Simulation also nie mehr als

$$\underbrace{(1 + \log_2 |\Gamma|) \cdot c'}_{\text{konstant}} \cdot f(|x|) + \underbrace{1 + \log_2 |Z|}_{\text{konstant}}$$

viele Felder (wobei Z die Zustandsmenge von M_x bezeichnet). Nach Lemma 6.3.2 gibt es eine natürliche Zahl $n_0' \in \mathbb{N}$, so dass für alle Wörter $x \in \Sigma^*$ mit $|x| \geq n_0'$ der Platzbedarf kleiner oder gleich $g(n)$ ist. Wieder soll n_0' die erwähnten Ausnahmen mit abdecken.

Nun taucht jede einbändige DTM unendlich oft in der Folge $M_\varepsilon, M_0, M_1, M_{00}, \ldots$ auf, insbesondere auch M'. Wir betrachten die erste DTM M_x mit $|x| \geq \max\{n_0, n_0'\}$, die M' entspricht. Wenn M_x das Wort x akzeptiert, so wird sie dies nach unserer Wahl von x in höchstens $2^{g(|x|)}$ vielen Schritten und unter Beachtung von $g(|x|)$ viel Speicher tun. (Bei mehr Schritten würde M_x mindestens eine Konfiguration doppelt durchlaufen und deshalb nie anhalten, insbesondere also x niemals akzeptieren.) Also kann M in diesem Fall die Simulation bis zu Akzeptanz von M_x erfolgreich zu Ende führen und akzeptiert deshalb ebenfalls. Wenn umgekehrt in allen anderen Fällen M_x das Wort x verwirft, dann verwirft auch M. Also akzeptiert M_x das Wort x genau dann, wenn x von M und damit *nicht* von $\overline{M}$ akzeptiert wird. Oder anders ausgedrückt: M_x und damit M' akzeptiert das Wort x genau dann, wenn es nicht in $L = L(\overline{M})$ liegt. Also wird M' die Sprache L nicht korrekt erkennen. Dies widerspricht der Wahl von M'. □

Die soeben vorgestellte Beweisidee mit der Konstruktion einer simulierenden Turingmaschine M funktioniert prinzipiell auch für den Nachweis einer entsprechenden Hierarchie in NSPACE, wenn also M nichtdeterministisch ist. Bei der Konstruktion der komplementären Turingmaschine $\overline{M}$ stoßen wir aber auf eine entscheidende Schwierigkeit. Solange M ein Wort x akzeptiert, kann $\overline{M}$ wie gehabt dieses Wort verwerfen. Sollte aber M das Wort x nicht akzeptieren, dann ist nicht klar, ob M das Wort x nur mit der soeben gewählten (geratenen) Berechnung oder mit *allen möglichen* Berechnungen verwirft (nur dann dürfen wir $\overline{M}$ akzeptieren lassen). Im deterministischen Fall gab es immer nur eine mögliche Berechnung, so dass dieser Fall dort nicht auftrat. Kann also für jede NTM M eine passende NTM $\overline{M}$ konstruiert werden, die bei gleicher Platzkomplexität die zugehörige komplementäre Sprache akzeptiert?

Die Antwort ist „ja“, aber diese Frage war mehr als 20 Jahre lang ein ungelöstes Problem. Die meisten Forscher waren sogar von der Antwort „nein“ ausgegangen, und so kam die folgende Lösung, welche von Immerman [16] und Szelepcsényi [26] unabhängig voneinander im Jahr 1987 gefunden wurde, einer Sensation gleich. Beide Wissenschaftler erhielten 1995 für ihre Leistung den *Gödel–Preis*.

6.3.4 Satz Für eine konstruierbare Funktion $f = \Omega(\log n)$ ist $\mathrm{NSPACE}(f(n))$ konstruktiv unter Komplementbildung abgeschlossen.

Beweis: Sei M eine NTM, die eine konstruierbare Platzkomplexität $f = \Omega(\log n)$ besitzt. Aus M heraus geben wir die Konstruktion einer NTM $\overline{M}$ an, die mit dem gleichen Platz auskommt und die komplementäre Sprache $\overline{L(M)}$ akzeptiert. $\overline{M}$ muss also ein Wort x genau dann akzeptieren, wenn M es nicht akzeptiert. Wir beschreiben, was $\overline{M}$ angesetzt auf x macht.

Für $m = 0, 1, 2, \ldots$ bestimmt $\overline{M}$, wieviele Konfigurationen M von der Startkonfiguration z_0x aus in höchstens m Schritten erreichen kann. Wir bezeichen diese Anzahl mit $c(m)$.

Wenn $c(m)$ bereits bekannt ist, kann M leicht die Anzahl $c(m+1)$ wie folgt bestimmen. Sie geht alle Konfigurationen durch, die mit $O(f(n))$ viel Speicher darstellbar sind, indem sie einen entsprechenden Bandbereich absteckt (dies kann sie, da f konstruierbar ist) und über alle Wörter dieser Länge iteriert. Für jedes dieser Wörter wird sie wie nachfolgend beschrieben prüfen, ob es sich um eine in maximal $m+1$ vielen Schritten erreichbare Konfiguration handelt. Falls ja, erhöht $\overline{M}$ einen Zähler für den späteren Wert von $c(m+1)$ um 1. Möglicherweise werden dabei wie beim Satz 6.2.4 von SAVITCH Wörter durchlaufen, die allein von ihrem Aufbau her gar keine Konfigurationen sein können, aber dieser Zeitverlust soll uns auch hier nicht weiter stören. Wenn alle Konfigurationen überprüft wurden, enthält der Zähler den Wert $c(m+1)$.

Für jede Konfiguration K geht $\overline{M}$ in einer inneren Schleife nochmals alle Konfigurationen durch. Für jede dieser „inneren“ Konfigurationen K' versucht $\overline{M}$ eine Berechnung der Länge $\leq m$ ausgehend von der Startkonfiguration zu raten. $\overline{M}$ schreibt diese Berechnung aber nicht auf, sondern simuliert einfach auf $O(f(n))$ viel Platz die NTM M. Falls K' tatsächlich durch eine solche Berechnung von z_0x aus ereichbar ist, erhöht sie einen dafür bereit gestellten Zähler t um 1 und prüft anhand der Übergangsfunktion von M, ob $K = K'$ oder $K \vdash K'$ gilt, um die Erreichbarkeit von K in höchstens $m+1$ vielen Schritten zu dokumentieren. Dann fährt sie mit der nächsten Konfiguration K' der inneren Schleife fort.

Wenn $\overline{M}$ in der inneren Schleife bei allen $c(m)$ vielen ereichbaren Konfigurationen eine passende Berechnung geraten hat, erreicht der Zähler t am Ende den Wert $c(m)$. Somit kann sich $\overline{M}$ sicher sein, keine in maximal m Schritten erreichbare Konfiguraton vergessen zu haben. Wenn also am Ende der inneren Schleife t einen kleineren Wert hat, ist genau dieses „Missgeschick“ passiert, und $\overline{M}$ verwirft. Mit dem richtigen Raten zum richtigen Zeitpunkt wird $\overline{M}$ aber Erfolg haben und $c(m+1)$ korrekt ermitteln.

Der erste Wert $c(0) = 1$ ist offensichtlich, denn in 0 Schritten ist nur die Startkonfiguration selbst erreichbar. Mit der obigen Methode bestimmt $\overline{M}$ nun die Werte $c(1)$, $c(2)$, $c(3)$, usw. Die Berechnung der Zahlenfolge endet, sobald sich ein Wert gegenüber dem Vorgängerwert nicht mehr erhöht. Dann hat $\overline{M}$ die Anzahl C aller überhaupt von M erreichbaren Konfigurationen ermittelt. Wenn sich $\overline{M}$ den Vorgängerwert in einer Variable C' merkt, sieht das bisher beschriebene Verfahren also so wie auf der nächsten Seite dargestellt aus.

Beachten Sie, dass M nach Lemma 6.2.2 für passend gewählte Konstanten c und d nur höchstens $c \cdot d^{f(|x|)}$ verschiedene Konfigurationen annehmen kann (von endlich vielen Ausnahmen für zu kurze Wörter x abgesehen). Wegen

$$c \cdot d^{f(|x|)} = c \cdot 2^{\log_2 d \cdot f(|x|)} < 2^c \cdot 2^{\log_2 d \cdot f(|x|)} = 2^{\log_2 d \cdot f(|x|)+c}$$

benötigt $\overline{M}$ zur Darstellung jeder Zahl $c(i)$ also nicht mehr als

$$\log_2 d \cdot f(|x|) + c = O(f(|x|))$$

viele Bits. Weil $\overline{M}$ zur Ermittlung von $c(m+1)$ zudem nur den Vorgängerwert $c(m)$ benötigt, kann die Berechnung aller $c(i)$–Werte in $O(f(|x|))$ viel Platz erfolgen.

```
int m := 0;
int C := 1;
int C';
Sei K_0 die Startkonfiguration von M für das Wort x;
repeat {
   C' := C;
   C := 0;
   for jede Konfiguration K von M do {
      int t := 0;
      bool r := false;
      for jede Konfiguration K' von M do {
         Simuliere eine geratene Berechnung von K_0 aus
            mit höchstens m vielen Schritten;
         if die geratene Berechnung endet mit K' then {
            t := t + 1;
            if (K' = K) or (K' ⊢ K) then {
               r := true;
            }
         }
      }
      if t < C' then {
         Stoppe Verarbeitung und verwerfe;
      }
      if r then {
         C := C + 1;
      }
   }
   m := m + 1;
} until C = C';
```

Sobald die Zahl C feststeht, geht $\overline{M}$ mit zwei ähnlichen verschachtelten `for`–Schleifen nochmals alle erreichbaren Konfigurationen durch, überspringt diesmal jedoch in der äußeren Schleife alle akzeptierenden Konfigurationen. Ermittelt sie dann immer noch die gleiche Anzahl C wie zuvor, so ist keine der erreichbaren Konfigurationen akzeptierend. Also gilt $x \notin L(M)$, und genau dann lassen wir $\overline{M}$ akzeptieren.

Somit akzeptiert $\overline{M}$ genau die komplementäre Sprache von M. □

Damit ist für eine NTM M das Problem der Konstruktion einer komplementären NTM $\overline{M}$ mit $L(\overline{M}) = \overline{L(M)}$ gelöst. Wir erhalten unseren zweiten Hierarchiesatz:

6.3.5 Satz Seien f und g zwei Funktionen mit $f = \Omega(\log n)$ und $f = o(g)$. Sei ferner g konstruierbar. Dann gilt

$$\mathrm{NSPACE}(f(n)) \subset \mathrm{NSPACE}(g(n)) \ .$$

Beweis: Wir können den Beweis von Satz 6.3.3 mit Hilfe von Satz 6.3.4 analog auf NTMs übertragen. □

Nun fehlen noch die beiden Hierarchiesätze für die deterministische und nichtdeterministische Zeit. Leider können wir die bis jetzt verwendete Beweisidee für die Zeithierarchien nicht weiter verwenden. Es gibt zwei gravierende Probleme.

Die simulierende Turingmaschine M aus den Sätzen 6.3.3 und 6.3.5 besitzt eine feste Anzahl von Arbeitsbändern, auf denen jede Turingmaschine M_x simuliert werden muss. Wenn M_x mehr Bänder als M hätte, wäre dies ein Problem. Zum Glück konnten wir nach Satz 6.1.3 jede TM M_x als einbändig annehmen, ohne dafür mehr Platz aufwenden zu müssen. Also reichte eine Aufzählung einbändiger TMs in den Sätzen 6.3.3 und 6.3.5 aus. Bzgl. der Zeit ist die in Satz 4.3.2 verwendete Reduktionstechnik auf ein Band jedoch mit einem quadratischen Zeitverlust verbunden. Dies haben wir im letzten Absatz oberhalb des Satzes 6.1.3 auf Seite 217 bereits erwähnt. Für einen zu Satz 6.3.3 analogen Beweis für deterministische Zeit müssten wir dann statt $f = o(g)$ sogar $f^2 = o(g)$ verlangen. Immerhin könnten wir so z.B. noch

$$\mathrm{DTIME}(n^2) \subset \mathrm{DTIME}(n^5)$$

beweisen, nicht aber mehr z.B.

$$\mathrm{DTIME}(n^2) \subset \mathrm{DTIME}(n^3) \ .$$

Wir müssen uns aber bei den zu simulierenden TMs M_x nicht unbedingt auf einbändige TMs beschränken. Die simulierende TM M darf eine genügend große Anzahl von Bänder besitzen, die aber vorab fest gewählt sein muss. Somit dürften alle TMs M_x z.B. zumindest zweibändrig sein. Tatsächlich ist die Simulation einer k–Band TM durch eine zweibändige TM nur mit einem logarithmischen Zeitverlust verbunden [11], so dass wir in Wirklichkeit nur $f \cdot \log(f) = o(g)$ statt $f^2 = o(g)$ verlangen müssen. Also können wir doch

$$\mathrm{DTIME}(n^2) \subset \mathrm{DTIME}(n^3)$$

beweisen, da $n^2 \log(n^2) = 2n^2 \log n = o(n^3)$ gilt. Für den Nachweis von z.B.

$$\mathrm{DTIME}(n^2) \subset \mathrm{DTIME}(n^2 \log n)$$

wären unsere Argumente aber immer noch zu schwach.

Das zweite Problem betrifft Satz 6.3.4, für den kein äquivalentes Resultat im Fall der nichtdeterministischen Zeit bekannt ist. Für die Komplexitätsklasse NTIME kommen wir also mit unserem bisherigen Ansatz nicht weiter. Dies ist insofern „ärgerlich“, weil wir den soeben genannten logarithmischen Zeitverlust bei nichtdeterministischer Zeit gar nicht hinnehmen müssten. Es gilt nämlich folgender Satz:

6.3.6 Satz Jede k–bändige NTM M mit einer konstruierbaren Zeitkomplexität $O(f(n))$ lässt sich durch eine zweibändige NTM M' mit gleicher Zeitkomplexität simulieren.

Beweis: M' rät einfach $O(f(n))$ viele Übergänge von M der Form $\delta(z,a) = (z',b,x)$ mit $x \in \{L,N,R\}$ und schreibt sich diese passend kodiert hintereinander auf eines ihrer beiden Arbeitsbänder. Da jeder Übergang nur eine konstante Größe besitzt, benötigt M' dafür nur $O(f(n))$ viel Zeit. Nun geht sie nacheinander alle geratenen Übergänge nochmals durch und simuliert gleichzeitig auf ihrem anderen Arbeitsband die Entwicklung des ersten Arbeitsbandes von M. Aus den geratenen Übergängen liest sie also nur die Änderungen und Kopfbewegungen für das erste Band ab und vollzieht sie entsprechend nach, die anderen Informationen bzgl. der restlichen Bänder ignoriert sie zunächst. Wenn irgendwelche Schwierigkeiten auftreten (z.B. weil ein zu lesendes Zeichen nicht auf dem Band steht oder der aktuelle Zustand nicht zum nächsten Übergang passt), so verwirft M' — dassselbe hätte auch M gemacht. Wenn alles gut geht, simuliert M' anschließend nacheinander die anderen Arbeitsbänder von M auf die gleiche Art und Weise. Von jeder Simulation merkt sie sich am Ende noch das aktuelle Symbol unter dem Kopf des Simulationsbandes. Dies kann wieder durch eine Erweiterung ihrer Zustandsmenge erfolgen. Für die am Ende k gemerkten Symbole prüft sie noch, ob es sich um eine haltende und akzeptierende Konfiguration handelt. Insgesamt hat sie damit die gesamte Verarbeitung von M in $k \cdot O(f(n)) = O(f(n))$ viel Zeit nachvollzogen. □

Wir zeigen nun im Fall $f = o(g)$, dass eine bestimmte Sprache L existiert, die von einer $(k+1)$–bändigen DTM mit Zeitkomplexität $O(g)$ akzeptiert werden kann, nicht aber von einer k–bändigen DTM mit Zeitkomplexität $O(f)$. Wir bezeichnen die entsprechenden Komplexitätsklassen mit $\mathrm{DTIME}_{k+1}(g)$ bzw. $\mathrm{DTIME}_k(f)$. Mit dieser feineren Unterteilung der deterministischen Zeitkomplexitätsklassen können wir die dichten Hierarchien aufrechterhalten.

6.3.7 Satz Seien f und g zwei Funktionen mit $f = \Omega(\log n)$ und $f = o(g)$. Sei ferner g konstruierbar. Dann gilt

$$\mathrm{DTIME}_k(f(n)) \subset \mathrm{DTIME}_{k+1}(g(n)) \ .$$

Beweis: Zunächst gilt wieder $\mathrm{DTIME}_k(f(n)) \subset \mathrm{DTIME}_{k+1}(g(n))$, da wir mit mehr Zeit und einem zusätzlichen Band mindestens genauso viel berechnen können. Nun greifen wir die Beweisidee aus Satz 6.3.3 wieder auf und konstruieren eine $(k+1)$–bändige DTM M, die k–bändige DTMs aus einer entsprechende Aufzählung $M_\varepsilon, M_0, M_1, M_{00}, \ldots$ simulieren wird.

Angesetzt auf ein Eingabewort $x \in \{0,1\}^*$ markiert M zunächst den Wert $g(|x|)$ auf ihrem $(k+1)$–sten Band (dies kann sie, da g konstruierbar ist). Anschließend simuliert sie M_x auf x und ihren ersten k Bändern. Wie in Satz 6.3.3 kodieren wir jedes Zeichen aus dem Bandalphabet Γ von M_x durch $1 + \log_2 |\Gamma|$ viele Bits in entsprechend vielen Feldern. Auch die Zustände von M_x werden wieder logarithmisch kodiert. Beachten Sie außerdem, dass M die Kodierung von M_x in einer zweiten Spur auf z.B. dem ersten

Band mitführen kann. Als Kodierungsinformation reicht dafür die „eigentliche“ Kodierung aus, also ohne den führenden Präfix in Form einer Nullensequenz (vgl. Schritt 7 unserer Kodierungsvorschrift auf Seite 228). Immer wenn sich dort der Kopf um eine Position bewegt, wird die gesamte Kodierung ebenfalls um eine Position in die gleiche Richtung umkopiert. Da M mindestens zwei Bänder zur Verfügung hat, kostet sie diese Kopieraktion nur proportional zur Kodierungslänge viel Zeit.

Nach jedem Simulationsschritt und einer evtl. fälligen Umkopieraktion für die Kodierung von M_x schiebt M den Kopf auf ihrem $(k+1)$-ten Band eine Position weiter. Wenn die Zeit dort abgelaufen ist, verwirft M. Ansonsten akzeptiert M genau dann, wenn auch M_x akzeptiert.

Nun konstruieren wir wieder die komplementäre DTM $\overline{M}$. Dies ist kein Problem, da M deterministisch ist. Die Sprache $L := \Sigma^* \setminus L(M)$ liegt wegen der Zeitmessung auf dem $(k+1)$–sten Band offenbar in $\mathrm{DTIME}_{k+1}(g(n))$. Sie liegt aber nicht in $\mathrm{DTIME}_k(f(n))$, was wir nachfolgend durch Widerspruch beweisen.

Sei M' eine $O(f(n))$–zeitbeschränkte DTM mit $L(M') = L$. Sei u die Länge der „eigentlichen“ Kodierung von M', Γ ihr Bandalphabet und Z ihre Zustandsmenge. Sollte also M' durch M bzw. $\overline{M}$ simuliert werden, verursacht jeder Simulationsschritt Zeitkosten, die proportional zu

$$(1 + \log_2 |\Gamma|) + (1 + \log_2 |Z|) + u$$

sind. Für eine passende Konstante c sind dies nichr mehr als

$$c \cdot (2 + \log_2 |\Gamma| + \log_2 |Z| + u)$$

viele Schritte von M bzw. $\overline{M}$. Nun können wir weiter für genügend große n (d.h. bis auf endlich viele Ausnahmen) und eine passende Konstante c' die Laufzeit von M' durch $c' \cdot f(n)$ abschätzen. Also kostet die gesamte Simulation einer Berechnung von M' durch M bzw. $\overline{M}$ nicht mehr als

$$c \cdot c' \cdot (2 + \log_2 |\Gamma| + \log_2 |Z| + u) \cdot f(n)$$

viel Zeit. Nach Lemma 6.3.2 gibt es eine natürliche Zahl $n_0 \in \mathbb{N}$, so dass für alle Zahlen $n \geq n_0$ diese Zeitschranke kleiner oder gleich $g(n)$ ist und die oben genannten Ausnahmen mit berücksichtigt werden.

M' taucht unendlich oft in der Aufzählung der TMs auf. Wir betrachten die erste DTM M_x mit $|x| \geq n_0$, die M' entspricht. Mit dieser Wahl von x kann $\overline{M}$ die Simulation von M_x auf x und damit von M' auf x vollständig durchführen. Da sie aber wie in Satz 6.3.3 das Wort x genau dann akzeptieren wird, wenn M' es verwirft, muss $L(M') \neq L$ gelten. Dies ist ein Widerspruch zur Wahl von M'. □

Das zusätzliche Band von M wird übrigens nicht wirklich benötigt. Mit einer sehr geschickten Konstruktion von Fürer [9] kann nämlich die Zeitmessung auch auf einem der anderen k Bänder erfolgen. Damit erhalten wir das folgende schärfere Ergebnis:

6.3.8 Satz Seien f und g zwei Funktionen mit $f = \Omega(\log n)$ und $f = o(g)$. Sei ferner g konstruierbar. Dann gilt

$$\mathrm{DTIME}_k(f(n)) \subset \mathrm{DTIME}_k(g(n)) \ .$$

Leider funktioniert unser Beweis für nichtdeterministisch zeitbeschränkte Turingmaschinen immer noch nicht, da er wie in den anderen beiden Hierarchiesätzen auf der Existenz einer komplementären TM $\overline{M}$ basiert und uns ein Zeitanalogon zu Satz 6.3.4 fehlt. Aber mit der folgenden interessanten Konstruktion werden wir unser Ziel erreichen, auch wenn die resultierende Hierarchie nicht ganz so dicht wie in allen anderen Fällen ist. Das Resultat geht auf Seiferas, Fischer und Meyer zurück [24].

6.3.9 Satz Seien f und g zwei Funktionen mit $f(n+1) = o(g(n))$. Sei ferner g konstruierbar. Dann gilt

$$\mathrm{NTIME}(f(n)) \subset \mathrm{NTIME}(g(n)) \ .$$

Beweis: Wir konstruieren eine NTM M, die k–bändige NTMs aus einer entsprechenden Aufzählung $M_\varepsilon, M_0, M_1, M_{00}, \ldots$ in ihre Simulationsarbeit mit einbezieht. Nach Satz 6.3.6 dürfen wir diese Aufzählung auf zweibändige NTMs begrenzen. Unsere NTM M wird zwei Bänder für entsprechende Simulationen bereitstellen. Weiter wird sie ein drittes Band wie in Satz 6.3.7 zur Zeitmessung von $g(n)$ vielen Schritten verwenden, wenn sie auf ein Wort der Länge n angesetzt wird.

Wir beweisen die Aussage

$$\mathrm{NTIME}(f(n)) \subset \mathrm{NTIME}(g(n))$$

indirekt, d.h. wir führen die Annahme

$$\mathrm{NTIME}(g(n)) \subseteq \mathrm{NTIME}(f(n))$$

zu einem Widerspruch.

Sei $L \subseteq \{0,1\}^*$ irgendeine Sprache, die von einer k–bändigen DTM M' entschieden wird. M' hält also für jede Eingabe immer an. Wir werden uns am Ende des Beweises eine konkrete Sprache für L aussuchen.

M arbeitet auf dem Eingabealphabet $\Sigma = \{0,1,\#\}^*$. Angesetzt auf ein Wort x der Form $z = x\#y\#^p$ mit $x, y \in \{0,1\}^*$ und $p \in \mathbb{N}_0$ verhält sich M wie folgt:

- Zunächst simuliert M die DTM M' auf der Eingabe y für maximal $g(|z|)$ viele Schritte. Wenn die Zeit für die Simulation bis zum Halt von M' ausreicht, akzeptiert M genau dann, wenn M' akzeptiert.
- Wenn die Zeit nicht ausreicht, simuliert M anschließend die NTM M_x auf dem Wort
$$z\# = x\#y\#^p\# = x\#y\#^{p+1} \ .$$
Die Simulation wird wieder für maximal $g(|z|)$ viele Schritte ausgeführt. Wenn diese Zeit ausreicht, akzeptiert M genau dann, wenn M_x akzeptiert.

- Wenn die Zeit wieder nicht ausreicht, verwirft M.

Für alle y akzeptiert oder verwirft M' das Wort y in endlicher Zeit, da M' immer hält. Also existiert für alle Wörter $x, y \in \{0,1\}^*$ eine kleinste Zahl p_y, so dass M angesetzt auf $x\#y\#^{p_y}$ die erste Simulation von M' angesetzt auf y korrekt zu Ende führt, weil die Zeit $g(|x\#y\#^{p_y}|) = g(|xy| + 1 + p_y)$ dann dafür ausreicht.

Offenbar ist M $O(g(n))$–zeitbeschränkt, da sie maximal zweimal die Zeit $g(n))$ verbraucht. Also erkennt sie eine Sprache $L' \in \mathrm{NTIME}(g(n))$. Wegen unserer Annahme

$$\mathrm{NTIME}(g(n)) \subseteq \mathrm{NTIME}(f(n))$$

existiert eine $O(f(n))$–zeitbeschränkte zweibändige NTM N mit $L(N) = L'$ (siehe Satz 6.3.6). Aufgrund dieser Zeitbeschränkung können wir für genügend große n und eine passende Konstante c die Laufzeit von N durch $c \cdot f(n)$ abschätzen. Insbesondere wird sie angesetzt auf ein Wort $z\#$ der Form $z\# = x\#y\#^{p+1}$ nur $c \cdot f(|z| + 1)$ viel Zeit verbrauchen, wenn allein das Infix y lang genug ist. Wegen $f(n + 1) = o(g(n))$ und Lemma 6.3.2 wird für genügend lange Infixe y dieser Zeitverbrauch kleiner oder gleich $g(|z|)$ sein.

Die NTM N taucht irgendwo als x–te NTM M_x zum ersten Mal in der Aufzählung auf. Ist in einem Wort $z = x\#y\#^p$ dann das Infix y lang genug, die Zahl p aber kleiner als p_y, so wird nach den bisherigen Erkenntnissen M angesetzt auf z die erste Simulation nicht zu Ende führen, die zweite aber schon. Also berechnet M in diesem Fall das Ergebnis von M_x angesetzt auf $z\#$. Aber M_x ist die TM N, und diese berechnet die gleiche Sprache wie M selbst. Also akzeptiert M das Wort $z = x\#y\#^p$ im Fall $p < p_x$ genau dann, wenn sie das Wort $z\# = x\#y\#^{p+1}$ akzeptiert. Im Fall $p = p_y$ berechnet M dagegen das Ergebnis von M' angesetzt auf y, da dann die Zeitschranke für die erste Simulation ausreicht. Also erhalten wir folgenden Zusammenhang für alle $y \in \Sigma^*$:

$$\begin{aligned}
& M \text{ akzeptiert das Wort } x\#y = x\#y\#^0 \\
\iff\ & M \text{ akzeptiert das Wort } x\#y\#^1 \\
\iff\ & M \text{ akzeptiert das Wort } x\#y\#^2 \\
\iff\ & M \text{ akzeptiert das Wort } x\#y\#^3 \\
\iff\ & \ldots \\
\iff\ & M \text{ akzeptiert das Wort } x\#y\#^{p_y} \\
\iff\ & M' \text{ akzeptiert das Wort } y \\
\iff\ & y \in L
\end{aligned}$$

Kurz gesagt akzeptiert also M das Wort $x\#y$ genau dann, wenn y in L enthalten ist. Also kann man leicht eine NTM M_L angeben, die die Sprache L erkennt. Diese ist im Wesentlichen mit M identisch, stellt aber jeder Eingabe $y \in \{0,1\}^*$ zuerst das Präfix $x\#$ voran. Da M eigentlich $O(g(n))$–zeitbeschränkt ist, sich die Eingabe durch die genannte Modifikation aber um $|x\#| = |x|+1$ viele Zeichen verlängert, hat M_L eine nichtdeterministische Zeitkomplexität von $O(g(n + |x| + 1))$. Also liegt L in $\mathrm{NTIME}(g(n + |x| + 1))$.

Doch die obige Konstruktion ist prinzipiell für jede entscheidbare Sprache L möglich. Also liegt *jede* entscheidbare Sprache L in NTIME($g(n + |x| + 1)$). Nach Satz 6.3.5 gilt aber z.B.

$$\text{NSPACE}(g(n + |x| + 1)) \subset \text{NSPACE}(2^{g(n+|x|+1)}) \ ,$$

d.h. es gibt eine Sprache $L \in \text{NSPACE}(2^{g(n+|x|+1)})$, die nicht in NSPACE($g(n+|x|+1)$) und daher wegen Satz 6.2.1 auch nicht in NTIME($g(n + |x| + 1)$) liegt. Aus diesem Widerspruch erkennen wir, dass unsere Annahme

$$\text{NTIME}(g(n)) \subseteq \text{NTIME}(f(n))$$

falsch und somit die Behauptung

$$\text{NTIME}(f(n)) \subset \text{NTIME}(g(n))$$

richtig sein muss. □

Satz 6.3.9 verlangt als Voraussetzung die etwas strengere Annahme $f(n + 1) = o(g(n))$ statt nur $f(n) = o(g(n))$, wie es bei allen vorherigen Hierarchiesätzen der Fall war. In vielen Fällen kann man diesen Unterschied vernachlässigen. Betrachten Sie z.B. den Fall $f(n) := 2^n$ und $g(n) := 2^n \log\log n$. Dann gilt $f = o(g)$, aber auch $f(n + 1) = o(g(n))$ wegen

$$f(n + 1) = 2^{n+1} = 2 \cdot 2^n = o(2^n \log\log n) \ ,$$

so dass wir nicht nur z.B.

$$\text{NSPACE}(2^n) \subset \text{NSPACE}(2^n \log\log n) \ ,$$

sondern auch

$$\text{NTIME}(2^n) \subset \text{NTIME}(2^n \log\log n)$$

folgern können. In den Übungen werden Sie jedoch ein Beispiel präsentiert bekommen, wo sich dieser kleine Unterschied auswirken wird.

6.4. Der Lückensatz von Borodin

Im letzten Abschnitt haben wir gesehen, dass das Hinzufügen von sehr wenig Berechnungsressourcen (z.B. von nur $\log\log\log n$ viel Platz) ausreichend ist, um neue Sprachen zu erkennen. Diesem Ergebnis steht scheinbar das folgende Resultat entgegen. Es behauptet nämlich, dass für jede noch so schnell wachsende Funktion f eine Komplexitätsfunktion g existiert, so dass DSPACE($g(n)$) mit DSPACE($f(g(n))$) zusammen fällt. Wenn beispielsweise $f(n) = 2^n$ gilt, so gibt es eine Komplexitätsfunktion g, so dass ausgehend von DSPACE($g(n)$) keine weitere Sprache erkannt werden kann, obwohl man exponentiell viel mehr Platz bereitstellt. Für die anderen drei Komplexitätsmaße gelten analoge Ergebnisse.

Wir benötigen für den Beweis des Lückensatzes das folgende Lemma.

6.4.1 Lemma Es gibt eine durch eine DTM M realisierbare Testfunktion

$$\texttt{bool platzÜberschreitung(Kodierung } x\texttt{, int } n\texttt{, int } m\texttt{)} \ ,$$

die für die Kodierung x einer DTM M_x und zwei Zahlen $n, m \in \mathbb{N}_0$ in endlicher Zeit entscheidet, ob es ein Wort y der Länge n gibt, so dass M_x während ihrer Berechnung auf der Eingabe y mehr als m viele Felder beansprucht.

Beweis: M testet nacheinander mit dem nachfolgend beschriebenen Verfahren für alle Wörter y der Länge n, ob M_x angesetzt auf y mehr als m Felder benötigt. Ist die Anwort zumindest einmal "ja", so ist auch die in der Behauptung gestellte Frage positiv beantwortet, d.h. M kann bei dem ersten erfolgreichen Test für ein konkretes Wort y den gesamten Test abbrechen und „ja" als Antwort zurückgeben. Ansonsten gibt sie nach der (negativen) Prüfung aller Wörter die Antwort „nein" zurück.

Jeder Einzeltest arbeitet wie folgt. M steckt für ein konkretes Wort y genau m viele Felder auf ihrem Band ab und simuliert anschließend M_x auf der Eingabe y. Sollte M_x während der Simulation versuchen, den abgesteckten Bereich zu verlassen, so kann M die Simulation abbrechen und die Antwort „ja" zurück geben (was wie oben beschrieben zu der Gesamtantwort „ja" führt). Sollte M_x auf der Eingabe y ohne eine Grenzverletzung anhalten, so lautet die Antwort „nein" (dabei ist es egal, ob y akzeptiert oder verworfen wird). Problematisch könnte allerdings eine dritte Situation werden. Es könnte nämlich sein, dass M_x in eine Endlosschleife gerät, ohne jemals den abgesteckten Bereich zu verlassen. Dann lautet die korrekte Antwort in diesem Einzelfall zwar trotzdem „nein", aber wegen der unendlich lange laufenden Simulation würde M erst gar keine Antwort geben. Doch M kann einfach jede auftretende Konfiguration zwischenspeichern. Wegen des begrenzten Platzes sind nur endlich viele verschiedene Konfigurationen möglich. Wenn zum ersten Mal eine Konfiguration doppelt auftritt, wird die simulierte DTM M_x die dazwischen liegenden Konfigurationen in einer Schleife immer wieder durchlaufen und niemals mehr anhalten. Also kann M eine solche Situation in endlicher Zeit erkennen, die Simulation für diesen Einzeltest abbrechen und die richtige Antwort "nein" zurückgeben. □

Nun können wir den Lückensatz von Borodin [6] beweisen:

6.4.2 Satz Sei $f : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ eine total berechenbare Funktion, d.h. wir können f z.B. mit einem C–Programm oder auch einer DTM ermitteln, und f ist für alle Argumente definiert. Ferner erfülle f die Bedingung $f(n) \geq n$ für alle $n \in \mathbb{N}_0$. Dann existiert eine total berechenbare Funktion $g : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ mit

$$\mathrm{DSPACE}(g(n)) = \mathrm{DSPACE}(f(g(n))) \ .$$

Also existiert zwischen den Komplexitätsklassen $\mathrm{DSPACE}(g(n))$ und $\mathrm{DSPACE}(f(g(n)))$ eine „Lücke", d.h. es gibt trotz des evtl. enormen Unterschieds zwischen dem Wachstumsverhalten von $g(n)$ und $f(g(n))$ keine einzige Sprache, die in $\mathrm{DSPACE}(f(g(n)))$, aber nicht zugleich auch in $\mathrm{DSPACE}(g(n))$ liegt.

Beweis: Der Programmcode für die Berechnung der Funktion $g : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ sieht wie folgt aus (eine Erklärung erfolgt im Anschluss):

```
int g(int n) {
   int m := 1;
   int m';
   repeat {
      m' := m;
      for jedes Wort x ∈ {0,1}* mit |x| ≤ n do {
         if platzÜberschreitung(x, n, m) then {
            if not platzÜberschreitung(x, n, f(m)) then {
               m := f(m);
            }
         }
      }
   } until m = m';
   return m;
}
```

Die Routine prüft, mit welcher minimalen Platzkomplexität Wörter der Länge n von bestimmten DTMs beaebeitet werden. Konkret werden alle DTMs M_x, deren Kodierungslänge $|x|$ kleiner oder gleich n ist, diesem Test unterzogen. Ziel ist es, eine Zahl $m \geq 1$ zu finden, so dass Wörter der Länge n mit einer minimalen Platzkomplexität p bearbeitet werden, die (für jede DTM M_x individuell) entweder $p \leq m$ oder $p > f(m)$ erfüllt. Zur Erreichung dieses Ziels wird innerhalb der `for`–Schleife mit den zwei ineinander geschachtelten `if`–Abfragen geprüft, ob für M_x das Gegenteil der Fall ist. Konkret prüft die erste Abfrage, ob es mindestens ein Wort der Länge n gibt, welches mehr als m viele Felder erfordert. Die zweite Abfrage stellt sicher, dass kein solches Wort mehr als $f(m)$ viele Felder benutzt. In diesem Fall liegt die Platzkomplexität für Wörter der Länge n also zwischen $m + 1$ und $f(m)$, und m durch $f(m)$ ersetzt. Hierdurch wird m größer, denn zum einen gilt nach Voraussetzung $f(m) \geq m$, und zum anderen kann der Fall $f(m) = m$ ausgeschlossen werden, da sich ansonsten die beiden `if`–Abfragen widersprochen hätten. Da sich die Routine den alten Wert von m in der Variable m' gemerkt hat, wird später die `repeat ... until`–Schleife nochmals durchlaufen, so dass alle DTMs einem weiteren Test unterzogen werden. Insbesondere wird also auch jede DTM M_x nochmals geprüft, die eine Erhöhung von m verursacht hat.

Beachten Sie, dass jede der endlich vielen überprüften DTMs M_x für Wörter der Länge n eine bestimmte minimale (im Zweifelsfall unendlich große) Platzkomplexität p besitzt. Sollte eine solche DTM für eine Erhöhung von m verantwortlich sein, also $m < p \leq f(m)$ gelten, so wird dies für den gleichen Wert von n und der gleichen Turingmaschine M_x nicht noch einmal passieren. Durch die Erhöhung von m auf den neuen Wert $f(m)$ wird nämlich die betroffene minimale Platzkomplexität zukünftig kleiner oder gleich m sein, so dass für diese konkrete Situation der Test ab jetzt bestanden wird. Also nimmt die

Anzahl der möglichen Problemfälle mit jeder Erhöhung von m ab. Da es nur endlich viele solche Fälle gibt, werden am Ende für ein konkretes n alle Turingmaschinen den Test bestehen und die Methode terminiert. Dies zeigt, dass g eine totale und berechenbare Funktion ist.

Nun können wir die Aussage $\mathrm{DSPACE}(g(n)) = \mathrm{DSPACE}(f(g(n)))$ beweisen. Die eine Inklusion $\mathrm{DSPACE}(g(n)) \subseteq \mathrm{DSPACE}(f(g(n)))$ ist wegen der Voraussetzung $f(n) \geq n$ offensichtlich. Sei also nun $L \in \mathrm{DSPACE}(f(g(n)))$ beliebig gewählt und M eine passende zugehörige DTM mit $L(M) = L$, die $O(f(g(n)))$–platzbeschränkt ist. Nach Satz 6.1.1 können wir annehmen, dass für eine bestimmte Konstante $n_0 \in \mathbb{N}$ alle Berechnungen für Wörter mit mindestens $n \geq n_0$ Zeichen nicht mehr als $f(g(n))$ viel Band verbrauchen. M taucht irgendwo als x–te DTM zum ersten Mal in der Aufzählung aller DTMs auf. Sei $n_0' := |x|$ die Länge der Kodierung von M_x. Dann wird M_x und damit M für alle Wörter der Länge $n \geq n_0'$ in die Tests bei der Berechnung von $g(n)$ mit einbezogen. Also gilt für alle Wörter $y \in \{0,1\}^n$ mit $n \geq n_0'$ nach Konstruktion von $g(n)$ folgende Bedingung:

- Entweder benötigen alle Berechnungen höchstens $g(n)$ viel Platz oder
- es gibt mindestens ein solches Wort, dessen Berechnung mehr als $f(g(n))$ viel Platz erfordert.

Doch für alle Wörter $y \in \{0,1\}^n$ mit $n \geq \max\{n_0, n_0'\}$ ist der zweite Fall unmöglich, also werden bis auf endlich viele Ausnahmen (die nur Wörter mit einer Länge kleiner als $\max\{n_0, n_0'\}$ betreffen können) alle Wörter der Länge n mit höchstens $g(n)$ viel Platz akzeptiert. Folglich gilt sogar $L \in \mathrm{DSPACE}(g(n))$.

Es sei noch bemerkt, dass man $g = \Omega(h)$ für irgendeine berechenbare Funktion h garantieren kann, indem man in der ersten Zeile des Programmcodes für die Funktion `g` die Variable m nicht mit dem Wert 1, sondern mit $h(n)$ initialisiert. □

Wie passt dieses Ergebnis mit Satz 6.3.3 zusammen? Nehmen Sie z.B. an, wie hätten als Funktion $f(n) := n^2$ ausgewählt. Dann kann die laut Satz 6.4.2 existierende Funktion $g : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ nicht z.B. $g(n) := n^3$ erfüllen, denn dann würde

$$\mathrm{DSPACE}(n^3) = \mathrm{DSPACE}(g(n)) = \mathrm{DSPACE}(f(g(n))) = \mathrm{DSPACE}(n^6)$$

folgen, was Satz 6.3.3 widerspricht. Genauer gesagt kann überhaupt keine *konstruierbare* Funktion aus $\Omega(\log n)$ für g in Frage kommen, weil ansonsten offenbar auch $f(g(n)) = g^3(n)$ konstruierbar wäre (siehe Übungen) und wir den gleichen Widerspruch erhalten würden. Andererseits ist g jedoch problemlos *berechenbar*. Somit kommen wir zwangsläufig zu folgendem Schluss:

6.4.3 Korollar Es gibt berechenbare, aber nicht konstruierbare Funktionen.

6.4.4 Satz Der Lückensatz 6.4.2 gilt analog auch für die anderen drei Komplexitätsmaße NSPACE, DTIME und NTIME.

Beweis: Die Beweisidee ist identisch. Der Unterschied besteht nur in einer anderen Ausführung der Routine

$$\texttt{bool platzÜberschreitung(Kodierung } x\texttt{, int } n\texttt{, int } m\texttt{)} \ ,$$

die entsprechend modifiziert werden muss. Im Fall NSPACE kann es z.B. viel mehr zu simulierende Berechnungspfade geben. Da es aber wegen des begrenzten Platzes trotzdem nur endlich viele sind, können wir wie zuvor alle Möglichkeiten durchprobieren und den verbrauchten Platz messen.

Für die beiden Zeitkomplexitätsklassen ist eine entsprechende Routine

$$\texttt{bool zeitÜberschreitung(Kodierung } x\texttt{, int } n\texttt{, int } m\texttt{)}$$

sogar noch einfacher zu programmieren, da sie nur prüfen muss, ob jede mögliche Berechnung nach maximal m Schritten anhält. Das Problem mit evtl. vorhandenen Endlosschleifen stellt sich hier also nicht. □

Der Lückensatz von Borodin hat interessante Folgerungen, z.B. diese hier:

6.4.5 Korollar Es gibt eine total berechenbare Funktion $g : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ mit

$$\mathrm{DSPACE}(g(n)) = \mathrm{NSPACE}(g(n)) \ .$$

Beweis: Mit der Wahl $f(n) := n^2$ garantiert uns Satz 6.4.2 eine total berechenbare Funktion g mit

$$\mathrm{DSPACE}(g(n)) = \mathrm{DSPACE}(g^2(n)) \ ,$$

wobei wir gemäß der abschließenden Bemerkung im Beweis $g = \Omega(\log n)$ annehmen dürfen. Doch nach Satz 6.2.1 und Korollar 6.2.5 gilt

$$\mathrm{DSPACE}(h(n)) \subseteq \mathrm{NSPACE}(h(n)) \subseteq \mathrm{DSPACE}(h^2(n))$$

für alle Funktionen $h = \Omega(\log n)$, was in diesem Spezialfall

$$\mathrm{DSPACE}(g(n)) \subseteq \mathrm{NSPACE}(g(n)) \subseteq \mathrm{DSPACE}(g^2(n)) = \mathrm{DSPACE}(g(n))$$

bedeutet, also insbesondere

$$\mathrm{DSPACE}(g(n)) = \mathrm{NSPACE}(g(n))$$

und damit die Behauptung. □

In den Übungen werden Sie ganz ähnlich sogar die Existenz einer total berechenbaren Funktion $g : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ mit

$$\mathrm{DTIME}(g(n)) = \mathrm{NTIME}(g(n)) = \mathrm{DSPACE}(g(n)) = \mathrm{NSPACE}(g(n))$$

nachweisen, bei der also sogar alle vier Komplexitätsklassen zusammenfallen.

6.5. Der Blum'sche Beschleunigungssatz

Abschließend besprechen wir noch eine weitere scheinbar „pathologische“ Eigenschaft unserer vier Komplexitätsmaße, welche von BLUM bereits 1967 nachgewiesen wurde [3]. Es gibt nämlich Sprachen, die sich beliebig schnell bzw. mit beliebig wenig Speicherplatz erkennen lassen, d.h. man kann keine optimalen Programme angeben. Wenn man also für eine solche Sprache L eine passende Turingmaschine M mit $L(M) = L$ gefunden hat, die z.B. $O(f(n))$ viel Zeit verbraucht, so existiert auch eine Turingmaschine, die z.B. nur $O(\sqrt{f(n)})$ viel Zeit benötigt. Aber auch diese lässt sich noch weiter beschleunigen, usw.

Wir werden wegen des erforderlichen technischen Aufwands dieses Phänomen nur für ein Komplexitätsmaß beweisen und wählen dafür wieder den Fall DSPACE aus. Die grundlegende Idee ist jedoch wie beim Lückensatz in allen vier Fällen dieselbe.

Zunächst müssen wir unsere Kenntnisse über die Testfunktion

`bool platzÜberschreitung(Kodierung` x`, int` n`, int` m`)`

aus Lemma 6.4.1 etwas verfeinern. Wir benötigen insbesondere eine Analyse des benötigten Platzes. Dieser wird auch von der Kodierungslänge $|x|$ der jeweils analysierten Turingmaschine M_x abhängen, die nach Satz 4.3.2 ohne Einschränkung einer Aufzählung $M_\varepsilon, M_0, M_1, M_{00}, \ldots$ einbändiger DTMs über dem Bandalphabet $\{0, 1, \square\}$ entstammt. Weiterhin ersetzen wir den dritten Parameter m durch einen von n abhängigen Funktionswert, um dann anschließend die Platzkomplexität dieser Funktion in Abhängigkeit von n zu messen. Zur Optimierung des benötigten Platzbedarfs müssen wir zudem die in Lemma 6.4.1 vorgeschlagene Implementierung noch etwas modifizieren.

6.5.1 Lemma Sei f eine konstruierbare Funktion mit $f = \Omega(\log n)$. Dann existiert ein durch eine DTM M realisierbares Prüfverfahren mit diesen Eigenschaften:

- Angesetzt auf die Kodierung x einer DTM M_x und eine Zahl $n \in \mathbb{N}_0$ wird überprüft, ob es ein Wort y der Länge n gibt, so dass M_x während ihrer Berechnung auf der Eingabe y mehr als $f(n)$ viele Felder beansprucht.
- M benötigt für die Berechnung der Antwort höchstens $O(f(n)) + |x|$ viele Bandfelder.

Diesen Test werden wir während des Beweises des Beschleunigungssatzes wieder durch einen Aufruf der Form `platzÜberschreitung(`x`, `n`, `$f(n)$`)` referenzieren.

Beweis: M testet wie in Lemma 6.4.1 für alle Wörter y der Länge n, ob M_x angesetzt auf y mehr als $f(n)$ Felder benötigt. Der dafür benötigte Platz kann für jeden Einzeltest wiederverwendet werden, d.h. es kommt darauf an, mit möglichst wenig Platz beim Einzeltest auszukommen.

Für ein konkretes Wort y der Länge n steckt M wie gehabt den zu testenden Platzbedarf (hier also $f(n)$ viele Felder) auf ihrem Band ab und simuliert anschließend M_x auf der Eingabe y. Wie zuvor bricht M bei einer Überschreitung der Platzgrenze durch M_x die

Simulation ab und gibt „ja“ zurück. Ebenso antwortet sie „nein“, wenn M_x ohne eine Grenzverletzung anhält. Nur zur Erkennung von Endlosschleifen geht M anders als in Lemma 6.4.1 vorgeschlagen vor, weil das Abspeichern aller durchlaufenen Konfigurationen zu teuer ist. Stattdessen zählt M wieder die simulierten Schritte von M_x mit. In Anlehnung an die in Lemma 6.2.2 vorgebrachten Argumente gibt es wegen dem dreielementigen Bandalphabet nur $3^{f(n)}$ viele Bandinhalte, $f(n)$ viele Kopfpositionen auf dem Arbeitsband, $n+2$ Kopfpositionen auf dem Eingabeband sowie $|Z|$ Zustände, wobei Z die Zustandsmenge von M_x bezeichnet. Sicherlich können wir die Anzahl $|Z|$ durch $2^{|x|}$ abschätzen, da insbesondere die gesamte Zustandsmenge von M_x irgendwie innerhalb von x kodiert sein muss und eine größere Zahl als $2^{|x|}$ binär gar nicht gespeichert werden kann. Also gibt es nur höchstens $3^{f(n)} \cdot f(n) \cdot (n+2) \cdot 2^{|x|}$ viele Konfigurationen, die wir wegen $f = \Omega(\log n)$ mit einem binären Zähler der Länge

$$\begin{aligned} &\log_2 \left(3^{f(n)} \cdot f(n) \cdot (n+2) \cdot 2^{|x|}\right) \\ &= \log_2 3 \cdot f(n) + \log_2(f(n)) + \log_2(n+2) + |x| = O(f(n)) + |x| \end{aligned}$$

überwachen können. Wenn M einen solchen Zähler bereitstellt und bei jedem simulierten Schritt von M_x um eins erhöht, so erkennt sie bei einem Überlauf, dass M_x eine Konfiguration doppelt betreten haben muss und deshalb nie mehr anhalten wird. Also kann M wie zuvor diese Situation erkennen und die Simulation mit der korrekten Antwort „nein“ abbrechen.

Welchen Speicherbedarf hat die eigentliche Simulation an sich? Da M selbst mit dem gleichen dreielementigen Bandalpahbet $\{0, 1, \square\}$ arbeitet, benötigt eine simulierte Konfiguration von M_x nur

$$f(n) + \log_2(f(n)) + \log_2(n+2) + |x| = O(f(n)) + |x|$$

viel Platz, nämlich für den Inhalt, die binäre Kodierung der Kopfpositionen auf dem Arbeits– und dem Eingabeband sowie für die Kodierung des aktuellen Zustands. Also benötigt M für den Zähler und die eigentliche Simulation insgesamt nur $O(f(n)) + |x|$ viel Platz. □

6.5.2 Satz Es gibt eine Sprache $L \subseteq \{0\}^*$, so dass für jede L akzeptierende DTM M_0 mit einer Platzkomplexität $O(f_0(n))$ eine weitere DTM M_1 mit einer Platzkomplexität von nur $O(\sqrt{f_0(n)})$ existiert. Bezeichnen wir diese Komplexität mit $f_1(n) := \sqrt{f_0(n)}$, so lässt sich eine noch bessere DTM M_2 mit Platzkomplexität $O(\sqrt{f_1(n)}) = O(\sqrt[4]{f_0(n)})$ nachweisen, usw., so dass sich eine unendlich lange Folge von DTMs

$$M_0, M_1, M_2, M_3, M_4, \ldots$$

ergibt. Jede DTM M_r besitzt die Platzkomplexität $O(f_r(n))$ mit

$$f_r(n) := \sqrt{f_{r-1}(n)} \qquad \left(\text{also } f_r(n) = \sqrt[2^r]{f_0(n)}\right)$$

und ist damit platzeffizienter als alle ihre Vorgängermaschinen, aber nicht so effizient wie ihre weiteren Nachfolger.

Beweis: Sei $M_\varepsilon, M_0, M_1, M_{00}, \ldots$ eine Aufzählung aller einbändigen DTMs über dem Bandalphabet $\{0, 1, \square\}$. Unser erstes Ziel ist es, für jede Turingmaschine M_x sicherzustellen, dass sie

- entweder die später erzeugte Sprache L nicht akzeptiert, also $L \neq L(M_x)$ gilt, oder
- eine Platzkomplexität von mindestens $2^{2^{n-|x|}} + 1$ für fast alle (d.h. bis auf endlich viele) $n \in \mathbb{N}_0$ besitzt.

Die folgende Prozedur zur Erzeugung der Sprache L realisiert dieses Ziel auf eine naheliegende Art und Weise. Sie prüft wiederholt nach und nach für jede Turingmaschine M_x, ob sie für ein konkretes n mit $2^{2^{n-|x|}}$ viel Platz auskommt. Falls ja, so könnte M_x diese Schranke evtl. auch für alle größeren n erfüllen und daher möglicherweise die zweite Bedingung verletzen. Sicherheitshalber wird deshalb dann die erste Bedingung sichergestellt. Hierfür wird ein bestimmtes Wort in der Sprache L genau dann platziert, wenn M_x dieses nicht akzeptiert. Also wird M_x — egal wie sich die restliche Sprache weiter entwickelt — L nicht mehr korrekt erkennen. Ihre Kodierung x wird zusätzlich in einer Menge T aufgenommen. Jede dort enthaltenen Turingmaschine braucht nicht weiter betrachtet zu werden, da sie die erste Bedingung erfüllt.

Für die Entscheidung, ob das Wort 0^n in die Sprache L mit aufgenommen werden soll, werden nacheinander alle DTMs M_x mit der Kodierungslänge $|x| = 0, 1, 2, \ldots, n$ diesem Test unterworfen. Die aktuelle Kodierungslänge $|x|$ ist in der Variablen m gespeichert.

```
T := ∅;
int n := 0;
while (true) {     // Endlosschleife
   int m := 0;
   repeat {
      for jedes Wort x ∈ {0,1}* mit |x| = m do {
         if x ∉ T then {
            if not platzÜberschreitung(x, n, 2^(2^(n-|x|))) then {
               T := T ∪ {x};
               if 0^n ∉ L(M_x) then L := L ∪ {0^n};
               m := n;        // hiermit werden beide
               break for;     // inneren Schleifen verlassen
            }
         }
      }
      m := m + 1;
   } until m > n;
   n := n + 1;
}
```

Wir zeigen nun, dass die hiermit erzeugte Sprache L wie gewünscht die obigen Anforderungen an alle DTMs realisiert. Sei dazu M_y irgendeine DTM aus der Aufzählung. Wir nehmen an, dass M_y eine Bandkomplexität von höchstens $2^{2^{n-|y|}}$ für unendlich viele Zahlen $n = n_0, n_1, n_2, \ldots$ besitzt, ansonsten würde M_y die zweite Bedingung erfüllen und es wäre nichts mehr zu beweisen. Für diese Zahlen n und dem Wort $x = y$ würde der in der Mitte des Programms befindliche Test

$$\texttt{platzÜberschreitung(}x\texttt{, }n\texttt{, }2^{2^{n-|x|}}\texttt{)}$$

also die Antwort „nein“ zurückgeben. Sollte also dieser Test während des Programmablaufs mit genau dieser Kodierung y und einem passenden n ausgeführt werden, so wird anschließend das Wort 0^n genau dann in L mit aufgenommen, wenn es von M_y verworfen wird. Da anschließend beide innere Schleifen verlassen werden, wird n anschließend erhöht und niemals mehr den gleichen Wert erhalten. Also wird M_y die Sprache L wegen dem Wort 0^n nicht korrekt erkennen und damit die erste Bedingung erfüllen, ganz gleich, welche anderen (längeren) Wörter noch nach L gelangen oder nicht.

Zur Erreichnung des ersten Ziels müssen wir also nur noch zeigen, dass der besagte Test für M_y wirklich einmal mit den passenden Parametern zur Ausführung kommt. Wir warten zunächst mehrere Iterationen der äußeren Endlosschleife ab, bis die Variable n den Wert $|y|$ erreicht. Ab dann wird M_y in die Tests mit einbezogen, wobei allerdings Turingmaschinen mit einer kürzeren Kodierung eine höhere Testpriorität genießen. Konkret werden immer erst alle Turingmaschinen M_x mit einer Kodierung der Länge $m = 0$ getestet (davon gibt es nur eine, nämlich M_ε), danach alle DTMs der Länge 1 (davon gibt es nur zwei, nämlich M_0 und M_1), danach alle der Länge 2 (davon gibt es nur vier), usw. Also gibt es nur endlich viele Turingmaschinen M_x, die vor M_y geprüft werden. Jede davon kann zwar selbst die Situation verursachen, dass sie den Platztest besteht und deshalb wegen ihr die beiden inneren Schleifen verlassen werden, noch bevor M_y zum Zuge kommt. Vorher wird jedoch die zugehörige Kodierung x in der Menge T mitprotokolliert. Also wird M_x nicht noch einmal einen vorzeitigen Abbruch verursachen, da alle Turingmaschinen M_x mit $x \in T$ von dem Testverfahren ausgeschlossen sind. Irgendwann werden alle solchen vor M_y liegenden Turingmaschinen M_x abgearbeitet und von dem Testverfahren ausgeschlossen sein, sofern deren Kodierung überhaupt jemals nach T gelangt. Sei n der Wert der gleichnamigen Variable zu diesem Zeitpunkt. Spätestens ab dann wird der Test auch immer zu M_y vorstoßen. Sei $n_i \geq n$ die nächstgrößere der obigen Zahlen. Sobald die Variable n diesen Wert n_i erreicht, wird der Test die zu geringe Platzkomplexität von M_y entdecken und diese von der Akzeptanz von L ausschließen. Also wird irgendwann die erste Bedingung für M_y erfüllt sein.

Als Nächstes überzeugen wir uns, dass für alle $k \in \mathbb{N}_0$ eine DTM für L mit Bandkomplexität $O(2^{2^{n-k}})$ existiert. Dazu betrachten wir die Menge aller DTMs M_x, die eine Kodierung der Länge $|x| < k$ besitzen und die früher oder später in die Menge T aufgenommen werden. Sei T_k die Menge aller zugehörigen Kodierungen. Nun starten wir in Gedanken das obige Programm und warten ab, bis $T_k \subseteq T$ gilt, d.h. bis zum letzten Mal eine Kodierung $x \in T_k$ zu T hinzugefügt wurde. Die Konstruktion von T ist dann

für alle Kodierungen x mit einer Länge von weniger als k komplett. Seien nun weiter L_k und n_k die Inhalte von der Menge L und der Variablen n zu diesem Zeitpunkt. Die spätere Sprache L wird nur Wörter aus L_k sowie Wörter mit einer Länge größer als n enthalten, da nur noch solche längeren Wörter zu L hinzugefügt werden. Also können wir mit Hilfe der Größen T_k, L_k und n_k das obige Programm wie folgt modifizieren, um zu prüfen, ob ein Wort $y \in \{0\}^*$ in L enthalten ist oder nicht. Die Idee ist, dass wir

- für alle Wörter y mit einer Länge $|y| \leq n_k$ einfach nur die Menge L_k befragen und
- für alle längeren Wörter die obige Konstruktion nur für $n = n_k + 1, n_k + 2, \ldots, |y|$ durchführen, da für alle $n \leq n_k$ die Ergebnisse bereits vorausberechnet vorliegen.

Die charakteristische Funktion χ_L von L lässt sich demnach wie auf der nächsten Seite dargestellt realisieren. Die vorausberechneten Größen T_k, L_k und n_k gehen als (große) Konstanten zwar in die Länge des Programmcodes ein, haben aber keine Auswirkung auf die nun analysierte Platzkomplexität.

Sei y das zu prüfende Wort, mit dem die Funktion χ_L aufgerufen wird. Dann hat der dritte Parameter bei jedem Aufruf der Testfunktion `platzÜberschreitung(`x`, `n`, `$2^{2^{n-|x|}}$`)` wegen $|x| = m \geq k$ und $n \leq |y|$ höchstens den Wert $2^{2^{|y|-k}}$. Nach Lemma 6.5.1 benötigt jeder Test somit höchstens $O(2^{2^{|y|-k}}) + |x|$ viel Platz. Nun ist die Kodierung x jeder getesteten Turingmaschine höchstens so lang wie das Eingabewort y, da sich aus den Schleifengrenzen die Abschätzung $|x| = m \leq n \leq |y|$ ergibt. Also vereinfacht sich die genannte Platzkomplexität von jedem Test zu

$$O(2^{2^{|y|-k}}) + |x| = O(2^{2^{|y|-k}} + |y|) = O(2^{2^{|y|-k}}) \ ,$$

da die doppelte Exponentialfunktion $f(|y|) := 2^{2^{|y|-k}}$ viel schneller als $|y|$ wächst. Die gleiche Platzkomplexität gilt dann auch für alle insgesamt im Programm ausgeführten Tests, da wie jedesmal den gleichen Platz wieder verwenden können.

Die charakteristische Funktion muss sich auch noch die Menge T aller gespeicherten Kodierungen merken. Der dafür benötigte Speicherplatz fällt jedoch kaum ins Gewicht. Wenn nämlich T um eine Kodierung x erweitert wird, dann hat diese gerade die Länge m, wobei sich m immer im Bereich zwischen k und $n \leq |y|$ bewegt. Es gibt höchstens 2^m Kodierungen der Länge m, d.h. T benötigt im schlimmsten Fall Platz für

$$\sum_{m=k}^{|y|} 2^m \leq \sum_{m=0}^{|y|} 2^m = 2^{|y|+1} - 1$$

Kodierungen, wobei jede Kodierung aus höchstens $|y|$ vielen Zeichen besteht. Also reicht zur Speicherung von T

$$O(|y| \cdot (2^{|y|+1} - 1)) = O((|y| + 1) \cdot 2^{|y|+1}) = O(2^{|y|+1} \cdot 2^{|y|+1}) = O(4^{|y|+1})$$

viel Platz aus. Da jede Exponentialfunktion langsamer als eine doppelte Exponentialfunktion wächst, benötigt die Routine insgesamt nicht mehr als

```
bool χ_L(Eingabewort y) {
   if |y| ≤ n_k then {
      if y ∈ L_k then {
         return true;
      } else {
         return false;
      }
   }
   T := ∅;
   int n;
   for n := n_k + 1 to |y| do {
      int m := k;
      repeat {
         for jedes Wort x ∈ {0,1}* mit |x| = m do {
            if x ∉ T then {
               if not platzÜberschreitung(x, n, 2^(2^(n-|x|))) then {
                  T := T ∪ {x};
                  if n = |y| then {
                     if 0^n ∉ L(M_x) then {
                        return true;
                     } else {
                        return false;
                     }
                  }
                  m := n;        // hiermit werden beide
                  break for;     // inneren Schleifen verlassen
               }
            }
         }
         m := m + 1;
      } until m > n;
   }
}
```

$$O(2^{2^{|y|-k}}) + O(4^{|y|+1}) = O(2^{2^{|y|-k}})$$

viel Platz. Also gibt es für alle $k \in \mathbb{N}_0$ eine Implementierung der charakteristischen Funktion χ_L durch ein Turingprogramm, welches für Eingaben der Länge n eine Platzkomplexität von $O(2^{2^{n-k}})$ besitzt.

Nach diesen Vorarbeiten können wir den Beweis des Beschleunigungssatzes überraschend einfach abschließen. Sei M_0 irgendeine DTM mit $L(M_0) = L$. Sie taucht irgendwo zum ersten Mal als x–te Turingmaschine M_x in unserer Aufzählung aller DTMs auf. Die Bandkomplexität von M_x beträgt bestenfalls $O(2^{2^{n-|x|}})$, da es gemäß der zweiten Bedin-

gung unendlich viele Zahlen n gibt, so dass M_x für je mindestens ein Wort der Länge n mindestens

$$2^{2^{n-|x|}} + 1 = \Omega(2^{2^{n-|x|}})$$

viele Felder benötigt. Doch wie soeben gesehen kann man L für alle k auch durch ein passendes Turingprogramm mit Platzkomplexität $O(2^{2^{n-k}})$ entscheiden. Setzen wir $f_0(n) := 2^{2^{n-|x|}}$, so garantiert uns die Wahl von $k = |x|+1, |x|+2, |x|+3, \ldots$ die Existenz von Turingmaschinen $M_1, M_2, M_3, \ldots$, so dass die r–te DTM M_r eine Platzkomplexität von

$$O(2^{2^{n-(|x|+r)}}) = O(2^{2^{n-|x|}/2^r}) = O(\sqrt[2^r]{f_0(n)}) = O(f_r(n))$$

besitzt. Damit ist alles bewiesen. □

A

Beweisführungen durch vollständige Induktion

A.1. Das Prinzip der vollständigen Induktion

Eine wichtige Beweistechnik wurde in dem Abschnitt 1.4 aus Zeitgründen nicht behandelt, nämlich die der sogenannten *vollständigen Induktion*. Sie eignet sich zum Beweis von vielen Behauptungen über natürliche Zahlen oder einen (unendlichen) Teilbereich davon. Betrachten Sie z.B. folgende Aussage. Sie besagt, dass die Summe der ersten n ungeraden Zahlen gleich der n–ten Quadratzahl ist:

$$1 + 3 + 5 + \cdots + (2 \cdot n - 1) = n^2 \ .$$

Abkürzend kann man diese Gleichung auch mit Hilfe der Summendarstellung

$$\sum_{i=1}^{n} (2i - 1) = n^2$$

notieren. Hierbei ist i wieder ein Index, der alle ganzen Zahlen zwischen den beiden Grenzen durchläuft, die unter und über dem großen Sigma–Symbol angegeben sind (hier also 1 und n). Der jeweilige Wert wird für alle Vorkommen von i in dem Term hinter dem Sigma–Symbol eingesetzt, und die resultierenden Ergebnisse werden aufaddiert. Hier ergibt sich also wie gewünscht die Summe aus den Werten $2 \cdot 1 - 1 = 1, 2 \cdot 2 - 1 = 3$, usw.

Wir können die Behauptung durch direktes Nachrechnen probeweise überprüfen:

$$n = 1: \quad \sum_{i=1}^{1} (2i - 1) = 1 = 1^2$$

$$n = 2: \quad \sum_{i=1}^{2} (2i - 1) = 1 + 3 = 4 = 2^2$$

$$n = 3: \quad \sum_{i=1}^{3} (2i - 1) = 1 + 3 + 5 = 9 = 3^2$$

$$
\begin{aligned}
n = 4: &\quad \sum_{i=1}^{4}(2i-1) = 1+3+5+7 = 16 = 4^2 \\
n = 5: &\quad \sum_{i=1}^{5}(2i-1) = 1+3+5+7+9 = 25 = 5^2 \\
n = 6: &\quad \sum_{i=1}^{6}(2i-1) = 1+3+5+7+9+11 = 36 = 6^2 \\
&\quad \dots
\end{aligned}
$$

Die Formel scheint also richtig zu sein. Die obigen „Tests“ sind aber kein Beweis, denn vielleicht stimmt ja die Aussage für $n = 7$ schon nicht mehr. Andererseits können wir natürlich auch nur endlich viele Tests vornehmen. Die obige Behauptung besteht aber in Wirklichkeit aus unendlich vielen Behauptungen, nämlich je einer Aussage für jede konkrete natürliche Zahl n. Durch direktes Nachrechnen kann der obige Satz also nicht gezeigt werden.

Die Beweistechnik der vollständigen Induktion besteht aus zwei separat zu beweisenden Teilen:

- Als erstes ist die Behauptung für $n = 1$ zu beweisen. Diesen Schritt nennt man die sogenannte *Induktionsbasis* oder *Induktionsverankerung*, manchmal auch den *Induktionsanfang*. Meistens ist dieser Schritt sehr einfach.
- Anschließend ist der *Induktionsschritt* durchzuführen. Für alle $n \in \mathbb{N}$ ist hierfür zu zeigen: falls die Behauptung bereits für n wahr ist (die sogenannte *Induktionsannahme* bzw. *Induktionshypothese*), so ist sie auch für den Fall $n + 1$ richtig.

Hat man beide Teile gezeigt, so ist die Aussage für alle $n \in \mathbb{N}$ gültig. Denn zunächst einmal gilt sie für $n = 1$, was man in der Induktionsverankerung nachgewiesen hat. Da die Aussage für $n = 1$ gilt, gilt sie durch Ausnutzung des Induktionsschritts dann auch für den Fall $n = 1 + 1 = 2$. Dann aber gilt sie auch für den Fall $n = 3$ durch erneute Ausnutzung des Induktionsschritts. Genauso kann man nun weiter für $n = 4, 5, 6, \dots$ argumentieren, d.h. die Aussage gilt für alle $n \in \mathbb{N}$.

Wir wenden diese Technik jetzt für das obige Beispiel an.

Induktionsbasis: Für $n = 1$ war, wie bereits gesehen, die Aussage richtig ($1 = 1^2$).

Induktionsschritt: n sei nun eine beliebige natürliche Zahl. Wir können aufgrund der Induktionshypothese annehmen, dass die Aussage für diese konkrete Zahl n bereits bewiesen ist. Es gilt also:

$$\sum_{i=1}^{n}(2i-1) = n^2 \ .$$

Wir müssen zeigen, dass sie dann auch für den Fall $n + 1$ stimmt. Hierfür ersetzen wir überall das Vorkommen von „n“ durch „$n + 1$“, also

$$\sum_{i=1}^{n+1}(2i-1) = (n+1)^2 \ .$$

Zum Beweis dieser Aussage kann man nun die Induktionshypothese geschickt ausnutzen. Es gilt nämlich

$$\begin{aligned}
&\sum_{i=1}^{n+1}(2i-1)\\
=\ &\Big(\sum_{i=1}^{n}(2i-1)\Big)+(2\cdot(n+1)-1)\\
=\ &n^2+(2\cdot(n+1)-1) \qquad \text{(Verwendung der Induktionshypothese)}\\
=\ &n^2+2n+2-1\\
=\ &n^2+2n+1\\
=\ &(n+1)^2
\end{aligned}$$

Der induktive Beweis ist damit vollständig geführt. □

Man kann beim Induktionsstart auch eine andere Basis als $n = 1$ wählen, z.B. $n = 0$. Dann gilt die Aussage entsprechend für alle $n \in \mathbb{N}_0$ statt nur für $n \in \mathbb{N}$. Genauso darf man beim Induktionschritt für den Fall $n+1$ nicht nur den Fall n als Induktionshypothese zurückgreifen, sondern auch alle vorherigen Fälle $n-1, n-2, \ldots, 1$, die ja ebenfalls nach dem obigen Prinzip bereits wahr sein müssen.

A.1.1 Beispiel Wir zeigen durch vollständige Induktion über $n \in \mathbb{N}_0$:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \ .$$

Diese Formel wird z.B. auf Seite 80 bei der Abschätzung der Tabellengröße benutzt.

Induktionsbasis: Im Fall $n = 0$ ist die Aussage wegen

$$\sum_{i=1}^{0} i = 0 = \frac{0(0+1)}{2}$$

richtig.

Induktionsschritt: Für den Induktionsschritt $n \to n+1$ folgern wir weiter

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= \Big(\sum_{i=1}^{n} i\Big) + (n+1) = \frac{n(n+1)}{2} + n + 1\\
&= \frac{n(n+1)+2n+2}{2} = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2} \ .
\end{aligned}$$

Also ist die Aussage auch für den Fall $n+1$ richtig und damit vollständig bewiesen. Beachten Sie, dass wir am Ende der ersten Zeile wieder die Induktionshypothese verwendet haben.

A.1.2 Beispiel Die *Fibonacci–Zahlen* (sprich: „Fiehbohnatschi“) sind eine unendliche Folge von Zahlen $F_0, F_1, F_2, \ldots$ und wie folgt definiert:

$$F_n := \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & \text{falls } n \geq 2 \end{cases}$$

Es gilt also $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, usw. Wir beweisen durch vollständige Induktion die folgende Formel, mit der man direkt die n–te Fibonacci–Zahl ausrechen kann:

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] .$$

Hierzu benötigen wir zunächst die beiden Identitäten:

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = 1 + \left(\frac{1+\sqrt{5}}{2}\right) \qquad \text{und} \qquad \left(\frac{1-\sqrt{5}}{2}\right)^2 = 1 + \left(\frac{1-\sqrt{5}}{2}\right) .$$

Diese sind wegen

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = \left(\frac{1}{2}+\frac{\sqrt{5}}{2}\right)^2 = \frac{1}{4}+2\cdot\frac{1}{2}\cdot\frac{\sqrt{5}}{2}+\frac{5}{4} = \frac{6}{4}+\frac{\sqrt{5}}{2} = 1+\frac{1}{2}+\frac{\sqrt{5}}{2} = 1+\left(\frac{1+\sqrt{5}}{2}\right)$$

und

$$\left(\frac{1-\sqrt{5}}{2}\right)^2 = \left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^2 = \frac{1}{4}-2\cdot\frac{1}{2}\cdot\frac{\sqrt{5}}{2}+\frac{5}{4} = \frac{6}{4}-\frac{\sqrt{5}}{2} = 1+\frac{1}{2}-\frac{\sqrt{5}}{2} = 1+\left(\frac{1-\sqrt{5}}{2}\right)$$

korrekt.

Nun prüft man für den Induktionsstart die beiden Fälle $n = 0$ und $n = 1$ nach. Im Gegenzug werden wir dann beim Induktionsschritt für $n + 1$ auf die beiden vorherigen Fälle $n - 1$ und n als Induktionshypothesen zurückgreifen können. Tatsächlich gilt

$$F_0 = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0\right] = \frac{1}{\sqrt{5}}(1-1) = 0$$

sowie

$$F_1 = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1\right]$$

$$= \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}\right] = \frac{1}{\sqrt{5}}\left[\frac{1}{2}+\frac{\sqrt{5}}{2}-\frac{1}{2}+\frac{\sqrt{5}}{2}\right] = \frac{1}{\sqrt{5}}\left[2\cdot\frac{\sqrt{5}}{2}\right] = 1 .$$

Für $n = 0$ und $n = 1$ ist die Formel also richtig. Für den Induktionsschritt nach $n + 1$ können wir dann wie bereits erwähnt annehmen, dass die Formel für den Fall $n - 1$ und n bereits gültig ist, d.h. wie dürfen die Gleichungen

$$F_{n-1} = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right]$$

und

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] .$$

als gültig voraussetzen. Wir addieren nun diese beiden Gleichungen zusammen. Auf der linken Seite ergibt sich wegen $F_{n-1} + F_n = F_{n+1}$ gerade der Wert F_{n+1}, und für die rechten Seite erhalten wir

$$\begin{aligned}
& \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right] + \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\left(1+\left(\frac{1+\sqrt{5}}{2}\right)\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\left(1+\left(\frac{1-\sqrt{5}}{2}\right)\right)\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\left(\frac{1+\sqrt{5}}{2}\right)^{2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\left(\frac{1-\sqrt{5}}{2}\right)^{2}\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right] .
\end{aligned}$$

(Zwischen der vierten und fünften Zeile haben wir dabei die vorab gezeigten Identitäten ausgenutzt.) Insgesamt gilt also

$$F_{n+1} = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right] ,$$

und dies ist gerade die zu beweisende Formel für den Fall $n + 1$.

In den meisten Beweisen werden der Induktionsstart und der Induktionsschritt nicht wie oben explizit herausgestellt, sondern die benötigten Argumente für die beiden Beweisteile werden in einem fließenden Text einfach hintereinander geschrieben. Auch die Verwendung der Induktionsannahme wird nicht besonders erwähnt. Betrachten Sie dazu die folgenden beiden abschließenden Beispiele.

A.1.3 Beispiel Wir zeigen, dass jede dritte Fibonacci–Zahl $F_0, F_3, F_6, \ldots$ (also die Folge F_{3n} für $n \in \mathbb{N}_0$) immer eine gerade Zahl ist. Für $n = 0$ ist nämlich $F_0 = 0$ und damit eine gerade Zahl. Weiterhin gilt

$$F_{3(n+1)} = F_{3n+3} = F_{3n+2} + F_{3n+1} = F_{3n+1} + F_{3n} + F_{3n+1} = 2 \cdot F_{3n+1} + F_{3n} .$$

Nun ist $2 \cdot F_{3n+1}$ sicherlich ebenso eine gerade Zahl wie F_{3n}(Induktionshypothese!), so dass sich $F_{3(n+1)}$ also aus einer Summe von zwei geraden Zahlen zusammensetzt. Demnach muss $F_{3(n+1)}$ ebenfalls gerade sein.

A.1.4 Beispiel Für alle $n \in \mathbb{N}_0$ ist $5^n + 3$ durch 4 teilbar.

Beweis: Im Fall $n = 0$ gilt $5^n + 3 = 1 + 3 = 4$, und 4 ist durch 4 teilbar. Ferner gilt

$$5^{n+1} + 3 = 5 \cdot 5^n + 3 = 5 \cdot 5^n + 15 - 12 = 5 \cdot (5^n + 3) - 12 \ ,$$

und da sowohl 12 als auch $5^n + 3$ durch 4 teilbar sind, ist auch $5 \cdot (5^n + 3) - 12$ durch 4 teilbar. □

Nicht immer ist die Variable n, über die man die Induktion führen möchte, direkt in der zu beweisenden Aussage enthalten. Man muss sie sich dann erst „geschickt“ wählen. Dies gilt vor allem für sog. *strukturelle Induktionen*, die sich auf den Aufbau von bestimmten Objekten beziehen. Eine solche strukturelle Induktion wird z.B. (ohne dass wir es dort explizit erwähnt haben) in dem Beweis von Satz 2.3.5 auf Seite 57 ausgeführt.

B

Untere Schranken

Dieses Kapitel befasst sich mit dem Nachweis einiger *unterer Schranken*, d.h. für diverse algorithmische Problemstellungen soll nachgewiesen werden, dass sie sich nicht schneller als mit einem gewissen Zeitaufwand berechnen lassen. Dieser Aufwand liegt in der Struktur der jeweiligen Probleme begründet.

Das Aufspüren von unteren Schranken ist häufig sehr schwierig. Denken Sie nur an die vielen $\mathcal{NP}$–vollständigen Probleme aus dem fünften Kapitel. Für keines dieser Probleme konnte man bislang eine exponentielle untere Schranke nachweisen, die die Vermutung $\mathcal{P} \neq \mathcal{NP}$ (siehe Seite 188) bestätigen würde.

Für ein konkretes Problem sind untere Schranken im Allgemeinen deshalb schwer zu finden, weil man gegen alle möglichen Lösungsalgorithmen argumentieren muss — und davon gibt es unendlich viele. Für eine obere Schranke muss man dagegen „nur“ ein entsprechend schnelles Verfahren finden.

In den folgenden Abschnitten werden wir drei Probleme vorstellen, bei denen der Nachweis einer nichttrivialen unteren Schranke trotzdem gelingt.

B.1. Zur Zeitkomplexität vergleichender Sortierverfahren

Sortieren ist eine wichtige Aufgabe in der Informationstechnologie. Immer wieder müssen Objekte in eine bestimmte Reihenfolge gebracht werden. Sie kennen dies aus der täglichen Arbeit mit Computern, bei denen man sich z.B. den Inhalt von Unterverzeichnissen nach den Dateinamen oder Änderungsdatum sortiert anzeigen lassen kann.

Untersuchungen von Computerherstellern wie z.B. IBM zeigen seit vielen Jahren, dass mehr als ein Viertel der kommerziell verbrauchten Rechenzeit auf Sortiervorgänge entfällt. Natürlich wurden und werden deshalb große Anstrengungen unternommen, möglichst effiziente Verfahren zum Sortieren von Daten zu entwickeln. Es gibt ganze Bücher, die sich ausschließlich mit dem Sortierproblem befassen, z.B. der erste Teil des klassi-

schen Buchs von Donald Knuth [17] auf fast 400 Seiten.

Sortierverfahren wurden in dieser Vorlesung nur beiläufig besprochen. Im Beispiel 5.1.1 auf Seite 176 wurde der Programmcode von *Bubblesort* angegeben und nebenbei erwähnt, dass es erheblich bessere Sortierverfahren wie z.B. *Heapsort* gibt, die für das Anordnen von n Objekten nur $O(n \log n)$ viel Zeit benötigen.

Bubblesort, Heapsort und viele weitere Verfahren haben eine gemeinsame Eigenschaft: sie greifen auf die zu sortierenden Objekte nur *vergleichend* zu. Für je zwei Objekte wird also wiederholt abgefragt, ob das eine Objekt kleiner, größer, gleich, usw. dem anderen ist. Die Auswertung anderer Objekteigenschaften erfolgt nicht. Ein solcher Algorithmus heißt dann auch *vergleichendes Sortierverfahren*. Im erwähnten Bubblesort–Algorithmus erfolgen diese Vergleiche z.B. in der vierten Zeile des Programmcodes auf Seite 176.

Ziel dieses Abschnitts ist der Beweis des folgenden Satzes.

B.1.1 Satz Das Sortieren von n Objekten mit einem vergleichenden Verfahren benötigt $\Omega(n \log n)$ viel Zeit.

Beweis: Um die Notation zu erleichtern, gehen wir o.B.d.A. davon aus, dass Zahlen sortiert werden sollen. Wir übergeben dann dem Sortierverfahren die ersten n Zahlen in einer durcheinander gewürfelten Reihenfolge und erwarten, dass der Algorithmus die Zahlen neu positioniert, so dass nach allen Umpositionierungen die Zahlen in einer z.B. aufsteigend sortierten Reihenfolge vorliegen. Für $i = 1, \ldots, n$ bezeichne $\sigma(i)$ die neue Position, die von der i–ten Zahl nach der Sortierung eingenommen wird. Die „Vertauschungsvorschrift“ $\sigma : \{1, \ldots, n\} \longrightarrow \{1, \ldots, n\}$ ist also eine *Permutation* (beginnend mit Seite 180 haben wir bereits mit solchen Permutationen gearbeitet).

Die Permutation ist eindeutig bestimmt, da alle Zahlen paarweise verschieden sind. Somit muss der Algorithmus unter allen möglichen $n!$ Permutationen diese eine bestimmte Permutation finden. Grundsätzlich kann dabei jede Permutation σ die richtige sein. Beispielweise ist bei acht Zahlen die Permutation

i	1	2	3	4	5	6	7	8
$\sigma(i)$	4	6	1	7	8	5	2	3

genau dann die gesuchte Vertauschungsvorschrift, wenn die Ausgangszahlen in der Reihenfolge 4, 6, 1, 7, 8, 5, 2 und 3 angeordnet sind. Dadurch wird nämlich die erste Zahl (die 4) auf die vierte Position versetzt, die zweite Zahl (die 6) auf die sechste Position usw., so dass sich am Ende genau die sortierte Reihenfolge von 1 bis 8 ergibt.

Wie schnell kann der Algorithmus die gesuchte Permutation finden? Anfangs sind theoretisch alle $n!$ Kandidaten möglich. Nach einem Vergleich zwischen zwei Zahlen wird diese Auswahl eingeschränkt, denn manche Permutationen sind mit dem Ergebnis inkompatibel. Wenn z.B. tatsächlich acht Zahlen vorgegeben sind und der Algorithmus bei einem Vergleich feststellt, dass die zweite Zahl kleiner als die achte ist, so kann die obige Beispielpermutation sicher ausgeschlossen werden. Sie würde nämlich wegen $\sigma(2) = 6$ und

$\sigma(8) = 3$ die zweite Zahl hinter die achte Zahl positionieren, d.h. die zweite Zahl müßte dann auch größer als die achte sein, was dem Vergleichsergebnis widerspricht.

Jeder Vergleich zwischen zwei Zahlen teilt deshalb die Menge der noch möglichen Permutationen in zwei Teilmengen auf. Die einen Permuationen sind mit dem positiven, die anderen mit dem negativen Vergleichsergebnis kompatibel. Da auf jede Permutation entweder das eine oder das andere zutrifft, enthält mindestens eine Teilmenge mindestens die Hälfte aller vorherigen Permutationen (ansonsten würden beide Mengen zusammen nicht die komplette vorherige Menge ergeben).

Nun kann sich die gesuchte Permutation in der größeren Teilmenge befinden. Wenn der Algorithmus mit einer für ihn möglichst ungünstigen Startreihenfolge „belastet“ wird, tritt diese Situation auch bei allen weiteren Vergleichsschritten auf. Da jedesmal die Kandidatenmenge höchstens halbiert wird, werden somit insgesamt mindestens (aufgerundet)

$$\log_2(n!)$$

Vergleichsschritte benötigt, bis nur noch eine (die gesuchte) Permuatation übrig bleibt.

Wegen

$$n! = n \cdot (n-1) \cdots 1 = \underbrace{n \cdot (n-1) \cdots}_{n\text{–mal}} \geq \underbrace{n \cdot (n-1) \cdots}_{(n/2)\text{–mal}} \geq \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdots \frac{n}{2}}_{(n/2)\text{–mal}} = \left(\frac{n}{2}\right)^{n/2}$$

folgt

$$\log_2(n!) \geq \frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \frac{n}{2}(\log_2 n - 1) = \frac{1}{2} n \log_2 n - \frac{n}{2} = \Omega(n \log n)$$

und somit die Behauptung. □

Somit ergibt sich unmittelbar:

B.1.2 Korollar Vergleichende Verfahren mit einer Zeitkomplexität von $O(n \log n)$ wie z.B. *Heapsort* arbeiten optimal. □

B.2. Zur Optimalität der Umwandlung mehrbändiger TMs

Wir haben auf Seite 125 in der Def. 4.3.1 das Modell einer mehrbändigen Turingmaschine vorgestellt und dabei von Anfang an betont, dass die prinzipielle „Berechungskraft“ einer solchen TM nicht über der einer normalen TM liegt. Tatsächlich wurde im Anschluss durch Satz 4.3.2 auf Seite 126 gezeigt, dass sich jede mehrbändige TM durch eine einbändige TM simulieren lässt. Allerdings steigt die Laufzeit durch diese Simulation quadratisch an, wie wir gleich sehen werden. Also stellt sich sofort die Frage, ob man diesen Zeitverlust nicht irgendwie vermeiden kann. Die Antwort ist (leider) nein, und der zugehörige Beweis ist der zentrale Inhalt dieses Abschnitts.

Zunächst überzeugen wir uns davon, dass die Simulation aus Satz 4.3.2 mit einem (nur) quadratischen Zeitverlust auskommt.

B.2.1 Satz Sei M eine mehrbändige DTM für eine Sprache L, deren Laufzeit bei einer Eingabe der Länge n durch $O(f(n))$ begrenzt ist (dabei sei $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine beliebige Komplexitätsfunktion). Dann benötigt die durch Satz 4.3.2 skizzierte DTM während der Simulation höchstens $O(f^2(n) + n)$ viel Zeit (wobei $f^2(n)$ eine Kurzform für $f(n) \cdot f(n)$ ist), d.h. für den „üblichen“ Fall $f = \Omega(\sqrt{n})$ beträgt die Laufzeit $O(f^2(n))$.

Beweis: Die im Beweis von Satz 4.3.2 skizzierte DTM simuliert jeden Schritt der ursprünglichen DTM, indem sie einmal von links nach rechts über den bislang belegten Bandinhalt mit allen Spuren hinwegläuft und dabei an den Kopfpositionen die entsprechenden Symbole einsammelt. Danach läuft sie von rechts nach links zurück und führt an den markierten Positionen diejenigen Modifikationen aus, die dem Übergang der ursprünglichen DTM entsprechen. Jeder Original–Übergang wird also nun durch viele Einzelschritte simuliert, wobei die Anzahl dieser Einzelschritte (in etwa) dem Doppelten des maximalen Abstands der simulierten Kopfpositionen entspricht. Wie groß aber kann dieser Abstand nach k simulierten Schritten werden? Anders gefragt: wie lautet eine obere Schranke für die Breite (in Feldern gemessen) desjenigen Bandbereichs, in dem sich alle simulierten Köpfe aufhalten?

Unmittelbar vor dem ersten Simulationschritt stehen alle simulierten Köpfe untereinander, so wie es in der Abbildung auf Seite 127 zu sehen ist. Die Breite beträgt anfangs also 1. Der schlechteste Fall tritt nun offensichtlich dann ein, wenn fortan mindestens ein Kopf sich nur noch nach rechts und ein anderer nur noch nach links bewegt. Nach einem simulierten Schritt beträgt die Breite dann drei, nach zwei simulierten Schritten fünf und allgemein nach k simulierten Schritten maximal $2k + 1$ Felder. Folglich ist die Breite stets durch

$$2 \cdot O(f(n)) + 1 = O(f(n))$$

viele Felder begrenzt, da die Turingmaschine gar nicht länger läuft. Jeder der $O(f(n))$ vielen Schritte der ursprünglichen Turingmaschine kann also in jeweils maximal $O(f(n))$ vielen Schritten simuliert werden, so dass für die eigentliche Simulation $O(f^2(n))$ viel Zeit ausreicht. Hinzu kommen noch $O(n)$ Schritte für die Initialisierung der Startkonfiguration sowie $O(f(n))$ viele Schritte zum Schluss für den Abbau der simulierten Spuren und der Ablage des berechneten Wortes, so dass die Gesamtlaufzeit wie behauptet durch $O(f^2(n) + n)$ begrenzt ist, im Fall $f = \Omega(\sqrt{n})$ also durch $O(f^2(n))$. □

Nun wollen wir im Gegenzug eine Sprache L finden, bei der eine quadratischer Zeitverlust nicht zu vermeiden ist. Wir müssen also zeigen, dass zum einen für eine passend gewählte Komplexitätsfunktion $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ mit $f = \Omega(\sqrt{n})$ eine mehrbändige DTM existiert, die L in $O(f(n))$ viel Zeit erkennt, zum anderen aber eine einbändige DTM mindestens $\Omega(f^2(n))$ Zeit dafür benötigt. Hierfür benötigen wir das Konzept der sog. *Kreuzungsfolgen*.

Sei M eine einbändige DTM, die irgendeine Sprache L akzeptiert. Wir können uns die Felder das Turingbandes mit Indizes $\ldots, -2, -1, 0, 1, 2, \ldots$ durchnummeriert vorstellen, wobei das Startfeld mit dem ersten Zeichen des Eingabeworts den Index 1 zugewiesen bekommt. Während der Verarbeitung von einem Wort $w \in L$ kann man sich die sog. *Kreuzungsfolge* $\ell_{w,i\leftrightarrow i+1}$ von Zuständen aufschreiben, die M unmittelbar nach dem Überschreiten der Grenze zwischen dem i–ten und dem $(i+1)$–sten Feld einnimmt (egal ob von links oder von rechts), wobei $i \in \mathbb{Z}$ ein beliebiger Index ist. Für die Verarbeitung von $w = ab$ durch eine DTM mit der Turingtafel

	a	b	$\Box$
z_0	(z_0, b, R)	(z_1, a, L)	(z_1, a, L)
z_1	(z_1, a, L)	$(z_1, \Box, R)$	$(z_2, \Box, L)$
z_2	$-$	$-$	$-$

gilt beispielsweise

$$z_0ab \vdash bz_0b \vdash z_1ba \vdash z_1a \vdash z_1\Box a \vdash z_2\Box\Box a$$

und deshalb $\ell_{ab,1\leftrightarrow 2} = (z_0, z_1, z_1, z_1)$ und $\ell_{ab,0\leftrightarrow 1} = (z_2)$. Alle anderen Kreuzungsfolgen sind leer.

B.2.2 Lemma Eine TM $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ benötigt zur Verarbeitung eines Wortes $w \in \Sigma^*$ mindestens

$$\sum_{i=1}^{|w|} |\ell_{w,i\leftrightarrow i+1}|$$

viele Schritte, wobei $|\ell_{w,i\leftrightarrow i+1}|$ die Anzahl der aufgeführten Zustände in der Kreuzungsfolge $\ell_{w,i\leftrightarrow i+1}$ bezeichnet (es gilt also z.B. für die obige DTM $|\ell_{ab,1\leftrightarrow 2}| = 4$ und $|\ell_{ab,0\leftrightarrow 1}| = 1$).

Beweis: Für jeden Eintrag in einer Kreuzungsfolge hat M einen Schritt mit einer verschiebenden Kopfbewegung ausgeführt. Die Summe der Längen aller Kreuzungsfolgen in dem Bereich des Eingabewortes ist also sicherlich ein untere Abschätzung für die Gesamtzahl aller durchgeführten Schritte. Wir haben dabei zwar alle verschiebenden Kopfbewegungen außerhalb des Eingabebereichs sowie alle Übergänge mit neutralen Kopfbewegungen außer Acht gelassen, für unsere Zwecke reicht die im Lemma behauptete Abschätzung jedoch aus. □

Wenn bei der Verarbeitung zweier Wörter x und y an einer bestimmten Bandposition nun die gleichen Kreuzungsfolgen auftreten, so lassen sich interessante Rückschlüsse ziehen:

B.2.3 Lemma Es seien $x := x_1x_2 \ldots x_m$ und $y := y_1y_2 \ldots y_m$ zwei gleich lange Wörter, die von einer DTM M akzeptiert werden. Wenn dann an einer Position $1 \leq k < m$ die Kreuzungsfolgen $\ell_{x,k\leftrightarrow k+1}$ und $\ell_{y,k\leftrightarrow k+1}$ zwischen den Feldern k und $k+1$ identisch sind, akzeptiert M auch das „vermischte“ Wort $x_1x_2 \ldots x_ky_{k+1}y_{k+2} \ldots y_m$.

Beweis: Wenn M auf das Wort $x_1x_2\dots x_ky_{k+1}y_{k+2}\dots y_m$ angesetzt wird, so verhält sie sich zunächst genauso, als wenn sie auf dem Wort x angesetzt worden wäre. Denn links von der Grenze zwischen dem k–ten und $(k+1)$–sten Feld (also auf den ersten k Zeichen $x_1x_2\dots x_k$) sind die beiden Wörter identisch.

Wenn nun M zum ersten Mal die besagte Grenze nach rechts hin überschreitet, so verhält sie sich ab diesem Zeitpunkt genauso, als wäre sie auf das Wort y angesetzt worden und hätte dort zum ersten Mal die Grenze überschritten. In beiden Fällen findet M nämlich den gleichen Suffix $y_{k+1}y_{k+2}\dots y_m$ auf dem Band vor, und außerdem befindet sich M nach dem Überschreiten der Grenze jeweils in dem gleichen Zustand, da die beiden Kreuzungsfolgen $\ell_{x,k\leftrightarrow k+1}$ und $\ell_{y,k\leftrightarrow k+1}$ identisch sind (der erste Eintrag in diesen Folgen gibt den Zustand von M nach dem Überschreiten der Grenze an).

Auch für den weiteren Verlauf gilt, dass M sich links bzw. rechts von der Grenze wie auf x bzw. y angesetzt verhält. Denn M kann nur lokal auf den aktuellen Bandsymbolen arbeiten und sieht deshalb nicht, dass sich jeweils jenseits der Grenze eine andere Berechnung als auf dem gesamten ursprünglichen Wort x bzw. y abspielt. Sobald M aber die Grenze überschreitet, ist durch die identischen Kreuzungsfolgen sichergestellt, dass die jeweils vorgefundene lokale Konfiguration genau der jeweiligen Originalkonfiguration entspricht.

Am Ende befindet sich der Kopf entweder links oder rechts von der Grenze, und M wird genau dasselbe machen, was sie auch angesetzt auf x bzw. y gemacht hätte — sie akzeptiert. Also gilt insgesamt wie behauptet $x_1x_2\dots x_ky_{k+1}y_{k+2}\dots y_m \in L(M)$. □

Man kann eine einfache Abschätzung für die Anzahl der verschiedenen Kreuzungsfolgen angeben, sofern deren maximalen Längen begrenzt werden.

B.2.4 Lemma Sei M eine DTM mit $q \geq 2$ Zuständen. Dann gibt es nicht mehr als q^{m+1} Kreuzungsfolgen der Länge $\leq m$.

Beweis: Eine Kreuzungsfolge der Länge j besteht aus einer Sequenz von j Zuständen, wobei jeder Zustand einer von q möglichen ist. Also gibt es insgesamt q^j viele Konstellationen für eine Kreuzungsfolge der Länge j. Die Anzahl aller Kreuzungsfolgen, die höchstens m Zustände lang sind, kann also nicht mehr als

$$q^0 + q^1 + q^2 + \cdots + q^m = \sum_{j=0}^{m} q^j$$

betragen.

Zur Vereinfachung der obigen Summe berechnen wir zunächst den $(q-1)$–fachen Wert hiervon. Es ergibt sich:

$$(q-1)\sum_{j=0}^{m} q^j = q\cdot\sum_{j=0}^{m} q^j - \sum_{j=0}^{m} q^j = \sum_{j=0}^{m} q^{j+1} - \sum_{j=0}^{m} q^j = \sum_{j=1}^{m+1} q^j - \sum_{j=0}^{m} q^j \ .$$

Wegen

$$\sum_{j=1}^{m+1} q^j - \sum_{j=0}^{m} q^j = \sum_{j=1}^{m} q^j + q^{m+1} - q^0 - \sum_{j=1}^{m} q^j = q^{m+1} - q^0 = q^{m+1} - 1$$

erhalten wir also insgesamt

$$(q-1)\sum_{j=0}^{m} q^j = q^{m+1} - 1 < q^{m+1} \ .$$

Nun gilt $q \geq 2$, denn M besitzt gemäß Voraussetzung mindestens zwei Zustände. Also folgt $\frac{1}{q-1} \leq 1$ und deshalb

$$\sum_{j=0}^{m} q^j = \frac{1}{q-1} \cdot (q-1)\sum_{j=0}^{m} q^j \leq (q-1)\sum_{j=0}^{m} q^j < q^{m+1} \ .$$

Ingesamt kann es also nicht mehr (sondern — um genau zu sein — sogar nur echt weniger!) als q^{m+1} Kreuzungsfolgen der Länge $\leq m$ geben. Das war zu zeigen. □

Mit diesen Vorbereitungen können wir nun den nächsten Schritt angehen und eine Sprache L angeben, von der wir später zeigen werden, dass eine einbändige DTM M für deren Akzeptanz quadratisch viel Zeit benötigt. Wir wählen:

$$L := \{w \in \{a,b\}^{2n} \,|\, n \in \mathbb{N} \wedge w \text{ ist von vorne und hinten gelesen gleich}\} \ .$$

L ist also die Sprache aller sog. *Palindrome* gerader Länge über dem binären Alphabet $\Sigma := \{a,b\}$. Es gilt z.B. $abba \in L$ oder $bbaabb \in L$, nicht aber $abab \in L$ (kein Palindrom) oder $abbba \in L$ (keine gerade Länge).

Sei nun M eine einbändige DTM M mit $L(M) = L$. Unser Ziel ist zu zeigen, dass M durchschnittlich quadratisch viel Zeit für die Akzeptanz eines Palindroms aus L benötigt, ganz gleich, wie man M wählt. Wir betrachten dazu zunächst für alle $n \in \mathbb{N}$ die Sprache

$$L_n := \{w \in L \,|\, |w| = 2n\} \ .$$

L_n enthält alle Palindrome $w = w_1 w_2 \ldots w_{2n} \in L$ der Länge $2n$ und somit genau 2^n Wörter. Die ersten n Zeichen bis zur Mitte sind nämlich frei wählbar (jeweils entweder a oder b), die restlichen n Zeichen aber nicht — sie sind durch die ersten n Zeichen eindeutig vorgegeben, wenn sich insgesamt noch ein Palindrom ergeben soll.

Wir analysieren nun genauer, wie M die einzelnen Palindrome $w \in L_n$ verarbeitet. Für $i = 1, 2, \ldots, n$ sei dazu $m(i,n)$ die mittlere Länge aller Kreuzungsfolgen zwischen den Zeichen w_i und w_{i+1}, d.h. wir definieren

$$m(i,n) := \frac{1}{2^n} \sum_{w \in L_n} |\ell_{w,i \leftrightarrow i+1}| \ .$$

Das folgende Lemma besagt, dass mindestens die Hälfte dieser Kreuzungsfolgen nicht länger als das Doppelte dieses Mittelwerts sind.

B.2.5 Lemma Seien $i, n \in \mathbb{N}$ mit $1 \leq i \leq n$ beliebig gewählt. Dann gilt $|\ell_{w,i}| \leq 2m(i,n)$ für mindestens 2^{n-1} der insgesamt 2^n Palindrome w aus L_n.

Beweis: Sei p die Anzahl aller Kreuzungsfolgen, die länger als $2m(i,n)$ sind. Dann gilt für die Summe der Längen aller 2^n Kreuzungsfolgen die Abschätzung

$$\sum_{w \in L_n} |\ell_{w,i \leftrightarrow i+1}| \geq p \cdot (2 \cdot m(i,n)) \ ,$$

d.h. wir können für den Mittelwert $m(i,n)$ die Ungleichung

$$m(i,n) = \frac{1}{2^n} \sum_{w \in L_n} |\ell_{w,i \leftrightarrow i+1}| \geq \frac{2 \cdot p \cdot m(i,n)}{2^n}$$

aufstellen, was wiederum

$$m(i,n) \geq \frac{2 \cdot p \cdot m(i,n)}{2^n} \iff 1 \geq \frac{2 \cdot p}{2^n} \iff p \leq \frac{2^n}{2} = 2^{n-1}$$

bedeutet. Höchstens die Hälfte aller Kreuzungsfolgen ist also länger als $2m(i,n)$. Oder anders ausgedrückt: bei mindestens der Hälfte aller Kreuzungsfolgen ist deren Länge kleiner oder gleich $2m(i,n)$. Das war zu beweisen. □

B.2.6 Lemma Seien $i, n \in \mathbb{N}$ mit $1 \leq i \leq n$ beliebig gewählt. Dann gibt es mindestens

$$\frac{2^{n-1}}{q^{2m(i,n)+1}}$$

vielen Palindrome $w \in L_n$, die eine identische Kreuzungsfolge $\ell_{w,i \leftrightarrow i+1}$ aufweisen.

Beweis: Die DTM M besitzt zumindest einen Endzustand (sonst könnte sie kein einziges Wort akzeptieren). Der Startzustand von M ist aber kein Endzustand, denn gemäß unserer Definition von DTMs sind in Endzuständen keine Übergänge möglich (siehe Seite 120), d.h. M könnte ganz offensichtlich L nicht korrekt erkennen. Daher besitzt M ganz sicher mindestens zwei Zustände, d.h. wir von der Annahme $q \geq 2$ ausgehen. Nach Lemma B.2.4 gibt es somit höchstens $q^{2m(i,n)+1}$ Kreuzungsfolgen der Länge $\leq 2m(i,n)$. Wäre nun die Behauptung falsch, so könnte es insgesamt nur weniger als

$$\frac{2^{n-1}}{q^{2m(i,n)+1}} \cdot q^{2m(i,n)+1} = 2^{n-1}$$

viele Palindrome w aus L_n geben, deren Kreuzungsfolge $\ell_{w,i \leftrightarrow i+1}$ nicht länger als $2m(i,n)$ ist. Dies würde aber dem Ergebnis von Lemma B.2.5 widersprechen. □

B.2.7 Lemma Seien $i, n \in \mathbb{N}$ mit $1 \leq i \leq n$ beliebig gewählt. Seien weiter

$$v = v_1 v_2 \dots v_{2n} \qquad \text{und} \qquad w = w_1 w_2 \dots w_{2n}$$

zwei Palindrome aus L_n mit identischen Kreuzungsfolgen $\ell_{v,i\leftrightarrow i+1} = \ell_{w,i\leftrightarrow i+1}$. Dann gilt

$$v_{i+1} v_{i+2} \dots v_n \neq w_{i+1} w_{i+2} \dots w_n \ .$$

Beweis: Angenommen nicht. Dann gilt also

$$v_{i+1} v_{i+2} \dots v_n = w_{i+1} w_{i+2} \dots w_n \ ,$$

d.h. bei beiden Wörtern sind die letzten $n - i$ Zeichen links von der Mitte identisch. Dasselbe gilt dann auch für die ersten $n - i$ Zeichen rechts von der Mitte, da es sich um zwei Palindrome handelt. Da aber v und w verschieden sind, müssen sie sich folglich auf den ersten i (und damit auch auf den letzten i) Zeichen unterscheiden, d.h. es gilt

$$v_1 v_1 \dots v_i = w_1 w_2 \dots w_i \ .$$

Das ab der $(i + 1)$–ten Position neu zusammengesetzte Wort

$$v_1 v_2 \dots v_i w_{i+1} w_{i+2} \dots w_{2n}$$

wäre somit kein Palindrom mehr. Aufgrund der identischen Kreuzungsfolgen

$$\ell_{v,i\leftrightarrow i+1} = \ell_{w,i\leftrightarrow i+1}$$

würde M dieses Wort gemäß Lemma B.2.3 aber akzeptieren, was nicht sein kann. Also muss doch wie behauptet

$$v_{i+1} v_{i+2} \dots v_n \neq w_{i+1} w_{i+2} \dots w_n$$

gelten. □

B.2.8 Lemma Seien $i, n \in \mathbb{N}$ mit $1 \leq i \leq n$ beliebig gewählt. Dann gilt

$$m(i, n) \geq \frac{i-1}{2 \log_2 q} - \frac{1}{2} \ .$$

Beweis: Nach Lemma B.2.6 gibt es mindestens

$$\frac{2^{n-1}}{q^{2m(i,n)+1}}$$

Palindrome $w \in L_n$ mit identischen Kreuzungsfolgen $\ell_{w,i\leftrightarrow i+1}$. Je zwei dieser Palindrome müssen sich gemäß Lemma B.2.7 auf ihren $n - i$ Zeichen links von der Mitte unterscheiden. Da es aber überhaupt nur 2^{n-i} verschiedene Wörter der Länge $n - i$ gibt (an jeder Position steht entweder ein a oder ein b), muss die Ungleichung

$$\frac{2^{n-1}}{q^{2m(i,n)+1}} \leq 2^{n-i}$$

erfüllt sein. Mit Hilfe der Umformungen

$$\frac{2^{n-1}}{q^{2m(i,n)+1}} \leq 2^{n-i} \iff q^{2m(i,n)+1} \geq \frac{2^{n-1}}{2^{n-i}} \iff q^{2m(i,n)+1} \geq 2^{i-1}$$

$$\iff \log_2(q^{2m(i,n)+1}) \geq \log_2(2^{i-1}) \iff (2m(i,n)+1)\log_2 q \geq i-1$$

$$\iff 2m(i,n)+1 \geq \frac{i-1}{\log_2 q} \iff m(i,n) \geq \frac{i-1}{2\log_2 q} - \frac{1}{2}$$

erhalten wir das behauptete Ergebnis. □

Damit können wir unsere Hauptresultate formulieren:

B.2.9 Satz Eine einbändige DTM M benötigt durchschnittlich quadratisch viel Zeit zur Akzeptanz eines Wortes w aus der Sprache

$$L := \{w \in \{a,b\}^{2n} \mid n \in \mathbb{N} \wedge w \text{ ist von vorne und hinten gelesen gleich}\} \ .$$

Beweis: Nach Lemma B.2.2 ist

$$\sum_{i=1}^{|w|} |\ell_{w,i\leftrightarrow i+1}|$$

eine untere Schranke für die benötigte Laufzeit zur Akzeptanz eines Wortes $w \in L$. Die *durchschnittliche* Laufzeit zur Akzeptanz aller 2^n Wörter aus $L_n \subseteq L$ beträgt daher mindestens

$$\frac{1}{2^n} \sum_{w\in L_n} \sum_{i=1}^{|w|} |\ell_{w,i\leftrightarrow i+1}| = \frac{1}{2^n} \sum_{w\in L_n} \sum_{i=1}^{2n} |\ell_{w,i\leftrightarrow i+1}| \geq \frac{1}{2^n} \sum_{w\in L_n} \sum_{i=1}^{n} |\ell_{w,i\leftrightarrow i+1}| \ .$$

Wegen

$$\frac{1}{2^n} \sum_{w\in L_n} \sum_{i=1}^{n} |\ell_{w,i\leftrightarrow i+1}| = \sum_{i=1}^{n} \Big(\frac{1}{2^n} \sum_{w\in L_n} |\ell_{w,i\leftrightarrow i+1}|\Big) = \sum_{i=1}^{n} m(i,n)$$

können wir die durchschnittliche Laufzeit auf eine uns bereits bekannte Größe zurückführen. Da die Anzahl q der Zustände einer konkreten DTM M konstant ist, erhalten wir nun

$$\sum_{i=1}^{n} m(i,n) \ \geq \ \sum_{i=1}^{n} \Big(\frac{i-1}{2\log_2 q} - \frac{1}{2}\Big)$$

$$= \ \frac{1}{2\log_2 q} \sum_{i=1}^{n} (i-1) - \sum_{i=1}^{n} \frac{1}{2} \ = \ \frac{1}{2\log_2 q} \cdot \frac{n(n-1)}{2} - \frac{n}{2} \ = \ \Omega(n^2) \ .$$

Damit ist alles gezeigt. □

Beachten Sie, dass somit die sog. *Worst–Case–Komplexität* für die Sprache L ebenfalls mindestens quadratisch sein muss (diese beschreibt, welche maximale Laufzeit ein Eingabewort der Länge n verursacht). Umgekehrt gilt dies im Allgemeinen nicht, d.h. wir haben sogar ein schärferes Resultat als eingangs behauptet bewiesen.

B.2.10 Satz Eine mehrbändige DTM M kann die Sprache

$$L := \{w \in \{a,b\}^{2n} \mid n \in \mathbb{N} \wedge w \text{ ist von vorne und hinten gelesen gleich}\} \ .$$

in Linearzeit erkennen.

Beweis: Bereits eine zweibändige DTM M erreicht dieses Ziel mühelos. Wir geben eine Konstruktionsskizze an, die man leicht in ein konkretes Turing–Programm umsetzen kann:

a) M bewegt sich zunächst über dem Eingabewort w nach rechts und rückt dabei nach jeweils zwei Schritten auch den anderen Schreib–Lese–Kopf um ein Feld nach rechts vor, wobei sie ein Zeichen (z.B. \$) auf dem zweiten Band hinterlässt. Somit hat sie auf diesem Band die halbe Länge von w abgesteckt, wenn sie das Ende von w erreicht. Natürlich überprüft sie dabei gleich noch, ob w eine gerade Länge besitzt (d.h. ob sie immer jeweils zwei Zeichen auf dem ersten Band vorgefunden hat) — falls nicht, verwirft sie.

b) Nun bewegt M beide Köpfe synchron nach links und kopiert dabei die rechte Worthälfte von w auf das zweite Band. M kann die Wortmitte von w leicht erkennen, da gleichzeitig auf dem zweiten Band die \$–Markierungszeichen aufhören. Nun steht sie mit ihrem ersten Kopf in der Mitte von w (genauer gesagt soll sie sich unter dem ersten Zeichen links von der Wortmitte positionieren) und mit ihrem zweiten Kopf wieder auf der Ausgangsposition.

c) Nun muss M nur noch den ersten Kopf weiter nach links sowie den zweiten zurück nach rechts bewegen und dabei alle gelesenen Zeichen von beiden Bändern vergleichen. Sollten irgendwelche Unstimmigkeiten auftreten, so verwirft M. Wenn dagegen alles stimmt, muss w ein Palindrom sein, und M akzeptiert.

Gemessen an der Länge von w benötigt jeder Schritt nur linear viel Zeit, d.h. die Gesamtkomplexität beträgt offensichtlich $O(n)$. □

Damit haben wir unser Ziel erreicht:

B.2.11 Korollar Die Konstruktion aus Satz 4.3.2 ist optimal.

Beweis: In Satz B.2.1 haben wir gezeigt, dass die Zeitkomplexität der vorgestellten Simulation (nur) um einen quadratischen Faktor ansteigt. Dies ist im allgemeinen nicht zu vermeiden, denn mit

$$L := \{w \in \{a,b\}^{2n} \mid n \in \mathbb{N} \wedge w \text{ ist von vorne und hinten gelesen gleich}\}$$

haben wir ein Beispiel für eine Sprache gefunden, die mit einer mehrbändigen DTM in $O(n)$, mit einer einbändigen DTM aber in nicht weniger als $\Omega(n^2)$ viel Zeit zu erkennen ist. □

B.3. Beweis einer unteren Schranke für die GNF–Umwandlung

In dritten Kapitel wurde neben der *Chomsky–Normalform* (*CNF*, siehe Def. 3.3.1 auf Seite 99) auch die *Greibach–Normalform* (*GNF*, siehe Def. 3.3.7 auf Seite 107) für kontextfreie Grammatiken vorgestellt. Ein Aspekt, der bislang vernachlässigt wurde, betrifft die *Größe* der Grammtiken, wenn man diese in CNF oder GNF umwandelt. Die Größe einer Grammatik ist dabei die Summe der Längen der linken und rechten Wörter von allen Grammatikregeln. Da eine kontextfreie Grammatik G auf jeder linken Produktionsseite nur eine einzelne Variable aufweist (d.h. die Länge beträgt dort immer 1), ergibt sich bei p Regeln mit den rechten Seitenlängen $\alpha_1, \alpha_2, \ldots, \alpha_p$ folgende Formel:

$$|G| := \sum_{i=1}^{p} (1 + |\alpha_i|) \ .$$

Eine Grammatik kann bei der Umwandlung in CNF oder GNF deutlich größer werden. Bzgl. CNF–Grammatiken ist bislang kein Algorithmus bekannt, welcher besser arbeitet als das von uns besprochene Verfahren (siehe Satz 3.3.3 auf Seite 100). Wir notieren hier ohne Beweis, dass durch die Eliminierung der Kettenregeln die Ausgangsgrammatik G auf quadratische Größe $O(|G|^2)$ anschwellen kann. Bei GNF–Grammatiken tritt der Vergrößerungseffekt noch stärker auf, denn das bislang beste Verfahren begrenzt die Größe der erzeugten GNF–Grammatik auf $O(|G|^4)$ [5, 29]. Weitere Details hierzu befinden sich am Schluss dieses Anhangs.

Auf der Suche nach besseren Algorithmen stellt sich natürlich die Frage, wie stark weitere Verbesserungen überhaupt möglich sind. Anders ausgedrückt: Ist eine gewisse Vergrößerung einer Grammatik bei ihrer Umwandlung in CNF oder GNF unvermeidbar? Dies ist zumindest bei der GNF tatsächlich der Fall – wir werden in diesem Anhang nachweisen, dass ein quadratisches Größenwachstum in Kauf genommen werden muss.

Wir beginnen unsere Analyse mit kontextfreien Grammatiken, die nur ein einziges Wort w ableiten können. $G = (V, \Sigma, P, S)$ sei eine solche Grammatik, und o.B.d.A. habe G unter allen $L := \{w\}$ erzeugenden kontextfreien Grammatiken eine möglichst geringe Größe.

B.3.1 Lemma Eine sog. *nutzlose Variable* taucht entweder in keiner Satzform auf oder ist nicht in der Lage, ein Wort aus Σ^* zu erzeugen. G enthält jedoch keine solchen nutzlosen Variablen.

Beweis: Dies ist offensichtlich, ansonsten könnte man die Grammatik entsprechend bereinigen und dadurch kleiner machen, was der Annahme der Größenminimalität von G widerspricht. □

B.3.2 Lemma Kontextfreie Grammatiken enthalten per Definition keine „ε–Regeln“ der Form $A \to \varepsilon$ (vgl. die Ausführungen auf S. 39). Aber selbst wenn ε–Regeln erlaubt wären, würde G keine enthalten.

Beweis: Angenommen, es gäbe eine solche Regel, etwa $A \to \varepsilon$. Dann betrachten wir eine Ableitung der Form $S \Rightarrow xAz \Rightarrow xz = w$ (eine solche Ableitung existiert, da A nach Lemma B.3.1 nützlich ist). Aus A kann man dann kein anderes nichtleeres Wort y erzeugen, ansonsten wäre auch das Wort xyz ableitbar, was wegen $xyz \neq xz = w$ und $L(G) = \{w\}$ ein Widerspruch wäre. Folglich kann man alle Regeln bereinigen, bei denen auf der rechten Seite die Variable A auftaucht, indem man jeweils dieses Vorkommen von A ersatzlos streicht (und es gibt mindestens eine solche Regel, die in der obigen Ableitung zu der Satzform xAz geführt hat). Dies aber würde zu kürzeren rechten Seiten und damit insgesamt zu einer kleineren Grammatik G führen, was erneut der Annahme der Größenminimalität von G widerspricht. □

B.3.3 Lemma In G sind Ableitungen der Form $A \Rightarrow^+ A$ unmöglich, ganz gleich, welche Variable A gewählt wird.

Beweis: Nach Lemma B.3.2 gibt es keine ε–Regeln in P. Folglich impliziert eine Ableitung der Form $A \Rightarrow^+ A$, dass ausgehend von A nur Kettenregeln zum Einsatz kommen, die schließlich wieder in A münden. Aber dann wären alle dabei auftretenden Variablen gleichwertig, und man könnte sie durch eine einzige Variable ersetzen (dies war auch der erste Schritt bei der Umwandlung einer Grammatik in CNF, siehe Satz 3.3.3 auf S. 100). Insbesondere wären alle Kettenregeln hinfällig, und die so verkleinerte Grammatik stünde ein weiteres Mal im Widerspruch zur Größenminimalität von G. □

B.3.4 Lemma G enthält keine rekursiven Variablen, d.h. für jede Variable A sind Ableitungen der Form $A \Rightarrow^+ \alpha A \beta$ mit $\alpha, \beta \in (V \cup \Sigma)^*$ unmöglich.

Beweis: Den Fall $\alpha = \beta = \varepsilon$ haben wir soeben in Lemma B.3.3 ausgeschlossen. Nehmen wir also an, es würde eine Ableitung der Form $A \Rightarrow^+ \alpha A \beta$ mit $\alpha \neq \varepsilon$ oder $\beta \neq \varepsilon$ (oder beides) geben. Da es gemäß Lemma B.3.1 keine nutzlosen Variablen in α oder β gibt, gilt sogar $A \Rightarrow^* \alpha A \beta \Rightarrow^* uAv$ für passende Wörter u und v. Die Variable A ist gemäß Lemma B.3.1 ebenfalls nützlich. Also existiert eine Ableitung der Form $S \Rightarrow^* xAz \Rightarrow^* xyz = w$. In diese Ableitung könnten wir die vorherige Ableitung einsetzen und so auch z.B. das Wort

$$S \Rightarrow^* xAz \Rightarrow^* x\alpha A\beta z \Rightarrow^* xuAvz \Rightarrow^* xuyvz$$

erzeugen. Nach Lemma B.3.2 gibt es aber keine ε–Regeln, d.h. jede Variable erzeugt immer zumindest ein Zeichen. Folglich gilt $|\alpha| \leq |u|$ und $|\beta| \leq |v|$. Wegen $\alpha\beta \neq \varepsilon$ muss demnach mindestens eines der beiden Wörter u und v nichtleer sein, d.h. es gilt mit Sicherheit $xuyvz \neq xyz = w$. Somit wäre G in der Lage, ein nicht erlaubtes Wort abzuleiten. Also kann es keine rekursiven Variablen in G geben. □

B.3.5 Lemma Sei A eine Variable und x ein aus A ableitbares Wort. Sei ferner p die Anzahl der *verschiedenen* Produktionen, die in einer zugehörigen Ableitung $A \Rightarrow^* x$ zum Einsatz kommen (wobei eine Regel durchaus mehrmals verwendet werden darf). Weiter seien $\alpha_1, \ldots, \alpha_p$ die rechten Seiten dieser p Regeln. Dann ist die Länge von x durch das Produkt

$$|\alpha_1| \cdot |\alpha_2| \cdots |\alpha_p|$$

nach oben begrenzt.

Beweis: Wir beweisen die Behauptung durch Induktion über die Regelanzahl p (eine Einführung in Induktionsbeweise wird in Anhang A gegeben).

Im Fall $p = 1$ gibt es nur eine Regel, deren rechte Seite wir mit α_1 bezeichnen. Auf der linken Seite muss die Regel die Variable A aufweisen, da sonst mit dieser einzigen Regel die Ableitung $A \Rightarrow^* x$ unmöglich wäre. Also ist die Regel von der Form $A \to \alpha_1$ mit $\alpha_1 \in (V \cup \Sigma)^*$. In α_1 kann allerdings die Variable A nicht noch einmal auftreten, ansonsten könnte die Ableitung niemals enden, da es ja nur diese eine Regel gibt. Genauso sind auch alle anderen Variablen in α_1 unmöglich, denn dann könnte die Ableitung nicht mehr zum Wort x fortgeführt werden (außer $A \to \alpha_1$ ist ja keine Regel vorhanden). Also besteht α_1 nur als Terminalen und muss somit gleich x sein. Die einzige Regel hat demnach die Form $A \to x$ und erfüllt damit trivialerweise die Behauptung wegen $|x| \leq |x| = |\alpha_1|$.

Im Fall $p > 1$ besteht die Ableitung aus mindestens zwei Schritten. Die zuerst verwendete Regel hat die Form $A \to \alpha_i$ für ein passendes i mit $1 \leq i \leq p$. Wir können o.B.d.A. $i = p$ annehmen (denn wir sind in der Wahl frei, in welcher Reihenfolge die rechten Produktionsseiten mit $\alpha_1, \ldots, \alpha_p$ bezeichnet werden). Somit lässt sich die Ableitung von A nach x durch

$$A \Rightarrow \alpha_p \Rightarrow^* x$$

beschreiben. Die Regel $A \to \alpha_p$ kommt jedoch kein zweites Mal zum Einsatz. Wäre dies der Fall, so müßte zwischenzeitlich eine Satzform $\beta A \gamma$ (mit passenden Wörtern $\beta, \gamma \in (V \cup \Sigma)^*$) durchlaufen werden. Eine solche Satzform ist jedoch aus A gemäß Lemma B.3.4 nicht ableitbar.

Wenn die Regel $A \to \alpha_p$ also nicht nochmals verwendet wird, so kommen in der Ableitung $\alpha \Rightarrow^* x$ nur noch die $p - 1$ anderen Produktionen mit den rechten Seiten $\alpha_1, \ldots, \alpha_{p-1}$ vor. Jede in α_p vorhandene Variable kann gemäß Induktionsannahme höchstens ein

$$|\alpha_1| \cdot |\alpha_2| \cdots |\alpha_{p-1}|$$

langes Wort ableiten, und da es nicht mehr als $|\alpha_p|$ viele Variablen in α_p gibt, kann x nicht länger als insgesamt

$$|\alpha_p| \cdot \Big(|\alpha_1| \cdot |\alpha_2| \cdots |\alpha_{p-1}|\Big) = |\alpha_1| \cdot |\alpha_2| \cdots |\alpha_p|$$

sein, womit der Induktionsschluss bewiesen wäre. □

Lemma B.3.5 greift insbesondere für die Startvariable S. Wenn G also p Regeln besitzt, deren rechte Seiten die Längen $\ell_1, \ldots, \ell_p$ haben, so ist die Länge von dem einzigen Satz w durch das Produkt

$$\ell_1 \cdot \ell_2 \cdots \ell_p$$

beschränkt. Ist ein festes Wort w vorgegeben, muss folglich

$$\ell_1 \cdot \ell_2 \cdots \ell_p \geq |w|$$

gelten. Dann aber kann man auch zeigen, dass die Größe der Grammatik

$$\sum_{i=1}^{p}(1 + \ell_i)$$

eine gewisse Mindestgröße nicht unterschreiten kann. Dies ist die Aussage des nächsten Lemmas. Wir legen uns dazu auf ein Wort w der Länge 4^k fest, wobei k irgendeine nichtnegative ganze Zahl ist.

B.3.6 Lemma Sei $k \in \mathbb{N}_0$ beliebig. Weiter seien p natürliche Zahlen $\ell_1, \ldots, \ell_p$ so gewählt, dass

$$\ell_1 \cdot \ell_2 \cdots \ell_p \geq 4^k$$

erfüllt ist. Dann gilt

$$\sum_{i=1}^{p}(1 + \ell_i) \geq 5k \ .$$

Beweis: Angenommen nicht. Dann müssten sich natürliche Zahlen k und $\ell_1, \ldots, \ell_p$ finden lassen, so dass die beiden Bedingungen

$$1) \qquad \ell_1 \cdot \ell_2 \cdots \ell_p \geq 4^k$$

$$2) \qquad \sum_{i=1}^{p}(1 + \ell_i) < 5k$$

zutreffen. Wir werden schrittweise aus einer solchen Lösung weitere Lösungen für evtl. andere Zahlen k und p konstruieren, die ebenfalls beide Gleichungen erfüllen, aber einfacher aufgebaut sind. Am Ende werden wir dann aber sehen, dass diese gar nicht existieren können.

Schritt 1: *Jede Lösung umfasst immer zumindest zwei Zahlen, d.h. die Fälle* $p = 1$ *oder* $p = 0$ *sind unmöglich.*

Im Fall $p = 1$ wäre nur eine Zahl ℓ_1 an der Lösung beteiligt. Gemäß 1) und 2) müßte dann zugleich

$$\ell_1 \geq 4^k \qquad \text{und} \qquad 1 + \ell_1 < 5k$$

gelten, insgesamt also

$$1 + 4^k < 5k \ ,$$

was für alle $k \in \mathbb{N}_0$ unmöglich ist.

Die „leere“ Lösung (mit $p = 0$) ist zwangsläufig ebenso falsch, denn dann müßte $1 \geq 4^k$ (also $k = 0$) und zugleich $0 < 5k$ (also $k \geq 1$) gelten.

Schritt 2: *Wenn es eine Lösung $\ell_1, \ldots, \ell_p \in \mathbb{N}$ für 1) und 2) gibt, dann auch eine solche, die höchstens eine 5 und ansonsten nur Zahlen ≤ 4 enthält.*

Angenommen, in einer Lösung mit

$$\begin{aligned} &1) \qquad \ell_1 \cdot \ell_2 \cdots \ell_p \geq 4^k \\ &2) \qquad \sum_{i=1}^{p} (1 + \ell_i) < 5k \end{aligned}$$

existieren zwei Zahlen $\ell_i, \ell_j \geq 5$. Da wir alle Zahlen $\ell_1, \ldots, \ell_p$ beliebig vertauschen können (denn auf die Bedingungen 1) und 2) hat dies keinen Einfluss), nehmen wir o.B.d.A. an, dass ℓ_1 und ℓ_2 größer oder gleich 5 sind.

Nun ersetzen wir ℓ_1 und ℓ_2 durch $\ell_1 - 2$ und $\ell_2 - 2$ und fügen eine weitere Zahl $\ell_{p+1} := 3$ zur Lösung hinzu. Hierdurch wird das Produkt in 1) größer, denn statt $\ell_1 \cdot \ell_2$ geht nun $(\ell_1 - 2) \cdot (\ell_2 - 2) \cdot 3$ in das Produkt mit ein, und es gilt

$$\begin{aligned} (\ell_1 - 2) \cdot (\ell_2 - 2) \cdot 3 &= 3\,\ell_1\ell_2 - 6(\ell_1 + \ell_2) + 12 = \ell_1\ell_2 + 2(\ell_1\ell_2 - 3(\ell_1 + \ell_2) + 9) - 6 \\ &= \ell_1\ell_2 + 2\underbrace{(\ell_1 - 3)}_{\geq 2}\underbrace{(\ell_2 - 3)}_{\geq 2} - 6 \geq \ell_1\ell_2 + 8 - 6 > \ell_1\ell_2 \ . \end{aligned}$$

Also gilt 1) nach der Modifikation noch immer. Auch 2) ist weiterhin erfüllt, denn bzgl. der Summe fallen nun $1 + \ell_1$ und $1 + \ell_2$ weg, dafür kommen die neuen Summanden $1 + (\ell_1 - 2)$, $1 + (\ell_2 - 2)$ und $1 + 3$ hinzu. Die Summe ändert sich wegen

$$1 + (\ell_1 - 2) + 1 + (\ell_2 - 2) + 1 + 3 = \ell_1 + \ell_2 + 2 = (1 + \ell_1) + (1 + \ell_2)$$

also nicht.

Somit ist es uns gelungen, auf Kosten einer zusätzlichen Zahl (der 3) zwei andere Zahlen der Lösung zu verringern. Wir wiederholen diesen Schritt solange, bis es keine zwei Zahlen mehr gibt, die größer oder gleich 5 sind.

Als Nächstes eliminieren wir die Vieren:

Schritt 3: *Wenn es eine Lösung $\ell_1, \ldots, \ell_p \in \mathbb{N}$ für 1) und 2) gibt, dann auch eine solche, die höchstens eine 5 und ansonsten nur Zahlen ≤ 3 enthält.*

Jede Vier in der Lösung können wir nämlich einfach weglassen, wenn wir dafür gleichzeitig die Zahl k um eins verringern. In Bedingung 1) werden dadurch beide Seiten durch 4 geteilt, und in Bedingung 2) verringern sich beide Seiten um 5. Also bleiben beide Bedingungen wahr.

Einsen können wir noch leichter eliminieren:

Schritt 4: *Wenn es eine Lösung $\ell_1, \ldots, \ell_p \in \mathbb{N}$ für 1) und 2) gibt, dann auch eine solche, die höchstens eine 5 und ansonsten nur Zweien und Dreien enthält.*

Einsen kann man nämlich ersatzlos streichen — an der erfüllten Bedingung 1) ändert sich dadurch überhaupt nichts, und Bedingung 2) bleibt genauso wahr, da lediglich die linke Seite kleiner wird.

Nun stört noch die Fünf:

Schritt 5: *Wenn es eine Lösung $\ell_1, \ldots, \ell_p \in \mathbb{N}$ für 1) und 2) gibt, dann auch eine solche, die nur aus Zweien und Dreien besteht.*

Falls bislang eine 5 in der Lösung auftaucht, so kann sie gemäß Schritt 1 nicht allein sein. Es ist also noch mindestens eine weitere Zahl vorhanden, die wir gemäß Schritt 4 als eine 2 oder 3 annehmen könnnen.

- Handelt es sich um eine 2, so ersetzen wir diese durch eine 3, streichen die 5 und verringern k um eins. Bedingung 1) bleibt dann wahr, denn die linke Seite wird nur um den Faktor
$$\frac{2 \cdot 5}{3} < 4$$
kleiner, während sich die rechte Seite um den Faktor 4 vermindert. Auch an der Wahrheit von Bedingung 2) ändert sich nichts, da links die Summanden $1+2$ und $1+5$ durch einen neuen Summanden $1+3$ ersetzt werden, was einer Verringerung um 5 gleichkommt — genauso wie auf der rechten Seite.
- Andernsfalls (die weitere Zahl ist eine 3) streichen wir sowohl diese 3 als auch die 5 ersatzlos und vermindern k um zwei. 1) bleibt dann wahr (die rechte Seite wird durch $4^2 = 16$ geteilt, die linke nur durch $3 \cdot 5 = 15$). In Bedingung 2) vermindert sich die linke Seite um $(1+3)+(1+5) = 10$, die rechte auch.

Jetzt schränken wir noch die Häufigkeit der Zweien und Dreien in drei abschließenden Schritten ein:

Schritt 6: *Wenn es eine Lösung $\ell_1, \ldots, \ell_p \in \mathbb{N}$ für 1) und 2) gibt, dann auch eine solche, die — abgesehen von höchstens einer Zwei — nur aus Dreien besteht.*

Sollten nämlich in einer Lösung aus Schritt 5 zwei oder mehr Zweien vorkommen, so streichen wir je zwei Zweien und vermindern k um eins. In 1) werden dann beide Seiten durch 4 geteilt, in 2) vermindert sich die linke Seite zweimal um $1+2$ (also insgesamt um 6), während die rechte Seite gleichzeitig nur um 5 kleiner wird.

Schritt 7: *Wenn es eine Lösung $\ell_1, \ldots, \ell_p \in \mathbb{N}$ für 1) und 2) gibt, dann auch eine solche, die — abgesehen von höchstens einer Zwei — nur aus höchstens vier Dreien besteht.*

Fünf Dreien tragen zum Produkt in 1) den Faktor $3^5 = 243$ und zu der Summe in 2) den Wert $5 \cdot (1+3) = 20$ bei. Wenn also je 5 Dreien gestrichen und gleichzeitig k um 4 gesenkt wird, bleiben beide Bedingungen wahr, da sich dann auch die rechte Seiten in 1) um den Faktor $4^4 = 256$ bzw. in 2) um den Wert $5 \cdot 4 = 20$ vermindern.

Schritt 8: *Wenn es eine Lösung $\ell_1, \ldots, \ell_p \in \mathbb{N}$ für 1) und 2) gibt, dann auch eine solche, die aus höchstens 4 Dreien oder aber aus einer Zwei und höchstens zwei Dreien besteht.*

Gegenüber Schritt 7 ist hierfür nur noch zu ergänzen, dass auch eine Zwei und drei Dreien weggelassen werden können, wenn man nämlich zugleich k um drei senkt. In 1)

wird die linke Seite durch den Faktor $2 \cdot 3 \cdot 3 \cdot 3 = 54$ geteilt, die rechte aber sogar durch $4^3 = 64$. In 2) vermindert sich die linke Summe um $(1 + 2) + 3 \cdot (1 + 3) = 15$, die rechte Seite wegen $5 \cdot 3 = 15$ ebenfalls.

Nun sind wir soweit, dass wir Lemma B.3.6 beweisen können. Wir haben gezeigt, dass — wenn die Ungleichungen 1) und 2) gleichzeitig erfüllbar wären — sogar eine Lösung existieren muss, die den starken Einschränkungen aus Schritt 8 genügt. Wenn man gleichzeitig bedenkt, dass jede Lösung aus mindestens zwei Zahlen bestehen muss (siehe Schritt 1), so kommen für die vereinfachte Lösung überhaupt nur noch fünf Möglichkeiten (bzw. deren Permutationen) in Betracht:

$$\ell_1 = 2 \text{ und } \ell_2 = 3 \quad \text{oder} \quad \ell_1 = 2\,,\ \ell_2 = 3 \text{ und } \ell_3 = 3 \quad \text{oder} \quad \ell_1 = 3 \text{ und } \ell_2 = 3$$
$$\text{oder} \quad \ell_1 = 3\,,\ \ell_2 = 3 \text{ und } \ell_3 = 3 \quad \text{oder} \quad \ell_1 = 3\,,\ \ell_2 = 3\,,\ \ell_3 = 3 \text{ und } \ell_4 = 3\ .$$

Aber alle diese Lösungen sind falsch. Beispielsweise können die drei Dreien die beiden Ungleichungen nicht erfüllen, denn 1) besagt dann

$$3 \cdot 3 \cdot 3 = 27 \geq 4^k\ ,$$

d.h. k müßte kleiner als 3 sein. Gleichzeitig müßte aber wegen 2) auch

$$(1 + 3) + (1 + 3) + (1 + 3) = 12 < 5k$$

gelten, d.h. k dürfte gerade nicht kleiner als 3 sein. Alle anderen „Lösungen“ kann man genauso einfach widerlegen.

Also lassen sich 1) und 2) niemals gleichzeitig erfüllen, und dies besagt, dass die Behauptung des Lemmas richtig sein muss. □

Damit haben wir ein wichtiges Zwischenresultat erzielt:

B.3.7 Korollar Sei $k \in \mathbb{N}$ beliebig. Dann hat jede kontextfreie Grammatik für die Sprache $L_k := \{a^{4^k}\}$ mindestens die Größe $5k$.

Beweis: Sei G eine beliebige kontextfreie Grammatik für L_k. G habe p Regeln, deren rechte Seiten die Längen $\ell_1, \ldots, \ell_p \in \mathbb{N}$ haben. Nach Lemma B.3.5 muss dann

$$\ell_1 \cdot \ell_2 \cdots \ell_p \geq 4^k$$

gelten. Mit Hilfe von Lemma B.3.6 folgt somit

$$|G| = \sum_{i=1}^{p} (1 + \ell_i) \geq 5k$$

und damit die Behauptung. □

Die Schranke in Korollar B.3.7 ist übrigens „scharf“, d.h. es gibt tatsächlich eine Grammatik G für die Sprache L_k mit $|G| = 5k$, nämlich

$$\begin{array}{rcl} S & \to & A_1A_1A_1A_1 \\ A_1 & \to & A_2A_2A_2A_2 \\ A_2 & \to & A_3A_3A_3A_3 \\ & \vdots & \\ A_{k-2} & \to & A_{k-1}A_{k-1}A_{k-1}A_{k-1} \\ A_{k-1} & \to & aaaa \ . \end{array}$$

Wir nähern uns unserem Ziel:

B.3.8 Satz Für jedes $k \geq 1$ existiert eine kontextfreie Sprache L_k mit diesen Eigenschaften:

- L_k lässt sich durch eine kontextfreie Grammatik der Größe $O(k)$ erzeugen.
- Jede kontextfreie Grammatik für L_k in GNF ist jedoch mindestens $\Omega(k^2)$ groß.

Beweis: Wir wählen $\Sigma := \{a_1, \ldots, a_k\}$ als ein Alphabet mit k Symbolen und ferner $L_k := \{a_1, \ldots, a_k\}^{4^k}$ als Sprache aller Wörter der Länge 4^k über diesem Alphabet. L_k lässt sich leicht durch folgende kontextfreie Grammatik erzeugen:

$$\begin{array}{rcl} S & \to & A_1A_1A_1A_1 \\ A_1 & \to & A_2A_2A_2A_2 \\ A_2 & \to & A_3A_3A_3A_3 \\ & \vdots & \\ A_{k-1} & \to & A_kA_kA_kA_k \\ A_k & \to & a_1 \,|\, a_2 \,|\, \ldots \,|\, a_k \ . \end{array}$$

Ihre Größe beträgt

$$5 + \underbrace{5 + 5 + \cdots + 5}_{(k-1)\text{–mal}} + 2k = 7k = O(k) \ ,$$

womit die erste Aussage bewiesen wäre.

Sei nun G_k eine kontextfreie Grammatik für L_k in GNF. Für jedes $i = 1, \ldots, k$ erzeugen wir aus G eine „abgespeckte“ Version G_i, die nur diejenigen Regeln enthält, die auf ihren rechten Seiten mit dem Symbol a_i anfangen. Dann ist G_i immerhin noch in der Lage, das Wort $a_i^{4^k}$ zu erzeugen, denn dieses Wort ist in L_k enthalten und war somit auch in G ableitbar. Alle dabei verwendeten Regeln sind aber auch in G_i verfügbar, denn alle anderen Regeln fangen auf ihren rechten Seiten nicht mit dem Symbol a_i an und kommen somit für eine Ableitung von $a_i^{4^k}$ nicht in Frage.

Andererseits erzeugt G_i auch wirklich nur dieses eine Wort $a_i^{4^k}$. Denn jedes weitere Wort könnte ebenfalls nur aus a_i–Symbolen bestehen (andere Terminalzeichen kommen in den Regeln von G_i ja nicht vor), d.h. ein solches anderes Wort könnte sich von $a_i^{4^k}$ nur durch

seine Länge unterscheiden. Somit wäre es nicht in L_k enthalten, aber trotzdem aus G ableitbar (dort sind ja auch alle Regeln aus G_i vefügbar) — ein Widerspruch.

Also gilt $L(G_i) = \{a_i^{4^k}\}$ für alle $i = 1, \ldots, k$. Aus Korollar B.3.7 folgt dann $|G_i| \geq 5k$, d.h. wir erhalten für G eine Mindestgröße von

$$|G| = \sum_{i=1}^{k} |G_i| \geq \sum_{i=1}^{k} 5k = 5k^2 = \Omega(k^2) \ .$$

Damit ist auch die zweite Behauptung bewiesen. □

B.3.9 Korollar Jedes allgemeine Verfahren zur Umwandlung von kontextfreien Grammatiken in die Greibach–Normalform muss einen quadratischen Größenanstieg in Kauf nehmen, d.h. es wird immer eine kontextfreie Grammatik geben, deren umgewandelte Grammatik quadratisch viel größer als die Ausgangsgrammatik ist.

Beweis: Wir betrachten nochmals die Grammatiken für die L_k–Sprachen aus Satz B.3.8. Dort wurden Grammatiken G_k der Größe $7k$ angegeben, die L_k erzeugen. Im Vergleich dazu hatte jede GNF–Grammatik für L_k mindestens die Größe $5k^2$. Wenn man das Umwandlungsverfahren also mit einer Grammatik G_k startet, so steigt die Größe der erzeugten GNF–Grammatik auf mindestens

$$5k^2 = 5\Big(\frac{|G_k|}{7}\Big)^2 = \frac{5}{49}|G_k|^2 = \Omega(|G_k|^2)$$

an, womit alles gezeigt ist. □

Das bislang beste Umwandlungsverfahren [5] vergrößert eine Ausgangsgrammatik G wie eingangs erwähnt auf bis zu $O(|G|^3)$ bzw. sogar $O(|G|^4)$, je nachdem, ob G kettenregelfrei ist oder nicht. Es verbleibt somit eine „Forschungslücke“, von der man bislang nicht weiß, ob sie durch eine schärfere untere Schranke oder durch ein besseres Verfahren zu schließen ist.

Der Originalartikel [5] ist recht technisch ausgeführt. Ein besser zu verstehender Zugang zu dem Verfahren ist in [29] enthalten.

B.4. Optimalität des Horner–Schemas

Das sogenannte *Horner–Schema* ist uns bereits auf Seite 141 begegnet. Es ging darum, eine binär kodierte Zahl effizient in die übliche Zahlendarstellung zur Basis 10 zu überführen. Das Horner–Schema basierte auf einer wiederholten Ausklammerung der Basis 2, so dass sich der auszuwertende Term als eine ineinander verschachtelte Folge von Produkten und Summen darstellen ließ. Die Dekodierung der Binärzahl $bin(13) = 1101_2$ stellte sich zum Beispiel wie folgt dar:

$$1101_2 = (((1 \cdot 2) + 1) \cdot 2 + 0) \cdot 2 + 1 = 13 \ .$$

Allgemein wird das Horner–Schema zur Auswertung von Polynomen der Form

$$\sum_{i=0}^{n} a_i x^i$$

eingesetzt, indem die Polynomvariable x wiederholt ausgeklammert wird:

$$\sum_{i=0}^{n} a_i x^i = (\ldots(((a_n x^n + a_{n-1})x + a_{n-2})x + a_{n-3})x + \ldots)x + a_0$$

Programmtechnisch lässt sich das Hornerschema also einfach wie folgt realisieren (dabei sei `y` eine zusätzliche Variable, die am Ende das Ergebnis enthält):

```
y := a_n * x + a_n-1;
y := y * x + a_n-2;
y := y * x + a_n-3;
   ...
y := y * x + a_0;
```

Hierbei werden (nur) n Multiplikationen verwendet. Rechnet man das Polynom dagegen auf dem „klassischen“ Weg aus, so benötigt man $2n - 1$ Multiplikationen:

- $n - 1$ Multiplikationen werden für die Bildung der Potenzen x^2, x^3, …, x^n verwendet.
- n weitere Multiplikationen dienen zur Bildung der Produkte dieser Potenzen mit ihren Koeffizienten.

Hinsichtlich der Anzahl der Additionen gibt es keinen Unterschied: In beiden Fällen werden n Additionen benötigt.

Nun sind das Horner–Schema und die „klassische“ Methode nur zwei von evtl. vielen weiteren Methoden, um ein Polynom auszuwerten. Es stellt sich somit die Frage, ob man auf einem anderen Weg die Anzahl der benötigten Multiplikationen und / oder Additionen weiter drücken kann.

Wir betrachten für diese Untersuchung Programme, die sich an die folgenden Rahmenbedingungen halten:

- Formal gesehen soll eine Funktion $f : \mathbb{R}^{n+2} \longrightarrow \mathbb{R}$ mit

  $$f(a_0, a_1, \ldots, a_n, x) := \sum_{i=0}^{n} a_i x^i$$

  auswertet werden. Neben der Zahl x sind also auch alle Koeffizienten $a_0, a_1, \ldots, a_n$ Eingabeparameter.
- Als arithmetische Operationen sind nur Additionen, Subtraktionen und Multiplikationen erlaubt.

- Zwischenergebnisse können gespeichert und beliebig oft wieder verwendet werden.
- Im Programmcode dürfen neben den Argumenten $a_0, a_1, \ldots, a_n$ und x auch beliebige Konstanten verwendet werden.
- Die Anzahl der durchgeführten Operationen hängt nur von n, nicht aber von den konkreten Werten der Argumente $a_0, a_1, \ldots, a_n$ und x ab. Die Zeitkomplexität lässt sich also durch eine Angabe der Form $O(t(n))$ beschreiben, wobei $t : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine geeignet gewählte Komplexitätsfunktion ist.

Wir werden zeigen, dass unter diesen Voraussetzungen das Horner–Schema optimal ist, d.h. für die Auswertung eines allgemeinen Polynoms von Grad n sind insgesamt n Additionen und Subtraktionen als auch n Multiplikationen zwingend erforderlich.

B.4.1 Satz (Ostrowski[1]) Die Auswertung eines Polynoms $f : \mathbb{R}^{n+2} \longrightarrow \mathbb{R}$ mit

$$f(a_0, a_1, \ldots, a_n, x) := \sum_{i=0}^{n} a_i x^i$$

erfordert insgesamt mindestens n Additionen und Subtraktionen.

Beweis: Jede Subtraktion kann durch eine Multiplikation mit -1 und eine anschließende Addition ersetzt werden. Dabei bleibt die Summe der im Programmcode auftretenden Additionen und Subtraktionen gleich. Wir eliminieren nun alle Subtraktionen aus dem Programm und können uns somit auf die verbleibenden Additionen konzentrieren.

Der Algorithmus muss für alle möglichen Eingaben korrekt funktionieren, insbesondere auch für den Fall $x = 1$, bei dem wegen

$$f(a_0, a_1, \ldots, a_n, 1) = \sum_{i=0}^{n} a_i$$

im Endeffekt „nur“ die Summe aller Koeffizienten $a_0, a_1, \ldots, a_n$ berechnet wird. Wir werden jedoch durch eine Induktion über n zeigen, dass allein für die Aufsummierung von $n + 1$ Koeffizienten bereits n Additionen erforderlich sind.

Der Induktionsanfang $n = 0$ ist trivial, denn natürlich kommt man für das „Aufsummieren“ von nur einem Koeffizient mit 0 Additionen aus. Sei also für den Induktionsschritt $n \geq 1$. O.B.d.A. benutze das Programm möglichst wenig Additionen. Dann ist trotzdem zumindest eine Addition erforderlich, denn ohne eine solche Operation kann das Programm nur Ausdrücke der Form

$$c \prod_{i=0}^{n} a_i^{e_i}$$

[1] Alexander M. Ostrowski, *1893 Kiew, †1986 Montagnola, russisch–deutsch–schweizerischer Mathematiker, Professor für Mathematik in Basel.

erzeugen, wobei jeweils c eine von den Argumenten unabhängige Konstante ist. Wird das Programm nun mit Eingaben der Form $a_2 = a_3 = \cdots = a_n = 0$ konfrontiert (d.h. es soll eigentlich nur $a_0 + a_1$ ausgerechnet werden), so sind alle Ausdrücke nutzlos, in denen die Argumente $a_2, a_3, \ldots, a_n$ auftauchen, da diese Ausdrücke dann verschwinden. Als einzig sinnvolle Ausdrücke bleiben also nur Terme der Art

$$c \cdot a_0^{e_0} \cdot a_1^{e_1}$$

übrig. Ein solcher Term kann aber nur für endlich viele Sonderfälle dem korrekten Funktionsergebnis

$$f(a_0, a_1, 0, 0, \ldots, 0, 1) = a_0 + a_1$$

entsprechen. Weiter kann das Programm in $O(t(n))$ viel Zeit nur endlich viele dieser Terme berechnen, insgesamt also nur endlich viele Sonderfälle abdecken. Mindestens eine Addition ist somit unvermeidlich.

Wir betrachten nun die erste im Programm auftretende Addition. Sie summiert zwei Terme der oben dargestellten Art, d.h. sie ist von der Form

$$c_1 \prod_{i=0}^{n} a_i^{d_i} + c_2 \prod_{i=0}^{n} a_i^{e_i} \quad .$$

Mindestens einer der Exponenten (also einer der d_i– und e_i–Werte) ist ungleich 0, ansonsten würde der Term lediglich die Summe der beiden Konstanten c_1 und c_2 ermitteln. Dann aber hätte das Programm ersatzweise auch gleich die Konstante $c := c_1 + c_2$ verwenden und die eine Addition einsparen können — ein Widerspruch zu der Annahme, dass das Programm möglichst wenig Additionen verwendet.

Sei also o.B.d.A. $d_j \neq 0$ für einen passend gewählten Index j. Wir modifizieren das Verfahren nun ein wenig.

- Ganz zu Beginn überschreiben wir das j–te Argument a_j mit der Zahl 0. Das Argument hat also fortan keine Bedeutung mehr.
- Die zuerst durchgeführte Addition ist nun überflüssig, da das linke Produkt verschwindet. Das Ergebnis können wir also überall durch das rechte Produkt

  $$\prod_{i=0}^{n} a_i^{e_i}$$

  ersetzen.

Das so geänderte Programm berechnet nun eine Funktion $g : \mathbb{R}^{n+1} \longrightarrow \mathbb{R}$ mit

$$g(a_0, a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n, x) := \sum_{i \neq j}^{n} a_i x^i \quad .$$

Speziell im Fall $x = 1$ wird also wegen

$$g(a_0, a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n, 1) := \sum_{i \neq j}^{n} a_i$$

die Summe von n Koeffizienten ermittelt. Folglich muss das veränderte Verfahren gemäß Induktionsannahme noch mindestens $n-1$ Additionen aufwenden. Da wir anfangs eine Addition gestrichen haben, besitzt das ursprüngliche Programm eine Addition mehr (insgesamt also mindestens n), was den Induktionsschluss beweist.

Also kann bei der Auswertung eines Polynoms vom Grad n kein Algorithmus, welcher die anfangs erwähnten Rahmenbedingungen erfüllt, mit weniger als n Additionen auskommen. □

Ganz ähnlich gehen wir vor, um den Mindestaufwand von n Multiplikationen zu beweisen. Das hierbei verwendete Argument ist jedoch etwas komplexer und beruht auf einigen Standard–Resultaten der linearen Algebra.

B.4.2 Satz (Pan[1]) Die Auswertung eines Polynoms $f : \mathbb{R}^{n+2} \longrightarrow \mathbb{R}$ mit

$$f(a_0, a_1, \ldots, a_n, x) := \sum_{i=0}^{n} a_i x^i$$

erfordert mindestens n Multiplikationen.

Es ist (überraschenderweise) einfacher, eine noch allgemeinere Aussage zu zeigen. Wir werden herleiten, dass für eine gegebene $(m+1) \times (n+1)$–Matrix A und ein Polynom $s : \mathbb{R} \longrightarrow \mathbb{R}$ beliebigen Grades die Auswertung der Funktion $g : \mathbb{R}^{n+2} \longrightarrow \mathbb{R}$ mit

$$g(a_0, a_1, \ldots, a_n, x) := (1, x, x^2 \ldots, x^m) \cdot A \cdot (a_0, a_1, \ldots, a_n)^t + s(x)$$

mindestens $rang(A) - 1$ Multiplikationen erfordert. Gilt beispielsweise

$$A := \begin{pmatrix} 2 & -1 \\ 1 & 3 \end{pmatrix} \qquad \text{und} \qquad s(x) := 4x^3 - x + 7 \ ,$$

so lautet die zugehörige Funktion

$$g(a_0, a_1, x) := (1, x) \begin{pmatrix} 2 & -1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} + 4x^3 - x + 7 \ ,$$

was ausmultipliziert

$$g(a_0, a_1, x) = 4x^3 + (a_0 + 3a_1 - 1)x + 2a_0 - a_1 + 7$$

entspricht. Zur Auswertung dieser Funktion wird dann wegen

$$\det A = \begin{vmatrix} 2 & -1 \\ 1 & 3 \end{vmatrix} = 2 \cdot 3 - 1 \cdot (-1) = 7 \neq 0 \qquad (\text{also } rang(A) = 2)$$

mindestens eine Multiplikation notwendig sein.

Um diese allgemeine Behauptung zu beweisen, legen wir unseren Fokus auf sogenannte *signifikante Multiplikationen*, die wir zunächst definieren müssen.

[1] Victor Ya. Pan, russischer und US–amerikanischer Mathematiker, Professor für Mathematik in New York.

B.4.3 Definition Sei ein Algorithmus zur Berechnung der obigen Funktion

$$g(a_0, a_1, \ldots, a_n, x) := (1, x, x^2, \ldots, x^m) \cdot A \cdot (a_0, a_1, \ldots, a_n)^t + s(x)$$

gegeben, der sich an die eingangs erwähnten Rahmenbedingungen hält. In die durchgeführte Berechnung fließen also die Argumente $a_0, a_1, \ldots, a_n$ und x sowie im Programm fest kodierte Konstanten ein. Für gegebene Argumente $a_0, a_1, \ldots, a_n, x \in \mathbb{R}$ schauen wir uns nun alle vom Algorithmus nacheinander durchgeführten arithmetischen Operationen an und definieren darauf basierend A– und X–Werte wie folgt:

- Jedes der $n+1$ Argumente $a_0, \ldots, a_n$ ist ein A–Wert.
- Das Argument x ist ein X–Wert.
- Wenn ein A–Wert als Argument in eine Addition, Subtraktion oder Multiplikation einfließt, so ist das Ergebnis ebenfalls ein A–Wert.
- Entsprechend wird auch die Eigenschaft eines X–Wertes von einem Argument auf das Ergebnis vererbt. Insbesondere kann es auch vorkommen, dass eine Summe, eine Differenz oder ein Produkt sowohl einen A– als auch einen X–Wert darstellt, nämlich dann, wenn ein A– und ein X–Wert miteinander verrechnet werden.
- Eine Multiplikation, die gemäß der letzten Anmerkung einen kombinierten A– und X–Wert erzeugt, heißt *signifikant*.

Ein A–Wert hängt also immer von einem der Parameter $a_0, \ldots, a_n$ ab, ein X–Wert von x (außer in „unsinnigen“ Spezialfällen, bei denen ein Ergebnis wieder mit 0 multipliziert wird usw.).

B.4.4 Beispiel Der folgende Programmcode berechnet einige (nicht unbedingt sinnvolle) Zwischenergebnisse und wertet in der letzten Zeile eine signifikante Multiplikation aus:

```
y := a3 + 17;     // y ist ein A–Wert
z := 3 * x;      // z ist ein X–Wert
t := y + a1;      // t ist ein A–Wert
y := t - y;      // t ist ein A–Wert
t := z + t;      // t ist ein kombinierter A– und X–Wert
z := y * z;      // z ist ein kombinierter A– und X–Wert
```

B.4.5 Satz Sei A eine $(m+1) \times (n+1)$–Matrix und $s : \mathbb{R} \longrightarrow \mathbb{R}$ ein Polynom beliebigen Grades. Jeder Algorithmus zur Berechnung der Funktion $g : \mathbb{R}^{n+2} \longrightarrow \mathbb{R}$ mit

$$g(a_0, a_1, \ldots, a_n, x) := (1, x, x^2, \ldots, x^m) \cdot A \cdot (a_0, a_1, \ldots, a_n)^t + s(x)$$

führt mindestens $rang(A) - 1$ signifikante Multiplikationen aus.

Beweis: Wir zeigen die Aussage durch vollständige Induktion über den Rang r von A. Im Fall $r = 0$ oder $r = 1$ ist dies klar, denn trivialerweise werden immer zumindest 0 signifikante Multiplikationen ausgeführt. Der Induktionsanfang ist damit bereits bewiesen. Sei für den Induktionsschritt also $r \geq 2$. Wir begründen zunächst, warum irgendwann überhaupt eine solche Multiplikation auftreten muss.

Ohne eine solche Operation können nur Ausdrücke der Form

$$p(x) + \sum_{i=0}^{n} c_i a_i$$

berechnet werden. Hierbei ist p ein Polynom, dessen Koeffizienten genauso wie alle c_i–Werte aus dem bisherigen Programmablauf resultierende Konstanten sind, also nicht von $a_0, a_1, \ldots, a_n$ abhängen (ansonsten hätte es schon vorher eine signifikante Multiplikation gegeben).

Wegen $r = \mathit{rang}(A) \geq 2$ hat A mindestens die Größe 2×2 (d.h. es gilt $m \geq 1$ und $n \geq 1$). Außerdem muss es für passende Argumente $a_0, a_1, \ldots, a_n$ möglich sein, mittels

$$(b_0, b_1, \ldots, b_m) := A \cdot (a_0, a_1, \ldots, a_n)^t$$

unendlich viele verschiedene Vektoren $(b_0, b_1, \ldots, b_m)$ zu erzeugen, bei denen mindestens einer der Werte $b_1, b_2, \ldots, b_m$ nicht verschwindet (wären nur Vektoren der Form $(b_0, 0, 0, \ldots, 0)$ möglich, so würde offensichtlich $\mathit{rang}(A) \leq 1$ gelten). Jeder b_i–Wert ungleich 0 hängt zudem von den Argumenten $a_0, a_1, \ldots, a_n$ ab, ist also ein A–Wert. Dann aber entstehen bei der Auswertung von dem Teilausdruck

$$(1, x, x^2, \ldots, x^m) \cdot A \cdot (a_0, a_1, \ldots, a_n)^t = \sum_{i=0}^{m} b_i x^i$$

nichtverschwindende Terme der Art

$$b_i \cdot \underbrace{x \cdot x \cdots x}_{i\text{–mal}}$$

mit $i \geq 1$, die Multiplikationen von A– und X–Werten und somit signifikante Multiplikationen darstellen. Die ohne solche signifikanten Multiplikationen darstellbaren Terme können nur in endlich vielen Sonderfällen mit den zu berechnenden Ausdrücken übereinstimmen. Wegen der beschränkten Laufzeit, die von den konkreten Argumenten unabhängig ist, können nur endlich viele solcher Sonderfälle abgefangen werden. Somit erkennen wir, dass im Fall $r \geq 2$ signifikante Multiplikationen unvermeidlich sind.

Die erste solche signifikante Multiplikation ist von der Form

$$\left(p(x) + \sum_{i=0}^{n} c_i a_i\right) \cdot \left(q(x) + \sum_{i=0}^{n} d_i a_i\right) \ ,$$

wobei wieder alle c_i– und d_i–Werte sowie alle Koeffizienten der Polynome p und q von $a_0, a_1, \ldots, a_n$ unabhängige Konstanten sind. Außerdem ist mindestens ein c_i– oder d_i–Wert ungleich Null, da wir ansonsten zwei X–Werte miteinander multiplizieren würden,

was keiner signifikanten Multiplikation entspricht. O.B.d.A. sei $c_j \neq 0$ für einen passend gewählten Index j.

Wir modifizieren nun wieder das Verfahren an einigen Stellen.

- Ganz zu Beginn fügen wir zusätzliche arithmetische Operationen ein, die den Wert

$$c := -\frac{1}{c_j}\Big(p(x) + \sum_{i \neq j} c_i a_i\Big)$$

  berechnen. Das Resultat benutzen wir, um das Original–Argument a_j zu überschreiben, d.h. der ursprünglich übergebene Wert a_j ist ab jetzt ohne Bedeutung. Die Berechnung erzeugt höchstens einen kombinierten A– und X–Wert, kommt aber ohne signifikante Multiplikationen aus. Weiter wird die scheinbar notwendige Division für den Bruch $\frac{1}{c_j}$ nicht wirklich benötigt, da c_j und somit auch der Bruch eine Konstante ist, die im Programm direkt kodiert werden kann. (Gleiches gilt auch für die im weiteren Verlauf auftretenden Brüche.)

- Die signifikante Multiplikation

$$\Big(p(x) + \sum_{i=0}^{n} c_i a_i\Big) \cdot \Big(q(x) + \sum_{i=0}^{n} d_i a_i\Big)$$

  ist nun überflüssig, da der linke Faktor verschwindet. Bei jeder Verwendung des ursprünglichen Resultats können wir ersatzweise den Wert 0 verwenden. Das Programm wird entsprechend modifiziert.

Das so geänderte Programm berechnet nun eine Funktion $h : \mathbb{R}^{n+1} \longrightarrow \mathbb{R}$ mit

$$\begin{aligned} &h(a_0, a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n, x) \\ &:= (1, x, x^2, \ldots, x^m) \cdot A \cdot (a_0, a_1, \ldots, a_{j-1}, c, a_{j+1}, \ldots, a_n)^t + s(x) \ . \end{aligned}$$

Wenn wir die $(n+1)$ Spalten der Matrix A als Vektoren $\ell_0, \ell_1, \ldots, \ell_n \in \mathbb{R}^{n+1}$ notieren, können wir den berechneten Funktionswert $h(a_0, a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n, x)$ auch als

$$(1, x, x^2, \ldots, x^m) \cdot \begin{pmatrix} \uparrow & \uparrow & \ldots & \uparrow \\ \ell_0 & \ell_1 & \ldots & \ell_n \\ \downarrow & \downarrow & \ldots & \downarrow \end{pmatrix} \cdot (a_0, a_1, \ldots, a_{j-1}, c, a_{j+1}, \ldots, a_n)^t + s(x)$$

schreiben. Der dabei auftretende Vektor

$$\begin{pmatrix} \uparrow & \uparrow & \ldots & \uparrow \\ \ell_0 & \ell_1 & \ldots & \ell_n \\ \downarrow & \downarrow & \ldots & \downarrow \end{pmatrix} \cdot (a_0, a_1, \ldots, a_{j-1}, c, a_{j+1}, \ldots, a_n)^t = \sum_{i \neq j} a_i \ell_i + c\ell_j$$

lässt sich wegen

$$c = -\frac{p(x)}{c_j} - \sum_{i \neq j} \frac{c_i}{c_j} a_i$$

zu

$$\sum_{i\neq j} a_i\ell_i - \frac{p(x)}{c_j}\ell_j - \sum_{i\neq j}\frac{c_i}{c_j}a_i\ell_j = -\frac{p(x)}{c_j}\ell_j + \sum_{i\neq j} a_i\Big(\ell_i - \frac{c_i}{c_j}\ell_j\Big)$$

umformulieren. Notieren wir weiter für $i \neq j$ die Vektoren $\ell'_i := \ell_i - \frac{c_i}{c_j}\ell_j$ als Spalten einer neuen Matrix

$$B := \begin{pmatrix} \uparrow & \uparrow & \dots & \uparrow & \uparrow & \dots & \uparrow \\ \ell'_0 & \ell'_1 & \dots & \ell'_{j-1} & \ell'_{j+1} & \dots & \ell'_n \\ \downarrow & \downarrow & \dots & \downarrow & \downarrow & \dots & \downarrow \end{pmatrix} ,$$

so erkennen wir, dass sich die neu berechnete Funktion h insgesamt durch die Vorschrift

$$\begin{aligned} &h(a_0, a_1, \dots, a_{j-1}, a_{j+1}, \dots, a_n, x) \\ &= (1, x, x^2, \dots, x^m) \cdot (B \cdot (a_0, a_1, \dots, a_{j-1}, a_{j+1}, \dots, a_n)^t - \tfrac{p(x)}{c_j}\ell_j) + s(x) \end{aligned}$$

beschreiben lässt. Der beim Ausmultiplizieren entstehende Term

$$-\frac{p(x)}{c_j}(1, x, x^2, \dots, x^m) \cdot \ell_j$$

entspricht dabei einem Polynom über der Variablen x, dessen Koeffizienten nur von festen Konstanten des Programms, nicht aber von den Parametern $a_0, a_1, \dots, a_{j-1}, a_{j+1}, \dots, a_n$ abhängen. Also können wir es zusammen mit dem Polynom $s(x)$ zu einem neuen Polynom $s'(x)$ zusammenfassen und erhalten insgesamt die Berechungsvorschrift

$$\begin{aligned} &h(a_0, a_1, \dots, a_{j-1}, a_{j+1}, \dots, a_n, x) \\ &= (1, x, x^2, \dots, x^m) \cdot B \cdot (a_0, a_1, \dots, a_{j-1}, a_{j+1}, \dots, a_n)^t + s'(x) \ . \end{aligned}$$

Nun schätzen wir die Anzahl der signifikanten Multiplikationen in dem modifizierten Programm ab. Bekanntlich ändert sich der Rang einer Matrix nicht, wenn ein Vielfaches einer Spalte zu einer anderen Spalte hinzuaddiert oder abgezogen wird. Genau so sind für alle $i \neq j$ die Spaltenvektoren $\ell'_i = \ell_i - \frac{c_i}{c_j}\ell_j$ aber entstanden. Folglich gilt für den Rang r der Matrix A die Gleichheit

$$r = \mathit{rang}\begin{pmatrix} \uparrow & \uparrow & \dots & \uparrow \\ \ell_0 & \ell_1 & \dots & \ell_n \\ \downarrow & \downarrow & \dots & \downarrow \end{pmatrix} = \mathit{rang}\begin{pmatrix} \uparrow & \uparrow & \dots & \uparrow & \uparrow & \uparrow & \dots & \uparrow \\ \ell'_0 & \ell'_1 & \dots & \ell'_{j-1} & \ell_j & \ell'_{j+1} & \dots & \ell'_n \\ \downarrow & \downarrow & \dots & \downarrow & \downarrow & \downarrow & \dots & \downarrow \end{pmatrix}$$

und deshalb für die Matrix B (welche die Spalte ℓ_j nicht enthält) die Abschätzung

$$\mathit{rang}(B) = \mathit{rang}\begin{pmatrix} \uparrow & \uparrow & \dots & \uparrow & \uparrow & \dots & \uparrow \\ \ell'_0 & \ell'_1 & \dots & \ell'_{j-1} & \ell'_{j+1} & \dots & \ell'_n \\ \downarrow & \downarrow & \dots & \downarrow & \downarrow & \dots & \downarrow \end{pmatrix} \geq r - 1 \ .$$

Dann aber muss gemäß Induktionsannahme das neue Programm mindestens $r-1$ signifikante Multiplikationen enthalten. Bei der Änderung des Programms haben wir jedoch

keine signifikante Multiplikationen hinzugefügt, sondern im Gegenteil auf eine verzichtet (die erste im bisherigen Programm berechnete signifikante Multiplikation wurde ja von uns gestrichen). Also enthält das bisherige Programm mindestens r solche Multiplikationen.

Damit ist der Induktionsschluss komplett. □

Nun ist es leicht, Satz B.4.2 zu beweisen:

Beweis: Man wähle in Satz B.4.2 für A die $(n+1)$–dimensionale Einheitsmatrix E_{n+1} und lasse das Polynom $s(x)$ verschwinden. Dann schrumpft die Funktion $g : \mathbb{R}^{n+2} \longrightarrow \mathbb{R}$ auf die Auswertung

$$g(a_0, a_1, \ldots, a_n, x) = \sum_{i=0}^{n} a_i x^i$$

zusammen. Wegen $rang(E_{n+1}) = n + 1$ folgt sofort die Behauptung. □

B.4.6 Korollar Das Horner–Schema zur Auswertung eines Polynoms $g : \mathbb{R}^{n+2} \longrightarrow \mathbb{R}$ mit

$$g(a_0, a_1, \ldots, a_n, x) := \sum_{i=0}^{n} a_i x^i$$

arbeitet unter den genannten Rahmenbedingungen optimal.

Beweis: Das Horner–Schema kommt mit je n Additionen und Multiplikationen aus. Weniger Operationen sind nach unseren Erkenntnissen aus den Sätzen B.4.1 und B.4.2 auch nicht möglich. □

Es sei zum Abschluss bemerkt, dass sich *scheinbar* Gegenbeispiele zu Satz B.4.2 finden lassen. Betrachten Sie z.B. das Polynom

$$3x^4 - 3x^3 + 13x^2 + 2x - 10 \ .$$

Wegen

$$3x^4 - 3x^3 + 13x^2 + 2x - 10 = (3x^2 - 2)(x^2 - x + 5)$$

kann es mühelos mit nur drei Multiplikationen ausgewertet werden (eine Multiplikation wird zur Bildung von x^2 benötigt, eine weitere zur Bildung von $3x^2$ und noch eine zur Berechnung des abschließenden Produkts). Dies steht scheinbar im Widerspruch zu Satz B.4.2, welches einen Mindestaufwand von vier Multiplikationen behauptet.

Der Widerspruch löst sich auf, wenn man bedenkt, dass hier nur eine Funktion

$$f(x) := 3x^4 - 3x^3 + 13x^2 + 2x - 10$$

ausgewertet wird, bei der also alle Koeffizienten fest vorgegeben sind und nicht mehr als frei wählbare Parameter in das Ergebnis mit eingehen. In diesem Fall kann man dann das Vorwissen auch gewinnbringend ausnutzen.

Literaturverzeichnis

[1] A. Asteroth, C. Baier, *Theoretische Informatik*, Pearson Studium, 2002.

[2] A. Beutelspacher, J. Schwenk, K.–D. Wolfenstetter, *Moderne Verfahren der Kryptographie*, 6. Auflage, Vieweg+Teubner, 2006.

[3] M. Blum, *A Machine–Independent Theory of the Complexity of Recursive Functions*, J. ACM, 14(2), 322–336, 1967.

[4] N. Blum, *Theoretische Informatik*, Oldenbourg Verlag, 2001.

[5] N. Blum, R. Koch, *Greibach Normal Form Transformation Revisited*, Information and Computation, (150):11-118, 1999.

[6] A. Borodin, *Computational Complexity and the Existence of Complexity Gaps*, J. ACM, 19(1), 158–174, 1972.

[7] Clay Mathematics Institute, *P vs NP Problem*, im Internet unter der Adresse `www.claymath.org` unter dem Reiter „Millennium Problems“ verfügbar.

[8] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Introduction To Algorithms*, 2nd Edition, MIT Press, 2001.

[9] M. Fürer, *Data Structures for Distributed Counting*, J. Computer and System Sciences, 28, 231–243, 1984.

[10] M. R. Garey, D. S. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[11] F. C. Hennie, R. E. Stearns, *Two–Tape Simulation of Multitape Turing Machines*, J. ACM, 13(4), 533–546, 1966.

[12] D. W. Hoffmanm, *Theoretische Informatik*, Hanser Verlag, 2. Auflage, 2011.

[13] J. E. Hopcroft, J. D. Ullman, *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*, Addison Wesley, 1990.

[14] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*, Pearson Studium, Addison Wesley, 2002.

[15] J. E. Hopcroft, W. J. Paul, L. G. Valiant, *On Time versus Space*, J. ACM, 24, 332–337, 1977.

[16] N. Immerman, *Nondeterministic Space is Closed Under Complementation*, SIAM Journal on Computing 17, 935–938, 1988.

[17] D. E. Knuth, *The Art of Computer Programming 3, Sorting and Searching*, Addison-Wesley, 2nd Edition, 1998.

[18] A. Kunert, *LR(k)–Analyse für Pragmatiker*, Technischer Report. Online unter `http://www2.informatik.hu-berlin.de/~kunert/papers/lr-analyse` verfügbar.

[19] Y. Matiyasevich, G. Senizergues, *Decision Problems for Semi–Thue Systems with a few Rules*, Proceedings, 11th Anual IEEE Symposium on Logic in Computer Science, 1996.

[20] K. R. Reischuk, *Einführung in die Komplexitätstheorie*, Teubner, 1990.

[21] H. Rogers, *Theory of Recursive Functions and Effective Computability*, McGraw, 1967.

[22] W. J. Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Computer and System Sciences, 4(2), 177–192, 1970.

[23] U. Schöning, *Theoretische Informatik kurzgefasst*, 4. Auflage, Spektrum, 2001.

[24] J. I. Seiferas, M. J. Fischer, A. R. Meyer, *Refinements of the Nondeterministic Time and Space Hierarchies*, Symposium on Switching and Automata Theory (FOCS), 130–137, 1973.

[25] M. Sipser, *Introduction to the Theory of Computation*, 2nd Edition, Cengage Learning / Thomson, 2005.

[26] R. Szelepcsényi, *The Method of Forcing for Nondeterministic Automata*, Bulletin of the EATCS 33, 96–100, 1987.

[27] G. Vossen, K.–U. Witt, *Grundlagen der Theoretischen Informatik mit Anwendungen*, Vieweg, 2000.

[28] I. Wegener, *Komplexitätstheorie*, Springer, 2003.

[29] A. C. Weisshaar, *Überlegungen zur Umwandlung kontextfreier Grammatiken in die Greibach–Normalform*, Bachelor–Thesis, Hochschule Karlsruhe Technik und Wirtschaft, 2012.

[30] Wikipedia, *Quines*, unter `de.wikipedia.org/wiki/Quine_(Computerprogramm)` online verfügbar.

Index

Symbols

A

B

C

D

E

F

L

M

N

O

P

Q

R

S

T