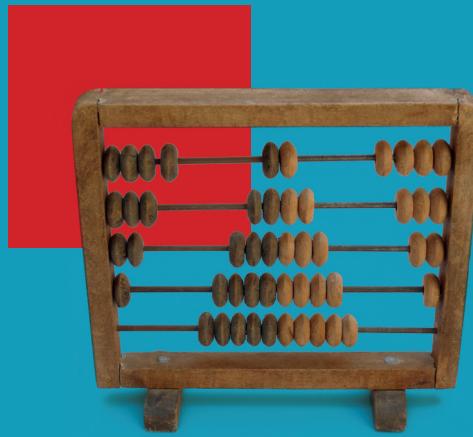


dirk w. HOFFMANN

THEORETISCHE INFORMATIK

2., aktualisierte Auflage



EXTRA: Mit kostenlosem E-Book

HANSER



Bonus: Lösungen zu den Übungsaufgaben,
Glossar und weiteres Zusatzmaterial im Internet



Bleiben Sie einfach auf dem Laufenden:
www.hanser.de/newsletter

Sofort anmelden und Monat für Monat
die neuesten Infos und Updates erhalten.

Dirk W. Hoffmann

Theoretische Informatik

2., aktualisierte Auflage

Mit 295 Bildern, 26 Tabellen und 108 Aufgaben

HANSER

Autor

Prof. Dr. Dirk W. Hoffmann, Hochschule Karlsruhe, Fakultät für Informatik

Alle in diesem Buch enthaltenen Programme, Verfahren und elektronischen Schaltungen wurden nach bestem Wissen erstellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund ist das im vorliegenden Buch enthaltene Programm-Material mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials oder Teilen davon entsteht.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information Der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung – mit Ausnahme der in den §§ 53, 54 URG genannten Sonderfälle –, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2011 Carl Hanser Verlag München

Lektorat: Mirja Werner

Herstellung: Franziska Kaufmann

Satz: Dirk W. Hoffmann, Karlsruhe

Coverconcept: Marc Müller-Bremer, www.rebranding.de, München

Coverrealisierung: Stephan Rönigk

Datenbelichtung, Druck und Bindung: Kösel, Krugzell

Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702

Printed in Germany

ISBN: 978-3-446-42639-9

E-Book-ISBN: 978-3-446-42854-6

www.hanser.de/computer

Vorwort

Für die meisten Menschen ist die Informatik fest mit der Entstehungsgeschichte des Computers verbunden; einer Technik, die von außen betrachtet keinen Grenzen zu unterliegen scheint. Wir erleben seit Jahren eine schier ungebremste Entwicklung und sind längst daran gewöhnt, dass der Computer von heute schon morgen überholt ist. Dass sich hinter der Computertechnik eine tiefgründige Wissenschaft verbirgt, die all die großen Erfolge erst möglich macht, bleibt vielen Menschen verborgen. Die Rede ist von der theoretischen Informatik.

In der Grundlagenausbildung hat die theoretische Informatik ihren festen Platz eingenommen. Viele Studierende begegnen ihr mit gemischten Gefühlen und von manchen wird sie gar als bedrohlich empfunden. Mitverantwortlich für diese Misere sind die historischen Wurzeln der theoretischen Informatik. Entstanden aus der Mathematik, wird sie häufig in einer Präzision dargestellt, die in der Informatik ihresgleichen sucht. Manch ein Leser verirrt sich schnell in einem Gewirr aus Definitionen, Sätzen und Beweisen, das die Sicht auf die eigentlichen Konzepte und Methoden unfreiwillig verdeckt. Dass die theoretische Informatik weder schwer noch trocken sein muss, versuche ich mit diesem Buch zu beweisen.

Die folgenden Kapitel werden von zwei Leitmotiven getragen. Zum einen möchte ich die grundlegenden Konzepte, Methoden und Ergebnisse der theoretischen Informatik vermitteln, ohne diese durch einen zu hohen Abstraktionsgrad zu vernebeln. Hierzu werden die Problemstellungen durchweg anhand von Beispielen motiviert und die Grundideen der komplizierteren Beweise an konkreten Probleminstanzen nachvollzogen. Zum anderen habe ich versucht, den Lehrstoff in vielerlei Hinsicht mit Leben zu füllen. An zahlreichen Stellen werden Sie Anmerkungen und Querbezüge vorfinden, die sich mit der historischen Entwicklung dieser einzigartigen Wissenschaftsdisziplin beschäftigen.

Bei allen Versuchen, einen verständlichen Zugang zu der nicht immer einfachen Materie zu schaffen, war es mir ein Anliegen, keinen Verlust an Tiefe zu erleiden. Das Buch ist für den Bachelor-Studiengang konzipiert und deckt die typischen Lehrinhalte ab, die im Grundstudium an den hiesigen Hochschulen und Universitäten unterrichtet werden.

Vorwort zur zweiten Auflage

Rund zwei Jahre sind seit dem Erscheinen der ersten Auflage vergangen. Seither habe ich von Studenten und Dozenten zahlreiche Zuschriften erhalten, für die ich mich an dieser Stelle herzlich bedanke. Die sehr positive Resonanz hat mich in der Überzeugung bestärkt, den richtigen didaktischen Ansatz gewählt zu haben. Mein ganz besonderer Dank gilt Herrn Prof. Dr. Wolfgang

Ertel von der Hochschule Ravensburg-Weingarten. Die zahlreichen erfrischenden Diskussionen, die ich mit ihm führen durfte, haben mir wertvolle Hinweise und Denkanstöße geliefert, wie das Manuskript weiter verbessert werden kann. Genauso herzlich möchte ich mich bei Herrn Prof. Dr. Hans Werner Lang von der Fachhochschule Flensburg bedanken. Von ihm stammt die Konstruktionsidee für die Grammatik auf Seite 191.

Die vor Ihnen liegende zweite Auflage ist von vielen kleinen und größeren Änderungen geprägt. Allem anderen voran habe ich die Neuauflage dazu genutzt, Fehler auszumerzen, die sich unbemerkt in die Erstauflage gemogelt hatten. Zum anderen habe ich die Gelegenheit am Schopf gepackt, das Manuskript an mehreren Stellen umzuschreiben oder zu erweitern. Tiefergehende inhaltliche Änderungen finden Sie im Kapitel *Logik* und im Kapitel *Komplexitätstheorie*. Die Kapitel *Formale Sprachen* und *Endliche Automaten* habe ich um den Satz von Myhill-Nerode ergänzt, den einige Leser in der ersten Auflage schmerzlich vermissten. Auch die Gesamtzahl der Übungsaufgaben ist in dieser Auflage merklich gewachsen. Eine weitere Änderung ist unscheinbar, zieht sich aber durch das gesamte Buch: Die natürlichen Zahlen beginnen jetzt bei 0, und nicht mehr bei 1.

Karlsruhe, im April 2011

Dirk W. Hoffmann

Symbolwegweiser



Definition



Leichte Übungsaufgabe



Satz, Lemma, Korollar



Mittelschwere Übungsaufgabe



Schwere Übungsaufgabe

Lösungen zu den Übungsaufgaben

In wenigen Schritten erhalten Sie die Lösungen zu den Übungsaufgaben:

1. Gehen Sie auf die Seite www.dirkwhoffmann.de/TH
2. Geben Sie den neben der Aufgabe abgedruckten Webcode ein
3. Die Musterlösung wird als PDF-Dokument angezeigt

Inhaltsverzeichnis

1 Einführung	11
1.1 Was ist theoretische Informatik?	11
1.2 Zurück zu den Anfängen	14
1.2.1 Die Mathematik in der Krise	14
1.2.2 Metamathematik	18
1.2.3 Die ersten Rechenmaschinen	22
1.2.4 Der Computer wird erwachsen	24
1.2.5 Berechenbarkeit versus Komplexität	26
1.3 Theoretische Informatik heute	32
1.4 Übungsaufgaben	34
2 Mathematische Grundlagen	37
2.1 Grundlagen der Mengenlehre	38
2.1.1 Der Mengenbegriff	38
2.1.2 Mengenoperationen	41
2.2 Relationen und Funktionen	44
2.3 Die Welt der Zahlen	52
2.3.1 Natürliche, rationale und reelle Zahlen	52
2.3.2 Von großen Zahlen	55
2.3.3 Die Unendlichkeit begreifen	57
2.4 Rekursion und induktive Beweise	65
2.4.1 Vollständige Induktion	66
2.4.2 Strukturelle Induktion	68
2.5 Übungsaufgaben	70
3 Logik und Deduktion	81
3.1 Aussagenlogik	82
3.1.1 Syntax und Semantik	82
3.1.2 Normalformen	91
3.1.3 Beweistheorie	96
3.1.3.1 Hilbert-Kalkül	98
3.1.3.2 Resolutionskalkül	104
3.1.3.3 Tableaukalkül	109
3.1.4 Anwendung: Hardware-Entwurf	112

3.2	Prädikatenlogik	117
3.2.1	Syntax und Semantik	118
3.2.2	Normalformen	122
3.2.3	Beweistheorie	124
3.2.3.1	Resolutionskalkül	130
3.2.3.2	Tableaukalkül	135
3.2.4	Anwendung: Logische Programmierung	138
3.3	Logikerweiterungen	145
3.3.1	Prädikatenlogik mit Gleichheit	146
3.3.2	Logiken höherer Stufe	147
3.3.3	Typentheorie	149
3.4	Übungsaufgaben	150
4	Formale Sprachen	161
4.1	Sprache und Grammatik	162
4.2	Chomsky-Hierarchie	168
4.3	Reguläre Sprachen	170
4.3.1	Definition und Eigenschaften	170
4.3.2	Pumping-Lemma für reguläre Sprachen	172
4.3.3	Satz von Myhill-Nerode	174
4.3.4	Reguläre Ausdrücke	176
4.4	Kontextfreie Sprachen	179
4.4.1	Definition und Eigenschaften	179
4.4.2	Normalformen	179
4.4.2.1	Chomsky-Normalform	179
4.4.2.2	Backus-Naur-Form	181
4.4.3	Pumping-Lemma für kontextfreie Sprachen	182
4.4.4	Entscheidungsprobleme	186
4.4.5	Abschlusseigenschaften	188
4.5	Kontextsensitive Sprachen	191
4.5.1	Definition und Eigenschaften	191
4.5.2	Entscheidungsprobleme	192
4.5.3	Abschlusseigenschaften	193
4.6	Phrasenstruktursprachen	193
4.7	Übungsaufgaben	195
5	Endliche Automaten	201
5.1	Begriffsbestimmung	202
5.2	Deterministische Automaten	204
5.2.1	Definition und Eigenschaften	204
5.2.2	Automatenminimierung	206
5.3	Nichtdeterministische Automaten	208

5.3.1	Definition und Eigenschaften	208
5.3.2	Satz von Rabin, Scott	210
5.3.3	Epsilon-Übergänge	212
5.4	Automaten und reguläre Sprachen	216
5.4.1	Automaten und reguläre Ausdrücke	217
5.4.2	Abschlusseigenschaften	218
5.4.3	Entscheidungsprobleme	220
5.4.4	Automaten und der Satz von Myhill-Nerode	221
5.5	Kellerautomaten	223
5.5.1	Definition und Eigenschaften	223
5.5.2	Kellerautomaten und kontextfreie Sprachen	226
5.5.3	Deterministische Kellerautomaten	228
5.6	Transduktoren	230
5.6.1	Definition und Eigenschaften	230
5.6.2	Automatenminimierung	231
5.6.3	Automatensynthese	233
5.6.4	Mealy- und Moore-Automaten	234
5.7	Petri-Netze	238
5.8	Zelluläre Automaten	243
5.9	Übungsaufgaben	246
6	Berechenbarkeitstheorie	253
6.1	Berechnungsmodelle	254
6.1.1	Loop-Programme	254
6.1.2	While-Programme	260
6.1.3	Goto-Programme	264
6.1.4	Primitiv-rekursive Funktionen	269
6.1.5	Turing-Maschinen	277
6.1.5.1	Einband-Turing-Maschinen	277
6.1.5.2	Einseitig und linear beschränkte Turing-Maschinen	285
6.1.5.3	Mehrspur-Turing-Maschinen	286
6.1.5.4	Mehrband-Turing-Maschinen	286
6.1.5.5	Maschinenkomposition	288
6.1.5.6	Universelle Turing-Maschinen	289
6.1.5.7	Zelluläre Turing-Maschinen	293
6.1.6	Alternative Berechnungsmodelle	295
6.1.6.1	Registermaschinen	296
6.1.6.2	Lambda-Kalkül	300
6.2	Church'sche These	302
6.3	Entscheidbarkeit	309
6.4	Akzeptierende Turing-Maschinen	312

6.5	Unentscheidbare Probleme	319
6.5.1	Halteproblem	319
6.5.2	Satz von Rice	322
6.5.3	Reduktionsbeweise	325
6.5.4	Das Post'sche Korrespondenzproblem	326
6.5.5	Weitere unentscheidbare Probleme	330
6.6	Übungsaufgaben	333
7	Komplexitätstheorie	341
7.1	Algorithmische Komplexität	342
7.1.1	O-Kalkül	349
7.1.2	Rechnen im O-Kalkül	352
7.2	Komplexitätsklassen	356
7.2.1	P und NP	359
7.2.2	PSPACE und NPSPACE	365
7.2.3	EXP und NEXP	367
7.2.4	Komplementäre Komplexitätsklassen	369
7.3	NP-Vollständigkeit	371
7.3.1	Polynomielle Reduktion	371
7.3.2	P-NP-Problem	372
7.3.3	Satz von Cook	373
7.3.4	Reduktionsbeweise	380
7.4	Übungsaufgaben	386
A	Notationsverzeichnis	397
B	Abkürzungsverzeichnis	401
C	Glossar	403
Literaturverzeichnis		419
Namensverzeichnis		423
Sachwortverzeichnis		425

1 Einführung

„Wir müssen wissen. Wir werden wissen.“

David Hilbert

1.1 Was ist theoretische Informatik?

Kaum eine andere Technologie hat unsere Welt so rasant und nachhaltig verändert wie der Computer. Unzählige Bereiche des täglichen Lebens werden inzwischen von Bits und Bytes dominiert – selbst solche, die noch vor einigen Jahren als elektronikfreie Zone galten. Die Auswirkungen dieser Entwicklung sind bis in unser gesellschaftliches und kulturelles Leben zu spüren und machen selbst vor der deutschen Sprache keinen Halt. Vielleicht haben auch Sie heute schon *gemailt*, *gesimst* oder *gegoogelt* (Abbildung 1.1). Die Digitalisierung unserer Welt ist in vollem Gange und eine Abschwächung der eingeschlagenen Entwicklung zumindest mittelfristig nicht abzusehen.

Die in der Retrospektive einzigartige Evolution der Computertechnik ist eng mit der Entwicklung der Informatik verbunden. Als naturwissenschaftliche Fundierung der Computertechnik untersucht sie die Methoden und Techniken, die eine digitale Welt wie die unsere erst möglich machen. In der gleichen Geschwindigkeit, in der Computer die Welt eroberten, konnte sich die Informatik von einer Nischendisziplin der Mathematik und Elektrotechnik zu einer eigenständigen Grundlagenwissenschaft entwickeln. War sie zu Anfang auf wenige Kernbereiche beschränkt, so präsentierte sich die Informatik mittlerweile als eine breit gefächerte Wissenschaftsdisziplin. Heute existieren Schnittstellen in die verschiedensten Bereiche wie die Biologie, die Medizin und sogar die bildenden Künste.

In Abbildung 1.2 sind die vier Säulen dargestellt, von denen die Informatik gegenwärtig getragen wird. Eine davon ist die theoretische Informatik. Sie beschäftigt sich mit den abstrakten Konzepten und Methoden, die sich hinter den Fassaden moderner Computersysteme verbirgen. Die theoretische Informatik ist vor der technischen Informatik die älteste Kernsäule und hat ihren direkten Ursprung in der Mathematik.

down|loa|den <engl.> (EDV - Daten von einem Computer, aus dem Internet herunterladen); ich habe downgeloadet

googeln (mit Google im Internet suchen); ich googelle;

mailen <engl.> (als E-Mail senden); gemailt

sim|sen (ugs. für eine SMS versenden)



Abbildung 1.1: Die zunehmende Technisierung des Alltagslebens macht auch vor der deutschen Sprache keinen Halt. Im Jahr 2004 schaffte es das neudeutsche Verb *googeln* in den Duden [105]. Dort finden sich auch die Worte *mailen*, *simsen* und *downloaden* wieder.

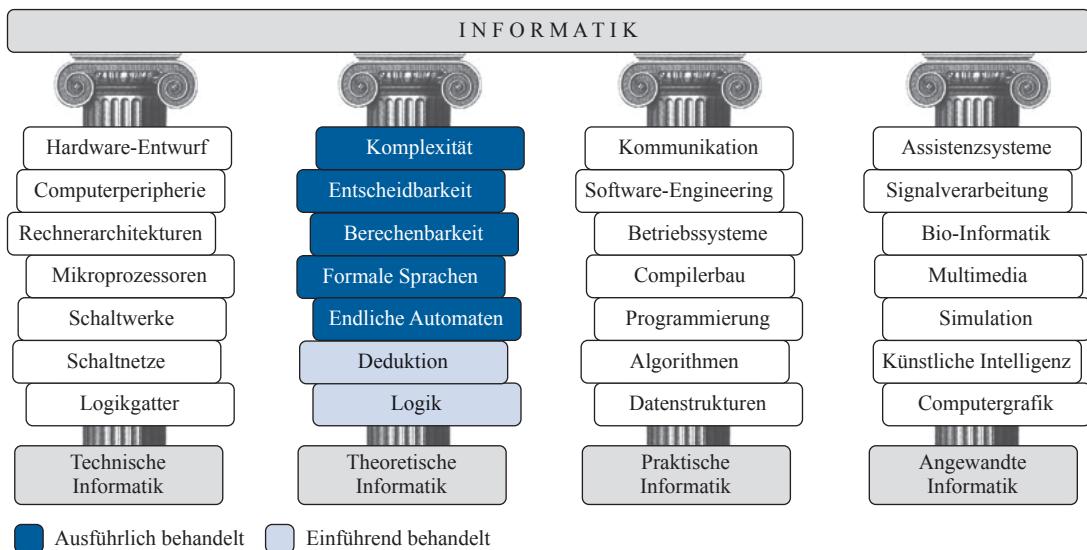


Abbildung 1.2: Die vier Säulen der Informatik

Trotz ihres relativen Alters hat dieser Zweig der Informatik nichts von seiner ursprünglichen Bedeutung verloren. Er bildet das konzeptionelle Fundament, auf dem die anderen Bereiche der Informatik solide ruhen und aus dessen Wissensfundus sie schöpfen.

Betrachten wir die inhaltlichen Themen der modernen theoretischen Informatik genauer, so lassen sich diese in die folgenden Teilgebiete untergliedern (vgl. Abbildung 1.3):

■ Logik und Deduktion (Kapitel 3)

Die Logik beschäftigt sich mit grundlegenden Fragestellungen mathematischer Theorien. Im Mittelpunkt steht die Untersuchung *formaler Systeme (Kalküle)*, in denen Aussagen aus einer kleinen Menge vorgegebener Axiome durch die Anwendung fest definierter Schlussregeln abgeleitet werden. Die Logik spielt nicht nur in der theoretischen Informatik, sondern auch in der technischen Informatik und der Software-Entwicklung eine Rolle. Mit der Aussagenlogik gibt sie uns ein Instrumentarium an die Hand, mit dem wir jede erdenkliche Hardware-Schaltung formal beschreiben und analysieren können. Ferner lässt sich mit der Prädikatenlogik und den Logiken höherer Stufe das Verhalten komplexer Hardware- und Software-Systeme exakt spezifizieren und in Teilen verifizieren.

■ Formale Sprachen (Kapitel 4)

Die Theorie der formalen Sprachen beschäftigt sich mit der Analyse, der Klassifikation und der generativen Erzeugung von Wortmengen. Künstliche Sprachen sind nach festen Regeln aufgebaut, die zusammen mit dem verwendeten Symbolvorrat eine formale Grammatik bilden. Die zugrunde liegende Theorie gibt uns die Methoden und Techniken an die Hand, die für den systematischen Umgang mit modernen Programmiersprachen und dem damit zusammenhängenden Compilerbau unabdingbar sind. Viele Erkenntnisse aus diesem Bereich haben ihre Wurzeln in der Linguistik und stoßen dementsprechend auch außerhalb der Informatik auf reges Interesse.

■ Automatentheorie (Kapitel 5)

Hinter dem Begriff des endlichen Automaten verbirgt sich ein abstraktes Maschinenmodell, das sich zur Modellierung, zur Analyse und zur Synthese zustandsbasierter Systeme eignet. Auf der obersten Ebene untergliedern sich endliche Automaten in Akzeptoren und Transduktoren. Erstere zeigen einen engen Bezug zu den formalen Sprachen, Letztere spielen im Bereich des Hardware-Entwurfs eine dominierende Rolle. Sie sind das mathematische Modell, mit dem sich das zeitliche Verhalten synchron getakteter Digitalschaltungen exakt beschreiben und analysieren lässt.

■ Berechenbarkeitstheorie (Kapitel 6)

Die Berechenbarkeitstheorie beschäftigt sich mit grundlegenden Untersuchungen über die algorithmische Lösbarkeit von Problemen. Die Bedeutung dieses Teilgebiets der theoretischen Informatik ist zweigeteilt. Zum einen wird das gesamte Gebiet der Algorithmentechnik durch die Definition formaler Berechnungsmodelle auf einen formalen Unterbau gestellt. Zum anderen ermöglicht uns die systematische Vorgehensweise, die Grenzen der prinzipiellen Berechenbarkeit auszuloten.

■ Komplexitätstheorie (Kapitel 7)

Während die Berechenbarkeitstheorie Fragen nach der puren Existenz von Algorithmen beantwortet, versucht die Komplexitätstheorie die Eigenschaften einer Lösungsstrategie quantitativ zu erfassen. Algorithmen werden anhand ihres Speicherplatzbedarfs und Zeitverbrauchs in verschiedene Komplexitätsklassen eingeteilt, die Rückschlüsse auf deren praktische Verwertbarkeit zulassen. Die Ergebnisse dieser Theorie beeinflussen den gesamten Bereich der modernen Software- und Hardware-Entwicklung.

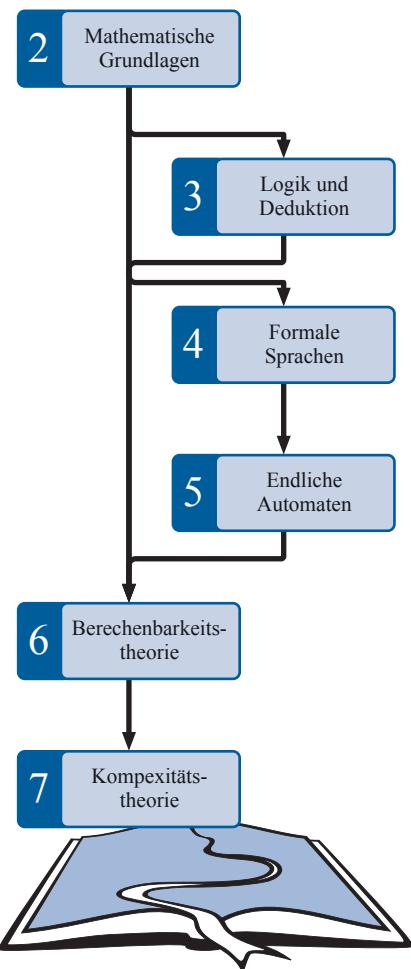


Abbildung 1.3: Kapitelübersicht. Die Pfeile deuten an, wie die einzelnen Kapitel inhaltlich zusammenhängen.

1. Axiom:

„Zu zwei Punkten gibt es genau eine Gerade, auf der sie liegen.“



2. Axiom:

„Jede gerade Strecke zwischen zwei Punkten lässt sich eindeutig verlängern.“



3. Axiom:

„Zu einem Punkt und einer Strecke kann man genau einen Kreis konstruieren.“



4. Axiom:

„Alle rechten Winkel sind gleich.“



5. Axiom:

„Zu einer Geraden und einem Punkt außerhalb der Geraden gibt es genau eine Gerade, die durch den Punkt geht und parallel zur ersten Geraden ist.“



Euklid von Alexandria
(ca. 365 v. Chr. – ca. 300 v. Chr.)

Abbildung 1.4: Die euklidischen Axiome

1.2 Zurück zu den Anfängen

Bevor wir uns ausführlich mit den Begriffen und Methoden der theoretischen Informatik beschäftigen, wollen wir in einem historischen Streifzug herausarbeiten, in welchem Umfeld ihre Teilgebiete entstanden sind und in welchem Zusammenhang sie heute zueinander stehen.

1.2.1 Die Mathematik in der Krise

Die theoretische Informatik hat ihre Wurzeln in der Mathematik. Ihre Geschichte beginnt mit der *Grundlagenkrise*, die Anfang des zwanzigsten Jahrhunderts einen Tiefpunkt in der mehrere tausend Jahre alten Historie der Mathematik markierte. Um die Geschehnisse zu verstehen, die die älteste aller Wissenschaften in ihren Grundfesten erschütterten, wollen wir unseren Blick zunächst auf das achtzehnte Jahrhundert richten. Zu dieser Zeit war die Mathematik schon weit entwickelt, jedoch noch lange nicht die abstrakte Wissenschaft, wie wir sie heute kennen. Fest in der realen Welt verankert, wurde sie vor allem durch Problemstellungen der physikalischen Beobachtung vorangetrieben. Zahlen waren nichts weiter als Messgrößen für reale Objekte und weit von den immateriellen Gedankengebildern der modernen Zahlentheorie entfernt. So wenig wie die Mathematik als eigenständige Wissenschaft existierte, so wenig gab es den reinen Mathematiker.

Im neunzehnten Jahrhundert änderte sich die Sichtweise allmählich in Richtung einer abstrakteren Mathematik. Zahlen und Symbole wurden von ihrer physikalischen Interpretation losgelöst betrachtet und entwickelten sich langsam, aber beharrlich zu immer abstrakter werdenden Größen. Mit der geänderten Sichtweise war es nun möglich, eine Gleichung der Form

$$c^2 = a^2 + b^2$$

völlig unabhängig von ihrer pythagoreischen Bedeutung zu betrachten. In ihrer abstraktesten Interpretation lässt sie sich als mathematisches Spiel begreifen, das uns erlaubt, die linke Seite durch die rechte zuersetzen. Die Variablen a , b und c degradieren in diesem Fall zu inhaltsleeren Größen, die in keinerlei Bezug mehr zu den Seitenlängen eines rechtwinkligen Dreiecks stehen.

Dass es richtig war, das mathematische Gedankengerüst von seiner physikalischen Interpretation zu trennen, wurde durch die Physik selbst untermauert. So machte die zu Beginn des zwanzigsten Jahrhunderts auf-

„Wenn es sich darum handelt, die Grundlagen einer Wissenschaft zu untersuchen, so hat man ein System von Axiomen aufzustellen, welche eine genaue und vollständige Beschreibung derjenigen Beziehungen enthalten, die zwischen den elementaren Begriffen jener Wissenschaft stattfinden. Die aufgestellten Axiome sind zugleich die Definitionen jener elementaren Begriffe und jede Aussage innerhalb des Bereiches der Wissenschaft, deren Grundlagen wir prüfen, gilt uns nur dann als richtig, falls sie sich mittelst einer endlichen Anzahl logischer Schlüsse aus den aufgestellten Axiomen ableiten lässt. Bei näherer Betrachtung entsteht die Frage, ob etwa gewisse Aussagen einzelner Axiome sich untereinander bedingen und ob nicht somit die Axiome noch gemeinsame Bestandteile enthalten, die man beseitigen muss, wenn man zu einem System von Axiomen gelangen will, die völlig voneinander unabhängig sind.“

Vor allem aber möchte ich unter den zahlreichen Fragen, welche hinsichtlich der Axiome gestellt werden können, dies als das wichtigste Problem bezeichnen, zu beweisen, dass dieselben untereinander widerspruchsfrei sind, d.h., dass man aufgrund derselben mittelst einer endlichen Anzahl von logischen Schlüssen niemals zu Resultaten gelangen kann, die miteinander in Widerspruch stehen.“



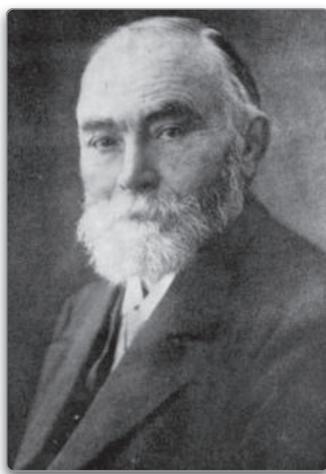
David Hilbert
(1862 – 1943)

Abbildung 1.5: Auszug aus Hilberts historischer Jahrhundertrede auf dem internationalen Kongress der Mathematiker in Paris

keimende Quantenmechanik deutlich, dass die damals wie heute merkwürdig anmutenden Effekte der Elementarteilchenphysik nur mithilfe abstrakter Modelle präzise erfasst werden können. Viele mathematische Konstrukte wie z. B. der *Hilbertraum* oder die *abstrakte Gruppe* konnten nachträglich zur Beschreibung der Natur eingesetzt werden, obwohl diese nichts mit unserer makroskopischen Anschauung gemeinsam haben.

Die zunehmende Abstraktion ließ Raum für Fragen zu, die sich in einer physikalisch gedeuteten Mathematik nicht stellen. Interpretieren wir z. B. die *euklidischen Axiome* (Abbildung 1.4) ausschließlich im Sinne der klassischen Geometrie, so erscheinen sie als reine Selbstverständlichkeit. Sie decken sich mit den Erfahrungen, die wir in der makroskopischen Welt tagtäglich machen und kaum jemand würde auf die Idee kommen, an ihnen zu zweifeln. Entsprechend lange galten die Axiome als unantastbar.

Die Situation ändert sich, sobald wir die Mathematik als ein abstraktes Wechselspiel von Symbolen und Regeln betreiben. Lösen wir uns von der intuitiven Interpretation der euklidischen Axiome, so stellt sich



Gottlob Frege
(1848 – 1925)

Abbildung 1.6: Gottlob Frege. Der im mecklenburgischen Wismar geborene Mathematiker zählt zu den Mitbegründern der mathematischen Logik und der analytischen Philosophie. Im Jahr 1879 eröffnete Frege mit seiner berühmten *Begriffsschrift* einen axiomatischen Zugang zur Logik [35]. Er führt darin die grundlegenden Konzepte und Begriffe ein, die wir auch heute noch in der Prädikatenlogik (Abschnitt 3.2) und den Logiken höherer Stufe (Abschnitt 3.3.2) verwenden. Sein Begriffsgerüst war deutlich weiter entwickelt als die Syllogismen des Aristoteles – der bis dato präzisesten Form des logischen Schließens. Die meiste Zeit seines Lebens war Frege ein überzeugter Verfechter des *Logizismus*. Er vertrat die Auffassung, dass die Mathematik ein Teil der Logik sei. In diesem Sinne müssen sich alle Wahrheiten auf eine Menge von Axiomen zurückführen lassen, die nach Freges Worten „*eines Beweises weder fähig noch bedürftig*“ seien. Er stand damit in einer Gegenposition zu anderen Mathematikern seiner Zeit, von denen viele die Logik als isoliertes Teilgebiet der Mathematik begriffen.

die Frage, ob diese ein vollständiges und widerspruchsfreies Gebilde ergeben. Im Jahr 1899 gelang es David Hilbert, diese Frage positiv zu beantworten. Er postulierte ein Axiomensystem, aus dem sich alle Sätze der euklidischen Geometrie ableiten lassen, ohne die verwendeten Symbole mit einer speziellen Interpretation zu versehen [46].

Inspiriert von den Anfangserfolgen stand die Mathematik um die Jahrhundertwende vollends im Zeichen der axiomatischen Methode. Das Führen eines Beweises wurde als der Prozess verstanden, Sätze durch die Anwendung wohldefinierter Schlussregeln aus einer kleinen Menge vorgegebener, *a priori* als wahr definierter Axiome abzuleiten. Eine spezielle Interpretation der Symbole war hierzu nicht erforderlich und im Grunde genommen auch gar nicht angestrebt. Die Mathematik wurde zu einem Spiel, das nach starren Regeln funktionierte, und das Führen eines Beweises zu einem mechanischen Prozess werden ließ.

Der deutsche Mathematiker David Hilbert war kein Unbekannter. Bereits zu Lebzeiten wurde er als Ikone gefeiert und beeinflusste wie kein anderer die Mathematik des beginnenden zwanzigsten Jahrhunderts. Im Jahr 1900 hielt Hilbert auf dem internationalen Kongress der Mathematiker in Paris eine wegweisende Rede, an der sich die weitere Stoßrichtung der gesamten Mathematik über Jahre hinweg orientieren sollte (vgl. Abbildung 1.5). In seiner Ansprache trug er 23 ungelöste Probleme vor, die für die Mathematik von immenser Wichtigkeit, aber bis dato ungelöst waren.

Bereits an zweiter Stelle forderte Hilbert die Weltgemeinschaft dazu auf, einen Beweis für die Widerspruchsfreiheit der arithmetischen Axiome zu liefern. Das Problem, das Hilbert hier ansprach, war von immenser Wichtigkeit für die gesamte Mathematik, schließlich adressieren die arithmetischen Axiome den vitalen Kern, auf dem alle Teilbereiche dieser Wissenschaft aufbauen. Solange es nicht gelingt, die Widerspruchsfreiheit formal zu beweisen, kann nicht mit Sicherheit ausgeschlossen werden, dass sich z. B. neben dem Theorem $1 + 1 = 2$ auch das Theorem $1 + 1 \neq 2$ aus den Axiomen ableiten lässt. Das verästelte Gebäude der Mathematik würde auf einen Schlag in Trümmern vor uns liegen.

Bereits wenige Jahre nach Hilberts Rede sollte die Wissenschaftsgemeinde erleben, wie real eine solche Gefahr wirklich war. Der deutsche Mathematiker Gottlob Frege (Abbildung 1.6) spürte sie am eigenen Leib, als er 1902 ein formales Axiomensystem für ein Teilgebiet der Mathematik aufstellte, das auf den ersten Blick so intuitiv und einfach erscheint wie kaum ein anderes. Die Rede ist von der *Mengenlehre*. Der zweite Band seiner *Grundgesetze der Arithmetik* schließt mit dem folgenden Nachwort [33, 34]:

„Einem wissenschaftlichen Schriftsteller kann kaum etwas Unerwünschteres begegnen, als dass ihm nach Vollendung einer Arbeit eine der Grundlagen seines Baues erschüttert wird.“

Doch wodurch wurde Frege's Arbeit so grundlegend erschüttert, dass er sein gesamtes Werk gefährdet sah? Die Antwort ist in einem Brief von Bertrand Russell zu finden, den er im Jahr 1902 an Frege schickte – just zu der Zeit, als dieser sein mathematisches Werk vollendete. Aufbauend auf den Begriffen der naiven Mengenlehre definierte Russell die Menge aller Mengen, die sich nicht selbst als Element enthalten:

$$M := \{M' \mid M' \notin M'\}$$

Die Definition von M ist mit der damals verwendeten Mengendefinition von Georg Cantor vereinbar, führt bei genauerer Betrachtung jedoch unweigerlich zu einem Widerspruch. Da für jedes Element a und jede Menge M entweder $a \in M$ oder $a \notin M$ gilt, muss auch M entweder in sich selbst enthalten sein oder nicht. Die Definition von M offenbart uns jedoch das folgende erstaunliche Ergebnis:

$$M \in M \Rightarrow M \notin M, \quad M \notin M \Rightarrow M \in M$$

Der als *Russell'sche Antinomie* bekannte Widerspruch entlarvte den Cantor'schen Mengenbegriff als in sich widersprüchlich und lies Frege's Gedankengerüst wie ein Kartenhaus in sich zusammenstürzen. Die Geschehnisse unterstrichen nachhaltig, wie wichtig eine widerspruchsfreie Fundierung der Mathematik tatsächlich war.

Zu den ersten, die sich der neugeborenen Herausforderung stellten, gehörten die britischen Mathematiker Bertrand Russell und Alfred North Whitehead. Sie starteten den Versuch, ein widerspruchsfreies Fundament zu errichten, auf dem die Mathematik für alle Zeiten einen sicheren Halt finden sollte. Nach zehn Jahren intensiver Arbeit war das Ergebnis greifbar: Die *Principia Mathematica* waren fertiggestellt (vgl. Abbildung 1.7). In einem dreibändigen Werk unternahmen Russell und Whitehead den Versuch, weite Bereiche der Mathematik mit den Mitteln der elementaren Logik formal herzuleiten. Ein großer Teil des Werks ist der *Typentheorie* gewidmet; einer widerspruchsfreien Konstruktion des Mengenbegriffs, mit dem die Art von Selbstbezug vermieden wird, die wenige Jahre zuvor die Mathematik in ihre größte Krise stürzte. Heute gilt die Typentheorie der Principia als überholt. An ihre Stelle tritt der formale axiomatische Aufbau der Mengenlehre durch Ernst Zermelo und Abraham Fraenkel, der die Russell'sche Antinomie ebenfalls beseitigt [32, 111].

Die mathematische Widerspruchsfreiheit ist eine unabdingbare Eigenschaft des mathematischen Schließens. Fehlt sie, so kommt jedes formale System zu einem wertlosen Gedankengebilde. Warum dies so ist, wollen wir im Folgenden kurz begründen. Nehmen wir an, es gebe eine Aussage R , für die sich sowohl R als auch ihre Negation $\neg R$ innerhalb des Kalküls ableiten lassen. Die Situation erscheint wenig bedrohlich, wenn es sich um eine Aussage handelt, die uns nicht weiter interessiert. Eventuell ist R eine Aussage der Russell'schen Art, die uns ohnehin suspekt erscheint. Können wir den Kalkül vielleicht trotzdem sinnvoll einsetzen, wenn wir Aussagen dieser Form schlicht außen vor lassen?

Dass sich widersprüchliche Aussagen in einem Kalkül nicht isoliert betrachten lassen, liegt an den Schlussregeln der klassischen Logik. In Kapitel 3 werden Sie erlernen, wie sich das Theorem

$$\neg P \rightarrow (P \rightarrow Q)$$

aus den elementaren Axiomen der Logik herleiten lässt. In Worten besagt es: Ist P falsch, so folgt aus der Wahrheit von P die Wahrheit von Q . Substituieren wir R für P , so erhalten wir das Theorem

$$\neg R \rightarrow (R \rightarrow Q).$$

Da $\neg R$ eine wahre Aussage ist, lässt sich mithilfe der Abtrennungsregel (*Modus ponens*) das Theorem

$$R \rightarrow Q$$

herleiten. Nach Voraussetzung ist R ebenfalls wahr, so dass eine erneute Anwendung der Abtrennungsregel das Theorem Q hervorbringt. Da die Wahl von Q keinen Einschränkungen unterliegt, können wir eine beliebige Aussage für Q substituieren. Kurzum: In einem widersprüchlichen Kalkül lassen sich ausnahmslose alle Aussagen als wahr beweisen.

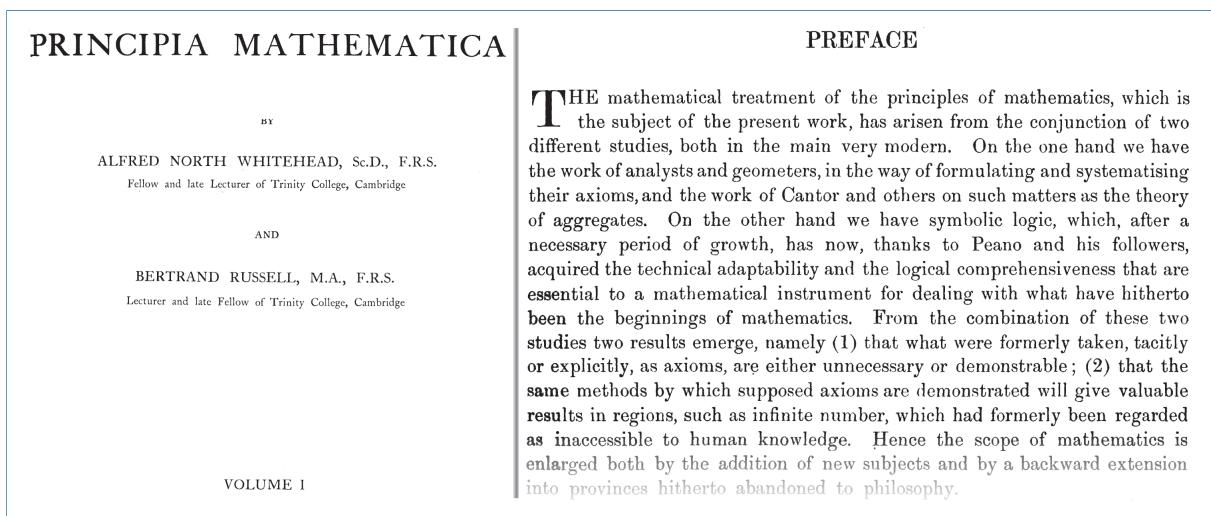


Abbildung 1.7: Die *Principia Mathematica*, erstmals erschienen in den Jahren 1910 bis 1913, ist eines der berühmtesten mathematischen Werke unserer Geschichte. Auf über 1800 Seiten, verteilt auf 3 Bände, unternahmen die Autoren den Versuch, alle mathematischen Erkenntnisse aus einer kleinen Menge von Axiomen systematisch herzuleiten.

Die Principia war in puncto Präzision jedem anderen Werk ihrer Zeit weit voraus. Sie fasste einen mathematischen Beweis als eine Folge von Regelanwendungen auf, durch die eine Aussage in endlich vielen Schritten aus einer festgelegten Menge von Axiomen abgeleitet wurde.

1.2.2 Metamathematik

Durch die zunehmende Beschäftigung mit den verschiedensten formalen Systemen entstand im Laufe der Zeit eine Metamathematik, die sich nicht mit der Ableitung von Sätzen *innerhalb* eines Kalküls beschäftigt, sondern mit Sätzen, die Aussagen *über* den Kalkül treffen. Drei Fragestellungen rückten in das Zentrum des Interesses:

■ Vollständigkeit

Ein formales System heißt *vollständig*, wenn jede wahre Aussage, die in der Notation des Kalküls formuliert werden kann, innerhalb desselben beweisbar ist. Mit anderen Worten: Für jede wahre Aussage P muss es eine endliche Kette von Regelanwendungen geben, die P aus den Axiomen deduziert. Beachten Sie, dass uns ein vollständiger Kalkül nicht preisgeben muss, wie eine solche Kette zu finden ist. Die Vollständigkeit garantiert lediglich deren Existenz.

■ Widerspruchsfreiheit

Ein formales System heißt *widerspruchsfrei*, wenn für eine Aussage P niemals gleichzeitig P und die Negation von P (geschrieben als $\neg P$) ableitet werden kann. Auf die immense Bedeutung der Widerspruchsfreiheit eines Kalküls sind wir weiter oben bereits eingegangen. Erfüllt ein formales System diese Eigenschaft nicht, so könnte es kaum wertloser sein. Es würde uns gestatten, jede beliebige Aussage zu beweisen.

■ Entscheidbarkeit

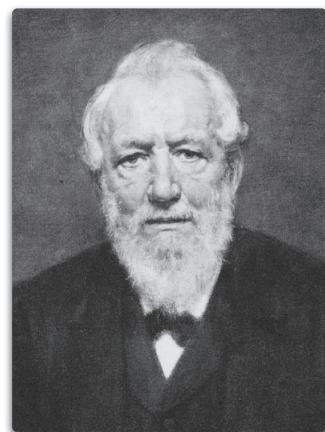
Ein formales System heißt *entscheidbar*, wenn ein systematisches Verfahren existiert, mit dem für jede Aussage entschieden werden kann, ob sie innerhalb des Kalküls beweisbar ist. Hinter der Eigenschaft der Entscheidbarkeit verbirgt sich nichts Geringeres als der Wunsch nach einer mechanisierten Mathematik. Wäre z. B. die Zahlentheorie vollständig und entscheidbar, so ließe sich für jede wahre zahlentheoretische Aussage auf maschinellem Wege ein Beweis konstruieren. Der Traum eines jeden Mathematikers würde wahr.

Hilbert war überzeugt, dass eine vollständige, widerspruchsfreie und entscheidbare Axiomatisierung der Mathematik möglich sei. Im Jahr 1929 wurden seine Hoffnungen durch die Arbeiten des jungen Mathematikers Kurt Gödel zusätzlich genährt, als dieser in seiner Promotionschrift die Vollständigkeit der Prädikatenlogik erster Stufe bewies [36]. Es war also möglich, einen Kalkül zu konstruieren, in dem sich jede wahre prädikatenlogische Formel in endlich vielen Schritten aus den Axiomen ableiten lässt. In diesen Tagen schien es nur eine Frage der Zeit zu sein, bis aus Hilberts Vermutungen Gewissheit werden würde.

1930 war das Jahr, in dem die Entwicklung eine abrupte Kehrtwende nehmen sollte. Am 8. September bekräftigte Hilbert vor der Versammlung Deutscher Naturforscher und Ärzte in seiner Heimatstadt Königsberg seine tiefe Überzeugung, dass es in der Wissenschaft keine unlösabaren Probleme gibt. Ein Auszug aus seiner Rede wurde in Form einer Radioansprache ausgestrahlt. Sie schließt mit den berühmten Worten:

„Wir dürfen nicht denen glauben, die heute mit philosophischer Miene und überlegenem Tone den Kulturuntergang prophezeien und sich in dem Ignorabimus gefallen. Für uns gibt es kein Ignorabimus, und meiner Meinung nach auch für die Naturwissenschaft überhaupt nicht. Statt des törichten Ignorabimus heiße im Gegenteil unsere Lösung: Wir müssen wissen, wir werden wissen.“

„*Ignoramus et ignorabimus.*“
(Wir wissen es nicht und wir werden es niemals wissen)



Emil Heinrich Du Bois-Reymond
(1818 – 1896)

„Für uns gibt es kein Ignorabimus.“
Mit diesem Satz bekräftigte David Hilbert seine Haltung, dass es in den Naturwissenschaften keine unbeweisbaren Wahrheiten gibt. Der Begriff *Ignorabimus* wurde durch den deutschen Gelehrten Emil Heinrich Du Bois-Reymond geprägt. Durch seine Leipziger Rede vor der Versammlung Deutscher Naturforscher und Ärzte löste er im Jahr 1872 einen Richtungsstreit aus, der auf Jahre hinweg zu kontroversen Diskussionen in der Wissenschaftsgemeinde führen sollte. Er vertrat die Meinung, dass Begriffe wie das Bewusstsein niemals mit naturwissenschaftlichen Methoden erklärbar sein werden. Kurzum: Die Wissenschaft besitzt unüberwindbare Grenzen. „*Ich werde jetzt, wie ich glaube, in sehr zwingender Weise durtun, dass nicht allein bei dem heutigen Stand unserer Kenntnis das Bewusstsein aus seinen materiellen Bedingungen nicht erklärbar ist, was wohl jeder zugibt, sondern dass es auch der Natur der Dinge nach aus diesen Bedingungen nicht erklärbar sein wird.*“ [8].

Der Unvollständigkeitsbeweis ist nicht nur aufgrund seiner inhaltlichen Tragweite von Bedeutung. Auch die trickreiche Beweisführung, mit der Gödel sein Resultat erzielte, zeugt von der Tiefe und Gründigkeit des Ergebnisses. Gödel konnte zeigen, dass mathematische Schlussregeln, die Aussagen über Zahlen machen, selbst als Zahl verstanden werden konnten. Damit war es möglich, die Ebene der Zahlentheorie mit ihren Metaebenen zu vermischen. Aussagen sind nichts anderes als Zahlen, die selbst Aussagen über Zahlen tätigen. Auf diese Weise gelang es Gödel, Metaaussagen wie „*Aussage XYZ ist beweisbar*“ innerhalb des Systems zu codieren.

Um die Unvollständigkeit zu beweisen, wandte Gödel einen Trick an. Er konstruierte Aussagen, die auf sich selbst Bezug nehmen und so eine Metaaussage über sich selbst beinhalten. Auf diese Weise gelang es ihm, eine Formel zu konstruieren, die der Metaaussage „*Diese Formel ist nicht beweisbar*“ entspricht. Ist die Formel wahr, so lässt sie sich nicht beweisen und das zugrunde liegende Axiomensystem ist unvollständig. Ist sie falsch, so würde ein Beweis für eine falsche Aussage existieren und das Axiomensystem wäre nicht widerspruchsfrei. Mit anderen Worten: Erfüllt ein Axiomensystem die Eigenschaft der Widerspruchsfreiheit, so ist es zwangsläufig unvollständig.

Der Unvollständigkeitssatz zeigte zudem, dass die Widerspruchsfreiheit eines hinreichend aussagekräftigen formalen Systems nicht innerhalb des Systems selbst bewiesen werden kann. Gödel nutzte aus, dass in einem widersprüchlichen System alle Aussagen wahr sind, d.h., ein Kalkül ist genau dann widerspruchsfrei, wenn es eine einzige Aussage gibt, die nicht bewiesen werden kann. Gödel konnte jedoch zeigen, dass eine Aussage der Form „*es existiert eine unbeweisbare Aussage*“ ebenfalls nicht innerhalb des Systems bewiesen werden kann.

Die Rede ist im Originalton erhalten und ein unschätzbares Dokument der Zeitgeschichte. Sie zeigt nachdrücklich, wie überzeugt Hilbert von der Durchführbarkeit seines ehrgeizigen Programms wirklich war.

Zum Zeitpunkt seiner Rede wusste Hilbert noch nichts von den Ereignissen, die sich am Vortag an anderer Stelle in Königsberg abspielten. Es war die große Stunde eines vierundzwanzigjährigen Mathematikers, der mit der Präsentation seines Unvollständigkeitssatzes die Mathematik aus den Angeln hob. Derselbe Kurt Gödel, der kurze Zeit zuvor die Vollständigkeit der Prädikatenlogik bewies, konnte zeigen, dass die Arithmetik aus fundamentalen Überlegungen heraus unvollständig sein musste. Sein Ergebnis war so allgemein, dass es auf jedes axiomatische System angewendet werden konnte, das ausdrucksstark genug ist, um die Zahlentheorie zu formalisieren. Damit war nicht nur gezeigt, dass der logische Apparat der Principia Mathematica unvollständig war, sondern auch, dass jeder Versuch, die Principia oder ein ähnliches System zu vervollständigen, von Grund auf zum Scheitern verurteilt ist. Gödel versetzte dem Hilbert'schen Programm einen schweren Schlag, von dem es sich nie erholen sollte.

Gödels Arbeit verwies die Mathematik zweifelsohne in ihre Grenzen, ließ jedoch Hilberts dritte Vermutung außen vor. Auch wenn wir nicht in der Lage sind, einen widerspruchsfreien und zugleich vollständigen Kalkül für die Theorie der Zahlen zu konstruieren, so könnte die Frage nach der Entscheidbarkeit eines Kalküls dennoch positiv beantwortet werden. Der Unvollständigkeitsbeweis schließt nicht aus, dass ein systematisches Verfahren existiert, das für jede Aussage bestimmt, ob es innerhalb des Systems beweisbar ist oder nicht.

Die Hoffnung, dass zumindest diese letzte Frage positiv beantwortet werden könnte, wurde im Jahr 1936 vollends zerstört, als der britische Mathematiker Alan Turing seine grundlegende Arbeit *On computable numbers, with an application to the Entscheidungsproblem* der Öffentlichkeit präsentierte (Abbildung 1.8). Turings Arbeit ist für die theoretische Informatik aus zweierlei Gründen von Bedeutung. Zum einen gelang es ihm als einer der Ersten, die Grenzen der Berechenbarkeit formal zu erfassen und das Entscheidungsproblem negativ zu beantworten; die Jagd nach dem mathematischen Perpetuum Mobile war zu Ende. Zum anderen konstruierte Turing für seinen Beweis ein abstraktes Maschinenmodell, das dem Funktionsprinzip moderner Computer bereits sehr nahe kam. Aus heutiger Sicht bildet das gedankliche Gebilde der *Turing-Maschine* die Nahtstelle zwischen der abstrakten Mathematik des frühen zwanzigsten Jahrhunderts und der Welt der realen Rechenmaschinen. In gewissem Sinne ersann Turing den *missing link*, der die Mathematik in Form des Computers zum Leben erweckte.

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO
THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

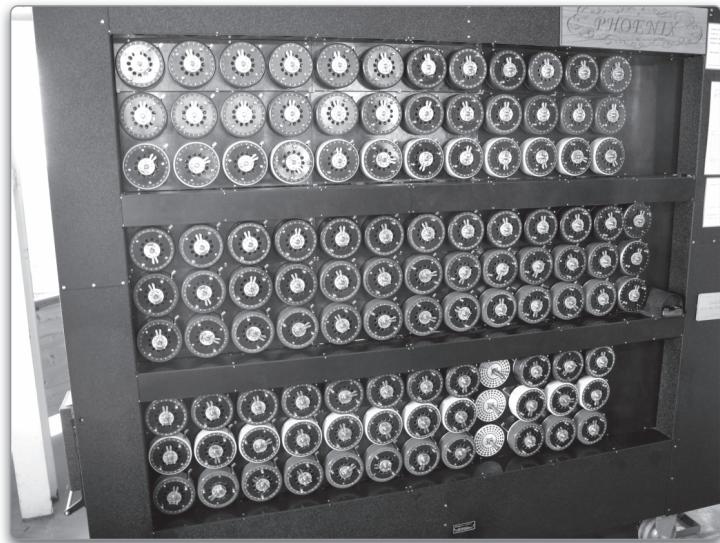
The “computable” numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

Abbildung 1.8: Im Jahr 1936 postulierte Alan Turing ein abstraktes Maschinenmodell, das die theoretische Informatik bis heute prägt [97]. Mit der *Turing-Maschine* legte er das formale Fundament, auf dem weite Teile der modernen Berechenbarkeitstheorien beruhen.

Turings theoretische Ergebnisse bilden den Kern der modernen Berechenbarkeitstheorie, die in der Informatik die gleiche Bedeutung besitzt wie die Relativitätstheorie in der Physik. Sie weist uns fundamentale Grenzen auf, die wir dem technologischen Fortschritt zum Trotz niemals überwinden können.

In der Tat existieren etliche reale Problemstellungen, die mit mechanisierten Verfahren nicht zu lösen sind. Ein Beispiel ist das *Halteproblem*, das wir in Kapitel 6 in all seinen Facetten untersuchen werden. Es besagt grob, dass kein mechanisiertes Verfahren existiert, mit dem wir immer korrekt entscheiden können, ob ein gegebenes Programm unter einer bestimmten Eingabe anhalten oder unendlich lange laufen wird. Die Unentscheidbarkeit des Halteproblems ist der Grund, dass selbst die modernsten Software-Compiler nicht zuverlässig entscheiden können, ob ein Programm eine Endlosschleife enthält oder nicht. Die Suche nach

Abbildung 1.9: Die Turing-Bombe war eine Spezialkonstruktion des britischen Militärs, die im Zweiten Weltkrieg zur Dechiffrierung des deutschen Nachrichtenverkehrs eingesetzt wurde. Die elektromechanisch arbeitende Apparatur bestand aus mehreren rotierenden Trommeln, von denen jeweils 3 eine Einheit bildeten. Jede der insgesamt 36 Einheiten war in der Lage, die Verschlüsselung einer Enigma zu simulieren. Die erste Bombe wurde im März 1940 in Betrieb genommen und im August des gleichen Jahres durch eine leistungsfähigere Variante ersetzt. Mit ihr gelang es den Briten, den deutschen Code in wenigen Stunden zu brechen. Die Abbildung zeigt einen Nachbau der Turing-Bombe aus Bletchley Park.



einem entsprechenden Algorithmus wäre vergebens; die Arbeit von Turing lehrt uns, dass ein solcher aus fundamentalen Überlegungen heraus nicht existieren kann. In Kapitel 6 werden wir uns mit der Berechenbarkeitstheorie im Detail auseinandersetzen und auch den Aufbau und die Funktionsweise der Turing-Maschine ausführlich kennen lernen.

1.2.3 Die ersten Rechenmaschinen

Die folgenden Jahre standen ganz im Zeichen der aufkeimenden Computertechnik. Im Jahr 1941 konstruierte Konrad Zuse mit der Z3 einen voll funktionsfähigen Rechner aus ca. 2000 elektromechanischen Relais. Die Maschine wurde mit einer Taktfrequenz von 5 bis 10 Hertz betrieben, kam auf ein Gesamtgewicht von ca. 1000 kg und besaß eine Leistungsaufnahme von 4000 Watt. Rückblickend gehört der Bau der Z3 zu den bedeutendsten Meilensteinen auf dem Weg in das Informationszeitalter.

In den kommenden Jahren diktierte der politische Umbruch den Lauf der Dinge. Der aufkeimende Zweite Weltkrieg holte die Wissenschaft in die reale Welt zurück – eine Welt, in der Not und Furcht keinen Platz für „mathematische Spielereien“ ließen. Turing, der mit seinem abstrakten Rechnermodell ein paar Jahre zuvor die Grenzen der Berechenbarkeit auslotete, konnte sich den weltgeschichtlichen Ereignissen nicht entziehen. Auch für ihn war die Zeit gekommen, sich aus der theoretischen Gedankenwelt zu verabschieden. Zusammen mit einer Reihe anderer

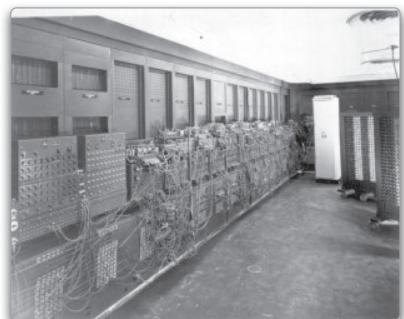
Wissenschaftler begab sich Turing nach Bletchley Park, einem idyllischen Landsitz in der Grafschaft Buckinghamshire. Dort suchte er als Mitglied eines geheimen Konsortiums im Dienst der britischen Regierung nach Möglichkeiten, um den militärischen Verschlüsselungscodes der deutschen Truppen zu brechen.

Während des Zweiten Weltkriegs chiffrierte die deutsche Wehrmacht den Nachrichtenverkehr mithilfe der *Enigma* – einer Maschine, die auf dem Prinzip der *polyalphabetischen Substitution* beruht [6, 28, 48]. Obwohl die Funktionsweise zunächst primitiv wirkt, hätte die manuelle Dechiffrierung viele Monate in Anspruch genommen.

Um den deutschen Nachrichtenverkehr zeitnah zu entschlüsseln, konstruierten die britischen Wissenschaftler mit der *Turing-Bombe* eine Rechenmaschine, die auf das Brechen des Enigma-Codes spezialisiert war (Abbildung 1.9). Nach anfänglichen Schwierigkeiten gelang es den Briten schließlich, die Funksprüche der deutschen Truppen in nur noch wenigen Stunden zu entschlüsseln. Für Turing hatte der Bau dieser Apparatur einen ganz besonderen Aspekt. Auch wenn sie sich in vielen Punkten von seinem abstrakten Maschinenmodell unterschied, wurde ein großer Teil seiner Idee plötzlich real. Realer als ihm wahrscheinlich selbst lieb war, denn natürlich hatte auch er sich unter den gegebenen Umständen gewünscht, dass eine solche Maschine nie hätte gebaut werden müssen.

Die Turing-Bombe war kein Computer im heutigen Sinne, da sie für die Lösung einer ganz speziellen Aufgabe konzipiert war. Die erste voll funktionsfähige Rechenmaschine, die nahezu allen Definitionen des modernen Computer-Begriffs standhält, war der *Electronic Numerical Integrator and Computer*, kurz ENIAC (Abbildungen 1.10 bis 1.12). Der Rechnerkoloss wurde unter der Leitung von J. Presper Eckert und John W. Mauchly an der Moore School of Electrical Engineering der University of Pennsylvania gebaut und 1946 der Öffentlichkeit vorgestellt [40, 93]. Die ENIAC beeindruckte bereits aufgrund ihrer Größe. Sie bestand aus insgesamt 30 Einheiten, die U-förmig über eine eigens errichtete Halle verteilt angeordnet waren. Die ca. 18.000 verbauten Vakuumröhren der knapp 30 Tonnen wiegenden Apparatur verursachten eine Leistungsaufnahme von sagenhaften 174.000 Watt.

Intern arbeiten alle Computer im Binärsystem, d. h., jede Berechnung lässt sich auf eine Folge von Verknüpfungen der Binärziffern 0 und 1 zurückführen. Formal lassen sich die Vorgänge innerhalb eines Computers mithilfe der *Aussagenlogik* beschreiben – einem Teilgebiet der mathematischen Logik, das bereits sehr gut untersucht war, bevor der erste Computer das Konstruktionslabor verließ. Damit wurde die mathemati-



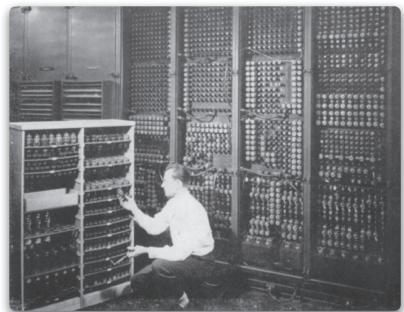
U.S.-Armee-Foto

Abbildung 1.10: Teilansicht der ENIAC (linker Raumhälfte)



U.S.-Armee-Foto

Abbildung 1.11: Teilansicht der ENIAC (rechte Raumhälfte)



U.S.-Armee-Foto

Abbildung 1.12: ENIAC-Techniker beim Austausch einer defekten Trioden-Röhre

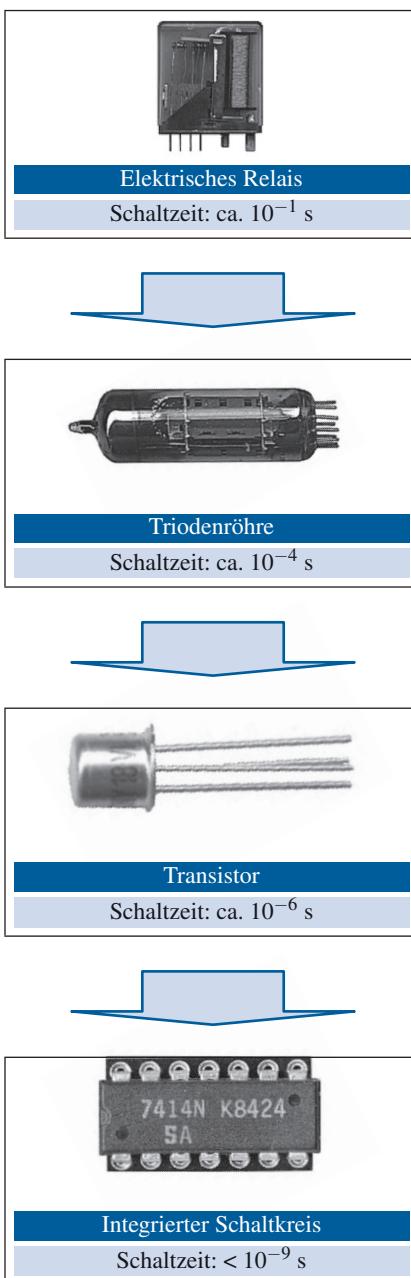


Abbildung 1.13: Von der ersten Rechenmaschine bis zum Supercomputer erlebte die Schaltkreis-Technik drei große Technologiewechsel.

sche Logik, die zu Beginn des Jahrhunderts die große Krise der Mathematik auslöste, plötzlich zur treibenden Kraft in der rasanten Fortentwicklung der Rechnertechnik. Seither ist die Logik nicht von der Informatik zu trennen. Sie war und ist die formale Grundlage für die Spezifikation, den Entwurf und die Analyse digitaler Hardware-Schaltungen. Kapitel 3 ist vollständig der Logik gewidmet und wird die grundlegende Bedeutung für die moderne Informatik aufzeigen.

Natürlich war es von der ENIAC bis zum modernen Computer noch ein langer Weg. Zur damaligen Zeit waren Rechenanlagen mit Triodenröhren bestückt, die den Entwicklern neben ihrer enormen Leistungsaufnahme vor allem wegen ihrer vergleichsweise geringen Zuverlässigkeit Kopfzerbrechen bereiteten. Erst durch die Erfindung des *Transistors* konnte die Computertechnik in Leistungsbereiche vordringen, die wir heute als selbstverständlich erachten (vgl. Abbildung 1.13). Der erste praxistaugliche Transistor wurde im Jahr 1948 von den Bell Laboratories in New York vorgestellt, allerdings sollten noch ein paar Jahre vergehen, bis er die alte Vakuumröhre vollständig verdrängen konnte. Der erste Computer auf Transistor-Basis wurde an der Manchester University Anfang der Fünfzigerjahre gebaut. Dem 1953 fertiggestellten Prototyp folgte eine deutlich erweiterte Variante, die zwei Jahre später in Betrieb genommen werden konnte.

1.2.4 Der Computer wird erwachsen

Der Übergang von der Röhre zum Transistor machte den Weg für die Computer der *zweiten Generation* frei. Die Technologietransition wurde von weiteren wichtigen Entwicklungen begleitet: Steuer- und Rechenwerke nahmen an Komplexität zu und bis dato fortschrittliche Konzepte wie die Verwendung indizierbarer Register oder der Einsatz von Gleitkomma-Hardware gehörten schnell zum Stand der Technik. Ferner hielt der *Ferritkernspeicher* Einzug, der bereits mehrere Kilobyte Daten aufnahm, allerdings in mühevoller Handarbeit gefertigt werden musste.

Die zunehmende Leistungsfähigkeit der konstruierten Computersysteme führte zu einer kontinuierlichen Veränderung in ihrer Bedienung. Wurden die Rechenmaschinen in den Pioniertagen noch direkt auf der Hardware-Ebene programmiert, arbeiteten auf den Computern der zweiten Generation bereits rudimentäre Betriebssysteme. Zeitgleich nahm die Entwicklung von Programmiersprachen an Fahrt auf. 1957 wurde der erste FORTRAN-Compiler ausgeliefert und 1960 die Programmiersprache COBOL verabschiedet. Verschiedene Spezialanwendungen sind auch heute noch in diesen Sprachen programmiert.

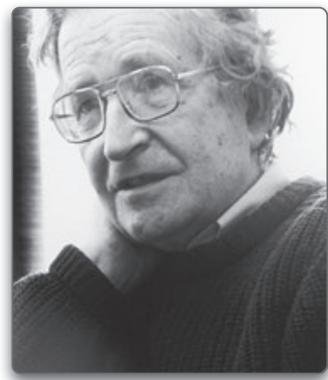
Hand in Hand mit der Evolution der Programmiersprachen wuchsen neue Teilbereiche heran, zu denen auch die Theorie der *formalen Sprachen* gehört. Dieser Teilbereich bündelt alle Methoden und Techniken, die sich mit dem Umgang mit künstlichen Sprachkonstrukten beschäftigen. Er stellt das theoretische Grundgerüst zur Verfügung, auf dem der moderne Compilerbau beruht [3].

Die Forschung auf diesem Gebiet wurde insbesondere durch die Arbeiten von Noam Chomsky (Abbildung 1.14) inspiriert, der im Jahr 1957 die *generative Linguistik* begründete [17]. Diese besagt im Kern, dass natürliche Sprachen elementaren Regeln unterliegen, die von uns unbewusst in der Analyse und Konstruktion von Sätzen eingesetzt werden. Auf diese Weise gelang es Chomsky zu erklären, dass wir richtige und falsche Sätze auch dann unterscheiden können, wenn wir sie noch nie in unserem Leben vorher gehört haben. Darüber hinaus führten Chomskys Arbeiten zu dem Ergebnis, dass die von Menschen gesprochenen Sprachen auf der unteren Ebene fast alle den gleichen Regeln unterliegen. Im Zuge seiner Analyse führte er den Begriff der *generativen Grammatik* ein, der kurze Zeit später Einzug in die theoretische Informatik hielt. Auch wenn Chomsky bei seiner Arbeit stets natürliche Sprachen im Sinn hatte, schuf er genau das Begriffsgerüst, das für den systematischen Umgang mit Computersprachen unabdingbar ist. In Kapitel 4 werden wir uns ausführlicher mit der Theorie der formalen Sprachen beschäftigen und die aus Sicht der Informatik wichtigsten Ergebnisse zusammenfassen.

Eng verwandt mit dem Gebiet der formalen Sprachen ist die *Automatentheorie*. In der Informatik beschreibt der Begriff des *endlichen Automaten* ein abstraktes Modell eines Zustandsübergangssystems, das in vielen Aspekten den Turing-Maschinen ähnelt, aber nicht an deren Ausdrucksstärke herankommt. Endliche Automaten stehen in einer engen Beziehung zu den formalen Sprachen, da sich die meisten Sprachklassen in ein äquivalentes Automatenmodell überführen lassen. In diesem Sinne wird der endliche Automat zu einer konkreten Beschreibungsform einer formalen Sprache. Moderne Compiler setzen Software-Implementierungen endlicher Automaten ein, um die Syntax eines Programmtextes zu überprüfen; Textverarbeitungen bedienen sich dieser Methodik, um Suchtexte in großen Texten effizient aufzuspüren.

Auch im Bereich des Hardware-Entwurfs spielt der endliche Automat eine zentrale Rolle. So lässt sich jede getaktete Hardware-Schaltung in einen funktional äquivalenten endlichen Automaten übersetzen und auf diese Weise einer formalen Analyse unterziehen. Umgekehrt beschreibt jeder endliche Automat eine Hardware-Schaltung und lässt sich automatisiert in ein *Schaltwerk* transformieren. Neben der Aussagenlo-

„There are two central problems in the descriptive study of language. One primary concern of the linguist is to discover simple and 'revealing' grammars for natural language. At the same time, by studying the properties of such successful grammars are clarifying the basic conceptions that underlie them, he hopes to arrive at a general theory of linguistic structure.“ [16]



Noam Chomsky
(1928)

Abbildung 1.14: Viele Erkenntnisse über formale Sprachen gehen auf die Arbeiten des amerikanischen Linguisten Noam Chomsky zurück. Bereits in den frühen Jahren seiner akademischen Laufbahn suchte Chomsky einen theoretischen Zugang zur Linguistik. Ein Kernelement seiner Herangehensweise war die Verwendung formaler Grammatiken – Metasprachen, die einen rekursiven Aufbau der Sprachausdrücke aus einer Menge von Grundelementen erlauben. Der formale Charakter seiner Methodik machte es erstmals möglich, mathematisch präzise Aussagen über linguistische Sprachen zu treffen. Insbesondere gelang es Chomsky, Sprachen anhand gewisser Eigenschaften ihrer Metasprache in verschiedene Typklassen einzuteilen. Die entstehende *Chomsky-Hierarchie* gehört heute zu den wichtigsten Klassifikationsmerkmalen formaler Sprachen.

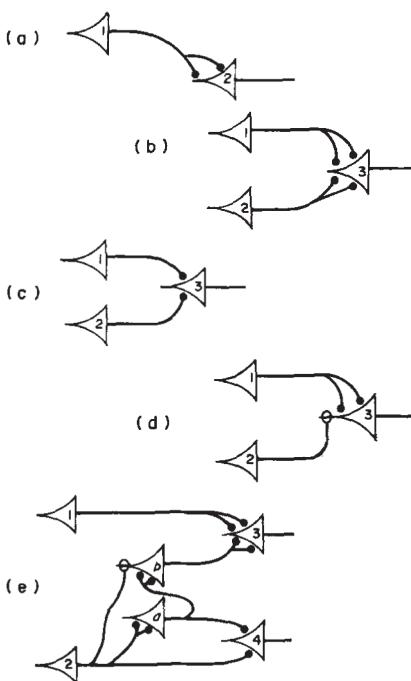


Abbildung 1.15: Nervennetze von McCulloch und Pitts [69]

gik ist die Automatentheorie die zweite tragende Säule des gesamten Hardware-Entwurfs. Aufgrund der großen Bedeutung widmet sich Kapitel 5 ausschließlich diesem Teilgebiet der theoretischen Informatik.

Im Gegensatz zu vielen anderen Begriffen der theoretischen Informatik ist die Entstehungsgeschichte des endlichen Automaten nicht an ein einzelnes Ereignis gebunden. Vielmehr haben wir es mit einem Begriff zu tun, der in einem evolutionären Prozess Schritt für Schritt entwickelt wurde. Zu den bemerkenswertesten Vorgängern zählen die *Nervennetze*, die im Jahr 1943 von Warren McCulloch und Walter Pitts zur Untersuchung neuronaler Strukturen eingeführt wurden (vgl. Abbildung 1.15). Die Arbeit ist für die Entwicklung des Automatenbegriffs in zweierlei Hinsicht von Bedeutung. Zum einen finden wir im Neuronenmodell von McCulloch und Pitts bereits viele Konzepte vor, die auch den modernen endlichen Automaten auszeichnen. Zum andere besaß die Arbeit für viele andere Wissenschaftler eine geradezu inspirierende Wirkung. So auch für den US-amerikanischen Mathematiker Stephen Cole Kleene, der das Nervennetz im Jahr 1956 zu einem allgemeinen Zustandsübergangsmodell weiterentwickelte [60]. In diesem Zusammenhang führte Kleene die *regulären Ausdrücke* ein, die bis heute die Standardnotation für die Beschreibung von Suchmustern bilden. Weiterer bedeutende Beiträge zur Automatentheorie wurden nahezu zeitgleich von George H. Mealy und Edward F. Moore publiziert [70, 75]. Aus Moores Arbeit stammt unter anderem das grafische Transitionsmodell, das wir auch heute noch zur Darstellung endlicher Automaten verwenden (vgl. Abbildung 1.16).

Den vielleicht größten Beitrag zur Automatentheorie lieferten Michael Oser Rabin und Dana Scott im Jahr 1959. Unter anderem sorgten sie für eine klare Abgrenzung zwischen endlichen Automaten und Turing-Maschinen. Hierzu wiesen sie nach, dass sich die zahlreichen, im Laufe der Zeit entstandenen Automatenmodelle allesamt aufeinander abbilden lassen und damit eine geringere Ausdrucksstärke besitzen als die Turing-Maschine. Des Weiteren führten sie das Konzept des Nichtdeterminismus ein, mit dem sie eine völlig neue Denkrichtung im Bereich der Berechenbarkeitstheorie schufen. Im Jahr 1976 wurden Rabin und Scott für ihre bedeutende Arbeit mit dem Turing-Award ausgezeichnet (Tabelle 1.1).

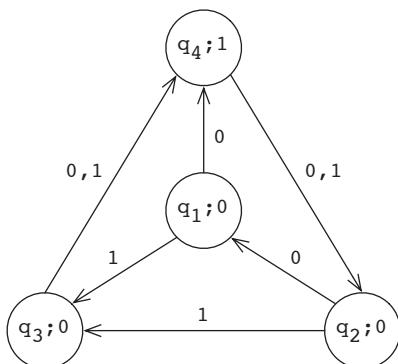


Abbildung 1.16: Endlicher Automat aus der Originalarbeit von Moore [75]

Die zunehmende Rechenleistung machte es in der zweiten Hälfte des zwanzigsten Jahrhunderts möglich, auch komplexe Probleme computer-

Jahr	Preisträger	Jahr	Preisträger	Jahr	Preisträger
1966	Alan J. Perlis	1984	Niklaus E. Wirth		Kristen Nygaard
1967	Maurice V. Wilkes	1985	Richard M. Karp	2002	Leonard M. Adleman
1968	Richard Hamming	1986	John E. Hopcroft, Robert E. Tarjan		Ronald L. Rivest, Adi Shamir
1969	Marvin Minsky	1987	John Cocke	2003	Alan Kay
1970	James H. Wilkinson	1988	Ivan Sutherland	2004	Vinton G. Cerf, Robert E. Kahn
1971	John McCarthy	1989	William Kahan	2005	Peter Naur
1972	Edsger W. Dijkstra	1990	Fernando J. Corbató	2006	Frances E. Allen
1973	Charles W. Bachman	1991	Robin Milner	2007	Edmund M. Clarke, E. Allen Emerson, Joseph Sifakis
1974	Donald E. Knuth	1992	Butler W. Lampson	2008	Barbara Liskov
1975	Allen Newell, Herbert A. Simon	1993	Juris Hartmanis, Richard E. Stearns	2009	Charles P. Thacker
1976	Michael O. Rabin, Dana S. Scott	1994	Edward Feigenbaum, Raj Reddy		
1977	John Backus	1995	Manuel Blum		
1978	Robert W. Floyd	1996	Amir Pnueli		
1979	Kenneth E. Iverson	1997	Douglas Engelbart		
1980	C. Antony R. Hoare	1998	Jim Gray		
1981	Edgar F. Codd	1999	Frederick P. Brooks, Jr.		
1982	Stephen A. Cook	2000	Andrew Chi-Chih Yao		
1983	Ken Thompson, Dennis M. Ritchie	2001	Ole-Johan Dahl,		



Tabelle 1.1: Der Turing-Award existiert seit 1966 und wird seitdem jährlich vergeben. Er ist die höchste Auszeichnung im Bereich der Informatik und hat eine ähnliche Bedeutung wie der Nobelpreis in anderen Wissenschaftsdisziplinen.

gestützt zu bearbeiten. Die Forschung auf dem Gebiet der Algorithmentechnik nahm an Fahrt auf und brachte immer neue Rechenvorschriften hervor, mit denen Probleme aus ganz unterschiedlichen Anwendungsbereichen maschinell gelöst werden konnten. Mit der Fähigkeit, immer größere Eingabemengen verarbeiten zu können, rückten die Laufzeit und der Platzverbrauch eines Programms stärker in das Bewusstsein der Soft- und Hardware-Entwickler. Damit ein Algorithmus für die Lösung praktischer Probleme eingesetzt werden konnte, musste er nicht nur *effektiv* sein, d. h. stets das korrekte Ergebnis berechnen, sondern auch *effizient*. Anders als in den Pioniertagen der theoretischen Informatik, in denen die Berechenbarkeitstheorie zu ihrer vollen Blüte heranreifte, war es nicht mehr länger ausreichend, die prinzipielle Lösbarkeit eines Problems zu zeigen. Eine algorithmische Lösung war faktisch nutzlos, wenn ein Computer mehrere Jahre oder Jahrzehnte für ihre Ausführung benötigte.

■ Problem



■ Algorithmus

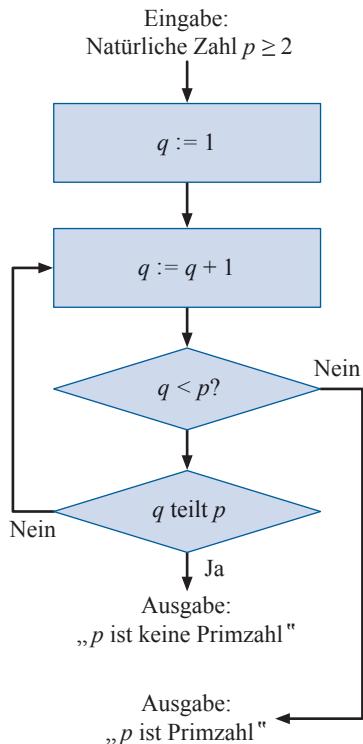


Abbildung 1.17: Das Problem PRIME lässt sich auf einfache Weise lösen, indem wir die Eingabe p nacheinander durch alle Zahlen q mit $2 \leq q < p$ teilen. Geht die Division niemals ohne Rest auf, so ist p eine Primzahl. Während das Programm für kleine Zahlen praktikabel arbeitet, nimmt die Rechenzeit für größere Werte von p drastisch zu. Kurzum: Der abgebildete Algorithmus ist effektiv, aber nicht effizient.

Die praktische Verwertbarkeit von Algorithmen lenkte die Aufmerksamkeit der Wissenschaftsgemeinde langsam, aber sicher von der Berechenbarkeitstheorie weg und hin zur Komplexitätstheorie. Die Grundlagen für die Untersuchung der Laufzeit- und Platzkomplexität von Algorithmen wurden in den Sechzigerjahren geschaffen [21, 26] und nachhaltig durch die Arbeit von Juris Hartmanis und Richard Stearns aus dem Jahr 1965 beeinflusst [42]. Mit ihren mathematisch präzisen Komplexitätsuntersuchungen von Turing-Maschinen legten die beiden US-amerikanischen Computerwissenschaftler den Grundstein für die formale Komplexitätstheorie, wie wir sie heute kennen.

Eine kleine Auswahl konkreter Beispiele soll verdeutlichen, mit welchen Fragestellungen sich dieser Zweig der theoretischen Informatik im Kern befasst. Als Erstes betrachten wir mit PRIME ein Problem der Zahlentheorie, das Mathematiker seit tausenden von Jahren beschäftigt. Für eine beliebige natürliche Zahl p gilt es zu entscheiden, ob es sich um eine Primzahl handelt oder nicht. Dass PRIME lösbar ist, liegt auf der Hand, schließlich können wir die Zahl p als Primzahl identifizieren, indem wir p nacheinander durch alle potenziellen Faktoren dividieren. Abbildung 1.17 fasst die Vorgehensweise in einem Flussdiagramm zusammen. Obwohl der konstruierte Algorithmus für alle Eingabewerte die korrekte Antwort liefert, nimmt die Rechenzeit für größere Eingabewerte dramatisch zu. Bezeichnet n die Anzahl der Bits in der Binärdarstellung von p , so müssen im schlimmsten Fall 2^n Divisionen berechnet werden, bis der Algorithmus terminiert. Selbst wenn wir eine Billion Einzeldivisionen pro Sekunde ausführen könnten, stünde das Ergebnis für eine 128-Bit-Binärzahl erst nach ca. 10^{19} Jahren fest. Wir müssten damit länger auf die Antwort warten, als unser Universum jemals existieren wird.

Als Nächstes wenden wir uns mit SAT einem Problem der Aussagenlogik zu (vgl. Abschnitt 3.2). Konkret geht es um die Frage, ob wir die Variablen einer aussagenlogischen Formel F so mit den Wahrheitswerten 1 (wahr) und 0 (falsch) belegen können, dass F wahr wird. Eine Formel mit dieser Eigenschaft heißt *erfüllbar (satisfiable)*.

Genau wie PRIME lässt sich auch SAT verblüffend einfach lösen. Da eine Formel F nur endlich viele aussagenlogische Variablen enthält und diese ausschließlich die Werte 1 oder 0 annehmen können, ist die Anzahl der möglichen Variablenbelegungen endlich. Somit können wir die Erfüllbarkeit einer Formel F entscheiden, indem wir alle Belegungskombinationen der Reihe nach durchprobieren (vgl. Abbildung 1.18). Genau dann, wenn wir eine Kombination finden, für die F den Wahrheitswert 1 annimmt ($F \equiv 1$), erfährt der Erfüllbarkeitstest eine positive Antwort. Wie schon im Falle von PRIME ist der gefundene Algorith-

mus effektiv, aber nicht effizient. Bezeichnet n die Anzahl der aussagenlogischen Variablen in F , so müssen im schlimmsten Fall alle 2^n Wertekombinationen betrachtet werden, um eine zuverlässige Aussage über die Erfüllbarkeit von F zu treffen.

Das Problem SAT spielt in der theoretischen Informatik eine weit größere Rolle, als es der ein oder andere Leser an dieser Stelle vermuten mag. In Abschnitt 7.3.3 werden wir zeigen, dass viele Algorithmen auf das Erfüllbarkeitsproblem reduziert werden können, indem der Programmablauf in eine aussagenlogische Formel hineincodiert wird. In diesem Sinne wird sich SAT als Universalproblem erweisen, aus dem sich viele Ergebnisse der modernen Komplexitätstheorie direkt ableiten lassen.

Obwohl die gesamte theoretische Informatik in der zweiten Hälfte des zwanzigsten Jahrhunderts durch die Erkenntnis geprägt war, dass ein praktisch verwertbarer Algorithmus nicht nur effektiv, sondern auch effizient arbeiten muss, waren die Fortschritte auf dem Gebiet der Algorithmentechnik unterschiedlicher Natur. Einige Probleme konnten mit Algorithmen gelöst werden, die auch für große Eingabelängen sehr schnell zu einem Ergebnis führten, andere widersetzen sich vehement einer effizienten Lösung. Es schien, als ob die untersuchten Problemstellungen individuelle Schwierigkeitsgrade besäßen, die selbst mit den größten Anstrengungen nicht umgangen werden konnten.

Wissenschaftler reagierten mit der Definition von *Komplexitätsklassen*, in die sich die untersuchten Problemstellungen anhand ihrer bisher bekannten Lösungsstrategien einordnen ließen. Die weiter oben eingeführten Algorithmen zur Lösung von PRIME und SAT sind sogenannte *Exponentialzeitalgorithmen*, da ihre Rechenzeit exponentiell mit dem Größenparameter n ansteigt. Probleme, für die ausschließlich Exponentialzeitalgorithmen bekannt sind, gelten in der Praxis als unlösbar.

Eine ungleich größere Bedeutung besitzen die *Polynomialzeitalgorithmen*, deren Rechenzeit durch ein Polynom $p(n)$ nach oben begrenzt ist. Da Polynome für steigende Werte von n deutlich langsamer gegen unendlich streben als Exponentialfunktionen, setzen viele Experten die Existenz eines Polynomialzeitalgorithmus mit der praktischen Lösbarkeit eines Problems gleich.

Welche Komplexität sich hinter einem spezifischen Problem verbirgt, ist diesem nicht unmittelbar anzusehen und bereits kleine Veränderungen der Fragestellung können zu einer schlagartigen Änderung der Problemkomplexität führen. Wir wollen dieses Phänomen am Beispiel des *Königsberger Brückenproblems* ergründen, das sich mit der Frage beschäftigt, ob in einem gegebenen Graph G ein *Euler-Kreis* existiert.

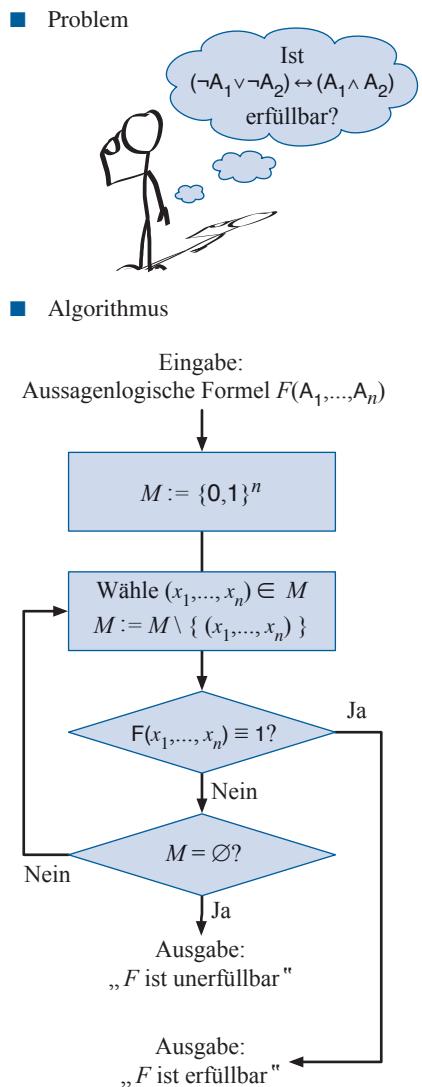
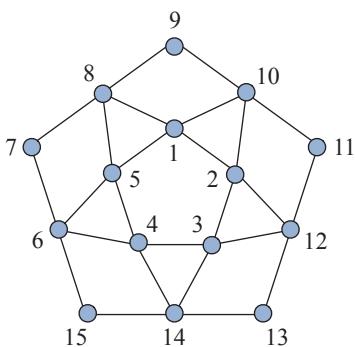
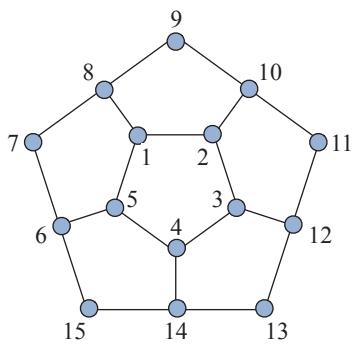


Abbildung 1.18: Das Problem SAT lässt sich lösen, indem der Funktionswert von F nacheinander für sämtliche Variablenbelegungen ausgerechnet wird. Da die Anzahl der Belegungen exponentiell mit der Anzahl der Variablen zunimmt, bleibt die praktische Anwendung dieser Methode auf Formeln mit wenigen Variablen beschränkt. Genau wie im Falle von PRIME ist der Algorithmus effektiv, aber nicht effizient.

■ Graph 1



■ Graph 2



■ Graph 3

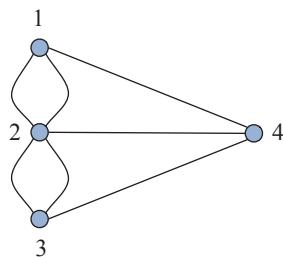


Abbildung 1.19: Hinter dem Königsberger Brückenproblem versteckt sich eine elementare Graph-Eigenschaft, die sich anhand des Euler-Kriteriums effizient nachweisen lässt. Von den dargestellten Graphen erfüllt nur der erste das Euler-Kriterium.

Plakativ gesprochen liegt ein solcher genau dann vor, wenn wir G auf einem Rundweg so durchlaufen können, dass alle Kanten genau einmal besucht werden. 1736 bewies der Schweizer Mathematiker Leonhard Euler, dass ein solcher Kreis genau dann existiert, wenn jeder Knoten eine gerade Anzahl ausgehender Kanten besitzt [7, 30].

Damit können wir das Problem für einen beliebigen Graphen mit n Knoten in linearer Zeit lösen, indem wir nacheinander die Anzahl der ausgehenden Kanten bestimmen. Genau dann, wenn wir in einem der Knoten eine ungerade Anzahl vorfinden, ist die Antwort negativ; einen Euler-Kreis kann es in diesem Fall nicht geben. Abbildung 1.19 demonstriert das Vorgehen anhand zweier Beispielgraphen, von denen nur der erste das Euler-Kriterium erfüllt. In den anderen Graphen existiert kein Euler-Kreis, da gleich mehrere Knoten eine ungerade Anzahl ausgehender Kanten aufweisen.

Euler motivierte seine Ergebnisse, indem er auf die besondere geografische Lage der Stadt Kaliningrad, das frühere Königsberg, zurückgriff. Die ehemalige Hauptstadt Ostpreußens wird durch den Fluss Pregel in vier Gebiete unterteilt, die zu Eulers Zeiten über 7 Brücken miteinander verbunden waren (vgl. Abbildung 1.20). Euler konnte mithilfe seiner Untersuchungen beweisen, dass es unmöglich war, Königsberg auf einem Rundweg zu erkunden, der jede Brücke exakt einmal besucht. Hierzu brauchte er nichts weiter zu tun, als die Kartentopologie in einen Graphen zu übersetzen, in dem jedes Stadtgebiet einem Knoten und jede Brücke einer Kante entspricht. Den entstehenden Graphen haben wir bereits kennen gelernt: Er ist mit dem dritten Beispiel aus Abbildung 1.19 identisch. Da dieser Graph keinen Euler-Kreis besitzt, kann es im ehemaligen Königsberg keinen entsprechenden Rundgang geben.

Für die Komplexitätstheorie ist das Königsberger Brückenproblem von großer Bedeutung, da eine geringfügige Abwandlung ein faktisch unlösbares Problem entstehen lässt. Es wurde im Jahr 1859 von Sir William Hamilton formuliert und ist dem Königsberger Brückenproblem zum Verwechseln ähnlich. Für einen gegebenen Graphen G ist zu entscheiden, ob wir diesen auf einem Rundweg so durchlaufen können, dass alle *Knoten* genau einmal besucht werden. Einen Weg mit dieser Eigenschaft bezeichnen wir als *Hamilton-Kreis*. Bezogen auf die Stadt-Karte von Königsberg lautet das Hamilton-Problem wie folgt: Gibt es einen Rundweg durch die Stadt, so dass kein Stadtteil zweimal betreten wird? In unserem konkreten Beispiel reicht ein gezielter Blick, um einen Hamilton-Kreis zu erkennen. Starten wir beispielsweise im Norden, so gelangen wir über die Pregel-Insel in den südlichen Stadtteil. Anschließend können wir über den östlichen Teil in den Norden zurückkehren, ohne die Insel erneut zu betreten.

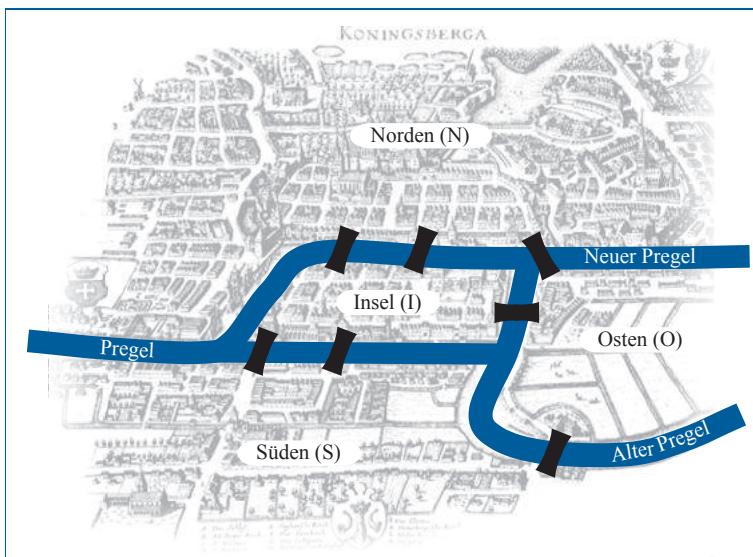


Abbildung 1.20: Im achtzehnten Jahrhundert wurde die Stadt Königsberg durch den Fluss Pregel in vier Gebiete aufgeteilt, die durch sieben Brücken miteinander verbunden waren. Bekannt wurde das Stadtbild durch den Schweizer Mathematiker Leonhard Euler, der es zur Veranschaulichung eines von ihm gelösten Graphenproblems benutzte. Er motivierte seine Arbeit mit der Frage, ob Königsberg auf einem Rundweg erkundet werden kann, auf dem jede Brücke genau einmal überquert wird. Im Jahr 1736 bewies er, dass ein solcher Weg nicht existiert. Mit seinen Untersuchungen begründete Euler die Graphentheorie, die heute sowohl in der theoretischen als auch in der praktischen Informatik ihren festen Platz eingenommen hat.

Obwohl die Lösung des Hamilton-Problems für kleine Graphen wie eine Fingerübung wirkt, ist es noch niemandem gelungen, einen deterministischen Polynomialzeitalgorithmus dafür zu formulieren. Ob ein solcher Algorithmus überhaupt existieren kann, ist gegenwärtig unbekannt. Um es vorwegzunehmen: Die Chancen, dass sich das Hamilton-Problem auf realen Rechenanlagen in Polynomialzeit lösen lässt, stehen aus heutiger Sicht nicht allzu gut, da es in die Klasse der *NP-vollständigen* Probleme fällt.

Die Klasse der NP-vollständigen Probleme ist eine Teilmenge der Klasse NP. Probleme aus NP zeichnen sich dadurch aus, dass eine potenzielle Lösung sehr einfach auf Korrektheit geprüft werden kann, das Finden derselben jedoch ungleich schwerer ist. Am Beispiel des Hamilton-Problems lässt sich diese Eigenschaft besonders gut beobachten. Ist eine Sequenz von Knoten vorgegeben, so können wir leicht feststellen, ob diese einen Hamilton-Kreis beschreibt. Die Berechnung der Knotensequenz gestaltet sich dagegen deutlich komplexer. Die Teilmenge der NP-vollständigen Probleme vereint alle Elemente aus NP, die mächtig genug sind, um damit alle anderen NP-Probleme ebenfalls zu lösen. Sie gehören damit zu den schwersten Problemen überhaupt und eine effiziente Lösung eines einzigen NP-vollständigen Problems würde die effiziente Lösbarkeit aller anderen Probleme aus NP nach sich ziehen.

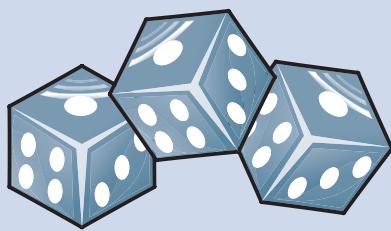
Für diesen Moment soll uns diese vage Beschreibung des Vollständigkeitsbegriffs genügen. Eine genaue Definition werden wir in Kapitel 7

Die Rechenzeit eines Exponentialzeitalgorithmus nimmt so schnell zu, dass seine Praxistauglichkeit deutlich eingeschränkt ist. Um größere Eingaben verarbeiten zu können, wurden für viele Probleme *randomisierte Algorithmen* entwickelt. Diese verfolgen die Grundidee, eine gewünschte Eigenschaft nicht für *alle*, sondern nur für die *meisten* Eingaben zu garantieren. Auf diese Weise entstehen Algorithmen, die aus theoretischer Sicht Lücken aufweisen, in der Praxis aber gute Ergebnisse liefern. Randomisierte Algorithmen lassen sich in zwei Klassen einteilen:

Las-Vegas-Algorithmen berechnen immer das korrekte Ergebnis, konsumieren für vereinzelte Eingaben jedoch überproportional viel Rechenzeit. In diese Gruppe fällt der bekannte Sortieralgorithmus Quicksort. Im statistischen Mittel benötigt dieser $n \cdot \log n$ Operationen, um eine Liste mit n Elementen zu sortieren. Bei einer ungünstigen Werteverteilung steigt die Laufzeit dagegen quadratisch mit der Anzahl der zu sortierenden Elemente an.

Monte-Carlo-Algorithmen garantieren für alle Eingaben die gleiche Laufzeitkomplexität, liefern dafür in manchen Fällen eine falsche Antwort. Mit dem Primzahltest von Robert Solovay und Volker Strassen enthält diese Klasse einen Algorithmus, der vor allem im Zusammenhang mit RSA-Kryptosystemen einen hohen Bekanntheitsgrad erreichte [92].

Der Primzahltest ist deutlich schneller als alle nicht-randomisierten Verfahren, in einzelnen Fällen werden jedoch auch faktorisierbare Zahlen als Primzahl klassifiziert.



nachholen, sobald das zum Verständnis notwendige Begriffsgerüst geschaffen wurde. So viel vorweg: Ob überhaupt ein NP-vollständiges Problem existiert, war lange Zeit unbekannt. Erst im Jahr 1971 gelang Stephen Cook und Leonid Levin unabhängig voneinander der Nachweis, dass das weiter oben skizzierte SAT-Problem NP-vollständig ist [24, 65, 66].

Ein Jahr später wies Richard Karp die NP-Vollständigkeit von 20 weiteren Problemen nach [54] und das im Jahr 1979 publizierte Buch von Michael Garey und David Johnson enthält bereits eine Zusammenfassung von ca. 300 NP-vollständigen Problemen. Können wir für nur ein einzelnes einen Polynomialzeitalgorithmus konstruieren, so sind alle anderen NP-vollständigen Probleme auf einen Schlag ebenfalls in Polynomialzeit lösbar. Damit steht fest: Sollte ein solcher Algorithmus tatsächlich irgendwann gefunden werden, wären seine Auswirkungen bis in alle Teilbereiche der Informatik hinein spürbar.

Die steigende Anzahl gefundener Probleme, die sich vehement einer effizienten Lösbarkeit entziehen, nährt jedoch in vielen Experten die Vermutung, dass es unmöglich ist, einen Polynomialzeitalgorithmus für ein NP-vollständiges Problem zu finden. Ein mathematischer Beweis dafür steht aber bis heute aus und wir dürfen gespannt sein, was die zukünftige Forschung in diesem Bereich an Überraschungen hervorbringen wird.

1.3 Theoretische Informatik heute

Die theoretische Informatik gehört zu den wenigen Teilgebieten der Informationswissenschaft, die über einen gefestigten Stoffumfang verfügen. Insbesondere im Vergleich mit der sich kontinuierlich wandelnden Software-Technik wirken die Erkenntnisse und Methoden überaus stabil. Trotzdem handelt es sich bei der theoretischen Informatik keinesfalls um einen abgeschlossenen Wissenschaftszweig und viele Problemstellungen sind nach wie vor ungelöst.

Eine davon ist die endgültige Klärung der weiter oben diskutierten Frage, ob sich NP-vollständige Probleme auf realen Rechenanlagen in Polynomialzeit lösen lassen. Dieses *P-NP-Problem* ist von so großer Bedeutung, dass es in die Riege der *7 Millennium-Probleme* aufgenommen wurde (Abbildung 1.21). Diese wurden am 24. Mai 2000 in Paris vorgestellt; in derselben Stadt, in der David Hilbert hundert Jahre zuvor in seiner historischen Rede die 23 dringlichsten Probleme der Mathematik formulierte. Die Forschung auf diesem Gebiet ist durchaus attraktiv.

Für jedes der 7 Millennium-Probleme hat das in Cambridge beheimatete Clay Mathematics Institute (CMI) ein Preisgeld von einer Million US-Dollar ausgelobt.

Wie auch immer die Antwort auf das P-NP-Problem lauten wird: Die Folgen sind nicht nur positiver Natur. Gelingt es, die heute mehrheitlich gehegte Vermutung zu bestätigen, dass sich NP-vollständige Probleme nicht in Polynomialzeit lösen lassen, so wäre die Existenz theoretisch beantwortbarer, aber praktisch unlösbarer Fragestellungen gesichert. Wir wüssten dann, dass Probleme wie das Hamilton'sche niemals effizient gelöst werden können. Doch auch die gegenteilige Antwort würde uns Kopfzerbrechen bereiten, da sich viele sicherheitskritische Algorithmen auf die praktische Unlösbarkeit NP-vollständiger Probleme stützen. Sollten wir jemals in der Lage sein, auch nur eine einzige NP-vollständige Fragestellung in Polynomialzeit zu beantworten, so ließen sich neben dem Hamilton-Problem auch Algorithmen aus dem Gebiet der Kryptographie effizient lösen. Die im Internetzeitalter so unabdingbar gewordene Verschlüsselung stände mit einem Schlag auf wackligen Füßen.

Wie Sie sehen, ist die theoretische Informatik weit mehr als mathematische Spielerei. Sie besitzt weitreichende Konsequenzen, die sich in alle Bereiche der Informatik auswirken. Dass die theoretische Informatik keine tote Wissenschaft ist, wird durch Ergebnisse der jüngsten Zeit untermauert. Eines davon bezieht sich auf das weiter oben eingeführte Problem PRIME, also auf die Frage, ob es sich bei einer gegebenen Zahl p um eine Primzahl handelt oder nicht. Obwohl Primzahlen seit tausenden von Jahren intensiv untersucht werden, wurde die Komplexität von PRIME erst im Jahr 2002 vollständig geklärt. In diesem Jahr gelang es den indischen Computerwissenschaftlern Manindra Agrawal, Neeraj Kayal und Nitin Saxena, einen polynomiellen Algorithmus für PRIME zu konstruieren [2]. Der AKS-Algorithmus erzeugte ein Echo weit über die Wissenschaftsgemeinde hinaus. Sogar die New York Times publizierte einen Artikel über die „*New Method Said to Solve Key Problem in Math*“ [86].

Die Arbeit von Agrawal, Kayal und Saxena ist aus zweierlei Gründen von Bedeutung. Zum einen zeigt sie, dass wir mit unseren Vermutungen oft falsch liegen. Viele Wissenschaftler waren vor 2002 der Überzeugung, dass für das Problem PRIME kein Polynomialzeitalgorithmus existiert. Zum anderen demonstriert sie, wie groß unsere Wissenslücken im Fundament der Informatik noch immer sind. Die Ansicht, wir hätten es hier mit einer vollständig untersuchten, in sich abgeschlossenen Wissenschaft zu tun, trügt. Ganz im Gegenteil: Die theoretische Informatik lebt!

„Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. [...] This apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. [...]“

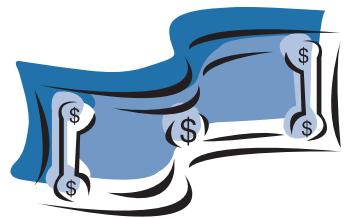


Abbildung 1.21: Für die endgültige Klärung des P-NP-Problems lobte das Clay Mathematics Institute im Jahr 2000 ein Preisgeld von einer Million US-Dollar aus.

1.4 Übungsaufgaben

Aufgabe 1.1



Webcode
1433

Anhand des *Barbier-Paradoxons* lässt sich die Russell'sche Antinomie mit Begriffen des Alltags veranschaulichen.



„Der Barbier von Sevilla rasiert genau diejenigen Männer von Sevilla, die sich nicht selbst rasieren.“

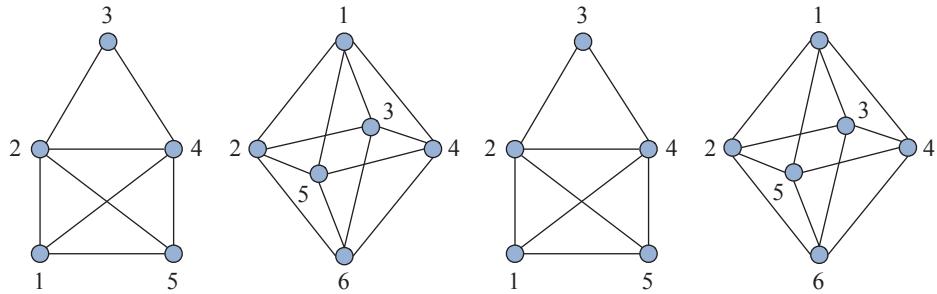
Stellen Sie fest, ob sich der Barbier selbst rasiert.

Aufgabe 1.2



Webcode
1861

Gegeben seien die folgenden Graphen:



- Versuchen Sie, in den beiden linken Graphen einen Euler-Kreis einzuziehen.
- Versuchen Sie, in den beiden rechten Graphen einen Hamilton-Kreis einzuziehen.

Aufgabe 1.3



Webcode
1181

Im Gegensatz zum Königsberger Brückenproblem haben wir in diesem Kapitel keine konkrete Berechnungsvorschrift für die Lösung des Hamilton-Problems angegeben.

- Skizzieren Sie einen Algorithmus, der für jeden Graphen korrekt entscheidet, ob ein gegebener Rundweg ein Hamilton-Kreis ist oder nicht.
- Skizzieren Sie einen Algorithmus, der für jeden Graphen korrekt entscheidet, ob er einen Hamilton-Kreis besitzt oder nicht.
- Wie hängt der Aufwand der formulierten Algorithmen mit der Größe des Graphen zusammen? Könnten Sie Ihren Algorithmus aus Teil b) beschleunigen, wenn Sie die Möglichkeit hätten, Rundwege mit bestimmten Eigenschaften zu erraten?

In diesem Kapitel haben Sie erfahren, wie der Logizismus die Mathematik zu Beginn des zwanzigsten Jahrhunderts verändert hat. Die zuvor physikalisch geprägte Mathematik wurde durch *formale Systeme* verdrängt, die sich aus *Axiomen* und *Regeln* zusammensetzen. Ein Theorem ist bewiesen, wenn es sich durch die Anwendung einer endlichen Folge von Regelanwendungen aus den Axiomen herleiten lässt. In einem solchen *Kalkül* wird das Führen eines mathematischen Beweises zu einem mechanischen Prozess.

Um das Gesagte mit Leben zu füllen, wollen wir ein konkretes formales System betrachten. Es stammt aus Douglas Hofstadters Meisterwerk *Gödel, Escher, Bach* und gibt einen erstklassigen Eindruck von der Grundidee des logischen Schließens [51]:

Aufgabe 1.4**Webcode****1904****■ Axiome**

1. **MI** ist ein Satz

■ Schlussregeln

1. Mit **xI** ist auch **xIU** ein Satz
2. Mit **Mx** ist auch **Mxx** ein Satz
3. In jedem Satz darf **III** durch **U** ersetzt werden
4. In jedem Satz darf **UU** entfernt werden

In dem soeben definierten *MIU-System* lässt sich z. B. der Satz **MUIIU** wie folgt beweisen:

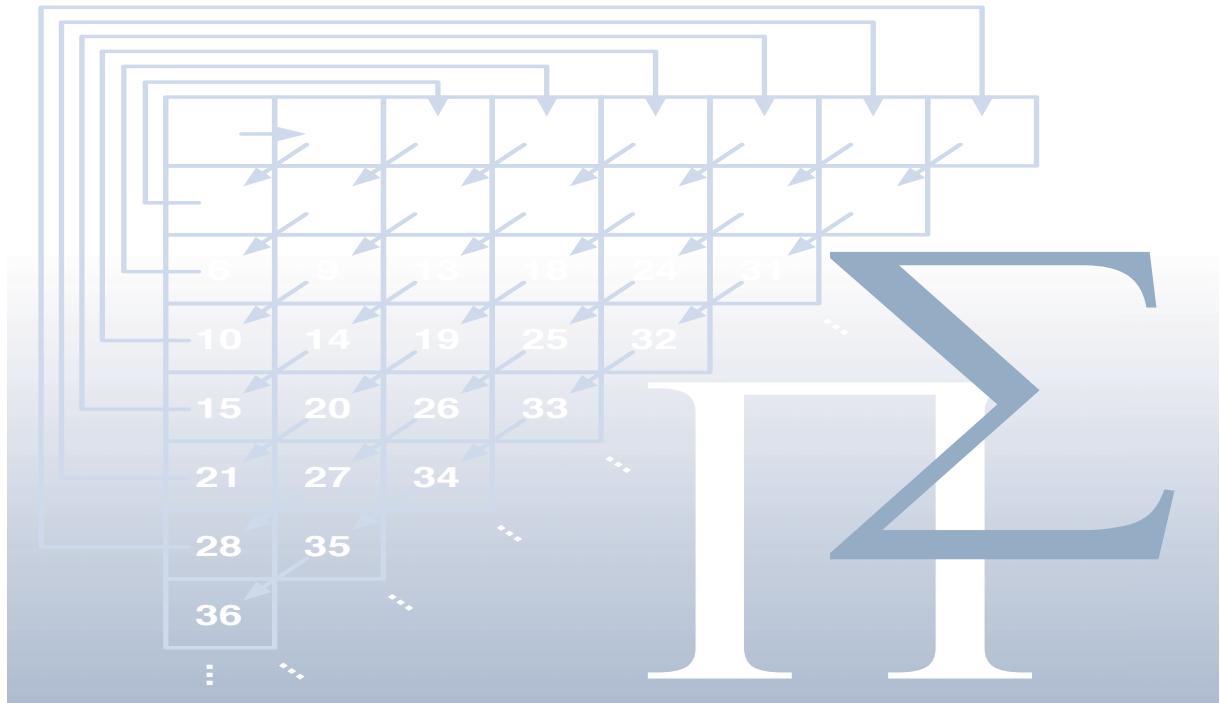
MI	(Axiom)
\Rightarrow III	(Regel 2)
\Rightarrow IIII	(Regel 2)
\Rightarrow MUI	(Regel 3)
\Rightarrow MUIU	(Regel 1)
\Rightarrow MUIUIU	(Regel 2)
\Rightarrow MUIIU	(Regel 4)

- a) Lassen sich die Sätze **MIU** und **MIIIIU** innerhalb des Kalküls ableiten?
- b) Ist die Reihenfolge der angewendeten Regeln immer eindeutig bestimmt?
- c) Versuchen Sie zu ermitteln, ob sich der Satz **MU** aus den Axiomen ableiten lässt. Konstruieren Sie einen Beweis in Form einer konkreten Ableitungssequenz oder begründen Sie, warum ein solcher Beweis nicht existieren kann.

2 Mathematische Grundlagen

In diesem Kapitel werden Sie ...

- die Cantor'sche Definition der Menge ergründen,
 - die grundlegenden Eigenschaften von Relationen und Funktionen kennen lernen,
 - die natürlichen, rationalen und reellen Zahlen untersuchen,
 - den systematischen Umgang mit der Unendlichkeit erlernen,
 - induktive Definitionen und Beweise verstehen.





Georg Cantor
(1845 – 1918)

Abbildung 2.1: Der deutsche Mathematiker Georg Cantor wurde am 3. März 1845 in Sankt Petersburg geboren. Nach seiner Ausbildung in Zürich, Göttingen und Berlin folgte er einem Ruf an die Universität Halle, an der er über 40 Jahre lang lehrte und forschte. Cantor gehört zu den bedeutendsten Mathematikern des späten neunzehnten und frühen zwanzigsten Jahrhunderts. Mit seiner *Mannigfaltigkeitslehre* begründete er die Mengenlehre und legte mit dem Begriff der *Kardinalität* den Grundstein für den Umgang mit der Unendlichkeit. Der Begriff der *Abzählbarkeit* geht genauso auf Cantor zurück wie die *Diagonalisierungsmethode*, mit deren Hilfe sich viele Erkenntnisse der theoretischen Informatik auf anschauliche Weise erklären lassen. Im Alter von 39 Jahren erkrankt Cantor an manischer Depression – ein Leiden, das ihn bis zu seinem Lebensende begleiten sollte. Kurz nach seinem siebzigsten Geburtstag wird er nach einem erneuten Krankheitsausbruch in die Universitätsklinik Halle eingewiesen. Dort stirbt Georg Cantor am 6. Januar 1918 im Alter von 72 Jahren.

2.1 Grundlagen der Mengenlehre

2.1.1 Der Mengenbegriff

Wir beginnen unseren Streifzug durch die Grundlagen der Mathematik mit einem Abstecher in das Gebiet der Mengenlehre. Für jeden von uns besitzt der Begriff der *Menge* eine intuitive Interpretation, die nicht zuletzt durch unser Alltagsleben geprägt ist. So fassen wir die 22 Akteure auf dem Fußballplatz wie selbstverständlich zu zwei Elfergruppen zusammen und wissen auch in anderen Lebenslagen Äpfel von Birnen zu unterscheiden. Die Zusammenfassung einer beliebigen Anzahl von Dingen bezeichnen wir als *Menge* und jedes darin enthaltene Objekt als *Element*.



Definition 2.1 (Mengendefinition nach Cantor)

„Unter einer *Menge* verstehen wir jede Zusammenfassung M von bestimmten wohl unterschiedenen Objekten unserer Anschauung oder unseres Denkens (welche die *Elemente* von M genannt werden) zu einem Ganzen.“

(Georg Cantor)

Dies ist der Originalwortlaut, mit dem der deutsche Mathematiker Georg Cantor (Abbildung 2.1) im Jahr 1885 den Mengenbegriff formulierte [13]. Die hierauf begründete mathematische Theorie wird als *Cantor'sche Mengenlehre* bezeichnet. Ebenfalls üblich sind die Begriffe der *anschaulichen, intuitiven* oder *naiven Mengenlehre*, um sie von den später entwickelten, streng axiomatisch definierten Mengenbegriffen abzgrenzen.

Wir schreiben $a \in M$, um auszudrücken, dass a ein Element von M ist. Entsprechend drückt die Notation $a \notin M$ aus, dass a nicht zu M gehört. Die abkürzende Schreibweise $a, b \in M$ bzw. $a, b \notin M$ besagt, dass sowohl a als auch b Elemente von M sind bzw. beide nicht zu M gehören. Zwei Mengen M_1 und M_2 gelten als gleich ($M_1 = M_2$), wenn sie exakt dieselben Elemente enthalten. Im Umkehrschluss existiert für zwei ungleiche Mengen M_1 und M_2 stets ein Element in M_1 oder M_2 , das nicht in der anderen Menge enthalten ist. Wir schreiben in diesem Fall $M_1 \neq M_2$. Offensichtlich gilt für jedes Objekt a und jede Menge M entweder $a \in M$ oder $a \notin M$.

Im Gegensatz zur umgangssprachlichen Bedeutung des Begriffs der Menge spielt es im mathematischen Sinn keine Rolle, ob darin wirklich

viele Objekte zusammengefasst sind. Wir reden selbst dann von einer Menge, wenn diese überhaupt keine Elemente enthält. Für diese *leere Menge* ist das spezielle Symbol \emptyset reserviert.

Mengen können ein einzelnes Objekt niemals mehrfach beinhalten und genauso wenig besitzen ihre Elemente einen festen Platz; Mengen sind inhärent ungeordnet. Im Übungsteil dieses Kapitels werden Sie sehen, dass der Mengenbegriff trotzdem stark genug ist, um geordnete Zusammenfassungen zu modellieren, die zudem beliebig viele Duplikate enthalten dürfen.

In der Praxis haben sich zwei unterschiedliche Schreibweisen etabliert, um die Elemente einer Menge zu definieren:

■ Aufzählende Beschreibung

Die Elemente einer Menge werden explizit aufgelistet. Selbst unendliche Mengen lassen sich aufzählend (enumerativ) beschreiben, wenn die Elemente einer unmittelbar einsichtigen Regelmäßigkeit unterliegen. Die nachstehenden Beispiele bringen Klarheit:

$$\mathbb{N} := \{0, 1, 2, 3, \dots\}$$

$$\mathbb{N}^+ := \{1, 2, 3, 4, \dots\}$$

$$\mathbb{Z} := \{\dots, -2, -1, 0, 1, 2, \dots\}$$

$$M_1 := \{0, 2, 4, 6, 8, 10, \dots\}$$

$$M_2 := \{0, 1, 4, 9, 16, 25, \dots\}$$

\mathbb{N} heißt die Menge der *natürlichen Zahlen* oder die Menge der *nicht-negativen ganzen Zahlen*. \mathbb{N}^+ beginnt mit der 1 und wird die Menge der *positiven ganzen Zahlen* genannt. \mathbb{Z} ist die Menge der *ganzen Zahlen*. M_1 enthält alle geraden natürlichen Zahlen und die Menge M_2 die Quadrate der ganzen Zahlen.

■ Deskriptive Beschreibung

Die Mengenzugehörigkeit eines Elements wird durch eine charakteristische Eigenschaft beschrieben. Genau jene Elemente sind in der Menge enthalten, auf die die Eigenschaft zutrifft.

$$M_3 := \{n \in \mathbb{N} \mid n \bmod 2 = 0\}$$

$$M_4 := \{n^2 \mid n \in \mathbb{N}\}$$

Demnach enthält die Menge M_3 alle Elemente $n \in \mathbb{N}$, die sich ohne Rest durch 2 dividieren lassen, und die Menge M_4 die Werte n^2 für alle natürlichen Zahlen $n \in \mathbb{N}$. Die Mengen M_3 und M_4 sind damit nichts anderes als eine deskriptive Beschreibung der im vorherigen Beispiel eingeführten Mengen M_1 und M_2 .

Auf den ersten Blick scheint der Mengenbegriff intuitiv erfassbar zu sein, auf den zweiten entpuppt er sich als komplexes Gebilde. Bereits im Einführungskapitel konnten wir den Cantor'schen Mengenbegriff mithilfe der *Russell'schen Antinomie* als widersprüchlich entlarven. Damit die Mathematik nicht auf wackligen Füßen steht, wurde mit der *axiomatischen Mengenlehre* eine formale Theorie geschaffen, die Inkonsistenzen der Russell'schen Art beseitigen soll. Einen wichtigen Grundstein legte der deutsche Mathematiker Ernst Zermelo, als er im Jahr 1907 ein entsprechendes Axiomensystem formulierte. Die *Zermelo-Mengenlehre* bestand aus insgesamt 7 Axiomen, die noch umgangssprachlich formuliert waren [111]. Das System wurde 1921 von Abraham Fraenkel um das Ersetzungssaxiom und 1930 von Zermelo um das Fundierungsaxiom ergänzt [32, 112]. Die 9 Axiome bilden zusammen die *Zermelo-Fraenkel-Mengenlehre*, kurz ZF, wie sie heute in weiten Teilen der Mathematik Verwendung findet (Abbildung 2.2). Wird ZF zusätzlich um das *Auswahlaxiom* (*axiom of choice*) erweitert, so entsteht die ZFC-Mengenlehre (*Zermelo-Fraenkel with Choice*). Das zusätzliche zehnte Axiom besagt das Folgende: Ist M eine Menge von nichtleeren Mengen, dann gibt es eine Funktion f , die aus jeder Menge $M' \in M$ genau ein Element auswählt. Das Auswahlaxiom ist unabhängig von allen anderen. 1937 zeigte Kurt Gödel, dass es sich widerspruchsfrei zu den ZF-Axiomen hinzufügen lässt [37]. 1963 kam Paul Cohen zu dem erstaunlichen Ergebnis, dass die Negation des Auswahlaxioms die Widerspruchsfreiheit von ZF ebenfalls nicht zerstört [23].

Mit dem ehemaligen Mengenbegriff von Cantor hat die axiomatische Mengenlehre nur wenig gemein. Sie gehört heute zu den schwierigsten Teilgebieten der Mathematik und nur wenigen ist es vergönnt, sie vollständig zu durchdringen.

■ Axiom der Bestimmtheit (Zermelo, 1908)

$$\forall x \forall y (x = y \leftrightarrow \forall z (z \in x \leftrightarrow z \in y))$$

„Ist jedes Element einer Menge M gleichzeitig Element von N und umgekehrt, ist also gleichzeitig $M \subset N$ und $N \subset M$, so ist immer $M = N$. Oder kürzer: jede Menge ist durch ihre Elemente bestimmt.“

■ Axiom der leeren Menge (Zermelo, 1908)

$$\exists x \forall y y \notin x$$

„Es gibt eine (uneigentliche) Menge, die ‚Nullmenge‘ \emptyset , welche gar keine Elemente enthält.“

■ Axiom der Paarung (Zermelo, 1908)

$$\forall x \forall y \exists z \forall u (u \in z \leftrightarrow u = x \vee u = y)$$

„Sind a, b irgend zwei Dinge des Bereichs, so existiert immer eine Menge $\{a, b\}$, welche sowohl a als [auch] b , aber kein von beiden verschiedenes Ding x als Element enthält.“

■ Axiom der Vereinigung (Zermelo, 1908)

$$\forall x \exists y \forall z (z \in y \leftrightarrow \exists (w \in x) z \in w)$$

„Jeder Menge T entspricht eine Menge $\mathfrak{S}T$ (die ‚Vereinigungsmenge‘ von T), welche alle Elemente der Elemente von T und nur solche als Elemente enthält.“

■ Axiom der Aussonderung (Zermelo, 1908)

$$\forall x \exists y \forall z (z \in y \leftrightarrow z \in x \wedge \varphi(z))$$

„Durch jede Satzfunktion $f(x)$ wird aus jeder Menge m eine Unterlage m_f ausgesondert, welche alle Elemente x umfasst, für die $f(x)$ wahr ist. Oder: Jedem Teil einer Menge entspricht selbst eine Menge, welche alle Elemente dieses Teils enthält.“

■ Axiom des Unendlichen (Zermelo, 1908)

$$\exists x (\emptyset \in x \wedge \forall (y \in x) \{y\} \in x)$$

„Der Bereich enthält mindestens eine Menge Z , welche die Nullmenge als Element enthält und so beschaffen ist, dass jedem ihrer Elemente a ein weiteres Element der Form $\{a\}$ entspricht.“

■ Axiom der Potenzmenge (Zermelo, 1908)

$$\forall x \exists y \forall z (z \in y \leftrightarrow z \subseteq x)$$

„Jeder Menge m entspricht eine Menge $\mathfrak{U}m$, welche alle Untermengen von m als Elemente enthält, einschließlich der Nullmenge und m selbst.“

■ Axiom der Ersetzung (Fraenkel, 1922)

$$(\forall (a \in x) \exists_1 b \varphi(a, b)) \rightarrow (\exists y \forall b (b \in y \leftrightarrow \exists (a \in x) \varphi(a, b)))$$

„Ist M eine Menge und wird jedes Element von M durch ‚ein Ding des Bereichs \mathfrak{B} ‘ ersetzt, so geht M wiederum in eine Menge über.“

■ Axiom der Fundierung (Zermelo, 1930)

$$\forall x (x \neq \emptyset \rightarrow \exists (y \in x) x \cap y = \emptyset)$$

„Jede (rückschreitende) Kette von Elementen, in welcher jedes Glied Element des vorangehenden ist, bricht mit endlichem Index ab bei einem Urelement. Oder, was gleichbedeutend ist: Jeder Teilbereich T enthält wenigstens ein Element t_0 , das kein Element t in T hat.“

■ Axiom der Auswahl (Zermelo, 1930)

$$(\forall (u, v \in x) (u \neq v \rightarrow u \cap v = \emptyset) \wedge \forall (u \in x) u \neq \emptyset) \rightarrow \exists y \forall (z \in x) \exists_1 (w \in z) w \in y$$

„Ist T eine Menge, deren sämtliche Elemente von \emptyset verschiedene Mengen und untereinander elementfremd sind, so enthält ihre Vereinigung $\cup T$ mindestens eine Unterlage S_1 , welche mit jedem Element von T ein und nur ein Element gemein hat.“

Abbildung 2.2: Axiome der Zermelo-Fraenkel-Mengenlehre

Von den ganzen Zahlen \mathbb{Z} wissen wir, dass sie sich mit den Vergleichsoperatoren \leq und \geq in eine Ordnung bringen lassen. Mengen lassen sich mithilfe der *Teil- oder Untermengenbeziehung* \subseteq und der *Obermengenbeziehung* \supseteq auf ähnliche Weise ordnen:

$$M_1 \subseteq M_2 : \Leftrightarrow \text{Aus } a \in M_1 \text{ folgt } a \in M_2$$

$$M_1 \supseteq M_2 : \Leftrightarrow M_2 \subseteq M_1$$

Beachten Sie, dass die Teilmengenbeziehung nach dieser Definition immer auch dann gilt, wenn die linke Menge überhaupt keine Elemente enthält. Mit anderen Worten: Die leere Menge \emptyset ist eine Teilmenge jeder anderen Menge. Des Weiteren ist jede Menge auch eine Teilmenge von sich selbst. Folgerichtig gelten die Beziehungen $\emptyset \subseteq M$, $M \supseteq \emptyset$, $M \subseteq M$ und $M \supseteq M$.

Mithilfe der eingeführten Operatoren können wir die Mengengleichheit wie folgt charakterisieren:

$$M_1 = M_2 \Leftrightarrow M_1 \subseteq M_2 \text{ und } M_2 \subseteq M_1$$

Zusätzlich vereinbaren wir die Operatoren \subset (*echte Teilmenge*) und \supset (*echte Obermenge*):

$$M_1 \subset M_2 : \Leftrightarrow M_1 \subseteq M_2 \text{ und } M_1 \neq M_2$$

$$M_1 \supset M_2 : \Leftrightarrow M_1 \supseteq M_2 \text{ und } M_1 \neq M_2$$

Offensichtlich gilt für die weiter oben eingeführten Mengen die Beziehung $M_1 \subset \mathbb{N} \subset \mathbb{Z}$. Dagegen gilt weder $M_1 \subset M_2$ noch $M_2 \subset M_1$.

2.1.2 Mengenoperationen

Bestehende Mengen lassen sich durch die Anwendung von *Mengenoperationen* zu neuen Mengen verknüpfen. In den nachstehenden Betrachtungen seien M_1 und M_2 Teilmengen einer nichtleeren *Universal-* bzw. *Trägermenge* T . Die *Vereinigungsmenge* $M_1 \cup M_2$ und die *Schnittmenge* $M_1 \cap M_2$ sind wie folgt definiert:

$$M_1 \cup M_2 := \{a \mid a \in M_1 \text{ oder } a \in M_2\}$$

$$M_1 \cap M_2 := \{a \mid a \in M_1 \text{ und } a \in M_2\}$$

Zwei Mengen M_1 und M_2 heißen *disjunkt*, falls $M_1 \cap M_2 = \emptyset$ gilt.

Die Definition lässt sich auf die Vereinigung bzw. den Schnitt beliebig vieler Mengen verallgemeinern. Für die endlich vielen Mengen

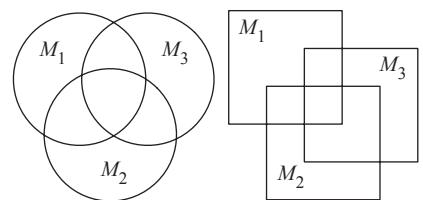
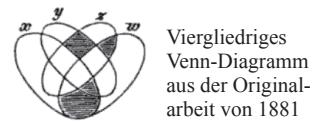
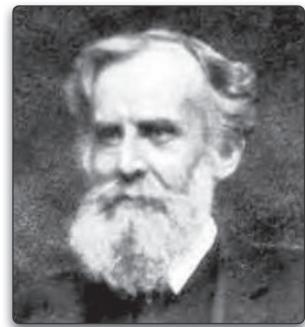


Abbildung 2.3: Venn-Diagramme sind ein anschauliches Hilfsmittel, um Beziehungen zwischen Mengen zu visualisieren. Eine Menge wird durch eine Fläche beschrieben, die durch einen geschlossenen Linienzug begrenzt wird. Jeder diskrete Punkt innerhalb der Fläche entspricht einem Element der Menge.



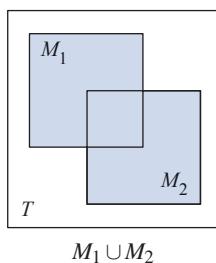
Viergliedriges Venn-Diagramm aus der Originalarbeit von 1881



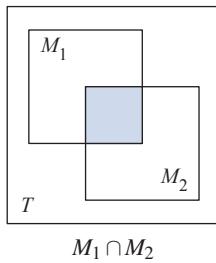
John Venn (1834 – 1923)

Abbildung 2.4: Das Venn-Diagramm wurde im Jahr 1881 durch den britischen Logiker und Philosophen John Venn eingeführt [100, 101]. Es bildet heute die am häufigsten eingesetzte Darstellungsform für die bildliche Repräsentation einer Menge.

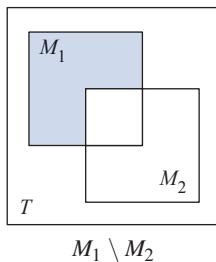
■ Vereinigung



■ Schnitt



■ Differenz



■ Komplement

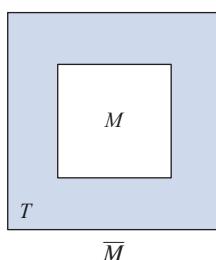


Abbildung 2.5: Elementare Mengenoperationen

M_1, \dots, M_n bzw. die unendlich vielen Mengen M_1, M_2, \dots vereinbaren wir die folgende Schreibweise:

$$\bigcup_{i=1}^n M_i := M_1 \cup \dots \cup M_n \quad \text{bzw.} \quad \bigcup_{i=1}^{\infty} M_i := M_1 \cup M_2 \cup \dots$$

$$\bigcap_{i=1}^n M_i := M_1 \cap \dots \cap M_n \quad \text{bzw.} \quad \bigcap_{i=1}^{\infty} M_i := M_1 \cap M_2 \cap \dots$$

Zusätzlich definieren wir die *Differenzmenge* $M_1 \setminus M_2$ sowie die *Komplementärmenge* \bar{M} wie folgt:

$$M_1 \setminus M_2 := \{a \mid a \in M_1 \text{ und } a \notin M_2\}$$

$$\bar{M} := T \setminus M$$

Viele Mengenbeziehungen lassen sich intuitiv mithilfe von *Venn-Diagrammen* veranschaulichen. Die Elemente einer Menge werden durch diskrete Punkte und die Mengen selbst als geschlossene Gebiete in der Ebene repräsentiert (vgl. Abbildungen 2.3 bis 2.5).

Die Vereinigungs-, Schnitt- und Komplementoperatoren begründen zusammen die *Mengenalgebra*. In der entstehenden algebraischen Struktur gilt eine Reihe von Gesetzen, die sich direkt aus der Definition der Operatoren ergeben. Insbesondere lassen sich die folgenden vier Verknüpfungsregeln ableiten:

■ Kommutativgesetze

$$M_1 \cup M_2 = M_2 \cup M_1$$

$$M_1 \cap M_2 = M_2 \cap M_1$$

■ Distributivgesetze

$$M_1 \cup (M_2 \cap M_3) = (M_1 \cup M_2) \cap (M_1 \cup M_3)$$

$$M_1 \cap (M_2 \cup M_3) = (M_1 \cap M_2) \cup (M_1 \cap M_3)$$

■ Neutrale Elemente

$$M \cup \emptyset = M$$

$$M \cap T = M$$

■ Inverse Elemente

$$M \cup \bar{M} = T$$

$$M \cap \bar{M} = \emptyset$$

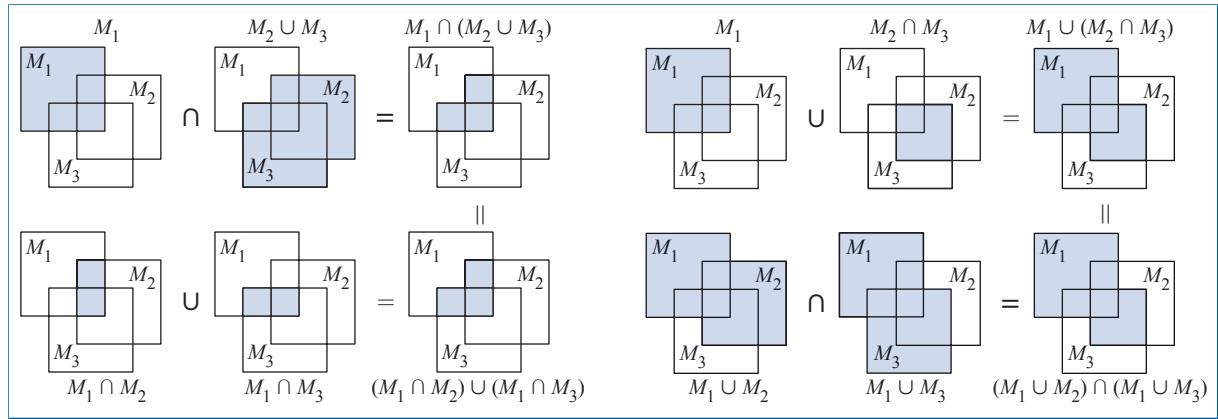


Abbildung 2.6: Veranschaulichung der Distributivgesetze anhand von Venn-Diagrammen

Von den vorgestellten Verknüpfungsregeln bedürfen nur die beiden Distributivgesetze eines zweiten Blickes, um sich von deren Richtigkeit zu überzeugen. Die Venn-Diagramme in Abbildung 2.6 liefern eine grafische Begründung für diese Regeln.

Die Mengenalgebra ist ein Spezialfall einer *booleschen Algebra* (Abbildung 2.7) [49]. Damit übertragen sich alle Gesetzmäßigkeiten, die in einer booleschen Algebra gelten, in direkter Weise auf die Mengenalgebra. Hierunter fallen insbesondere die folgenden Verknüpfungsregeln:

■ Assoziativgesetze

$$\begin{aligned} M_1 \cup (M_2 \cup M_3) &= (M_1 \cup M_2) \cup M_3 \\ M_1 \cap (M_2 \cap M_3) &= (M_1 \cap M_2) \cap M_3 \end{aligned}$$

■ Idempotenzgesetze

$$\begin{aligned} M \cup M &= M \\ M \cap M &= M \end{aligned}$$

■ Absorptionsgesetze

$$\begin{aligned} M_1 \cup (M_1 \cap M_2) &= M_1 \\ M_1 \cap (M_1 \cup M_2) &= M_1 \end{aligned}$$

■ Gesetze von De Morgan (Abbildung 2.8)

$$\begin{aligned} \overline{M_1 \cup M_2} &= \overline{M_1} \cap \overline{M_2} \\ \overline{M_1 \cap M_2} &= \overline{M_1} \cup \overline{M_2} \end{aligned}$$



George Boole
(1815 – 1864)

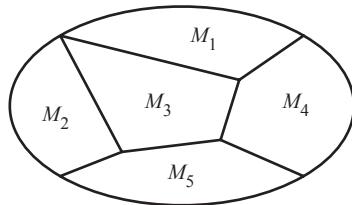
Abbildung 2.7: Der britische Mathematiker und Philosoph George Boole zählt zu den einflussreichsten Logikern des neunzehnten Jahrhunderts. Mit seinem historischen Werk *The laws of thought* legte er 1854 den Grundstein der mathematischen Logik [9]. Die nach ihm benannte boolesche Algebra ist die mathematische Grundlage für die Funktionsweise und die Konstruktion aller modernen Computeranlagen.

„The contrary of an aggregate is the compound of the contraries of the aggregants: the contrary of a compound is the aggregate of the contraries of the components.“



Augustus De Morgan (1806 – 1871)

Abbildung 2.8: Die Gesetze von De Morgan sind nach dem britischen Mathematiker Augustus De Morgan benannt, der neben George Boole als einer der bedeutendsten Mitbegründer der mathematischen Logik gilt. Auch in anderen Bereichen der Mathematik hat De Morgan Maßgebliches geleistet. Im Laufe seines Lebens verfasste er bedeutende Arbeiten in den Bereichen der Arithmetik und der Trigonometrie.



$$M = M_1 \cup \dots \cup M_5, M_i \neq \emptyset \text{ für } 1 \leq i \leq 5, \\ M_i \cap M_k = \emptyset \text{ für } i \neq k, 1 \leq i, k \leq 5$$

Abbildung 2.9: Eine Partition teilt eine Menge in paarweise disjunkte Äquivalenzklassen auf.

■ Auslöschungsgesetze

$$M \cup T = T$$

$$M \cap \emptyset = \emptyset$$

■ Gesetz der Doppelnegation

$$\overline{\overline{M}} = M$$

Zum Schluss wollen wir eine wichtige Mengenoperation einführen, die uns an zahlreichen Stellen in diesem Buch begegnen wird. Gemeint ist die Vereinigung aller Teilmengen zu einer neuen Menge 2^M . Diese wird als *Potenzmenge* bezeichnet und lässt sich mit der eingeführten Nomenklatur wie folgt charakterisieren:

$$2^M := \{M' \mid M' \subseteq M\}$$

Offensichtlich gelten für alle Mengen M die Beziehungen $\emptyset \in 2^M$ und $M \in 2^M$. Für eine nichtleere Menge M besitzt die Potenzmenge damit mindestens 2 Elemente.

Eine Teilmenge $P \subseteq 2^M$ ist eine *Partition* von M , wenn jedes Element aus M in einer und nur einer Menge aus P liegt. Die Elemente aus P werden als *Äquivalenzklassen* bezeichnet (vgl. Abbildung 2.9).

2.2 Relationen und Funktionen

Eine *Relation* setzt verschiedene Objekte in eine wohldefinierte Beziehung zueinander. Wir schreiben $x \sim_R y$, um auszudrücken, dass die Elemente x und y bezüglich der Relation R in Beziehung stehen. Um das Gegenteil auszudrücken, schreiben wir $x \not\sim_R y$. Die eingeführte Notation mag den Anschein erwecken, dass wir mit dem Relationenbegriff ein neues mathematisches Konzept einführen. Die folgenden Definitionen zeigen aber, dass sich die Relationentheorie vollständig auf den Schultern der Mengenlehre errichten lässt:



Definition 2.2 (Kartesisches Produkt)

Sei M eine beliebige Menge. Die Menge

$$M \times M := \{(x, y) \mid x, y \in M\}$$

nennen wir das *kartesische Produkt* von M .



Definition 2.3 (Relation)

Sei M eine beliebige Menge. Jede Menge R mit

$$R \subseteq M \times M$$

heißt Relation in M . Wir schreiben $x \sim_R y$ für $(x, y) \in R$ und $x \not\sim_R y$ für $(x, y) \notin R$.

Dieser Definition folgend, ist das kartesische Produkt $M \times M$ die Menge aller geordneten Paare (x, y) von Elementen aus M . Jede Relation können wir als diejenige Teilmenge von $M \times M$ auffassen, die für alle x, y mit $x \sim_R y$ das Tupel (x, y) enthält.

Relationen lassen sich auf verschiedene Weise beschreiben. Ist die Grundmenge M endlich, so werden neben der mathematisch geprägten Mengenschreibweise (vgl. Abbildung 2.10 oben) insbesondere die folgenden Darstellungen bemüht:

■ Graph-Darstellung

Die Elemente von M werden als Knoten in Form eines Punktes oder Kreises repräsentiert und jedes Element $(x, y) \in R$ als gerichtete Verbindungsgeraden (Pfeil) eingezeichnet. Elemente der Form (x, x) werden durch eine Schlinge symbolisiert, die den Knoten x mit sich selbst verbindet.

■ Matrix-Darstellung

In dieser Darstellung wird eine Relation durch eine binäre Matrix repräsentiert, die für jedes Element aus M eine separate Zeile und Spalte enthält. Jedes Matrixelement entspricht einem bestimmten Tupel (x, y) des kartesischen Produkts $M \times M$. Gilt $x \sim y$, so wird der Matrixkoeffizient an der betreffenden Stelle auf 1 gesetzt. Alle anderen Koeffizienten sind gleich 0.

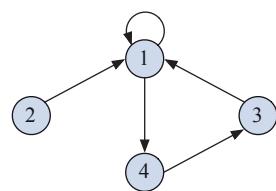
Abbildung 2.10 stellt die verschiedenen Repräsentationsformen gegenüber. Da jede Darstellung über individuelle Vor- und Nachteile verfügt, werden wir uns im Folgenden nicht auf eine einzige Repräsentation beschränken, sondern individuell auf die jeweils passende Darstellungsform zurückgreifen.

Viele praxisrelevante Relationen besitzen immer wiederkehrende, charakteristische Eigenschaften. Die folgenden *Relationenattribute* helfen, das Chaos zu ordnen:

■ Mengendarstellung

$$R := \{ (1, 1), (1, 4), (2, 1), (3, 1), (4, 3) \}$$

■ Graph-Darstellung



■ Tabellarische Darstellung

	1	2	3	4
1	1	0	0	1
2	1	0	0	0
3	1	0	0	0
4	0	0	1	0

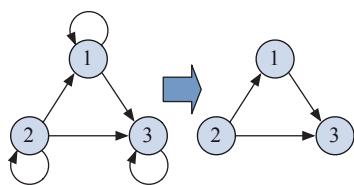
Adjazenztafel

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

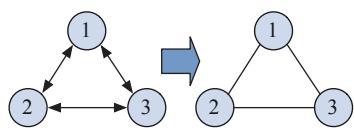
Adjazenzmatrix

Abbildung 2.10: Für die Beschreibung von Relationen haben sich verschiedene Darstellungsformen etabliert.

■ Reflexivität



■ Symmetrie



■ Transitivität

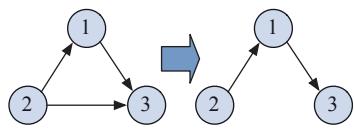


Abbildung 2.11: Vereinfachte Darstellung reflexiver, symmetrischer und transitiver Relationen



Definition 2.4 (Relationenattribute)

Eine Relation R in der Menge M heißt

- *reflexiv* in M , falls $x \sim x$ für alle $x \in M$ gilt,
- *irreflexiv* in M , falls $x \not\sim x$ für alle $x \in M$ gilt,
- *symmetrisch* in M , falls aus $x \sim y$ stets $y \sim x$ folgt,
- *asymmetrisch* in M , falls aus $x \sim y$ stets $y \not\sim x$ folgt,
- *antisymmetrisch* in M , falls aus $x \sim y$ und $y \sim x$ stets $x = y$ folgt,
- *transitiv* in M , falls aus $x \sim y$ und $y \sim z$ stets $x \sim z$ folgt,
- *linkstotal* in M , falls für alle $x \in M$ ein $y \in M$ existiert mit $x \sim y$,
- *rechtstotal* in M , falls für alle $y \in M$ ein $x \in M$ existiert mit $x \sim y$,
- *linkseindeutig* in M , falls aus $x \sim z$ und $y \sim z$ stets $x = y$ folgt,
- *rechtseindeutig* in M , falls aus $x \sim y$ und $x \sim z$ stets $y = z$ folgt.

Stellen wir eine Relation R als Graph oder in Form einer Adjazenzmatrix dar, so lassen sich viele der eingeführten Attribute auf den ersten Blick erkennen. Eine Relation R ist genau dann reflexiv, wenn alle Knoten des Relationengraphen mit einer Schlinge versehen sind, und eine symmetrische Relation liegt genau dann vor, wenn jede Kante in beiden Richtungen mit einer Pfeilspitze abschließt. Ähnliches gilt für die tabellarische Darstellung. Eine Relation R ist genau dann reflexiv, wenn in der Adjazenzmatrix sämtliche Koeffizienten der Hauptdiagonalen gleich 1 sind, und die Symmetrieeigenschaft ist gegeben, wenn die linke untere und die rechte obere Hälfte spiegelsymmetrisch zur Hauptdiagonalen liegen.

Für reflexive, symmetrische oder transitive Relationen R wird der Relationengraph gewöhnlich in einer vereinfachten Darstellung notiert (vgl. Abbildung 2.11). Reflexive Relationen werden dann ohne Schlingen gezeichnet und symmetrische Relationen durch einen ungerichteten Graph repräsentiert; an die Stelle der Doppelpfeile treten in diesem Fall einfache Linienverbindungen. Ähnliche Vereinfachungen gelten für transitive Relationen, die aus Gründen der Übersichtlichkeit um unnötige Kanten befreit werden dürfen. Eine Kante zwischen zwei Knoten x und y darf immer dann entfallen, wenn x und y bereits über andere Kanten miteinander verbunden sind.

Genau wie Mengen lassen sich auch Relationen durch die Anwendung elementarer Operationen zu neuen Relationen verknüpfen. Wichtig für unsere Betrachtungen sind vor allem die Berechnung des *Relationenprodukts* sowie die Bildung der inversen Relation.



Definition 2.5 (Relationenprodukt, inverse Relation)

R und S seien zwei Relationen in M . Das *Relationenprodukt* $R \cdot S$ und die *inverse Relation* R^{-1} sind wie folgt definiert:

$$R \cdot S := \{(x, y) \mid \text{es existiert ein } z \text{ mit } x \sim_R z \text{ und } z \sim_S y\}$$

$$R^{-1} := \{(y, x) \mid (x, y) \in R\}$$

Ferner treffen wir die folgenden Vereinbarungen:

$$R^0 := \{(x, x) \mid x \in R\}$$

$$R^n := R \cdot R^{n-1}$$

Entsprechend dieser Definition ist das Wertepaar (x, y) genau dann ein Element des Relationenprodukts, wenn x und y über ein drittes Zwischenelement z in Beziehung stehen. Offensichtlich gilt für drei beliebige Relationen S , R und T in einer Menge M das Folgende:

$$R \cdot (S \cdot T) = (R \cdot S) \cdot T$$

$$R \cdot R^0 = R$$

R^0 ist das neutrale Element des Relationenprodukts und wird als *Gleichheitsrelation* oder *Identität* bezeichnet.

Mithilfe der eingeführten Operationen lassen sich die weiter oben definierten Relationenattribute elegant charakterisieren. Für eine beliebige Relation R gelten die in Tabelle 2.1 zusammengefassten Beziehungen.

Manchmal ist es notwendig, eine Relation R um ein oder mehrere Attribute aus Definition 2.4 anzureichern. Hierzu wird R in eine größere Relation eingehüllt, die R als Teilmenge enthält. Von besonderer Bedeutung sind die *transitive* und die *reflexiv-transitive Hülle* von R :



Definition 2.6 (Transitive Hülle)

Sei R eine beliebige Relation in M . Die *transitive Hülle* R^+ ist die kleinste Relation, die R einschließt und die Eigenschaften der Transitivität erfüllt.

Reflexivität
„Für alle $x \in M$ gilt $x \sim x$ “ R ist reflexiv in M $\Leftrightarrow R^0 \subseteq R$
Irreflexivität
„Für alle $x \in M$ gilt $x \not\sim x$ “ R ist irreflexiv in M $\Leftrightarrow R^0 \cap R = \emptyset$
Symmetrie
„Aus $x \sim y$ folgt $y \sim x$ “ R ist symmetrisch in M $\Leftrightarrow R = R^{-1}$
Asymmetrie
„Aus $x \sim y$ folgt $y \not\sim x$ “ R ist asymmetrisch in M $\Leftrightarrow R \cap R^{-1} = \emptyset$
Antisymmetrie
„Aus $x \sim y$ und $y \sim x$ folgt $x = y$ “ R ist antisymmetrisch in M $\Leftrightarrow R \cap R^{-1} \subseteq R^0$
Transitivität
„Aus $x \sim y$ und $y \sim z$ folgt $x \sim z$ “ R ist transitiv in M $\Leftrightarrow R \cdot R \subseteq R$

Tabelle 2.1: Alternative Charakterisierung einiger Relationenattribute

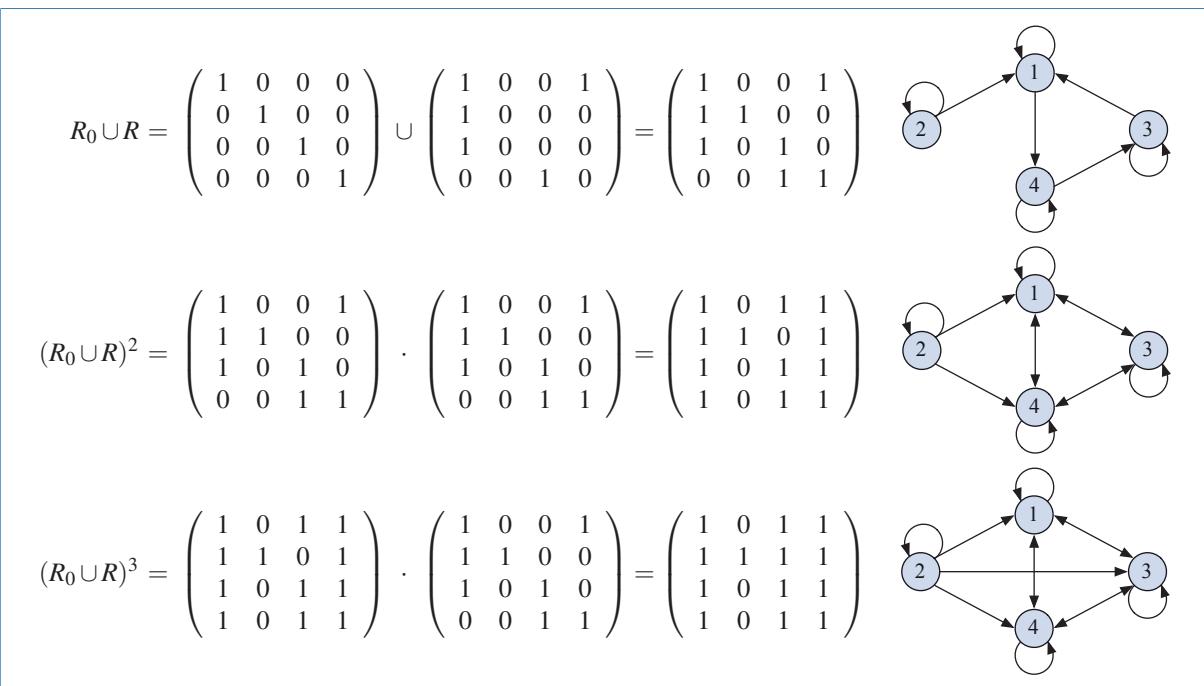


Abbildung 2.12: Iterative Berechnung der reflexiv-transitiven Hülle



Definition 2.7 (Reflexiv transitive Hülle)

Sei R eine beliebige Relation in M . Die *reflexiv-transitive Hülle* R^* ist die kleinste Relation, die R^+ einschließt und zusätzlich die Eigenschaften der Reflexivität erfüllt.

Aus der gegebenen Definition erschließt sich die folgende Überlegung: $x \sim_{R^+} y$ gilt genau dann, wenn $x \sim_R y$ ist oder ein $k \geq 1$ und Elemente z_1, \dots, z_k mit der folgenden Eigenschaft existieren:

$$x \sim_R z_1, z_1 \sim_R z_2, \dots, z_k \sim_R y$$

Damit lassen sich die transitive und die reflexiv-transitive Hülle in direkter Weise auf das Relationenprodukt reduzieren. Es gelten die folgenden Beziehungen:

$$R^+ = \bigcup_{i=1}^{\infty} R^i, \quad R^* = \bigcup_{i=0}^{\infty} R^i$$

Ist R eine Relation in einer endlichen Menge M mit n Elementen, so können wir die Gleichungen auf die folgende Form reduzieren:

$$R^+ = \bigcup_{i=1}^n R^i, \quad R^* = \bigcup_{i=0}^n R^i = (R_0 \cup R)^n$$

In dieser Form können wir die Gleichungen verwenden, um die transitive bzw. die reflexiv-transitive Hülle zu erzeugen. Für die Berechnung von R^* fügen wir zunächst die Elemente von R^0 zu R hinzu. Auf diese Weise wird R zu einer reflexiven Relation. Anschließend reichern wir die Menge um neue Wertepaare (x, y) an, indem wir die Relation so lange mit sich selbst multiplizieren, bis ein Fixpunkt erreicht ist. Die reflexiv-transitive Hülle ist berechnet, wenn keine neuen Wertepaare mehr hinzukommen.

Stellen wir die Relation R in Form einer Adjazenzmatrix dar, so lässt sich das Relationenprodukt durch eine modifizierte Matrizenmultiplikation erzeugen. Die Modifikation besteht darin, alle Produktkoeffizienten ungleich 0 als 1 in die Ergebnismatrix einzutragen. Hierdurch ist sichergestellt, dass die Multiplikation zweier Matrizen wieder eine binäre Matrix ergibt. Abbildung 2.12 demonstriert die Berechnung anhand eines konkreten Beispiels.

Wir werden nun die weiter oben eingeführten Attribute nutzen, um Relationen genauer zu klassifizieren. Die folgenden Klassen werden uns im Rest dieses Buchs immer wieder aufs Neue begegnen:

■ Äquivalenzrelationen

Eine Relation R ist eine *Äquivalenzrelation*, wenn sie gleichzeitig *reflexiv, symmetrisch* und *transitiv*

ist. Jede Äquivalenzrelation in einer Menge M besitzt die Eigenschaft, die Elemente aus M in Äquivalenzklassen einzuteilen. Hierzu wird jedes Element $x \in M$ der Äquivalenzklasse

$$[x]_\sim := \{y \mid x \sim y\}$$

zugeordnet. Die Reflexivität, Symmetrie und Transitivität führt dazu, dass die Äquivalenzklassen paarweise disjunkt sind. Die Menge der Äquivalenzklassen ist also eine Partition von M .

Die folgenden beiden Beispiele sind Äquivalenzrelationen in den natürlichen Zahlen:

$$R_1 := \{(x, y) \mid x, y \in \mathbb{N}, x \bmod 2 = y \bmod 2\}$$

$$R_2 := \{(x, x) \mid x \in \mathbb{N}\}$$

■ $\{(x, y) \mid x, y \in \mathbb{N}, x \bmod 2 = y \bmod 2\}$

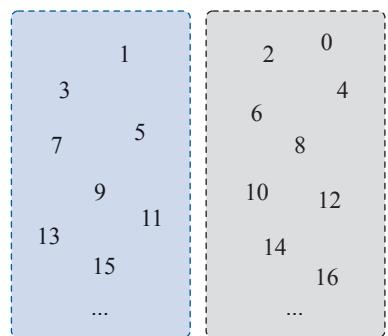


Abbildung 2.13: Äquivalenzrelationen partitionieren die Grundmenge in paarweise disjunkte Äquivalenzklassen.

■ $\{(x, y) \mid y = n \cdot x \text{ für ein } n \in \mathbb{N}\}$

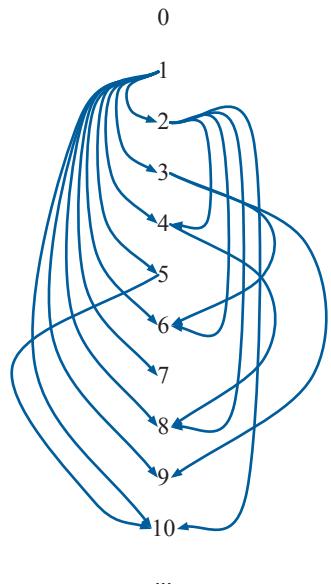
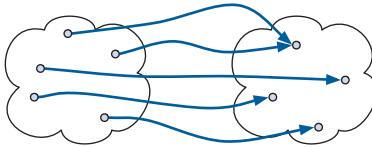


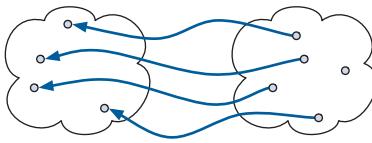
Abbildung 2.14: Ordnungsrelationen erzeugen eine (partielle) Hierarchie auf den Elementen der Grundmenge.

■ Surjektivität



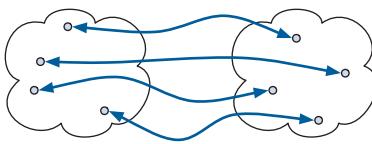
Ausnahmslos jedes Element der Zielmenge ist das Bild eines oder mehrerer Elemente der Definitionsmenge.

■ Injektivität



Es liegt eine umkehrbar eindeutige Zuordnung vor. Die Bilder zweier unterschiedlicher Elemente der Definitionsmenge sind stets verschieden.

■ Bijektivität



Die Eigenschaften der Surjektivität und Injektivität sind gleichermaßen erfüllt. Es besteht eine Eins-zu-eins-Zuordnung zwischen den Elementen der Definitionsmenge und der Zielmenge.

Abbildung 2.15: Surjektive, injektive und bijektive Funktionen

Die Relation R_1 separiert die Menge der natürlichen Zahlen in die Klasse der geraden Zahlen und die Klasse der ungeraden Zahlen (vgl. Abbildung 2.13). Hinter der Relation R_2 verbirgt sich nichts anderes als die bekannte Gleichheitsrelation ($=$). In diesem Fall bildet jedes Element x seine eigene Äquivalenzklasse.

■ Ordnungsrelationen

Eine Relation R ist eine *Ordnungsrelation*, wenn sie gleichzeitig reflexiv, transitiv und antisymmetrisch

ist. Die folgenden beiden Beispiele sind Ordnungsrelationen in den natürlichen Zahlen:

$$R_1 := \{(x, y) \mid x \leq y\}$$

$$R_2 := \{(x, y) \mid y = n \cdot x \text{ für ein } n \in \mathbb{N}^+\}$$

Die Relation R_1 entspricht der bekannten Kleiner-gleich-Relation. R_2 setzt zwei Zahlen x und y genau dann in Relation zueinander, wenn y ein ganzzahliges Vielfaches von x ist. Alle Ordnungsrelationen besitzen die Eigenschaft, die Elemente einer Menge in eine hierarchische Ordnung zu bringen. Die Definition lässt dabei ausdrücklich zu, dass zwei Elemente x und y in überhaupt keiner Beziehung zueinander stehen. Beispielsweise gilt für das obige Beispiel weder $2 \sim_{R_2} 3$ noch $3 \sim_{R_2} 2$ (vgl. Abbildung 2.14). Ordnungen mit dieser Eigenschaft werden auch als *partielle Ordnung* bezeichnet. Gilt dagegen, wie im Fall von R_1 , für zwei beliebige Elemente x und y entweder die Beziehung $x \sim y$ oder $y \sim x$, so sprechen wir von einer *Totalordnung* oder einer *linearen Ordnung*.

Neben den Äquivalenz- und Ordnungsrelationen existiert eine weitere wichtige Relationenklasse, die bisher gänzlich unerwähnt blieb. Die Rede ist von der Klasse der *Funktionen* (Abbildungen).



Definition 2.8 (Funktion)

Mit M und N seien zwei beliebige Mengen gegeben. Unter einer *Funktion* oder einer *Abbildung*

$$f : M \rightarrow N$$

verstehen wir eine Zuordnung, die jedem Element x der *Definitionsmenge* M höchstens ein Element $f(x)$ der *Zielmenge* N zuweist.



Definition 2.9 (Funktion (Fortsetzung))

Eine Funktion $f : M \rightarrow N$ heißt

- *total*, wenn für jedes $x \in M$ ein Element $f(x) \in N$ existiert.
- *partiell*, wenn sie für mindestens ein $x \in M$ undefiniert ist.

Das Element $f(x)$ heißt der *Funktionswert* von f an der Stelle x bzw. das *Bild* von x . Analog heißt x das *Urbild* von $f(x)$. Formal verbirgt sich hinter einer Funktion $f : M \rightarrow N$ nichts anderes als eine rechtseindeutige Relation $R_f \subseteq M \times N$. Die bekannte Schreibweise $f : x \mapsto y$ drückt aus, dass die Funktion f das Element x auf das Element y abbildet, und ist gleichbedeutend mit der relationalen Notation $x \sim_{R_f} y$. Gilt $N \subseteq M$, so sprechen wir von einer *Selbstabbildung*. Funktionen der Form $f : M^n \rightarrow M$ werden häufig als n -stellige *Operatoren* bezeichnet. In der Mathematik werden totale Funktionen anhand ihrer Abbildungseigenschaften wie folgt klassifiziert:



Definition 2.10 (Surjektivität, Injektivität, Bijektivität)

Sei $f : M \rightarrow N$ eine totale Funktion. f heißt

- *surjektiv*, falls für alle $y \in N$ ein $x \in M$ mit $f(x) = y$ existiert,
- *injektiv*, falls aus $f(x) = f(y)$ stets $x = y$ folgt,
- *bijektiv*, falls f sowohl injektiv als auch surjektiv ist.

Grob gesprochen besitzen surjektive Funktionen die Eigenschaft, die Zielmenge vollständig auszuschöpfen, d. h., jedes Element besitzt mindestens ein Urbild in der Definitionsmenge. Injektive Funktionen zeichnen sich dadurch aus, dass jedes Element der Zielmenge höchstens ein Urbild besitzt. Diese Eigenschaft ist die Voraussetzung, um aus einer Abbildung f die *Umkehrabbildung* f^{-1} zu konstruieren, die jedem Element $f(x)$ sein Urbild x zuordnet. Entsprechend werden injektive Funktionen auch als *umkehrbar eindeutig* oder einfach nur als *umkehrbar* bezeichnet. Eine bijektive Funktion f erfüllt beide Eigenschaften gleichzeitig, so dass für jedes Element x der Definitionsmenge genau ein Element y der Zielmenge mit $f(x) = y$ existiert. Mit anderen Worten: Bijektive Funktionen stellen eine Eins-zu-eins-Zuordnung zwischen Definitionsmenge und Zielmenge her. Abbildung 2.15 fasst die Eigenschaften surjektiver, injektiver und bijektiver Funktionen grafisch zusammen.

log.c

```
unsigned foo (unsigned x)
{
    int y = 0;
    while (x != 1) {
        x >= 1;
        y++;
    }
    return y;
}
```

Abbildung 2.16: Logarithmenberechnung in der Programmiersprache C

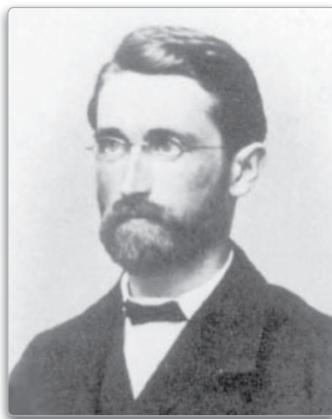
Der Funktionsbegriff wird in der theoretischen Informatik anders definiert als in der Mathematik. Dort ist eine Funktion eine Zuordnungsvorschrift, die ausnahmslos jedes Element der Definitionsmenge auf ein Element der Zielmenge abbildet. Mit anderen Worten: In der klassischen Mathematik sind alle Funktionen total. Der Begriff der partiellen Funktion wurde in der theoretischen Informatik eingeführt, um Programme zu beschreiben, die für gewisse Eingabewerte nicht terminieren. Als Beispiel betrachten wir die in Abbildung 2.16 dargestellte C-Funktion `foo`. Für $x > 0$ berechnet sie den ganzzahlig gerundeten Zweierlogarithmus des Eingabewerts x . Für $x = 0$ ist der Ausgabewert undefiniert, da das Programm in eine Endlossschleife gerät. Mithilfe von partiellen Funktionen lässt sich das Programmverhalten problemlos beschreiben:

$$f(x) = \begin{cases} \lfloor \log_2 x \rfloor & \text{für } x > 0 \\ \perp & \text{für } x = 0 \end{cases}$$

2.3 Die Welt der Zahlen

2.3.1 Natürliche, rationale und reelle Zahlen

„Die Zahlen sind freie Schöpfungen des menschlichen Geistes, sie dienen als Mittel, um die Verschiedenheit der Dinge leichter und schärfer aufzufassen. Durch den rein logischen Aufbau der Zahlenwissenschaft und durch das in ihr gewonnene stetige Zahlenreich sind wir erst in den Stand gesetzt, unsere Vorstellungen von Raum und Zeit genau zu untersuchen, indem wir dieselben auf dieses in unserem Geiste geschaffene Zahlenreich beziehen.“ [25]



Richard Dedekind
(1831 – 1916)

Abbildung 2.17: Im Jahr 1888 veröffentlichte der deutsche Mathematiker Richard Dedekind seine berühmte Abhandlung *Was sind und was sollen die Zahlen*. Hierin formulierte er fünf umgangssprachliche Axiome, über die sich die Struktur der natürlichen Zahlen eindeutig charakterisieren lässt.

Im vorherigen Abschnitt haben wir mit der Menge $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ die *natürlichen Zahlen* eingeführt. Der Umgang mit dieser Menge scheint uns in die Wiege gelegt, schließlich entspringt sie einer angeborenen Fähigkeit des Menschen: dem Abzählen von Dingen. Auch wenn uns die natürlichen Zahlen auf den ersten Blick völlig vertraut erscheinen, wollen wir sie an dieser Stelle durch eine formale Definition charakterisieren.

Die am weitesten verbreitete Definition geht auf den deutschen Mathematiker Richard Dedekind zurück. In seiner berühmten Abhandlung *Was sind und was sollen die Zahlen* aus dem Jahr 1888 charakterisierte er die natürlichen Zahlen über fünf Eigenschaften, die in moderner Sprechweise folgendermaßen lauten:



Definition 2.11 (Axiomatisierung der natürlichen Zahlen)

Die Menge der natürlichen Zahlen \mathbb{N} ist durch die folgenden fünf Axiome charakterisiert:

- P1) 0 ist eine natürliche Zahl.
- P2) Jede natürliche Zahl n hat genau einen Nachfolger $\text{succ}(n)$.
- P3) 0 ist kein Nachfolger einer natürlichen Zahl.
- P4) Die Nachfolger zweier verschiedener natürlicher Zahlen sind ebenfalls verschieden.
- P5) Enthält eine Teilmenge $M \subseteq \mathbb{N}$ die Zahl 0 und zu jedem Element n auch ihren Nachfolger $\text{succ}(n)$, so gilt $M = \mathbb{N}$.

Heute werden die Axiome P1 bis P5 als *Peano-Axiome* bezeichnet. Sie sind nach dem italienischen Mathematiker Giuseppe Peano benannt (Abbildung 2.18), der sie ein Jahr nach ihrer Formulierung durch Dedekind in eine symbolische Formelsprache übersetzte [80].

Die Menge der natürlichen Zahlen ist bezüglich der Addition und der Multiplikation abgeschlossen, d. h., für zwei beliebige Zahlen $a, b \in \mathbb{N}$ liegen auch die Summe $a + b$ sowie das Produkt $a \cdot b$ in \mathbb{N} . Aufgrund des

einseitig beschränkten Zahlenbereichs besitzt die Gleichung

$$a + x = b \quad (2.1)$$

jedoch nur für $a \leq b$ eine Lösung in \mathbb{N} . Abhilfe schafft die Menge der *ganzen Zahlen* \mathbb{Z} , die \mathbb{N} in den negativen Zahlenbereich fortsetzt:

$$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

In dieser Menge können wir Gleichung (2.1) für beliebige Werte von a und b lösen. Genau wie die natürlichen Zahlen sind auch die ganzen Zahlen bezüglich der Addition und der Multiplikation abgeschlossen. Dagegen ist die Division nur in Ausnahmefällen möglich. Der Wunsch, die Gleichung $a \cdot x = b$ für beliebige Werte $a, b \in \mathbb{Z}$ zu lösen, bringt uns auf direktem Weg zu den *rationalen Zahlen* \mathbb{Q} :

$$\mathbb{Q} := \{x \mid x = \frac{p}{q}, p, q \in \mathbb{Z}, q \neq 0\}$$

Jede rationale Zahl $a \in \mathbb{Q}$ lässt sich als unendlicher periodischer Dezimalbruch der Form

$$x = \underbrace{a_k \dots a_1 a_0}_{Vorkommastellen}, \underbrace{a_{-1} a_{-2} \dots a_{-l}}_{Nachkommastellen} \overbrace{a_{-l-1} a_{-l-2} \dots a_{-m}}^{\text{Periode}} \quad (2.2)$$

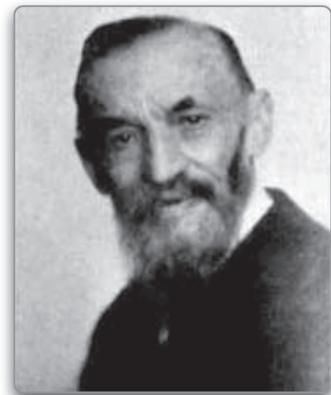
überführen und jede Zahl der Form (2.2) in der Form $\frac{p}{q}$ mit $p, q \in \mathbb{Z}$ darstellen. Kurzum: Die Menge der rationalen Zahlen ist mit der Menge der unendlichen periodischen Dezimalbrüche identisch. In diesen sind die endlichen Dezimalzahlbrüche als Spezialfall enthalten; sie entsprechen einem periodischen Dezimalbruch mit der Periode 0.

Die Menge der rationalen Zahlen verfügt über zwei wesentliche Eigenschaften. Zum einen sind die natürlichen und die ganzen Zahlen Teilmengen von \mathbb{Q} . Zum anderen können wir in dieser Menge uneingeschränkt rechnen. Für zwei beliebige rationale Zahlen liegen die Summe, die Differenz, das Produkt und der Quotient ebenfalls wieder in \mathbb{Q} . Es gelten die folgenden Rechenregeln:

$$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm cb}{bd}, \quad \frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}, \quad \frac{a}{b} : \frac{e}{d} = \frac{ad}{be}$$

Die gewonnene Bewegungsfreiheit, die wir in der Menge der rationalen Zahlen genießen, wirft die Frage auf, ob eine erneute Erweiterung des Zahlenbereichs sinnvoll ist. Die Antwort lautet „ja“, da die Menge \mathbb{Q} – obwohl sie jedes noch so kleine Intervall mit unendlich vielen Elementen überdeckt – Lücken besitzt. Ein Beispiel einer solchen Lücke ist die Zahl $\sqrt{2}$. Von dieser wissen wir zunächst nur, dass sie mit sich

„Questions that pertain to the foundations of mathematics, although treated by many in recent times, still lack a satisfactory solution. Ambiguity of language is philosophy's main source of problems. That is why it is of the utmost importance to examine attentively the very words we use.“



Giuseppe Peano
(1858 – 1932)

Abbildung 2.18: Der Italiener Giuseppe Peano gilt als einer der Wegbereiter der mathematischen Logik und der axiomatischen Methode. Peano war ein Akkurateur seines Faches und ein Verfechter symbolischer Sprachen. Nur diese verfügten seiner Ansicht nach über die nötige Präzision, um mathematische Sachverhalte prägnant und unmissverständlich zu formulieren. Unter anderem gehen die heute immer noch gebräuchlichen Symbole \cup , \cap , \in und \exists auf die Arbeiten von Peano zurück. Einen großen Teil seines wissenschaftlichen Engagements widmete er dem *Formulario-Projekt* – einem groß angelegten Vorhaben zur Formalisierung des mathematischen Wissens. Heute ist der Name Peano fest mit den fünf Axiomen verbunden, die uns die formale Grundlage für den Umgang mit den natürlichen Zahlen liefern.



In der modernen Mathematik ist die Null nicht wegzudenken. Ohne sie würden weder die elementaren Regeln der Arithmetik funktionieren noch wären wir in der Lage, Dezimalzahlen eindeutig zu notieren. Der Umgang mit ihr ist uns heute so vertraut, dass sie keinerlei Sonderstellung mehr bedarf. Kurzum: Die Null ist zu einer Zahl wie jede andere geworden.

Vor diesem Hintergrund verwundert es immer wieder, dass sie erst sehr spät ihren Platz in der Arithmetik fand. Babylonier, Ägypter, Griechen und Römer kamen kein Symbol für die Null und entsprechend schwer war es, in den verwendeten Zahlensystemen zu rechnen. Wahrscheinlich verhinderte ihre innerste Bedeutung den unvoreingenommenen Umgang mit dieser Zahl. Null stand für „nichts“ und jede Definition einer entsprechenden Zahl schien diese Eigenschaft ad absurdum zu führen.

Das geistige Fundament der Zahl Null, wie wir sie heute verstehen und verwenden, wurde in Indien im fünften bis siebten Jahrhundert n. Chr. geschaffen. Über Persien gelangte das Wissen nach Europa und über den Himalaja nach China.

Das Rechnen mit der Null sollte die Mathematiker noch über Jahre beschäftigen. Der indische Mathematiker und Astronom Brahmagupta war der Meinung, null geteilt durch null ergäbe null. Andere Mathematiker waren der Auffassung, dass eine Zahl unverändert bleibt, wenn sie durch null dividiert wird. Erst die Infinitesimalrechnung von Leibniz und Newton lieferte schließlich das mathematische Instrumentarium, um das unendlich Kleine zu beherrschen [12].

selbst multipliziert das Ergebnis 2 liefert. Den ungefähren Wert von $\sqrt{2}$ können wir ebenfalls beziffern:

$$\sqrt{2} \approx 1,41421$$

Wir können uns der Zahl $\sqrt{2}$ durch die Angabe weiterer Nachkommastellen beliebig nähern. Jeder Versuch, $\sqrt{2}$ exakt niederzuschreiben, ist jedoch von vorneherein zum Scheitern verurteilt. Schuld daran ist die Eigenschaft dieser Zahl, unendlich viele, unregelmäßig auftretende Nachkommaziffern zu besitzen. Mit anderen Worten: $\sqrt{2}$ besitzt keine periodische Dezimalbruchdarstellung und ist damit keine rationale Zahl. Diese Erkenntnis ist keinesfalls neu. Bereits Euklid von Alexandria konnte mit einem einfachen Widerspruchsargument zeigen, dass sich $\sqrt{2}$ nicht in Form eines Dezimalbruchs darstellen lässt und damit außerhalb von \mathbb{Q} liegen muss [29].

Schließen wir die Lücken in der Menge \mathbb{Q} durch die Hinzunahme aller Zahlen mit einer unendlichen, nichtperiodischen Dezimalbruchdarstellung, so erhalten wir die Menge der *reellen Zahlen* \mathbb{R} .

$$\mathbb{R} := \{x \mid x \text{ besitzt eine unendliche Dezimalbruchdarstellung}\} \quad (2.3)$$

Die Differenzmenge $\mathbb{R} \setminus \mathbb{Q}$ heißt die Menge der *irrationalen Zahlen*. Neben der Zahl $\sqrt{2}$ enthält $\mathbb{R} \setminus \mathbb{Q}$ weitere bekannte Größen. Beispiele sind die Kreiszahl π und die Euler'sche Konstante e .

Die soeben gegebene Definition der reellen Zahlen ist recht vage formuliert und im mathematischen Sinne nur bedingt belastbar. Aus diesem Grund wurden in der Vergangenheit erhebliche Anstrengungen unternommen, die Definition von \mathbb{R} zu präzisieren. Einen formalen und gleichzeitig anschaulichen Zugang zu den reellen Zahlen lieferte Dedekind im Jahr 1872 (Abbildung 2.17). Er ging von der Idee aus, dass die Zahlengerade durch jeden ihrer Punkte in eine linke und eine rechte Seite unterteilt wird. In entsprechender Weise lässt sich jeder Punkt durch zwei Mengen L und R mit $\mathbb{Q} = L \cup R$ und $l < r$ für alle $l \in L$ und alle $r \in R$ beschreiben. Das Tupel $(L|R)$ wird als *Dedekind'scher Schnitt* bezeichnet und steht stellvertretend für ein Element aus \mathbb{R} :

$$\begin{aligned} \frac{1}{3} &= (\{x \in \mathbb{Q} \mid x \leq \frac{1}{3}\} \mid \{x \in \mathbb{Q} \mid x > \frac{1}{3}\}) \\ \sqrt{2} &= (\{x \in \mathbb{Q} \mid x^2 < 2\} \mid \{x \in \mathbb{Q} \mid x^2 > 2\}) \end{aligned}$$

Obwohl jeder Schnitt ausschließlich aus rationalen Zahlen besteht, ist es uns gelungen, jede reelle Zahl eindeutig zu identifizieren.

Die Dedekind'schen Schnitte sind nicht das einzige Modell, um die reellen Zahlen greifbar zu machen. Andere Formalisierungen bedienen sich

des Prinzips der Intervallschachtelung oder definieren die reellen Zahlen, wie einst von Georg Cantor vorgeschlagen, als Äquivalenzklasse rationaler Cauchy-Folgen [45].

2.3.2 Von großen Zahlen

Standen in unseren bisherigen Untersuchungen die Zahlen im Mittelpunkt, wollen wir uns in diesem Abschnitt ausführlicher mit den *Operatoren* beschäftigen, ohne die jegliche Mathematik ein langweiliges Unterfangen wäre. Wir werden eine Reihe von Betrachtungen vornehmen, die für den reinen Mathematiker exotisch wirken mögen, im Bereich der Berechenbarkeitstheorie aber eine bedeutende Rolle spielen.

Nachfolgend setzen wir als Grundmenge die natürlichen Zahlen \mathbb{N} voraus. Die mit Abstand einfachste Operation auf einer Zahl $a \in \mathbb{N}$ ist die Berechnung des Nachfolgers $\text{succ}(a) = 1 + a$. Dass $\text{succ}(a)$ für alle Zahlen $a \in \mathbb{N}$ existiert, garantiert uns das zweite Peano-Axiom.

Unwesentlich komplexer ist die Addition. Im Grunde genommen handelt es sich dabei um keine neue Operation, da wir die Berechnung von $a + b$ auf die wiederholte Anwendung der Nachfolgeoperation $(1 + a)$ reduzieren können:

$$a + b = \underbrace{1 + (1 + (1 + (\dots + (1 + a)))))}_{b \text{ Kopien von } 1} \quad (2.4)$$

$$= 1 + (a + (b - 1)) \quad (2.5)$$

Die nächsthöhere Operation ist die Multiplikation. Analog zu den Gleichungen (2.4) und (2.5) können wir die Produktbildung auf die Addition zurückführen:

$$a \cdot b = \underbrace{a + (a + (a + (\dots + (a + a)))))}_{b \text{ Kopien von } a} \quad (2.6)$$

$$= a + (a \cdot (b - 1)) \quad (2.7)$$

Die wiederum nächsthöhere Operation ist die Potenzierung. Wir können dem eingeschlagenen Schema treu bleiben und die Potenz a^b wie folgt berechnen:

$$a^b = \underbrace{a \cdot (a \cdot (a \cdot (\dots \cdot (a \cdot a)))))}_{b \text{ Kopien von } a} \quad (2.8)$$

$$= a \cdot (a^{b-1}) \quad (2.9)$$

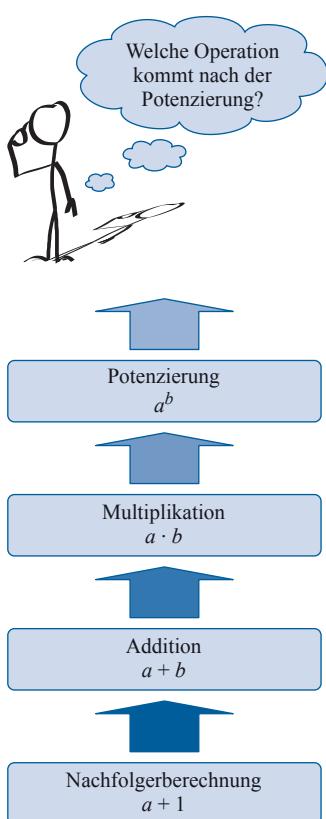


Abbildung 2.19: Hierarchie der arithmetischen Operatoren

Die arithmetischen Operationen scheinen ganz offensichtlich eine Hierarchie zu bilden (vgl. Abbildung 2.19). Doch wie geht diese Hierarchie nach oben weiter? Konkreter formuliert: Welche mathematische Operation kommt nach der Potenzierung? Dass den meisten von uns die Antwort auf diese Frage nicht auf der Zunge liegt, hat einen triftigen Grund: Der klassischen Mathematik fehlen Symbole, mit denen sich arithmetische Operationen höherer Grade niederschreiben lassen.

Der US-amerikanische Computerwissenschaftler Donald E. Knuth löste das Problem im Jahr 1976 durch die Einführung der *Up-Arrow-Notation* [61]. Mit dem \uparrow -Operator schuf er die Möglichkeit, Funktionen verschiedener Grade einheitlich zu benennen:

$$a \uparrow^n b := \begin{cases} a^b & \text{falls } n = 1 \\ 1 & \text{falls } b = 0 \\ a \uparrow^{n-1} (a \uparrow^n (b-1)) & \text{sonst} \end{cases}$$

Die Definition folgt exakt dem gleichen Konstruktionsprinzip, das uns bereits die Potenzierung auf die Multiplikation (Gleichung (2.9)), die Multiplikation auf die Addition (Gleichung (2.7)) und die Addition schließlich auf die Nachfolgeoperation (Gleichung (2.5)) reduzierten ließ.

Eine bekannte Funktion, die das Schema der herausgearbeiteten Berechnungsvorschrift aufgreift, ist die *Ackermann-Funktion* [1]. Sie wurde von dem deutschen Mathematiker Wilhelm Ackermann im Jahr 1928 publiziert und wird uns in Kapitel 6 als Hilfsmittel dienen, um die unterschiedliche Ausdrucksstärke von Loop- und While-Programmen zu beweisen. Die heute gebräuchliche Darstellungsform unterscheidet sich geringfügig von jener der Originalarbeit und ist durch die folgende rekursive Definition gegeben [44]:

$$\begin{aligned} \text{ack}(0, m) &= m + 1 \\ \text{ack}(n, 0) &= \text{ack}(n - 1, 1) \\ \text{ack}(n, m) &= \text{ack}(n - 1, \text{ack}(n, m - 1)) \end{aligned} \tag{2.10}$$

Gleichung (2.10) enthält den gleichen rekursiven Kern, auf dem auch die Definition des \uparrow -Operators beruht. Entsprechend sind wir in der Lage, die Ackermann-Funktion mithilfe des \uparrow -Operators auszudrücken. Es gelten die folgenden Beziehungen:

$$\begin{aligned} \text{ack}(1, m) &= 2 + (m + 3) - 3 \\ \text{ack}(2, m) &= 2 \cdot (m + 3) - 3 \end{aligned} \tag{2.11}$$

$\text{ack}(n, m)$	$m = 0$	$m = 1$	$m = 2$	$m = 3$	$m = 4$
$n = 0$	1	2	3	4	5
$n = 1$	2	3	4	5	6
$n = 2$	3	5	7	9	11
$n = 3$	5	13	29	61	125
$n = 4$	13	65533	$2^{65536} - 3$	$2^{2^{65536} - 3}$	$\text{ack}(3, 2^{2^{65536} - 3})$
$n = 5$	65533	$\text{ack}(4, 65533)$	$\text{ack}(4, \text{ack}(5, 1))$	$\text{ack}(4, \text{ack}(5, 2))$	$\text{ack}(4, \text{ack}(5, 3))$
$n = 6$	$\text{ack}(4, 65533)$	$\text{ack}(5, \text{ack}(5, 1))$	$\text{ack}(5, \text{ack}(6, 1))$	$\text{ack}(5, \text{ack}(6, 2))$	$\text{ack}(5, \text{ack}(6, 3))$

Tabelle 2.2: Ein kleiner Auszug aus der Wertetabelle der Ackermann-Funktion

$$\begin{aligned}\text{ack}(3, m) &= 2 \uparrow (m + 3) - 3 \\ \text{ack}(4, m) &= 2 \uparrow\uparrow (m + 3) - 3 \\ \text{ack}(5, m) &= 2 \uparrow\uparrow\uparrow (m + 3) - 3 \\ \text{ack}(6, m) &= 2 \uparrow\uparrow\uparrow\uparrow (m + 3) - 3\end{aligned}$$

$$\text{ack}(n, m) = 2 \underbrace{\uparrow\uparrow \dots \uparrow}_{(n-2)\text{-mal}} (m + 3) - 3 \quad (2.12)$$

Wie die Gleichungen (2.11) bis (2.12) offenbaren, verbirgt sich hinter dem harmlos anmutenden Rekursionsschema eine trickreiche Konstruktion. Die Ackermann-Funktion ist eine Art Universalfunktion, die die gesamte Operatorenhierarchie in sich vereint. Da der erste Parameter den Grad der auszuführenden Operation bestimmt, nimmt die Funktion selbst für kleine Parameter riesige Werte an. Tabelle 2.2 vermittelt einen Eindruck über das Wachstumsverhalten. Obwohl nur wenige Einträge gelistet sind, konnten die Funktionswerte nicht überall in ihrer Dezimaldarstellung eingetragen werden. So entspricht der Wert $\text{ack}(4, 2)$ bereits einer ca. 20.000-stelligen Zahl.

2.3.3 Die Unendlichkeit begreifen

In diesem Abschnitt wenden wir uns einem Thema zu, das in der theoretischen Informatik eine übergeordnete Rolle spielt und etliche Generationen von Mathematikern vor scheinbar unüberwindbare Rätsel stellte (vgl. Abbildung 2.20). Die Rede ist von der Unendlichkeit. Wir werden uns an diesen Begriff in zwei Schritten herantasten. Zunächst werden

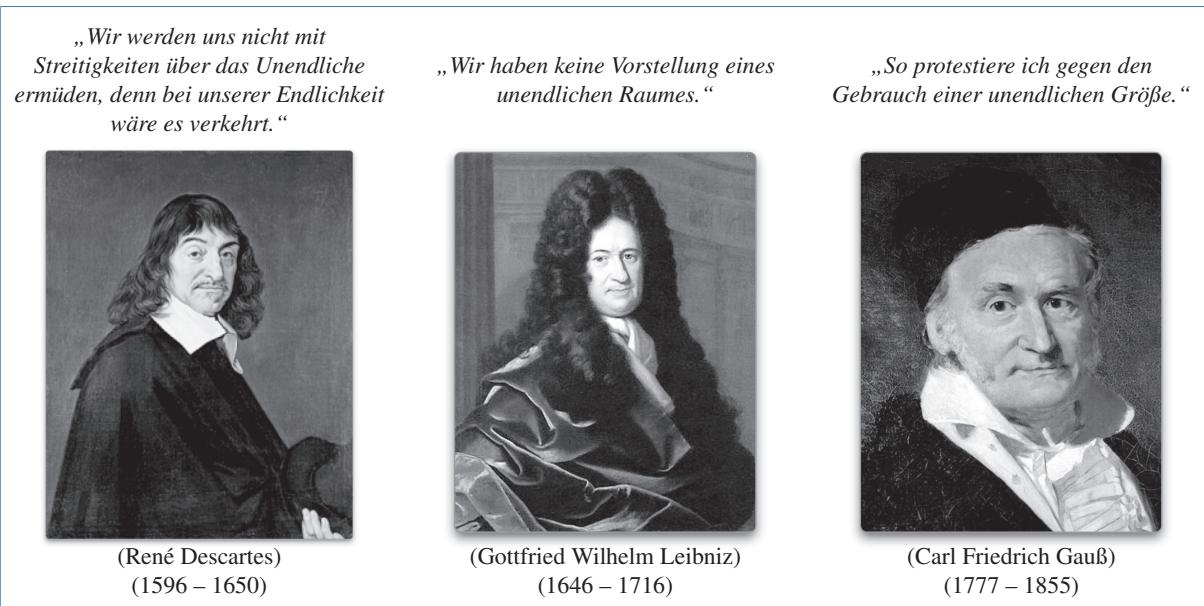


Abbildung 2.20: Der Begriff der Unendlichkeit stellte viele Mathematikergenerationen vor scheinbar unüberwindliche Rätsel. Erst die Arbeiten von Georg Cantor lieferten das notwendige Instrumentarium, um das Unendliche systematisch zu erfassen und beherrschbar zu machen.

wir definieren, was sich hinter dem nebulösen Terminus der Unendlichkeit genau verbirgt. Anschließend werden wir zeigen, dass Unendlichkeit nicht gleich Unendlichkeit ist. Sie mögen es vielleicht schon vermuten: Es gibt derer unendlich viele.

In der Mathematik ist die Unendlichkeit allgegenwärtig und in vielen Situationen scheint uns deren Anwesenheit nicht weiter zu stören. Wir gehen wie selbstverständlich mit der Menge der natürlichen Zahlen \mathbb{N} oder der Menge der ganzen Zahlen \mathbb{Z} um, obwohl wir niemals in der Lage sein werden, alle Zahlen niederzuschreiben. Auf die Frage, wie viele Elemente die Mengen \mathbb{N} und \mathbb{Z} wirklich besitzen, antworten wir fast schon reflexartig mit der Antwort „unendlich“. Viele von uns plagt dabei das Gefühl, dass die Menge der ganzen Zahlen \mathbb{Z} mehr Elemente enthalten müsste als die Menge der natürlichen Zahlen \mathbb{N} . Alle Elemente von \mathbb{N} sind schließlich in \mathbb{Z} enthalten, nicht jedoch umgekehrt. Damit ist an der Zeit, uns Gedanken zu machen, wie wir den schwer greifbaren Begriff der Unendlichkeit bändigen können.

Der Schlüssel für den Umgang mit dem Unendlichen liegt in der Beobachtung der *Mächtigkeit (Kardinalität)* einer Menge M . Diese wird

mit $|M|$ bezeichnet und entspricht für endliche Mengen schlicht der Anzahl ihrer Elemente.

$$\begin{aligned} M_1 = \emptyset &\Rightarrow |M_1| = 0 \\ M_2 = \{\square, \diamond, \circ\} &\Rightarrow |M_2| = 3 \\ M_3 = \{2, 3, 5\} &\Rightarrow |M_3| = 3 \end{aligned}$$

Die Mengen M_2 und M_3 sind *gleichmächtig*, da sie die gleiche Anzahl an Elementen enthalten. In diesem Fall sind wir in der Lage, die Elemente beider Mengen eindeutig einander zuzuordnen. Für unser Beispiel könnte die Zuordnung folgendermaßen aussehen:

$$\square \mapsto 2, \quad \diamond \mapsto 3, \quad \circ \mapsto 5$$

Stimmt die Anzahl der Elemente nicht überein, so ist jeder Versuch, eine derartige Zuordnung herzustellen, zum Scheitern verurteilt. Damit sind wir in der Lage, den Begriff der Mächtigkeit an die Existenz einer entsprechenden Abbildung zu knüpfen:



Definition 2.12 (Mengenvergleiche)

Mit M_1 und M_2 seien zwei beliebige Mengen gegeben. M_1 und M_2 heißen *gleichmächtig*, geschrieben als

$$|M_1| = |M_2|,$$

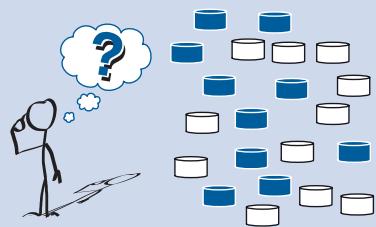
wenn eine bijektive Abbildung $f : M_1 \rightarrow M_2$ existiert. Jede unendliche Menge M heißt

- *abzählbar*, falls $|M| = |\mathbb{N}|$, und
- *überabzählbar*, falls $|M| \neq |\mathbb{N}|$ gilt.

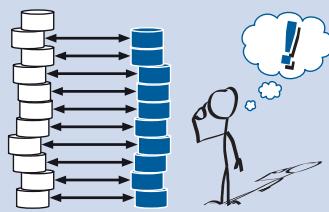
Zwei unendliche Mengen sind per Definition genau dann gleichmächtig, wenn sich ihre Elemente jeweils umkehrbar eindeutig einander zuordnen lassen. Auf den ersten Blick erscheint die Herangehensweise als unnatürlich und unnötig umständlich. Auf den zweiten Blick wird deutlich, dass die Definition darauf verzichtet, die Elemente einer Menge explizit zu zählen. Damit sind wir in der Lage, auch dann die Kardinalität zweier Mengen zu vergleichen, wenn diese unendlich viele Elemente enthalten.

Plakativ gesprochen drückt der Begriff der *Abzählbarkeit* aus, dass wir die Elemente einer unendlichen Menge durchnummerieren können. Mit welcher Nummer wir die einzelnen Elemente konkret belegen, spielt

Die Idee, die Gleichmächtigkeit zweier Mengen über die Existenz einer bijektiven Zuordnung zu entscheiden, ist kein Kunstprodukt der Mathematik. In Wirklichkeit ist uns das verwendete Prinzip vertrauter, als es auf den ersten Blick erscheinen mag. So setzen wir diese Methode in vielen Alltagssituationen unbewusst ein, um das vergleichsweise aufwendige Zählen großer Gegenstandsmengen zu umgehen. Auch Kinder, die des Rechnens noch überhaupt nicht mächtig sind, bedienen sich dieses Prinzips.

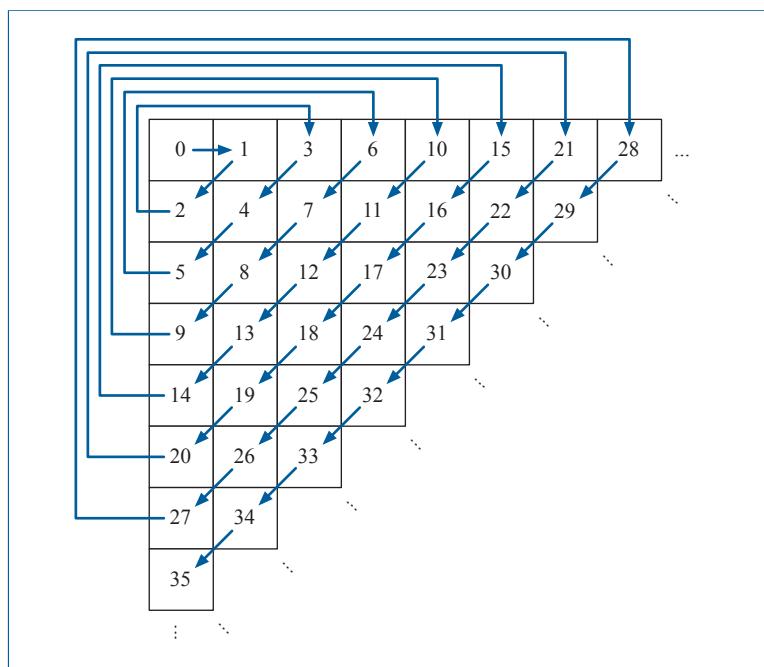


Um z. B. festzustellen, ob in einer Menge von Spielsteinen mehr blaue oder mehr weiße Steine enthalten sind, ist es ausreichend, diese farblich zu separieren und anschließend zu stapeln:



In der neuen Darstellung lässt sich mit einem einzigen Blick erkennen, dass die weißen Steine in der Überzahl sind. Beachtlich an dieser Methode ist vor allem die Tatsache, dass wir die Mächtigkeit zweier Mengen vergleichen können, ohne die konkrete Anzahl ihrer Elemente zu bestimmen. Erst hierdurch sind wir in der Lage, die Vorgehensweise auf Mengen mit unendlich vielen Elementen zu übertragen.

Abbildung 2.21: Die abgebildete Paarungsfunktion ordnet jedem Tupel $(p, q) \in \mathbb{N}^2$ eine Zahl $\pi_{\mathbb{N}}(p, q) \in \mathbb{N}$ zu. Die Abbildung ist bijektiv und beweist, dass \mathbb{N} und \mathbb{N}^2 gleichmächtig sind.



dabei keine Rolle – einzig die Existenz einer solchen Zuordnung macht eine Menge abzählbar. Überabzählbare Mengen besitzen diese Eigenschaft nicht, d. h., jeder Versuch, ihre Elemente durchzunummerieren, muss scheitern. Dass es solche Mengen überhaupt gibt, ist keinesfalls selbstverständlich.

Im Folgenden werden wir zeigen, dass solche Mengen tatsächlich existieren und sich sogar systematisch aus abzählbaren Mengen konstruieren lassen. Die Cantor'sche Definition der Kardinalität wird sich dabei als äußerst wertvoll erweisen, da wir hierüber in der Lage sind, die konstruierten Mengen miteinander zu vergleichen. Wie wir gleich sehen werden, hält die Unendlichkeit einige Überraschungen für uns bereit.

Als Erstes betrachten wir die beiden Mengen \mathbb{N} und \mathbb{Z} . Obwohl \mathbb{N} eine echte Teilmenge von \mathbb{Z} ist, lassen sich beide bijektiv aufeinander abbilden. Die folgende Zuordnung ist eine von – Sie werden es ahnen – unendlich vielen Möglichkeiten:

$$f : x \mapsto \begin{cases} -2x - 1 & \text{falls } x < 0 \\ 2x & \text{falls } x \geq 0 \end{cases}$$

Die Mengen der natürlichen und der ganzen Zahlen erweisen sich in der Tat als gleichmächtig. Doch damit nicht genug. Auch die Menge der ra-

tionalen Zahlen \mathbb{Q} lässt sich bijektiv auf \mathbb{N} abbilden. Dies liegt daran, dass wir eine Bijektion zwischen \mathbb{Q} und \mathbb{N}^2 herstellen können, und sich die Menge \mathbb{N}^2 bijektiv auf die Menge \mathbb{N} abbilden lässt. Wie dies funktioniert, ist in Abbildung 2.21 skizziert. Alle Elemente von \mathbb{N}^2 sind in einer Matrix angeordnet, die sich unendlich weit nach rechts und nach unten ausbreitet. Ein Element $(x, y) \in \mathbb{N}^2$ können wir eindeutig einem Element der Matrix zuordnen, indem wir x mit der Spalte und y mit der Zeile des entsprechenden Felds identifizieren. Wie die Abbildung zeigt, sind wir in der Lage, jedes Matrixelement (x, y) mit einer eindeutigen Nummer $\pi_{\mathbb{N}}(x, y)$ zu belegen, indem wir im Feld $(0, 0)$ beginnen und uns anschließend diagonal zwischen der oberen und der linken Seite hin- und herbewegen. Die entstehende Abbildung $\pi_{\mathbb{N}} : \mathbb{N}^2 \rightarrow \mathbb{N}$ heißt *Cantor'sche Paarungsfunktion* und lässt sich über die nachstehende Formel direkt berechnen:

$$\pi_{\mathbb{N}}(x, y) = y + \sum_{i=0}^{x+y} i = y + \frac{(x+y)(x+y+1)}{2}$$

Über die Existenz einer bijektiven Zuordnung zwischen \mathbb{N}^2 und \mathbb{N} haben wir gezeigt, dass beide Mengen die gleiche Mächtigkeit besitzen.

Mithilfe der Cantor'schen Paarungsfunktion lassen sich weitere Mengen als gleichmächtig identifizieren. Durch die rekursive Anwendung sind wir z. B. in der Lage, nicht nur jedem Paar $(x, y) \in \mathbb{N}^2$, sondern auch jedem Tripel $(x, y, z) \in \mathbb{N}^3$ ein eindeutiges Element in \mathbb{N} zuzuordnen. Hierzu definieren wir die Funktion $\pi_{\mathbb{N}}^3 : \mathbb{N}^3 \rightarrow \mathbb{N}$ wie folgt:

$$\pi_{\mathbb{N}}^3(x, y, z) := \pi_{\mathbb{N}}(\pi_{\mathbb{N}}(x, y), z)$$

Führen wir den Gedanken in dieser Richtung fort, so erhalten wir mit

$$\pi_{\mathbb{N}}^1(x_1) := x_1 \quad (2.13)$$

$$\pi_{\mathbb{N}}^{n+1}(x_1, \dots, x_n, x_{n+1}) := \pi_{\mathbb{N}}(\pi_{\mathbb{N}}^n(x_1, \dots, x_n), x_{n+1}) \quad (2.14)$$

eine bijektive Abbildung von \mathbb{N}^n auf \mathbb{N} . Damit ist bewiesen, dass der n -dimensionale Zahlenraum \mathbb{N}^n für $n \geq 1$ stets die gleiche Mächtigkeit besitzt wie die Grundmenge \mathbb{N} selbst – unabhängig davon, wie groß wir die Dimension $n \in \mathbb{N}^+$ auch wählen.

An dieser Stelle drängt sich unweigerlich die Frage auf, wie eine Menge M beschaffen sein muss, um mächtiger zu sein als die Menge der natürlichen Zahlen \mathbb{N} . In der Tat haben wir mit den reellen Zahlen \mathbb{R} eine solche Menge weiter oben bereits eingeführt. Die Überabzählbarkeit der reellen Zahlen wurde von Georg Cantor erstmals im Jahr 1874 formal gezeigt [14]. Der Beweis ist vergleichsweise abstrakt und nicht ganz einfach zu verstehen (siehe hierzu [50]).

Hilberts Hotel ist ein Gedankenspiel, das eine verblüffende Eigenschaft der Unendllichkeit anschaulich demonstriert [15]. Im Kern geht es um die Frage, ob in einem Hotel mit unendlich vielen Zimmern selbst dann ein neuer Gast aufgenommen werden kann, wenn es restlos ausgebucht ist.



Unter normalen Umständen würde Ihnen jeder Portier mit Schulterzucken begegnen. In Hilberts Hotel ist es dagegen kein Problem, ein freies Zimmer zu beschaffen. Um einen neuen Gast aufzunehmen, müssen wir lediglich alle bisherigen Gäste bitten, aus ihrem Zimmer mit der Nummer n in das Zimmer mit der Nummer $n + 1$ zu wechseln. Auf diese Weise wird das erste Zimmer frei, in das wir unseren neuen Gast einquartieren können.



Das Gedankenspiel stellt unsere menschliche Intuition auf eine harte Probe und mag zunächst mehr Verwirrung stiften als Klarheit schaffen. Auf den zweiten Blick wird deutlich, dass wir einen zusätzlichen Gast genau deshalb aufnehmen können, weil die Mengen \mathbb{N} und \mathbb{N}^+ gleichmächtig sind. Wenn jeder Gast sein Nachbarzimmer bezieht, entspricht dies der Anwendung der Funktion $f : n \mapsto n + 1$, die \mathbb{N} bijektiv auf die Menge \mathbb{N}^+ abbildet.

Abbildung 2.22: Das Diagonalisierungsargument. Gäbe es eine bijektive Abbildung von den natürlichen auf die reellen Zahlen, so müsste sich die (unendlich lange) Ziffernfolge jeder reellen Zahl in einer Zeile der Zuordnungsmatrix wiederfinden lassen. Unabhängig von der gewählten Zuordnung sind wir jedoch stets im Stande, die Ziffernfolge einer reellen Zahl zu konstruieren, die nicht in der Matrix vorkommt. Diese können wir erzeugen, indem wir uns entlang der Hauptdiagonalen von links oben nach rechts unten bewegen und die vorgefundene Ziffer um eins erhöhen oder erniedrigen. Die konstruierte Ziffernfolge kommt nirgends in der Matrix vor, da sie sich von jener der i -ten Zeile per Konstruktion in der i -ten Ziffer unterscheidet. Die Überlegung zeigt, dass eine bijektive Zuordnung der Elemente aus \mathbb{R} zu den Elementen aus \mathbb{N} nicht möglich ist. Kurzum: Die Menge der reellen Zahlen ist nicht abzählbar.

$f(0) =$	0	,	5	4	9	0	0	7	5	8	...
$f(1) =$	0	,	7	1	4	4	5	6	6	3	...
$f(2) =$	0	,	7	4	3	9	6	1	4	2	...
$f(3) =$	0	,	2	3	1	1	1	7	4	5	...
$f(4) =$	0	,	2	7	9	7	7	4	0	0	...
$f(5) =$	0	,	3	8	6	4	8	7	2	8	...
$f(6) =$	0	,	5	6	0	6	9	3	7	4	...
$f(7) =$	0	,	2	1	3	4	4	9	9	9	...
\vdots											

1877 bewies Cantor seine Aussage erneut – diesmal auf verblüffend einfache Weise. Den Kern des Beweises bildet das von ihm entwickelte *Diagonalisierungsargument*, eine genauso leistungsfähige wie intuitive Methode, um eine Menge als überabzählbar zu identifizieren. Cantor stellte die folgende Überlegung an: Angenommen, die beiden Mengen \mathbb{N} und \mathbb{R} sind gleichmächtig, so muss eine bijektive Abbildung $f : \mathbb{N} \rightarrow \mathbb{R}$ existieren, die jedes Element $x \in \mathbb{N}$ eineindeutig auf ein Element $f(x) \in \mathbb{R}$ abbildet. Listen wir die Funktionswerte $f(0), f(1), f(2), \dots$ von oben nach unten auf, so entsteht eine zweidimensionale Matrix, wie sie in Abbildung 2.22 skizziert ist. Formal entspricht das Element in Spalte x und Zeile y der x -ten Ziffer der Dezimalbruchdarstellung von $f(y)$. Natürlich können wir nur einen winzigen Ausschnitt der entstehenden Matrix zeichnen, da die Funktion f für unendlich viele Werte $y \in \mathbb{N}$ definiert ist und sich die Dezimalbruchdarstellung der reellen Zahlen $f(y)$ über unendlich viele Ziffern erstreckt. Kurzum: Die aufgestellte Matrix besteht aus unendlich vielen Spalten und Zeilen.

Mithilfe des Diagonalisierungsarguments können wir zeigen, dass die Matrix nie vollständig sein kann. Unabhängig von der konkreten Wahl von f existieren reelle Zahlen, die nicht in der Matrix enthalten sind und damit die Bijektivität von f ad absurdum führen. Wir konstruieren eine

solche Zahl, indem wir uns entlang der Hauptdiagonalen von links oben nach rechts unten bewegen und die vorgefundenen Ziffern jeweils um eins erhöhen oder erniedrigen. Die entstehende Ziffernfolge interpretieren wir als die Nachkommaziffern einer reellen Zahl r . Wäre f eine bijektive Abbildung von \mathbb{N} auf \mathbb{R} , so müsste auch die Zahl r in irgendeiner Zeile vorkommen. Aufgrund des gewählten Konstruktionsschemas ist jedoch sichergestellt, dass sich die reelle Zahl der i -ten Zeile in der i -ten Ziffer von r unterscheidet. Die Annahme, eine bijektive Zuordnung zwischen \mathbb{N} zu \mathbb{R} könnte existieren, führt zu einem unmittelbaren Widerspruch. Folgerichtig ist jeder Versuch, die reellen Zahlen nacheinander durchzumerken, zum Scheitern verurteilt. Kurzum: Die Mengen \mathbb{N} und \mathbb{R} stehen stellvertretend für zwei verschiedene Unendlichkeiten.

Trotzdem gelten einige der Eigenschaften, die wir für die Menge \mathbb{N} herausgearbeitet haben, auch in der Menge der reellen Zahlen. So sind wir auch hier in der Lage, ein Tupel $(x, y) \in \mathbb{R}$ bijektiv auf die Menge \mathbb{R} abzubilden. Abbildung 2.23 skizziert die zugrunde liegende Konstruktionsidee. Die beiden reellen Zahlen $x \in \mathbb{R}$ und $y \in \mathbb{R}$ werden zu einer gemeinsamen reellen Zahl $\pi_{\mathbb{R}}(x, y) \in \mathbb{R}$ verschmolzen, indem die Vor- und Nachkommaziffern reißverschlussartig miteinander verschrankt werden.

Erneut hat uns der Cantor'sche Zugang zur Unendlichkeit eine verblüffende Eigenschaft von Zahlenmengen offen gelegt. Die Gleichmächtigkeit von \mathbb{R} und \mathbb{R}^2 bedeutet, dass eine Gerade in der Ebene gleich viele Punkte besitzt wie die Ebene selbst. Wir sind damit in der Lage, die Punkte der Ebene verlustfrei auf die Punkte einer Geraden abzubilden. Ebenso ist es möglich, die Ebene lückenlos mit den Punkten einer Geraden zu belegen (vgl. Abbildung 2.24).

Kombinieren wir die Aufrufe von $\pi_{\mathbb{R}}$ wieder rekursiv miteinander, so entsteht für jede natürliche Zahl $n \in \mathbb{N}$ eine Abbildung $\pi_{\mathbb{R}}^n$, die den n -dimensionalen Zahlenraum \mathbb{R}^n bijektiv auf \mathbb{R} reduziert. Formal ist die Abbildung $\pi_{\mathbb{R}}^n$, in Analogie zu den Gleichungen (2.13) und (2.14), wie folgt definiert:

$$\begin{aligned}\pi_{\mathbb{R}}^1(x_1) &:= x_1 \\ \pi_{\mathbb{R}}^{n+1}(x_1, \dots, x_n, x_{n+1}) &:= \pi_{\mathbb{R}}(\pi_{\mathbb{R}}^n(x_1, \dots, x_n), x_{n+1})\end{aligned}$$

Am Beispiel der reellen Zahlen haben wir gesehen, dass eine Unendlichkeit existiert, die mächtiger ist als jene der natürlichen Zahlen. Das Ergebnis wirft die Frage auf, ob es eine weitere Unendlichkeit gibt, die wiederum mächtiger ist als jene der reellen Zahlen. Der folgende Satz von Cantor liefert uns eine positive Antwort auf diese Frage.

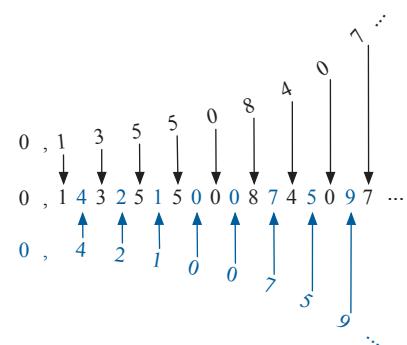


Abbildung 2.23: Im Reißverschlussverfahren lassen sich zwei reelle Zahlen zu einer einzigen reellen Zahl verschmelzen. Auf diese Weise lässt sich eine bijektive Abbildung von \mathbb{R}^2 auf \mathbb{R} konstruieren und damit die Gleichmächtigkeit der beiden Mengen zeigen.

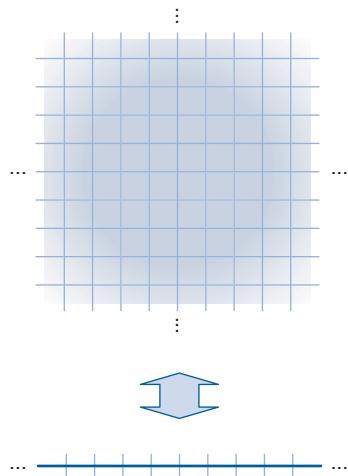


Abbildung 2.24: Obwohl die dargestellte Fläche mehr Druckfarbe verbraucht als das Liniensegment, besitzen die zweidimensionale Ebene und die eindimensionale Gerade die gleiche „Anzahl“ reeller Punkte. Jeder Punkt des einen geometrischen Objekts lässt sich eindeutig auf einen Punkt des anderen abbilden.

Mithilfe des Diagonalisierungsarguments konnte Cantor beweisen, dass die Menge der reellen Zahlen – das *Kontinuum* – eine größere Mächtigkeit besitzt als die Menge der natürlichen Zahlen. Zugleich hatte Cantor die Vermutung, dass es keine unendliche Menge gibt, die bez. ihrer Kardinalität zwischen \mathbb{N} und \mathbb{R} liegt. So sehr sich Cantor auch bemühte, blieb es ihm verwehrt, zu Lebzeiten einen Beweis für seine *Kontinuumshypothese* zu finden.

Der deutsche Mathematiker David Hilbert war von der Wichtigkeit der Cantor'schen Vermutung überzeugt. Auf der Liste der 23 wichtigsten mathematischen Probleme, die er im Jahr 1900 auf dem internationalen Mathematikerkongress in Paris vortrug, avancierte die Klärung der Kontinuumshypothese an erster Stelle.

Nach vielen erfolglosen Versuchen gelang Kurt Gödel im Jahr 1937 ein entscheidender Durchbruch. Wie im Fall des Auswahlaxioms konnte er zeigen, dass die Kontinuumshypothese mit den anderen Axiomen der Zermelo-Fraenkel-Mengenlehre vereinbar ist. Wird die Hypothesen den ZF-Axiomen oder den ZFC-Axiomen hinzugefügt, so bleibt die Widerspruchsfreiheit erhalten, sofern diese innerhalb der restlichen Axiome gegeben ist. Einen Beweis der Kontinuumshypothese konnte Gödel nicht liefern und nährte gleichermaßen die Vermutung, dass ein solcher überhaupt nicht existiert.

Die endgültige Gewissheit ließ bis zum Jahr 1963 auf sich warten, als der amerikanische Mathematiker Paul Cohen zeigen konnte, dass auch die Negation der Kontinuumshypothese nicht im Widerspruch zur Zermelo-Fraenkel'schen Mengenlehre steht. Damit war das Rätsel um die Cantor'sche Vermutung geklärt. Die Kontinuumshypothese ist innerhalb des klassischen Axiomensystems weder beweisbar noch widerlegbar und kann als zusätzliches Axiom widerspruchsfrei hinzugefügt werden.



Satz 2.1 (Satz von Cantor)

Für jede Menge M ist die Potenzmenge 2^M mächtiger als M selbst.

Wir können diese Aussage beweisen, indem wir ein ähnliches Diagonalisierungsargument verwenden, mit dem wir bereits die Überabzählbarkeit der reellen Zahlen zeigen konnten. Auch hier gehen wir zunächst wieder von der Existenz einer bijektiven Abbildung $f : M \rightarrow 2^M$ aus und führen die Annahme anschließend zu einem Widerspruch.

Sei nun f eine solche Funktion, die M bijektiv auf die Menge 2^M abbildet. Für jedes Element $m \in M$ können wir zwei Fälle unterscheiden: Entweder ist m im Bildelement $f(m)$ enthalten ($m \in f(m)$) oder nicht ($m \notin f(m)$). Alle Elemente, auf die Letzteres zutrifft, wollen wir in der Menge T zusammenfassen:

$$T = \{m \in M : m \notin f(m)\}$$

Da f bijektiv und damit insbesondere auch surjektiv ist, muss ein Urbild m_T existieren mit $f(m_T) = T$. Wie für alle Elemente aus M gilt auch für das Element m_T entweder die Eigenschaft $m_T \in T$ oder $m_T \notin T$. Beide Fälle führen jedoch unmittelbar zu einem Widerspruch:

$$\begin{aligned} m_T \in T &\Rightarrow m_T \notin f(m_T) \Rightarrow m_T \notin T \\ m_T \notin T &\Rightarrow m_T \in f(m_T) \Rightarrow m_T \in T \end{aligned}$$

Damit haben wir gezeigt, dass es eine bijektive Funktion $f : M \rightarrow 2^M$ nicht geben kann. Aus dem Cantor'schen Satz ergeben sich zwei wichtige Konsequenzen. Zum einen zeigt er, dass es keine *maximale Unendlichkeit* gibt, d. h., wir sind nicht in der Lage, eine Universalmenge zu konstruieren, die mächtiger ist als alle anderen Mengen. Es scheint, als ob es die Unendlichkeit abermals schafft, sich jeglichen Grenzen zu entziehen. Zum anderen bringt der Satz eine hierarchische Ordnung in die unendliche Menge der verschiedenen Unendlichkeiten:

$$|\mathbb{N}| < |2^{\mathbb{N}}| < |2^{2^{\mathbb{N}}}| < |2^{2^{2^{\mathbb{N}}}}| < |2^{2^{2^{2^{\mathbb{N}}}}}| < \dots$$

Cantor verwendete den hebräischen Buchstaben Aleph (\aleph), um die Mächtigkeit einer unendlichen Menge zu beschreiben. Die kleinste Unendlichkeit wird mit der *Kardinalzahl* \aleph_0 bezeichnet; sie entspricht der Kardinalität der natürlichen Zahlen. Eine kleinere Unendlichkeit als $|\mathbb{N}|$ kann es nicht geben, da sich alle unendlichen Teilmengen von \mathbb{N} bijektiv auf \mathbb{N} abbilden lassen. Die nächstgrößere Unendlichkeit wird durch die Kardinalzahl \aleph_1 beschrieben und so fort.

2.4 Rekursion und induktive Beweise

Immer dann, wenn ein Problem, eine Funktion oder ein Verfahren durch sich selbst beschrieben wird, sprechen wir von einer *rekursiven Definition*. In der Regel ist die Rekursion so konstruiert, dass sich das gesuchte Ergebnis aus einem oder mehreren Teilergebnissen *kleinerer Ordnung* berechnen lässt. Meist ist die Ordnung nach unten beschränkt, d. h., es existieren nicht weiter zerlegbare Basisfälle, die direkt berechnet werden können. Eine rekursive Definition besteht damit immer aus zwei Teilen. Im ersten Teil werden die Basisfälle definiert und im zweiten Teil die Rekursionsregeln festgelegt.

Abbildung 2.25 demonstriert das Rekursionsprinzip am Beispiel der Fakultätsfunktion. Da nahezu alle Programmiersprachen Selbstaufrufe auf Funktionsebene unterstützen, lassen sich rekursive Definitionen eins zu eins in ein Programm umsetzen. In einer rekursiv programmierter Funktion werden die Basisfälle explizit ausprogrammiert und alle anderen Funktionswerte durch eine Reihe von Selbstaufrufen ermittelt.

Von Programmieranfängern wird das Rekursionsprinzip gerne gemieden. Zu Unrecht, wie sich an etlichen Beispielen belegen lässt. Mit ein wenig Übung lassen sich viele Algorithmen deutlich übersichtlicher implementieren, als es die streng iterative Programmierung erlaubt.

Auch wenn das Rekursionsprinzip im Bereich der Informatik einen prominenten Platz einnimmt, ist es keine Erfindung des Informationszeitalters. In der Mathematik ist das Prinzip seit Langem verankert, wie das *Rad des Theodorus* belegt. Diese geometrische Figur wird aus mehreren aneinander gereihten Dreiecken T_1, \dots, T_n gebildet, die dem nachstehenden rekursiven Konstruktionsschema folgen:

- T_1 ist ein rechtwinkliges Dreieck mit der Seitenlänge 1.
- Die Hypotenuse von T_n ist ein Schenkel von T_{n+1} . Der andere Schenkel besitzt die Länge 1.

Werden die Dreiecke aneinander gereiht, so entsteht die in Abbildung 2.26 dargestellte Wurzelschnecke.

Im nächsten Abschnitt werden wir mit der vollständigen Induktion ein wichtiges Beweisprinzip vorstellen, das sich die rekursive Struktur der natürlichen Zahlen zunutze macht. Anschließend werden wir die Methode in Abschnitt 2.4.2 verallgemeinern und mit der strukturellen Induktion ein Hilfsmittel an die Hand bekommen, mit dem sich Aussagen über rekursiv definierte Datenstrukturen formal beweisen lassen.

■ Rekursives Definitionsschema

$$n! := \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

■ Rekursive Implementierung

fakultaet.c

```
int fakultaet( int n )
{
    if (n == 0)
        return 1;
    else
        return
            n * fakultaet(n-1);
}
```

1
2
3
4
5
6
7
8

Abbildung 2.25: Rekursive Implementierung der Fakultätsfunktion

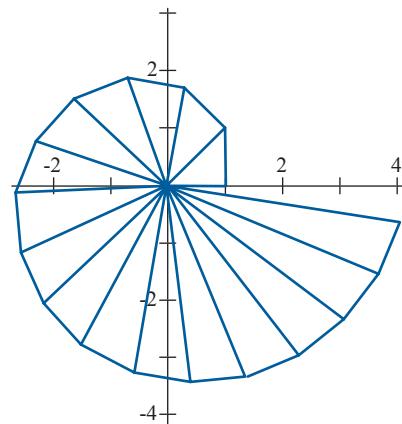


Abbildung 2.26: Das Rad des Theodorus (Wurzelschnecke) ist ein Beispiel einer rekursiv definierten geometrischen Struktur. Dem griechischen Gelehrten Theodorus (ca. 465 v. Chr.) gelang es mithilfe dieser Figur, die Zahlen $\sqrt{3}, \sqrt{5}, \sqrt{7}, \dots, \sqrt{17}$ als irrational zu identifizieren.

2.4.1 Vollständige Induktion

Die *vollständige Induktion* ist neben dem direkten Deduktionsbeweis und dem indirekten Widerspruchsbeweis die dritte grundlegende Beweistechnik der Mathematik. Ihre Berechtigung bezieht sie aus den in Abschnitt 2.3 eingeführten Peano-Axiomen. Das fünfte Peano-Axiom lautete wie folgt:

- P5) Enthält eine Teilmenge $M \subseteq \mathbb{N}$ die Zahl 0 und zu jedem Element n auch ihren Nachfolger $\text{succ}(n)$, so gilt $M = \mathbb{N}$.

Aus den elementaren Eigenschaften der natürlichen Zahlen lässt sich die folgende Verallgemeinerung herleiten:

- P5') Enthält eine Teilmenge $M \subseteq \mathbb{N}$ die Zahl n_0 und zu jedem Element $n \geq n_0$ auch ihren Nachfolger $\text{succ}(n)$, so gilt $\{n \mid n \geq n_0\} \subseteq M$.

Hieraus ergibt sich das Prinzip der vollständigen Induktion wie folgt:



Satz 2.2 (Vollständige Induktion)

Sei $n_0 \in \mathbb{N}$ und $A(n)$ eine parametrisierte Aussage über den natürlichen Zahlen. Wenn die folgenden beiden Aussagen gelten, so ist $A(n)$ für alle n mit $n \geq n_0$ wahr.

- $A(n_0)$ ist wahr.
- Für alle k mit $k \geq n_0$ gilt: Aus $A(k)$ folgt die Aussage $A(k+1)$.

Anwenden lässt sich die vollständige Induktion auf alle Aussagen, die von einem Parameter $n \in \mathbb{N}$ abhängen, wobei die Aussage *für alle* n ab einem gewissen Startwert n_0 bewiesen werden soll. Ein vollständiger Beweis setzt sich aus insgesamt drei Schritten zusammen:

- Induktionsanfang: Für den Startwert n_0 wird die Behauptung $A(n_0)$ bewiesen.
- Induktionsannahme: Für einen beliebigen Wert n mit $n \geq n_0$ wird die Aussage $A(n)$ als wahr angenommen.
- Induktionsschritt: Unter der Induktionsannahme wird der Beweis geführt, dass die Aussage $A(n+1)$ ebenfalls wahr ist.

Der Induktionsanfang, die Induktionsvoraussetzung und der Induktionsschritt stellen zusammen sicher, dass die Aussage *für alle n* Gültigkeit besitzt. Der Induktionsanfang beweist die Aussage für den Fall $n = n_0$. Die Gültigkeit für $n_0 + 1$ folgt durch Anwendung des Induktionsschritts auf den Fall n_0 , die Gültigkeit für $n_0 + 2$ durch Anwendung des Induktionsschritts auf den Fall $n_0 + 1$ und so fort.

Beachten Sie, dass wir die Induktionsannahme verstärken können, ohne die Gültigkeit des Beweisprinzips zu gefährden. Anstatt die Aussage $A(n)$ nur für ein einzelnes n als wahr anzunehmen, können wir die Gültigkeit von $A(k)$ für alle k mit $n_0 \leq k < n$ ebenfalls als wahr annehmen.

Um nicht im Nebel der Theorie zu versinken, wollen wir die folgende Aussage mit dem Mittel der vollständigen Induktion beweisen:

Satz 2.3

Ab einem gewissen Wert $n_0 \in \mathbb{N}$ gilt für alle $n \geq n_0$ die Abschätzung:

$$2^n < n! < n^n$$

Beweis

■ Induktionsanfang

Für $0 \leq n \leq 3$ ist die Abschätzung nicht erfüllt. Wir wählen $n_0 = 4$. Jetzt gilt $(2^4 = 16) < (4! = 24) < (4^4 = 256)$.

■ Induktionsvoraussetzung

Für ein gewisses $n \geq 4$ gelte die Abschätzung $2^n < n! < n^n$.

■ Induktionsschritt

Wir müssen zeigen, dass die Abschätzung

$$2^{n+1} < (n+1)! < (n+1)^{n+1}$$

erfüllt ist. Diese können wir mithilfe der Induktionsvoraussetzung und den elementaren Rechenregeln wie folgt herleiten:

$$\begin{aligned} 2^{n+1} &= 2 \cdot 2^n < (n+1) \cdot 2^n < (n+1) \cdot n! \\ &= (n+1)! \end{aligned}$$

$$\begin{aligned} (n+1)! &= (n+1) \cdot n! < (n+1)n^n < (n+1)(n+1)^n \\ &= (n+1)^{n+1} \end{aligned}$$

Damit ist die die Abschätzung für alle $n \geq 4$ erfüllt und die Behauptung bewiesen. \square

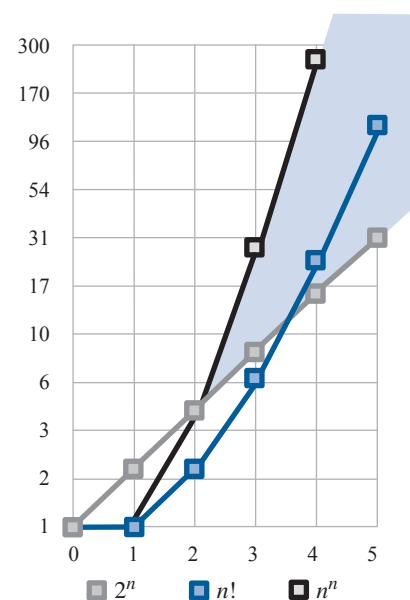
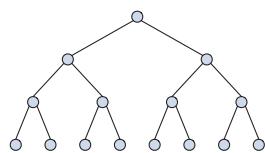
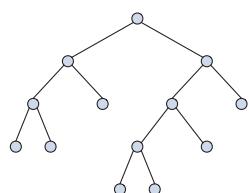


Abbildung 2.27: Die Funktionen 2^n , $n!$ und n^n im Vergleich. Ab $n = 4$ bilden 2^n und n^n einen Korridor, den die Fakultätsfunktion $n!$ nicht mehr verlässt. Mit der Technik der vollständigen Induktion lässt sich die Eigenschaft in wenigen Schritten beweisen.

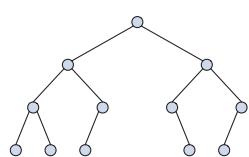
- Saturiert, balanciert



- Saturiert, nicht balanciert



- Nicht saturiert, balanciert



- Nicht saturiert, nicht balanciert

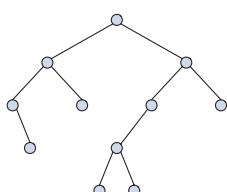


Abbildung 2.28: Balancierte Binäräbäume zeichnen sich dadurch aus, dass alle Pfade von der Wurzel zu den Blättern eine um höchstens 1 voneinander abweichende Länge aufweisen. Saturierte Binäräbäume besitzen die Eigenschaft, dass jeder Knoten entweder 0 oder 2 Söhne hat.

2.4.2 Strukturelle Induktion

Die strukturelle Induktion ist ein mathematisches Beweisschema, das uns erlaubt, Induktionsbeweise über beliebige rekursiv definierte Strukturen zu führen. Genau wie im Fall der vollständigen Induktion beweisen wir in einem strukturellen Induktionsbeweis die Behauptung zunächst für einen oder mehrere Basisfälle. Anschließend zeigen wir im Induktionsschritt, dass sich die Gültigkeit der Behauptung auf das nächstkomplexere Objekt übertragen lässt. Konkret umfassen in einem strukturellen Induktionsbeweis die Basisfälle alle nicht zusammengesetzten Elemente. Setzt sich ein Element aus mehreren Teilelementen zusammen, so wird die Behauptung unter der Annahme bewiesen, dass sie für alle Teilelemente bereits gezeigt wurde.

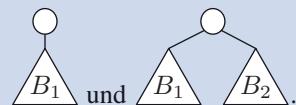
In der Informatik spielt die strukturelle Induktion die gleiche Rolle wie die vollständige Induktion in der Mathematik. Insbesondere im Bereich der Software-Technik lässt sich das Beweisprinzip gewinnbringend einsetzen, da viele Datenstrukturen einem rekursiven Konstruktionsschema folgen. Was wir unter einer solchen Definition konkret zu verstehen haben, wollen wir am Beispiel des *Binärbaums* herausarbeiten.



Definition 2.13 (Binärbaum)

Ein *Binärbaum* (*binary tree*) über einer *Blättermenge* L ist wie folgt definiert:

- Jedes Blatt $l \in L$ ist ein Binärbaum.
- Sind B_1 und B_2 Binäräbäume, dann sind es auch



Binäräbäume sind damit nichts anderes als spezielle Bäume mit der Eigenschaft, dass jeder Knoten höchstens 2 Söhne hat. Die Anzahl der Knoten eines Baums bezeichnen wir mit $|B|$. Die maximale Anzahl von Kanten, die wir von dem Knoten bis zu einem Blatt ablaufen können, heißt die *Tiefe* des Baums. Ein Binärbaum heißt *saturiert*, wenn jeder Knoten entweder 0 oder 2 Söhne hat. Ein Binärbaum heißt *balanciert*, wenn die Pfade von der Wurzel zu den Blättern eine um höchstens 1 voneinander abweichende Länge aufweisen. Beachten Sie, dass die Produktionsregel in Definition 2.13 nicht zwangsläufig einen balancierten Baum entstehen lässt (vgl. Abbildung 2.28).

Mithilfe der strukturellen Induktion können wir viele Eigenschaften von Binärbäumen beweisen. Als Beispiel wollen wir die Gültigkeit des folgenden Satzes zeigen, der uns in Abschnitt 4.4.3 im Zusammenhang mit dem Pumping-Lemma für kontextfreie Sprachen erneut begegnen wird.

Satz 2.4

Seien $k \in \mathbb{N}$ eine natürliche Zahl und B ein Binärbaum mit $|B| \geq 2^k$. Dann existiert in B mindestens ein Pfad der Länge $\geq k$.

Beweis

Wir beschränken uns darauf, den Satz für saturierte Binärbäume zu zei- gen. Der Beweis des allgemeinen Falls ergibt sich analog.

■ Induktionsanfang

Besteht ein Binärbaum B aus einem einzigen Blatt, so gilt

$$|B| = 1 = 2^0.$$

Der Baum besitzt trivialerweise einen Pfad der Länge 0.

■ Induktionsvoraussetzung

B_1 und B_2 seien zwei Binärbäume mit $|B_1| \geq 2^{k_1}$ und $|B_2| \geq 2^{k_2}$. Wir nehmen an, B_1 und B_2 erfüllen die zu beweisenden Eigenschaft, d. h., es gib einen Pfad P_1 in B_1 der Länge $\geq k_1$ und einen Pfad P_2 in B_2 der Länge $\geq k_2$.

■ Induktionsschritt

Wir kombinieren B_1 und B_2 zu einem Binärbaum B und zeigen, dass auch B die postulierte Eigenschaft erfüllt. Zwei Fälle sind zu unterscheiden (vgl. Abbildung 2.29):

■ Fall 1: B_1 enthält mindestens so viele Blätter wie B_2

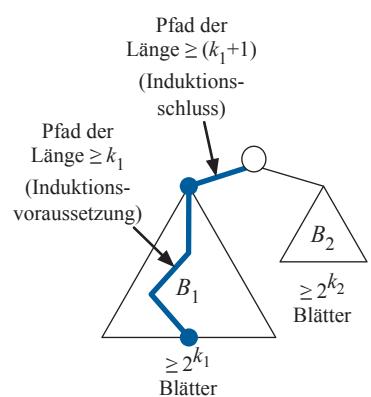
B hat dann maximal $2 \cdot 2^{k_1} = 2^{k_1+1}$ Knoten. Da in B_1 nach Induktionsvoraussetzung ein Pfad der Länge k_1 existiert, besitzt B einen Pfad der Länge $k_1 + 1$.

■ Fall 2: B_2 enthält mindestens so viele Blätter wie B_1

B hat dann maximal $2 \cdot 2^{k_2} = 2^{k_2+1}$ Knoten. Da in B_2 nach Induktionsvoraussetzung ein Pfad der Länge k_2 existiert, besitzt B einen Pfad der Länge $k_2 + 1$.

Damit ist die Behauptung für alle Binärbäume bewiesen. \square

■ Fall 1: $|B_1| \geq |B_2|$



■ Fall 2: $|B_2| \geq |B_1|$

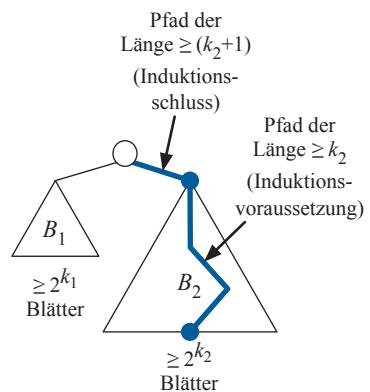


Abbildung 2.29: Grafische Veranschauli- chung des Induktionsschritts

2.5 Übungsaufgaben

Aufgabe 2.1

**Webcode
2859**

Die Mengen M' , M'' und M_k ($k \in \mathbb{N}$) seien wie folgt definiert:

$$M' := \{1, 3, 5, 7\}, \quad M'' := \{2, 3, 7, 11\}, \quad M_k := \{-k, -k+1, \dots, 0, \dots, k-1, k\}$$

Bestimmen Sie die Menge M mit

- | | | | |
|----------------------|---------------------------|------------------|-------------------------------------|
| a) $M = M' \cup M''$ | c) $M = M' \setminus M''$ | e) $M = 2^{M_1}$ | g) $M = \bigcup_{k=1}^{\infty} M_k$ |
| b) $M = M' \cap M''$ | d) $M = M'' \setminus M'$ | f) $M = 2^{M_2}$ | h) $M = \bigcap_{k=1}^{\infty} M_k$ |

Aufgabe 2.2

**Webcode
2339**

R_1 und R_2 seien wie folgt definiert:

$$R_1 := \{(2, 1), (3, 1), (4, 1), \dots, (3, 2), (4, 2), (5, 2), \dots, (4, 3), (5, 3), (6, 3), \dots\}$$

$$R_2 := \{(1, 2), (1, 3), (1, 4), \dots, (2, 3), (2, 4), (2, 5), \dots, (3, 4), (3, 5), (3, 6), \dots\}$$

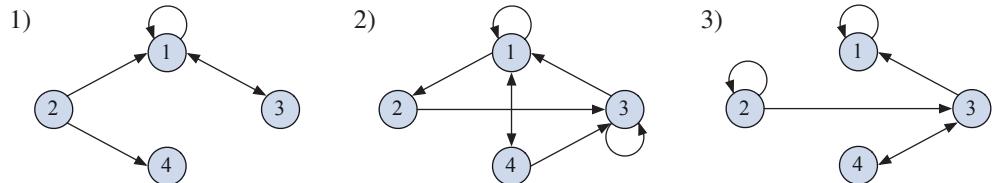
Welche bekannten Relationen verbergen sich hinter

- | | | | |
|----------|----------|-------------------|------------------------------|
| a) R_1 | b) R_2 | c) $R_1 \cup R_2$ | d) $\overline{R_1 \cup R_2}$ |
|----------|----------|-------------------|------------------------------|

Aufgabe 2.3

**Webcode
2184**

Gegeben seien die folgenden Relationen:



- a) Ordnen Sie jedem Relationengraphen eine der folgenden Adjazenzmatrizen zu:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

- b) Berechnen Sie für jede Relation die reflexiv-transitive Hülle.

Mengen sind von Natur aus ungeordnet und können ein und dasselbe Elemente immer nur einmal enthalten.

Aufgabe 2.4**Webcode****2688**

- Wie lassen sich trotzdem geordnete Folgen von Elementen darstellen, in denen Elemente doppelt vorkommen dürfen?
- Codieren Sie die Folge $\langle 1, 1, 2, 3, 4, 4 \rangle$ nach dem entwickelten Schema.

Die Menge der *komplexen Zahlen* \mathbb{C} ist eine Erweiterung der Menge der reellen Zahlen \mathbb{R} . Eine komplexe Zahl $c \in \mathbb{C}$ besteht aus einem *Realanteil* $x \in \mathbb{R}$ und einem *Imaginäranteil* $y \in \mathbb{R}$ und wird in der Form $z = x + yi$ notiert. i heißt die *imaginäre Einheit* und erfüllt die Eigenschaft $i^2 = -1$. In der Menge der komplexen Zahlen lässt sich gewohnt rechnen:

$$(3 + 2i) + (5 + 7i) = (3 + 5) + (2 + 7)i = 8 + 9i$$

$$(3 + 2i) \cdot (5 + 7i) = 15 + 21i + 10i + 14 \cdot i^2 = 1 + 31i$$

Zeigen oder widerlegen Sie, dass \mathbb{R} und \mathbb{C} die gleiche Kardinalität besitzen.

Aufgabe 2.5**Webcode****2195**

Die nachstehend abgebildeten Zahlenreihen demonstrieren, wie sich die natürlichen Zahlen mithilfe von Mengen darstellen lassen. Die linke Reihe heißt Zermelo'sche Zahlenreihe, benannt nach dem Mengentheoretiker Ernst Zermelo. Die rechte Reihe heißt Neumann'sche Zahlenreihe und geht auf den Mathematiker John von Neumann zurück.

Aufgabe 2.6**Webcode****2010**

■ Zermelo'sche Zahlenreihe

$$\begin{aligned} 0 &= \emptyset \\ 1 &= \{\emptyset\} \\ &= \{\emptyset\} \\ 2 &= \{1\} \\ &= \{\{\emptyset\}\} \\ 3 &= \{2\} \\ &= \{\{\{\emptyset\}\}\} \\ 4 &= \{3\} \\ &= \{\{\{\emptyset\}\}\} \\ &\dots \end{aligned}$$

■ Von Neumann'sche Zahlenreihe

$$\begin{aligned} 0 &= \emptyset \\ 1 &= \{\emptyset\} \\ &= \{\emptyset\} \\ 2 &= \{\{0\}\} \\ &= \{\emptyset, \{\emptyset\}\} \\ 3 &= \{\{0, 1\}\} \\ &= \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\ 4 &= \{\{0, 1, 2\}\} \\ &= \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\} \\ &\dots \end{aligned}$$

- Wie lauten die rekursiven Bildungsschemata, die sich hinter den beiden Zahlenreihen verbergen?

- b) Wie viele Elemente besitzt das n -te Glied der Neumann'schen Zahlenreihe? Welche Beziehung besteht zwischen diesem Wert und der repräsentierten Zahl?
- c) Sind die Elemente der Neumann'schen Zahlenreihe bezüglich der Teilmengenrelation \subseteq total geordnet? Lässt sich die Teilmengenbeziehung arithmetisch deuten?

Aufgabe 2.7
**Webcode
2098**

Mit $<$ sei eine Ordnungsrelation auf einer Menge M gegeben. M heißt *wohlgeordnet*, wenn

- M bezüglich $<$ total geordnet ist und zusätzlich
- jede nichtleere Teilmenge $N \subseteq M$ ein Element enthält, das bezüglich $<$ minimal ist.

In diesem Fall nennen wir $<$ eine *Wohlordnung*. Welche Mengen sind bezüglich der arithmetischen Kleiner-Relation wohlgeordnet?

- a) \mathbb{N}
b) \mathbb{Z}

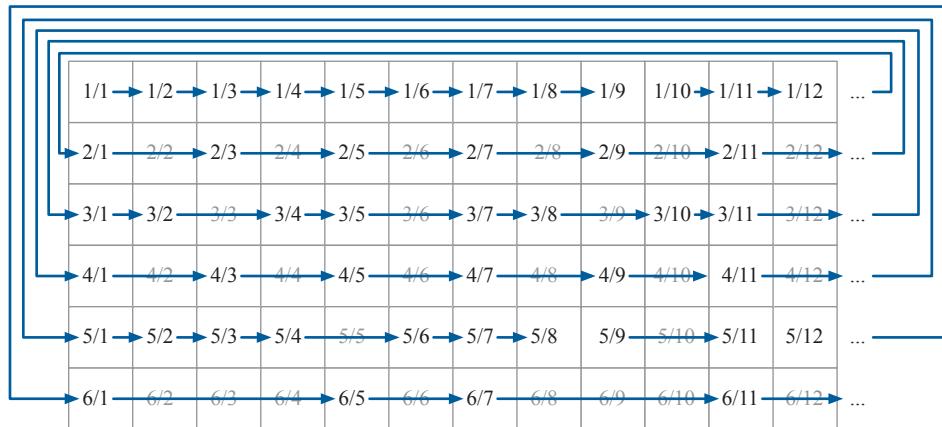
- c) \mathbb{Q}
d) \mathbb{R}

Aufgabe 2.8
**Webcode
2557**

Auf der Menge der rationalen Zahlen \mathbb{Q} definieren wir die Ordnungsrelation \prec wie folgt:

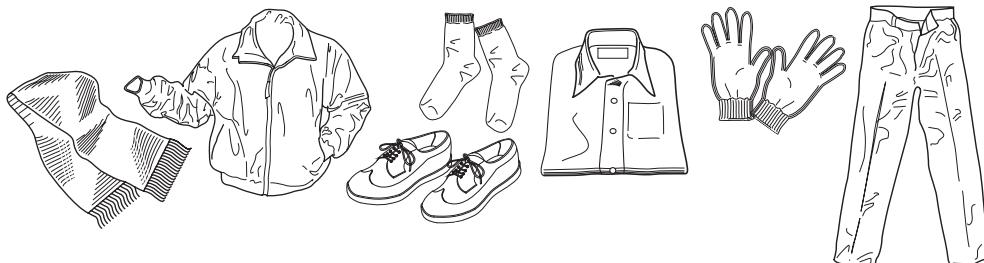
$$\frac{p_1}{q_1} \prec \frac{p_2}{q_2} : \Leftrightarrow \begin{cases} p_1 < p_2 & \text{falls } p_1 \neq p_2 \\ q_1 < q_2 & \text{falls } p_1 = p_2 \end{cases}$$

Hierin stehen $\frac{p_1}{q_1}$ und $\frac{p_2}{q_2}$ für zwei Brüche in gekürzter Darstellung. Grafisch lässt sich die Ordnungsrelation folgendermaßen veranschaulichen:



Ist die Ordnung total? Ist sie eine Wohlordnung?

Bilden Sie über der folgenden Menge von Gegenständen eine Ordnungsrelation R . Diese soll ausdrücken, in welcher Reihenfolge Sie die Kleidungsstücke anlegen müssen, um ordentlich angezogen das Haus zu verlassen:

Aufgabe 2.9**Webcode****2904**

Ist die von Ihnen erzeugte „Kleiderordnung“ partiell oder total?

Sei M eine endliche Menge mit n Elementen. Die Anzahl der Möglichkeiten, M in k Äquivalenzklassen zu unterteilen, wird durch die *Stirling-Zahl* $S(n, k)$ beschrieben. Sie ist nach dem schottischen Mathematiker James Stirling benannt.

Aufgabe 2.10**Webcode****2909**

- Wie viele Möglichkeiten existieren, um 2-elementige, 3-elementige und 4-elementige Mengen in 2 Klassen einzuteilen? Mit anderen Worten: Bestimmen Sie die Stirling-Zahlen $S(n, 2)$ für $2 \leq n \leq 4$.
- Für natürliche Zahlen n, k mit $1 \leq k \leq n$ lässt sich die Stirling-Zahl mithilfe der folgenden Formel berechnen:

$$S(n, k) = \frac{1}{k!} \sum_{i=0}^k (-1)^{k-i} \binom{k}{i} i^n$$

Werten Sie die Formel für alle n, k mit $1 \leq k \leq 4$ und $k \leq n \leq 8$ aus und tragen Sie die Funktionswerte in die folgende Tabelle ein. Verwenden Sie die ausgefüllte Tabelle, um Ihre Ergebnisse aus Teilaufgabe a) zu verifizieren.

	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$
$k = 1$								
$k = 2$								
$k = 3$								
$k = 4$								

Aufgabe 2.11**Webcode
2870**

Die Kreiszahl π und die Euler'sche Konstante e sind irrational, d. h., wir können ihren Wert in Dezimalbruchschreibweise nur annähern. Um sie exakt niederzuschreiben, müsste dieses Buch unendlich viele Seiten besitzen. Aus pragmatischen Gründen wollen wir uns deshalb mit der Angabe der ersten 8 Nachkommastellen begnügen:

$$\pi = 3,14159265\ldots \quad e = 2,71828182\ldots$$

Bis heute ist nicht bekannt, ob die Summe $\pi + e$ bzw. die Differenz $\pi - e$ ebenfalls irrational ist. Zeigen Sie, dass zumindest eine der beiden Zahlen irrational sein muss.

Aufgabe 2.12

Werten Sie die folgenden beiden Ausdrücke aus:

**Webcode
2390**

$$3^{(3^3)} = \boxed{\hspace{2cm}}$$

$$(3^3)^3 = \boxed{\hspace{2cm}}$$

Welcher Wert entspricht dem Ausdruck $3 \uparrow^2 3$?

Aufgabe 2.13**Webcode
2955**

Gegeben seien die folgenden Mengen:

- | | | | |
|---------------------|-----------------------|-----------------------|------------------------|
| 1) \mathbb{N} | 4) $2^{\mathbb{N}^2}$ | 7) $2^{\mathbb{Q}}$ | 10) \mathbb{R}^2 |
| 2) \mathbb{N}^2 | 5) \mathbb{Q} | 8) $2^{\mathbb{Q}^2}$ | 11) $2^{\mathbb{R}}$ |
| 3) $2^{\mathbb{N}}$ | 6) \mathbb{Q}^2 | 9) \mathbb{R} | 12) $2^{\mathbb{R}^2}$ |

- a) Welche der genannten Mengen sind gleichmächtig?
 b) Ordnen Sie die Mengen entsprechend ihrer Kardinalität hierarchisch an.

Aufgabe 2.14**Webcode
2598**

Sind die Mengen $[-1; 1]$ und $(-1; 1)$ gleichmächtig? Konstruieren Sie eine Bijektion zwischen den beiden Mengen oder begründen Sie, warum eine solche Abbildung nicht existieren kann.



Mit der *Cantor'schen Paarungsfunktion* $\pi_{\mathbb{N}}$ haben Sie eine bijektive Abbildung zwischen \mathbb{N} und \mathbb{N}^2 kennen gelernt. Die folgenden beiden Zuordnungsvorschriften sind der Cantor'schen Paarungsfunktion sehr ähnlich. Die linke definiert eine Bijektion zwischen \mathbb{N} und \mathbb{N}^2 und die rechte eine Bijektion zwischen \mathbb{N}^+ und \mathbb{N}^{+2} .

Aufgabe 2.15



Webcode

2787

Finden Sie geschlossene mathematische Formeln für die dargestellten Funktionen.

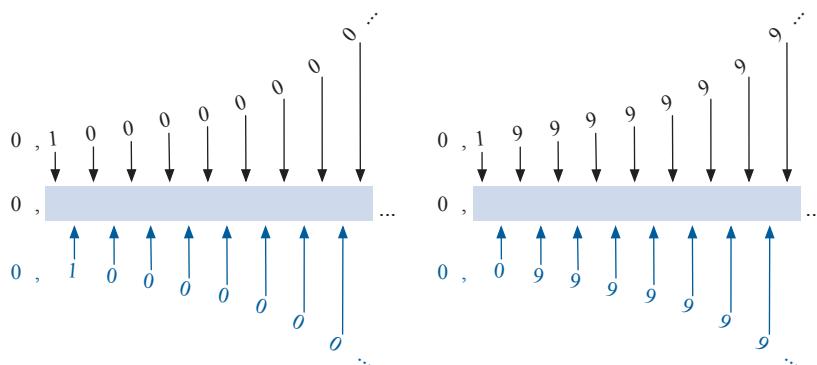
Auf Seite 63 haben wir dargelegt, dass eine bijektive Abbildung zwischen den Mengen \mathbb{R}^2 und \mathbb{R} konstruiert werden kann, indem die Ziffernfolgen zweier reeller Zahlen reißverschlussartig verschmolzen werden. Führen Sie die Konstruktion für die nachstehenden Beispiele durch.

Aufgabe 2.16



Webcode

2707



Was stellen Sie fest?

Aufgabe 2.17**Webcode
2532**

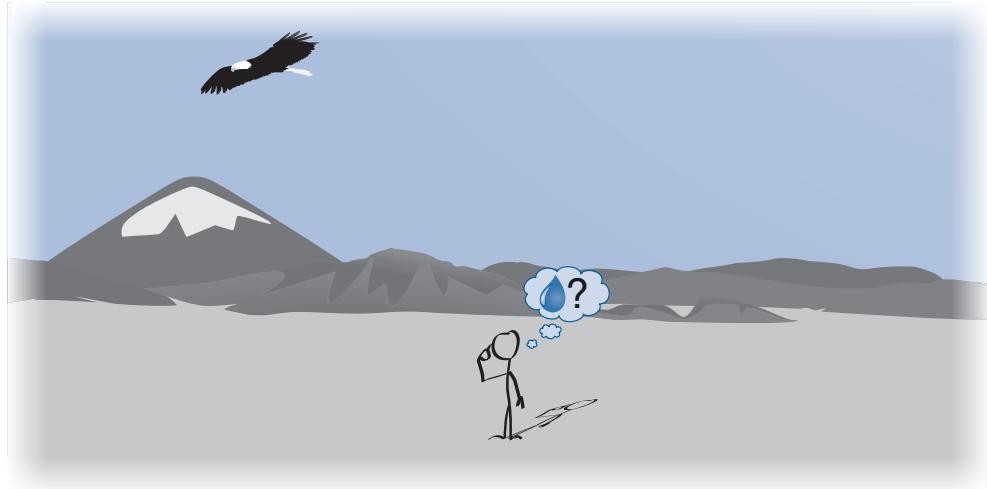
Stellen Sie sich vor, Sie sind der neue Pächter von Hilberts Hotel. Wie immer sind alle der unendlich vielen Zimmer restlos ausgebucht. Eines Tages hält *Hilberts Bus* vor Ihrem Hotel. Dieser verfügt über unendlich viele Sitzplätze, die alle mit einer eindeutigen Nummer $i \in \mathbb{N}$ versehen sind.



- Zu Ihrem Schrecken stellen Sie fest, dass der Bus vollständig besetzt ist. Beweisen Sie Ihr Organisationstalent und bringen Sie trotzdem alle neu angereisten Gäste in Ihrem Hotel unter. Welches Zimmer darf der Fahrgäst auf dem i -ten Sitzplatz beziehen?
- Die Güte Ihres Hotels spricht sich schnell herum und so dauert es nicht lange, bis eines Tages unendlich viele Hilbert-Busse gleichzeitig ihre Gäste vor Ihrem Haus abliefern. Auch hier trägt jeder Bus eine eindeutige Nummer $j \in \mathbb{N}$. Können Sie den großen Andrang immer noch bewältigen und alle Gäste aufnehmen?

Aufgabe 2.18**Webcode
2839**

Sie stehen inmitten der fiktiven *Hilbert-Wüste*, die sich in alle Richtungen unendlich weit ausbreitet. Sie wissen um die Existenz eines $1 \text{ km} \times 1 \text{ km}$ großen Wasserreservoirs, das sich an einer unbekannten Stelle in unbekannter Tiefe unter dem Wüstenboden verbirgt. Geben Sie eine Methode an, mit der Sie das Reservoir in endlicher Zeit finden werden!



Vorsicht bei Induktionsbeweisen! Die vollständige Induktion verfolgt ein einfaches Grundprinzip, ihre Anwendung sollte jedoch stets mit Bedacht geschehen. So lässt sich die Aussage

Aufgabe 2.19

Weocode
2422

„Wenn sich unter n Pferden ein Schimmel befindet, dann sind alle Pferde Schimmel.“

augenscheinlich mithilfe der vollständigen Induktion beweisen. Im Induktionsanfang zeigen wir die Aussage für $n = 1$. Besteht eine Gruppe aus einem einzigen Pferd und ist darin ein Schimmel enthalten, so sind offensichtlich alle Pferde dieser Gruppe Schimmel. Kurzum: Die Aussage ist für $n = 1$ richtig.

Jetzt nehmen wir als Induktionsvoraussetzung an, die Aussage sei für eine Gruppe von n Pferden richtig und zeigen im Induktionsschritt, dass die Aussage dann auch für Gruppen von $n + 1$ Pferden gilt. Zunächst wissen wir, dass sich in der Gruppe von $n + 1$ Pferden (mindestens) ein Schimmel befindet. Wir stellen nun alle Pferde derart in einer Reihe auf, dass ein Schimmel ganz vorne steht:



Jetzt betrachten wir die Gruppe der ersten n Pferde. Da unsere Aussage für n per Induktionsannahme richtig ist und ein Schimmel in dieser Gruppe ist, so müssen alle anderen Pferde dieser Gruppe ebenfalls Schimmel sein und wir erhalten das folgende Zwischenergebnis:



Jetzt können wir die Induktionsvoraussetzung auch auf die letzten n Pferde anwenden, da sich unter diesen auf jeden Fall auch mindestens ein Schimmel befindet:



Voilà: Alle Pferde sind Schimmel!

Selbstverständlich ist die bewiesene Aussage nicht richtig – Induktion hin oder her. Analysieren Sie die vorgebrachten Argumente und finden Sie den Fehler in der Beweiskette.

Aufgabe 2.20
Webcode
2657

Wir wollen in dieser Aufgabe das Hofstadter'sche MIU-System aus dem Übungsteil des ersten Kapitels wieder aufgreifen. Zur Erinnerung: Das MIU-System war durch ein einzelnes Axiom und vier Schlussregeln bestimmt:

■ Axiome

1. **MI** ist ein Satz

■ Schlussregeln

1. Mit $x\mathbf{I}$ ist auch $x\mathbf{IU}$ ein Satz
2. Mit $\mathbf{M}x$ ist auch $\mathbf{M}xx$ ein Satz
3. In jedem Satz darf **III** durch **U** ersetzt werden
4. In jedem Satz darf **UU** entfernt werden

Die rekursive Definition des MIU-Systems ermöglicht uns, Aussagen über alle ableitbaren Sätze mit dem Prinzip der strukturellen Induktion zu beweisen.

- a) Zeigen Sie durch strukturelle Induktion, dass alle ableitbaren Wörter die Eigenschaft erfüllen, dass die Anzahl der **I**'s nicht durch drei teilbar ist.
- b) Falls Sie die Übungsaufgabe des ersten Einführungskapitels nicht vollständig lösen konnten, versuchen Sie es erneut. Verwenden Sie die in Teilaufgabe a) gewonnene Erkenntnis.

Aufgabe 2.21
Webcode
2249

In der praktischen Informatik spielen fast ausschließlich diejenigen Rekursionsvorschriften eine Rolle, die nach endlich vielen Schritten terminieren. Ob eine rekursiv definierte Funktion diese Eigenschaft besitzt, ist nicht immer so einfach zu erkennen wie in den bisher diskutierten Beispielen. Um das Gesagte zu verdeutlichen, betrachten wir die *Collatz-Funktion* $c : \mathbb{N}^+ \rightarrow \mathbb{N}$, die folgendermaßen definiert ist:

$$c(n) = \begin{cases} 0 & \text{falls } n = 1 \\ 1 + c\left(\frac{n}{2}\right) & \text{falls } n > 0 \text{ und } n \text{ gerade ist} \\ 1 + c(3 \cdot n + 1) & \text{falls } n > 0 \text{ und } n \text{ ungerade ist} \end{cases}$$

- a) Berechnen Sie die Funktionswerte $c(3)$, $c(7)$ und $c(27)$.
- b) Terminiert die Rekursion Ihrer Meinung nach für alle $n \in \mathbb{N}^+$?

Der *Binomialkoeffizient* ist wie folgt definiert:

$$\binom{n}{k} := \frac{n!}{k! \cdot (n-k)!}$$

Aufgabe 2.22Webcode
2886

a) Beweisen Sie das folgende Additionstheorem:

$$\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}$$

b) Der *binomische Lehrsatz* lautet wie folgt:

$$(a+b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k, \quad a, b \in \mathbb{R}, n \in \mathbb{N}$$

Beweisen Sie den Lehrsatz durch vollständige Induktion.

Im Jahr 1843 veröffentlichte der französische Mathematiker Jacques Philippe Marie Binet die folgende Formel:

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

Zeigen Sie, dass die Binet-Funktion f_n für alle $n \in \mathbb{N}$ die nachstehende Rekursionsgleichung erfüllt:

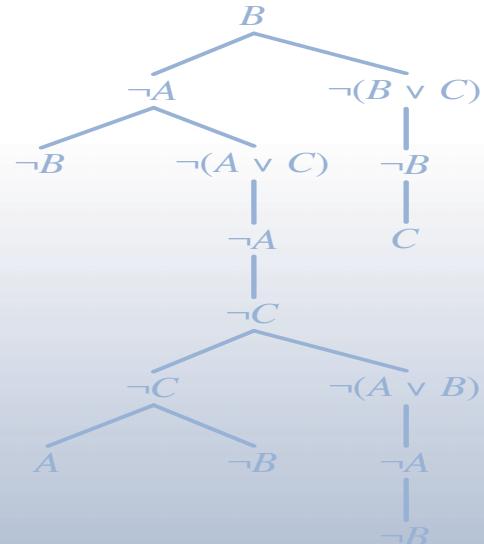
$$f_{n+2} = f_{n+1} + f_n$$

Aufgabe 2.23Webcode
2963

3 Logik und Deduktion

In diesem Kapitel werden Sie ...

- mit der Aussagenlogik die Grundlagen des logischen Schließens erlernen,
- formale Beweise im Hilbert-Kalkül führen,
- mit dem Resolutions- und dem Tableaukalkül zwei Widerspruchskalküle kennen lernen,
- die Aussagenlogik zur Prädikatenlogik erster Stufe erweitern,
- die Grundprinzipien der logischen Programmierung verstehen,
- einen Einblick in Logiken höherer Stufe erlangen.



In der formalen Logik haben wir es mit zwei Sprachebenen zu tun. Eine davon ist die *Objektebene*. Von hier aus betrachtet ist eine Logikformel nichts weiter als eine Zeichenkette, die nach formalen Bildungsregeln aufgebaut ist. Wie diese Regeln für die Aussagenlogik aussehen, haben wir in Definition 3.1 exakt festgelegt. Unter anderem ist die folgende Zeichenkette eine Formel der Aussagenlogik:

$$((A_1 \wedge A_2) \vee (\neg A_1 \wedge \neg A_2))$$

Im Folgenden werden wir die Logikebene mehrfach verlassen und Formeln von einem metatheoretischen Standpunkt aus betrachten. Beispielsweise könnte uns interessieren, unter welchen Bedingungen eine Formel erfüllbar ist, die sich als Disjunktion anderer Formeln darstellen lässt. Damit ist eine Formel der folgenden Bauart gemeint:

$$F \vee G$$

Anders als im ersten Beispiel ist diese Zeichenkette keine Formel der Aussagenlogik; sie ist eine Formel der *Metaebene*. Zu lesen ist sie als Schablone, die für eine Schar verschiedener Formeln steht. Erst wenn wir die Platzhalter F und G durch konkrete Teilausdrücke ersetzen, erhalten wir eine echte Formel.

Die Vermischung von Objekt- und Metaebene ist eine häufige Ursache von Verständnisschwierigkeiten im Bereich der Logik. Um Verwechslungen vorzubeugen, werden wir für die verschiedenen Formelbestandteile unterschiedlichen Schrifttypen verwenden. Alle Bestandteile der Objektebene werden in einer serifenlosen, steilen Schrift dargestellt (A, B, \dots). Platzhalter notieren wir, wie es in der Mathematik üblich ist, weiterhin in einer kursiven Schrift (F, G, \dots). Jetzt ist klar, warum die Symbole A_1 und A_2 in der ersten Formel steil, die Symbole F und G in der zweiten Formel dagegen kursiv dargestellt sind.

3.1 Aussagenlogik

Die Aussagenlogik (PL0) ist die einfachste Spielart des logischen Schließens. Gegenstand der Aussagenlogik sind *atomare Aussagen* („Es regnet“, „Die Straße ist nass“) und die Beziehungen, die zwischen solchen Aussagen bestehen („Wenn es regnet, dann ist die Straße nass“). Sie besticht durch eine einfachen Aufbau und eine klare Struktur, kommt jedoch nicht an die Ausdrucksstärke der anderen hier vorgestellten Logiken heran. Trotzdem ist die Bedeutung der Aussagenlogik beträchtlich. Zum einen ist sie die theoretische Grundlage des Hardware-Entwurfs, da sich das Verhalten einer kombinatorischen Hardware-Schaltung auf der Logikebene eins zu eins auf eine aussagenlogische Formel abbilden lässt. Zum anderen ist sie als Teilmenge in allen anderen Logiken enthalten und damit der kleinste gemeinsame Nenner, über den alle Logiken miteinander verbunden sind.

3.1.1 Syntax und Semantik

Wir nähern uns der Aussagenlogik in zwei Schritten. Zunächst fixieren wir die *Syntax*, d. h., wir definieren, nach welchen Regeln aussagenlogische Ausdrücke (Formeln) aufgebaut sind. Eine Formel ist in diesem Stadium nichts weiter als eine Folge von bedeutungsleeren Symbolen, die in einer festgelegten Art und Weise miteinander kombiniert werden dürfen. Erst die *Semantik* versieht die Ausdrücke mit einer Bedeutung; sie legt fest, wie wir die unterschiedlichen Zeichen und Symbolen einer Logikformel zu interpretieren haben.



Definition 3.1 (Syntax der Aussagenlogik)

Die Menge der *aussagenlogischen Formeln* über dem Variablenvorrat $V = \{A_1, A_2, \dots\}$ ist rekursiv definiert:

- 0 und 1 sind Formeln.
- Jede Variable $A_i \in V$ ist eine Formel.
- Sind F und G Formeln, dann sind es auch

$$(\neg F), (F \wedge G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G), (F \Leftrightarrow G).$$

Der Operator \neg ist die *Negation*, \wedge die *Konjunktion (UND-Operator)*, \vee die *Disjunktion (ODER-Operator)* und \rightarrow die *Implikation*. Ferner be-

zeichnen wir \leftrightarrow als Äquivalenz- und \leftrightarrow als Antivalenzoperator (XOR-Operator). Auch wenn die Namen bereits deutliche Hinweise geben, mit welcher Semantik wir die einzelnen Operatoren später belegen werden, sollten Sie versuchen, eine Formel zunächst als eine Aneinanderreihung von Symbolen zu betrachten, die noch keine konkrete Bedeutung besitzt.

Eine Formel F , die nicht weiter zerlegt werden kann, nennen wir *atomar*. In der Aussagenlogik ist die Menge der atomaren Formeln mit der Menge $V \cup \{0, 1\}$ identisch. Eine Formel, die als Teil einer anderen Formel vorkommt, bezeichnen wir als *Teilformel*. Wir verwenden die etwas informelle Notation $F \in G$, um auszudrücken, dass F eine Teilformel von G ist; andernfalls schreiben wir $F \notin G$. Variablen werden wir im Folgenden durchweg mit Großbuchstaben bezeichnen, allerdings von Fall zu Fall den Symbolvorrat anpassen (z. B. X, Y, Z anstelle von A_1, A_2, A_3).

Für die eingeführten Operatoren existiert keine einheitliche Notation. Zum einen wurde die Schreibweise in der Vergangenheit mehrfach geändert, zum anderen ist sie auch heute noch regional verschieden. Im oberen Teil von Abbildung 3.1 sind die Symbole zusammengefasst, wie sie zu Beginn des zwanzigsten Jahrhunderts üblich waren und zum Verständnis alter Forschungsbeiträge notwendig sind (vgl. Abbildung 3.2). Der untere Teil enthält die Symbole, wie sie insbesondere im US-amerikanischen Sprachraum verwendet werden. Die Notation \bar{F} für $\neg F$ hat sich auch im deutschen Sprachraum etabliert, da sie für viele Ausdrücke zu einer übersichtlicheren Darstellung führt.

Um die Lesbarkeit zu verbessern, werden wir auf die Angabe mancher Klammerpaare verzichten. Mehrdeutigkeiten werden, wie in Abbildung 3.3 gezeigt, über eine Reihe von Bindungs- und Kettenregeln beseitigt. Bindungsregeln teilen die Operatoren in schwächer bindende und stärker bindende Operatoren ein, Kettenregeln bestimmen den Umgang mit Ausdrücken, in denen der gleiche Operator mehrmals hintereinander vorkommt. Wir legen die übliche Konvention zugrunde, dass die Negation (\neg) stärker bindet als die Konjunktion (\wedge). Diese bindet wiederum stärker als die Disjunktion (\vee). Die Operatoren \rightarrow , \leftrightarrow und \leftrightarrow binden am schwächsten. Kommen sie in einem Ausdruck gemischt vor, erfolgt die Klammerung linksassoziativ. Werden Teilterme mit demselben Operator verknüpft, betrachten wir die entstehende Kette ebenfalls als linksassoziativ geklammert. Die einzige Ausnahme bildet der (einstellige) Negationsoperator, den wir rechtsassoziativ gruppieren.

Die zweistelligen Operatoren \wedge und \vee lassen sich auf natürliche Weise zu mehrstelligen Operatoren verallgemeinern. Hierzu vereinbaren wir

■ Frühere Schreibweise

Formel	Schreibweise
$\neg F$	$\sim F$
$F \wedge G$	$F \cdot G$
$F \vee G$	$F \vee G$
$F \rightarrow G$	$F \supset G$

■ Punkt-Strich-Notation

Formel	Schreibweise
$\neg F$	$\neg F, \bar{F}$
$F \wedge G$	$F \cdot G, FG$
$F \vee G$	$F + G$
$F \rightarrow G$	$F \rightarrow G$

Abbildung 3.1: Alternative Schreibweisen aussagenlogischer Formeln

■ Drei Formeln der Principia ...

- *2·03. $\vdash : p \supset \neg q \cdot \neg q \supset \neg p$
- *2·15. $\vdash : \neg p \supset q \cdot \neg q \supset \neg p$
- *2·16. $\vdash : p \supset q \cdot \neg q \supset \neg p$
- *2·17. $\vdash : \neg q \supset p \cdot p \supset q$

■ ... und deren moderne Schreibweise

- (2.03) $\vdash (p \rightarrow \neg q) \rightarrow (\neg q \rightarrow \neg p)$
- (2.15) $\vdash (\neg p \rightarrow q) \rightarrow (\neg q \rightarrow p)$
- (2.16) $\vdash (p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$
- (2.17) $\vdash (\neg q \rightarrow \neg p) \rightarrow (p \rightarrow q)$

Abbildung 3.2: In der Notation der Principia besitzt der Punkt eine Doppelbedeutung. In Abhängigkeit von seiner Position wird er für die konjunktive Verknüpfung oder zum Klammern von Teilausdrücken verwendet. Die Abbildung zeigt drei Formeln aus der Originalausgabe der Principia Mathematica sowie deren Übersetzung in die heute übliche Schreibweise.

■ Bindungsregeln

Ausdruck	Bedeutung
$\neg F \wedge G$	$((\neg F) \wedge G)$
$F \vee G \wedge H$	$(F \vee (G \wedge H))$
$F \vee G \rightarrow H$	$((F \vee G) \rightarrow H)$
$F \rightarrow G \leftrightarrow H$	$((F \rightarrow G) \leftrightarrow H)$
$F \leftrightarrow G \rightarrow H$	$((F \leftrightarrow G) \rightarrow H)$

■ Kettenregeln

Ausdruck	Bedeutung
$\neg\neg\neg F$	$\neg(\neg(\neg F))$
$F \wedge G \wedge H$	$(F \wedge G) \wedge H$
$F \vee G \vee H$	$(F \vee G) \vee H$
$F \rightarrow G \rightarrow H$	$(F \rightarrow G) \rightarrow H$
$F \leftrightarrow G \leftrightarrow H$	$(F \leftrightarrow G) \leftrightarrow H$
$F \leftrightarrow G \leftrightarrow H$	$(F \leftrightarrow G) \leftrightarrow H$

Abbildung 3.3: Zur Vereinfachung der Schreibweise dürfen Klammerpaare weggelassen werden. Zweideutigkeiten werden mithilfe von Bindungs- und Kettenregeln beseitigt. Erstere teilen die Operatoren in schwächer bindende und stärker bindende Operatoren ein, Letztere regeln den Umgang mit Ausdrücken, in denen der gleiche Operator mehrmals hintereinander vorkommt.

für die endlich vielen Formeln F_1, \dots, F_n die folgende Schreibweise:

$$\left(\bigwedge_{i=1}^n F_i \right) := F_1 \wedge \dots \wedge F_n \quad \left(\bigvee_{i=1}^n F_i \right) := F_1 \vee \dots \vee F_n$$

Die Semantik einer aussagenlogischen Formel wird über die *Modellrelation* \models festgelegt. Um diese formal definieren zu können, müssen wir vorab klären, was wir unter dem Begriff der *Interpretation* zu verstehen haben:



Definition 3.2 (Interpretation)

Sei F eine aussagenlogische Formel. A_1, \dots, A_n bezeichnen die in F vorkommenden Variablen. Jede Abbildung

$$I : \{A_1, \dots, A_n\} \rightarrow \{0, 1\}$$

heißt eine *Interpretation* von F .

Eine Interpretation ordnet jeder Variablen einer aussagenlogischen Formel F einen der beiden Wahrheitswerte 0 (*falsch*) oder 1 (*wahr*) zu und wird aufgrund dieser Eigenschaft auch als *Belegung* bezeichnet.

Der Begriff der Interpretation ist die Grundlage für die formale Definition der Semantik:



Definition 3.3 (Semantik der Aussagenlogik)

F und G seien aussagenlogische Formeln und I eine Interpretation. Die Semantik der Aussagenlogik ist durch die *Modellrelation* \models gegeben, die induktiv definiert ist:

$$\begin{aligned} I \models 1 \\ I \not\models 0 \\ I \models A_i &\Leftrightarrow I(A_i) = 1 \\ I \models (\neg F) &\Leftrightarrow I \not\models F \\ I \models (F \wedge G) &\Leftrightarrow I \models F \text{ und } I \models G \\ I \models (F \vee G) &\Leftrightarrow I \models F \text{ oder } I \models G \\ I \models (F \rightarrow G) &\Leftrightarrow I \not\models F \text{ oder } I \models G \\ I \models (F \leftrightarrow G) &\Leftrightarrow I \models F \text{ genau dann, wenn } I \models G \\ I \models (F \leftrightarrow G) &\Leftrightarrow I \not\models (F \leftrightarrow G) \end{aligned}$$

Eine Interpretation I mit $I \models F$ heißt *Modell* für F .

Wir können jede aussagenlogische Formel F mit n Variablen als *boolesche Funktion* $f^F : \{0, 1\}^n \rightarrow \{0, 1\}$ auffassen, die für eine Belegung I genau dann den Funktionswert 1 annimmt, wenn I ein Modell für F ist. Mit anderen Worten: Weist I den Variablen A_1, \dots, A_n die Wahrheitswerte b_1, \dots, b_n zu, dann ist der Funktionswert $f^M(b_1, \dots, b_n)$ durch die folgende Formel bestimmt:

$$f^F(b_1, \dots, b_n) = \begin{cases} 1 & \text{falls } I \models F \\ 0 & \text{falls } I \not\models F \end{cases}$$

Aufgrund des diskreten Definitionsbereichs lässt sich eine n -stellige boolesche Funktion in Form einer *Wahrheitstabelle* darstellen, indem alle möglichen Kombinationen der Eingangsvariablen A_1, \dots, A_n zusammen mit dem zugeordneten Funktionswert zeilenweise aufgelistet werden. Als Beispiele zeigt Abbildung 3.4 die Wahrheitstabellen der eingeführten aussagenlogischen Operatoren. Die Funktionswerte erschließen sich unmittelbar aus Definition 3.3. Wahrheitstabellen werden in der Literatur auch als *Wahrheitstafeln* oder *Funktions(wert)tabellen* bezeichnet; alle diese Begriffe bezeichnen die gleiche tabellarische Beschreibungsweise einer booleschen Funktion.

Abbildung 3.5 zeigt, wie sich Wahrheitstabellen für zusammengesetzte Ausdrücke erzeugen lassen. Ausgehend von den Basistermen werden zunächst die Teilformeln und anschließend der Gesamtausdruck ausgewertet. Die drei Beispiele wurden bewusst gewählt. Die Formel F_1 ist so beschaffen, dass sie genau zwei Modelle besitzt; sie ist genau dann wahr, wenn die Variablen A, B, C mit dem gleichen Wahrheitswert belegt werden. In der Terminologie der Aussagenlogik wird die Funktion als *erfüllbar* bezeichnet. F_2 ist ebenfalls erfüllbar, besitzt aber im Gegensatz zu F_1 die Eigenschaft, dass ausnahmslos alle Variablenbelegungen ein Modell sind. Solche Formeln heißen *allgemeingültig*. In entsprechender Weise bezeichnen wir F_3 als *unerfüllbare* Formel, da sie kein einziges Modell besitzt. Formal halten wir das Gesagte in der folgenden Definition fest:



Definition 3.4 (Erfüllbarkeit, Allgemeingültigkeit)

Eine aussagenlogische Formel F heißt

- *erfüllbar*, falls F mindestens ein Modell besitzt,
- *unerfüllbar*, falls F kein Modell besitzt,
- *allgemeingültig*, falls $\neg F$ unerfüllbar ist.

Eine allgemeingültige Formel bezeichnen wir auch als *Tautologie*.

Negation

	A	$\neg A$
0	0	1
1	1	0

Konjunktion

	A	B	$A \wedge B$
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

Disjunktion

	A	B	$A \vee B$
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	1

Implikation

	A	B	$A \rightarrow B$
0	0	0	1
1	0	1	1
2	1	0	0
3	1	1	1

Äquivalenz

	A	B	$A \leftrightarrow B$
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	1

Antivalenz

	A	B	$A \leftrightarrow B$
0	0	0	0
1	0	1	1
2	1	0	1
3	1	1	0

Abbildung 3.4: Wahrheitstabellen der aussagenlogischen Basisoperatoren

■ Beispiel 1: $F_1 = \underbrace{(\bar{A} \vee B)}_{G_1} \wedge \underbrace{(\bar{B} \vee C)}_{G_2} \wedge \underbrace{(\bar{C} \vee A)}_{G_3}$

$$\underbrace{\quad\quad\quad}_{G_4}$$

A	B	C	G_1	G_2	G_3	G_4	F_1
0	0	0	1	1	1	1	1
0	0	1	1	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	1	1	0	1	0
1	0	0	0	1	1	0	0
1	0	1	0	1	1	0	0
1	1	0	1	0	1	0	0
1	1	1	1	1	1	1	1

■ Beispiel 2: $F_2 = \underbrace{((A \rightarrow B) \wedge (B \rightarrow C))}_{G_1} \rightarrow \underbrace{(A \rightarrow C)}_{G_3}$

$$\underbrace{\quad\quad\quad}_{G_4}$$

A	B	C	G_1	G_2	G_3	G_4	F_2
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	1	0	1	0	1	0	1
0	1	1	1	1	1	1	1
1	0	0	0	1	0	0	1
1	0	1	0	1	1	0	1
1	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1

■ Beispiel 3: $F_3 = \underbrace{(A \leftrightarrow B) \wedge (A \leftrightarrow C)}_{G_1} \wedge \underbrace{(B \leftrightarrow C)}_{G_3}$

$$\underbrace{\quad\quad\quad}_{G_4}$$

A	B	C	G_1	G_2	G_3	G_4	F_3
0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	0
0	1	0	1	0	1	0	0
0	1	1	1	1	0	1	0
1	0	0	1	1	0	1	0
1	0	1	1	0	1	0	0
1	1	0	0	1	1	0	0
1	1	1	0	0	0	0	0

Abbildung 3.5: Wahrheitstabellen zusammengesetzter Funktionen

Abbildung 3.6 demonstriert den Zusammenhang zwischen den eingeführten Begriffen. Alle drei lassen sich in naheliegender Weise auf Mengen von aussagenlogischen Formeln erweitern. Die Formelmenge $M = \{F_1, \dots, F_n\}$ heißt erfüllbar, wenn eine Interpretation I existiert, die für alle $F_i \in M$ ein Modell ist. Beachten Sie, dass das Modell für alle Formeln das gleiche sein muss; es reicht also nicht aus, dass jede Formel

für sich erfüllbar ist. Die Unerfüllbarkeit und Allgemeingültigkeit von Formelmengen definieren wir analog. M ist unerfüllbar, wenn F_1, \dots, F_n kein gemeinsames Modell besitzen. Ist dagegen jede Interpretation ein Modell für die Elemente von M , so nennen wir M allgemeingültig.

Mithilfe der Modellrelation können wir den Begriff der *logischen Folgerung* formal definieren:



Definition 3.5 (Logische Folgerung)

$M := \{F_1, \dots, F_n\}$ sei eine Menge aussagenlogischer Formeln. Wir schreiben

$$M \models G \quad (\text{„aus } M \text{ folgt } G“}),$$

wenn jedes Modell von M auch ein Modell der aussagenlogischen Formel G ist. Ferner vereinbaren wir die Kurzschreibweise $\models G$ für $\emptyset \models G$ und $F \models G$ für $\{F\} \models G$.

Es gelten die folgenden Zusammenhänge:

- $\models G$ gilt genau dann, wenn G allgemeingültig ist.
- $F \models G$ gilt genau dann, wenn $F \rightarrow G$ allgemeingültig ist.
- $\{F_1, F_2, \dots, F_n\} \models G$ ist äquivalent zu $\{F_2, \dots, F_n\} \models F_1 \rightarrow G$.

In den kommenden Betrachtungen wird der Begriff der *Äquivalenz* immer wieder auftauchen:



Definition 3.6 (Äquivalenz)

Seien F und G zwei aussagenlogische Formeln. Die Relation \equiv ist wie folgt definiert:

$$F \equiv G \Leftrightarrow F \models G \text{ und } G \models F$$

Zwei Formeln F und G mit $F \equiv G$ heißen *äquivalent*.

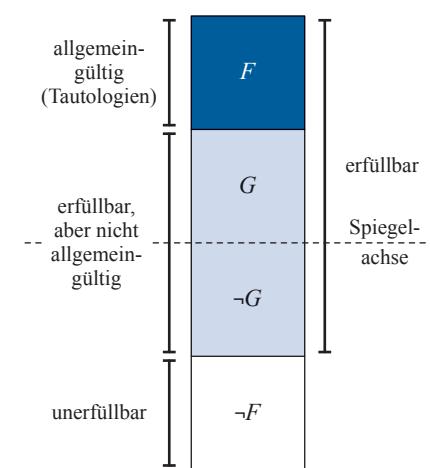


Abbildung 3.6: Das Spiegelungsprinzip visualisiert, wie sich die Eigenschaften der Formeln F und $\neg F$ gegenseitig beeinflussen. Ist F allgemeingültig, so ist $\neg F$ unerfüllbar. Ist F nicht allgemeingültig, aber dennoch erfüllbar, so gilt das Gleiche für $\neg F$. Damit ist die Allgemeingültigkeit eine exklusive Eigenschaft, die nur eine der beiden Formeln F oder $\neg F$ erfüllen kann. Im Gegensatz hierzu können sowohl F als auch $\neg F$ erfüllbar sein.

Damit sind zwei Formeln F und G genau dann äquivalent, geschrieben als $F \equiv G$, falls sie exakt dieselben Modelle besitzen. Im mathematischen Sinn ist \equiv eine Äquivalenzrelation auf der Menge der aussagenlogischen Formeln und besitzt die Eigenschaften der Reflexivität, Symmetrie und Transitivität:

Kommutativität	Neutralität
$F \wedge G \equiv G \wedge F$ $F \vee G \equiv G \vee F$	$F \wedge 1 \equiv F$ $F \vee 0 \equiv F$
Distributivität	Inversion
$F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$ $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$	$F \wedge \neg F \equiv 0$ $F \vee \neg F \equiv 1$
Assoziativität	Elimination
$F \wedge (G \wedge H) \equiv (F \wedge G) \wedge H$ $F \vee (G \vee H) \equiv (F \vee G) \vee H$	$F \wedge 0 \equiv 0$ $F \vee 1 \equiv 1$
De Morgan'sche Regeln	Idempotenz
$\neg(F \wedge G) \equiv \neg F \vee \neg G$ $\neg(F \vee G) \equiv \neg F \wedge \neg G$	$F \wedge F \equiv F$ $F \vee F \equiv F$
Absorption	Doppelnegation
$F \wedge (F \vee G) \equiv F$ $F \vee (F \wedge G) \equiv F$	$\neg\neg F \equiv F$

Tabelle 3.1: Wichtige Äquivalenzen aussagenlogischer Ausdrücke

- Reflexivität: Für alle Formeln F gilt $F \equiv F$.
- Symmetrie: Aus $F \equiv G$ folgt $G \equiv F$.
- Transitivität: Aus $F \equiv G$ und $G \equiv H$ folgt $F \equiv H$.

Es gelten die folgenden Zusammenhänge:

- F ist genau dann allgemeingültig, wenn $F \equiv 1$.
- F ist genau dann unerfüllbar, wenn $F \equiv 0$.

In Tabelle 3.1 sind wichtige Äquivalenzen zusammengefasst, die sich durch das Aufstellen von Wahrheitstafeln leicht verifizieren lassen. Die Bedeutung dieser Äquivalenzen ist zweigeteilt. Zum einen gestatten sie einen Einblick in die elementaren Zusammenhänge zwischen den eingeführten booleschen Operatoren, zum anderen dienen sie als wichtige Umformungsregeln für aussagenlogische Ausdrücke. Grundlage hierfür

ist das *Substitutionstheorem*, das uns gestattet, Teilformeln durch äquivalente Ausdrücke zu ersetzen, ohne die Modelle der Gesamtformel zu beeinflussen.

Satz 3.1 (Substitutionstheorem)

Seien F, G, G' aussagenlogische Formeln mit $G \in F$ und $G \equiv G'$.

$$F[G \leftarrow G']$$

bezeichnet diejenige Formel, die aus F entsteht, indem die Teilformel G durch G' ersetzt wird. Dann gilt:

$$F \equiv F[G \leftarrow G']$$

Mit dem Mittel der strukturellen Induktion aus Abschnitt 2.4.2 lässt sich das Substitutionstheorem induktiv über den Aufbau aussagenlogischer Formeln beweisen.

Vielleicht haben Sie sich gewundert, dass Tabelle 3.1 ausschließlich Rechenregeln für die aussagenlogischen *Elementaroperatoren* \neg , \wedge und \vee enthält. Hierbei handelt es sich um keine Einschränkung im eigentlichen Sinne, da sich alle anderen Operatoren auf diese drei zurückführen lassen (vgl. Tabelle 3.2). Die Menge $\{\neg, \wedge, \vee\}$ ist zudem ein *vollständiges Operatorensystem*, d. h., sie besitzt die Eigenschaft, dass sich jede boolesche Funktion durch einen aussagenlogischen Ausdruck beschreiben lässt, in dem ausschließlich Operatoren aus dieser Menge vorkommen. Ein Beweis der Vollständigkeit wird uns im nächsten Abschnitt ohne Zutun in die Hände fallen. Dort werden wir zeigen, wie sich eine entsprechende aussagenlogische Formel systematisch aus der Wahrheitstabelle einer booleschen Funktion erzeugen lässt.

Neben der Menge der Elementaroperatoren existieren weitere vollständige Operatorensysteme, wie z. B. die Menge $\{\neg, \rightarrow\}$. Um die Vollständigkeit zu beweisen, nutzen wir unser Wissen, dass die drei Elementaroperatoren \wedge , \vee und \neg zusammen ein vollständiges Operatorensystem bilden. Können wir zeigen, dass \wedge und \vee durch \neg und \rightarrow darstellbar sind, so lässt sich jede boolesche Funktion mit einem aussagenlogischen Ausdruck beschreiben, der ausschließlich die Operatoren \neg und \rightarrow enthält. Kurzum: $\{\neg, \rightarrow\}$ bildet dann ebenfalls ein vollständiges Operatorensystem. Tabelle 3.3 zeigt, wie die notwendigen Reduktionen durchgeführt werden können.

Mithilfe der Aussagenlogik lassen sich viele der kombinatorischen Zusammenhänge beschreiben, die wir im Bereich des mathematischen

Implikation
$F \rightarrow G \equiv \neg F \vee G$

Äquivalenz
$F \leftrightarrow G \equiv (\neg F \wedge \neg G) \vee (F \wedge G)$ $\equiv (\neg F \vee G) \wedge (F \vee \neg G)$

Antivalenz
$F \leftrightarrow G \equiv (\neg F \wedge G) \vee (F \wedge \neg G)$ $\equiv (\neg F \vee \neg G) \wedge (F \vee G)$

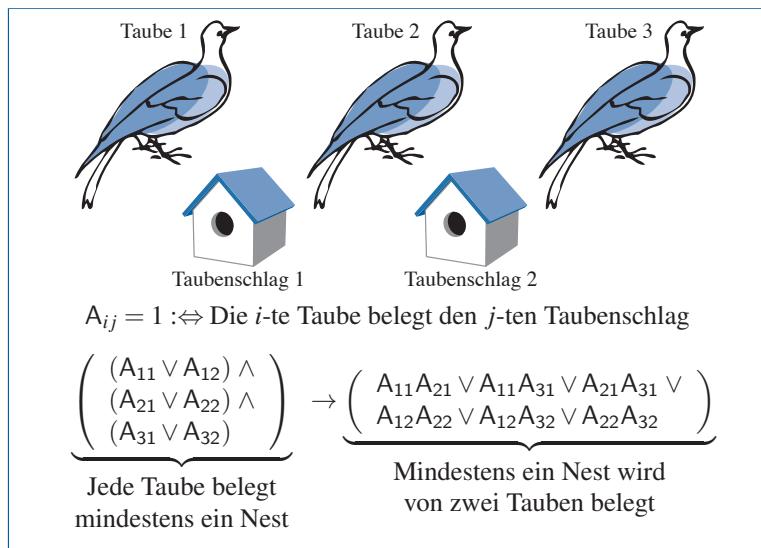
Tabelle 3.2: Reduktion der Operatoren \rightarrow , \leftrightarrow und \leftrightarrow auf die Elementaroperatoren \neg , \wedge und \vee

Reduktion von \wedge auf $\{\rightarrow, \neg\}$
$F \wedge G \equiv \neg(\neg F \vee \neg G)$ $\equiv \neg(F \rightarrow \neg G)$

Reduktion von \vee auf $\{\rightarrow, \neg\}$
$F \vee G \equiv \neg\neg F \vee G$ $\equiv \neg F \rightarrow G$

Tabelle 3.3: Reduktion der Elementaroperatoren \wedge und \vee auf \neg und \rightarrow

Abbildung 3.7: Das *Dirichlet'sche Schubfachprinzip* ist im angelsächsischen Raum unter dem Namen *pigeonhole principle (Taubenschlagprinzip)* bekannt. Verteilen sich, wie in diesem Beispiel, 3 Tauben auf 2 Taubenschläge, so muss mindestens ein Taubenschlag doppelt belegt werden. Mithilfe der Aussagenlogik lässt sich das Prinzip formalisieren und beweisen.



Schließens tagtäglich anwenden. Als Beispiel zeigt Abbildung 3.7 eine Formalisierung des *Dirichlet'schen Schubfachprinzips*. Dieses besagt, dass eine endliche Menge M nicht injektiv auf eine Menge N abgebildet werden kann, wenn N weniger Elemente enthält als M . Das Prinzip ist nach dem deutschen Mathematiker Johann Dirichlet (Abbildung 3.8) benannt und ein häufig angewandtes Beweisargument in der diskreten Mathematik. Jedem von uns ist das Schubfachprinzip aus dem Alltag geläufig. Verteilen wir m Gegenstände auf n Schubfächer und gilt $m > n$, so muss mindestens ein Schubfach mehrere Gegenstände enthalten. Im angelsächsischen Raum wird das Dirichlet'sche Schubfachprinzip als *pigeonhole principle (Taubenschlagprinzip)* bezeichnet. Auch hier ist die angestellte Überlegung die gleiche: Verteilen sich m Tauben auf n Nester und gilt $m > n$, so ist mindestens ein Taubenschlag mehrfach besetzt.

Mithilfe der Aussagenlogik können wir das Schubfachprinzip formalisieren und beweisen. Hierzu führen wir für jede mögliche Kombination von Gegenständen und Schubfächern eine aussagenlogische Variable A_{ij} ein, die genau dann den Wert 1 annimmt, wenn sich der i -te Gegenstand im j -ten Schubfach befindet. Um das Schubfachprinzip für n Gegenstände und $n - 1$ Schubfächer zu formalisieren, benötigen wir $n \cdot (n - 1)$ Variablen. Wir werden nun Schritt für Schritt herausarbeiten, wie sich das Schubfachprinzip mithilfe der booleschen Operatoren \neg , \wedge , \vee und \rightarrow formal nachbilden lässt:

- „Das i -te Element befindet sich in einem der Schubfächer“

$$\bigvee_{j=1}^{n-1} A_{ij}$$

- „Jedes Element befindet sich in einem der Schubfächer“

$$\bigwedge_{i=1}^n \bigvee_{j=1}^{n-1} A_{ij} \quad (3.1)$$

- „In Schubfach j befinden sich das i -te und k -te Element gleichzeitig“

$$A_{ij} \wedge A_{kj}$$

- „In Schubfach j befinden sich mindestens zwei Elemente“

$$\bigvee_{i=1}^{n-1} \bigvee_{k=i+1}^n (A_{ij} \wedge A_{kj})$$

- „In einem Schubfach befinden sich mindestens zwei Elemente“

$$\bigvee_{j=1}^{n-1} \bigvee_{i=1}^{n-1} \bigvee_{k=i+1}^n (A_{ij} \wedge A_{kj}) \quad (3.2)$$

Verbinden wir die Formeln (3.1) und (3.2) mit dem Implikationsoperator, so erhalten wir die Aussage des Schubfachprinzips: Befinden sich n Elemente in $n - 1$ Schubfächern, so enthält eines der Schubfächer mindestens zwei Elemente:

$$\bigwedge_{i=1}^n \bigvee_{j=1}^{n-1} A_{ij} \rightarrow \bigvee_{j=1}^{n-1} \bigvee_{i=1}^{n-1} \bigvee_{k=i+1}^n (A_{ij} \wedge A_{kj}) \quad (3.3)$$

Die in Abbildung 3.7 dargestellte Formel erhalten wir aus Formel (3.3) für den Spezialfall $n = 3$.

3.1.2 Normalformen

Weiter oben haben wir herausgearbeitet, dass wir eine aussagenlogische Formel F als boolesche Funktion interpretieren können, indem wir jede Interpretation I als Eingabebelegung auffassen und der Funktion genau dann den Wert 1 zuweisen, wenn I ein Modell für F ist. Die enge Beziehung, die zwischen aussagenlogischen Formeln auf der einen Seite



Johann Peter Gustav Lejeune Dirichlet
(1805 – 1859)

Abbildung 3.8: Das Schubfachprinzip erhielt seinen Namen durch Johann Peter Gustav Lejeune Dirichlet. Der deutsche Mathematiker wurde im nordrhein-westfälischen Düren geboren, damals Teil des Napoleonischen Kaiserreiches. 1822 nahm er das Studium der Mathematik in Paris auf. Bereits drei Jahre später demonstrierte er der Wissenschaftsgemeinde das erste Mal sein Talent, als er die Gültigkeit der Fermat'schen Vermutung für den Fall $n = 5$ bewies. Im Jahr 1826 kehrte er nach Deutschland zurück und graduierte 1827 an der Universität Bonn. Seine akademische Karriere führte ihn über Breslau und Berlin nach Göttingen, wo er im Jahr 1855 den Lehrstuhl von Carl-Friedrich Gauß übernahm. Dort sollte ihm das Schicksal nur eine kurze Schaffenszeit gewähren. Am 5. Mai 1859, nur fünf Monate nach seiner Frau Rebecca Mendelssohn Bartholdy, verstarb er im Alter von 54 Jahren. Dirichlet zählt zu den großen Mathematikern des neunzehnten Jahrhunderts und machte sich vor allem in den Gebieten der partiellen Differentialgleichungen und der algebraischen Zahlentheorie verdient. Zu seinen bedeutendsten Hinterlassenschaften gehört der Dirichlet'sche Einheitsatz, die Dirichlet-Funktion und die Dirichlet'sche Eta-Funktion.

■ Beispiel 1

$$F := (A \wedge B) \vee A$$

$$G := A$$

	A	B	F	G
0	0	0	0	0
1	0	1	0	0
2	1	0	1	1
3	1	1	1	1

■ Beispiel 2

$$F := (A \leftrightarrow B) \vee (A \leftrightarrow B)$$

$$G := 1$$

	A	B	F	G
0	0	0	1	1
1	0	1	1	1
2	1	0	1	1
3	1	1	1	1

Abbildung 3.9: Zwei aussagenlogische Formeln können selbst dann äquivalent sein, wenn sie unterschiedliche Variablen enthalten. Wie die Wahrheitstafeln belegen, besitzen F und G jeweils die gleichen Modelle.

und booleschen Funktionen auf der anderen besteht, ist nicht eindeutig. So repräsentieren die Formeln

$$F_1 := A$$

$$F_2 := A \wedge A$$

$$F_3 := A \wedge A \wedge A$$

...

$$F_n := A \wedge A \wedge \dots \wedge A$$

allesamt die gleiche boolesche Funktion. Die äußere Form lässt also keinerlei Rückschluss zu, ob zwei boolesche Formeln äquivalent zueinander sind oder nicht. Wie die Beispiele in Abbildung 3.9 zeigen, können zwei Formeln können sogar dann äquivalent sein, wenn sie unterschiedliche Variablen enthalten.

Um diesem Problem zu begegnen, wurden in der Vergangenheit mehrere *Normalformen* entwickelt, die eine Eins-zu-eins-Beziehung zwischen aussagenlogischen Formeln und booleschen Funktionen herstellen. Wichtige Normalformen sind die *kanonische konjunktive* und die *kanonische disjunktive Normalform*. Um diese formal zu definieren, benötigen wir die folgenden Hilfsbegriffe:



Definition 3.7 (Literal, Minterm, Maxterm)

Sei F eine aussagenlogische Formel mit den Variablen A_1, \dots, A_n .

- Jedes Vorkommen einer Variablen A_i oder ihrer Negation $\neg A_i$ bezeichnen wir als *Literal*, geschrieben als $(\neg)A_i$ oder L_i .
- Jeder Ausdruck der Form $(\neg)A_1 \wedge \dots \wedge (\neg)A_n$ heißt *Minterm*.
- Jeder Ausdruck der Form $(\neg)A_1 \vee \dots \vee (\neg)A_n$ heißt *Maxterm*.

Hat ein Literal L die Form $\neg A$, so sprechen wir von einem *negativen*, andernfalls von einem *positiven Literal*. Minterme und Maxterme besitzen die Eigenschaft, dass sämtliche Variablen einer Funktion konjunktiv bzw. disjunktiv miteinander verknüpft werden. Hieraus ergeben sich die folgenden beiden Eigenschaften:

- Ein Minterm ist für genau eine Variablenbelegung wahr.
- Ein Maxterm ist für genau eine Variablenbelegung falsch.

Mithilfe der eingeführten Begriffe definieren wir die kanonische disjunktive und die kanonische konjunktive Normalform wie folgt:



Definition 3.8 (KKNF und KDNF)

Sei F eine aussagenlogische Formel mit den Variablen

$$A_1, \dots, A_n$$

- F liegt in *kanonischer konjunktiver Normalform* (KKNF) vor, wenn sie die Form

$$F = \bigwedge_{i=1}^m ((\neg)A_1 \vee \dots \vee (\neg)A_n)$$

besitzt und alle Maxterme paarweise verschieden sind.

- F liegt in *kanonischer disjunktiver Normalform* (KDNF) vor, wenn sie die Form

$$F = \bigvee_{i=1}^m ((\neg)A_1 \wedge \dots \wedge (\neg)A_n)$$

besitzt und alle Minterme paarweise verschieden sind.

Die KKNF entspricht einer Kette UND-verknüpfter Maxterme und die KDNF einer Kette ODER-verknüpfter Minterme. Aus der Wahrheitstabelle einer aussagenlogischen Formel F lassen sich die kanonischen Normalformen auf einfache Weise ableiten. Für die KKNF wird für jede Belegung I mit $I \neq F$ ein Maxterm erzeugt. Anschließend werden diese konjunktiv miteinander verknüpft (vgl. Tabelle 3.4 links). Die KDNF entsteht analog, indem für jede Belegung I mit $I \models F$ ein Minterm erzeugt und diese anschließend disjunktiv miteinander verknüpft werden (vgl. Tabelle 3.4 rechts). Für das gezeigte Beispiel erhalten wir die in Abbildung 3.10 dargestellten Ergebnisse.

Das Konstruktionsschema besitzt zwei wesentliche Eigenschaften. Zum einen können wir es auf beliebige Wahrheitstabellen anwenden und somit zu jeder booleschen Funktion f eine äquivalente aussagenlogische Formel F in kanonischer konjunktiver oder kanonischer disjunktiver Normalform konstruieren. Da in F ausschließlich die drei Elementaroperatoren \neg , \wedge und \vee vorkommen, haben wir nebenbei bewiesen, dass die Menge $\{\neg, \wedge, \vee\}$ ein vollständiges Operatorenensemble bildet. Zum anderen ist das Konstruktionsschema deterministisch, d. h., wir hatten zu keiner Zeit eine Möglichkeit, die Konstruktion der Min- oder Maxterme in irgendeiner Weise zu beeinflussen. Da jede boolesche Funktion eine eindeutige Wahrheitstafeldarstellung besitzt, ist auch die erzeugte Formeldarstellung eindeutig.

Im strengen Sinn sind die kanonische konjunktive und die kanonische disjunktive Normalform keine echten kanonischen, d. h. syntaktisch eindeutigen, Darstellungen. Verantwortlich hierfür ist eine fehlende Ordnung zwischen den Literalen eines Minterms bzw. Maxterms und den Mintermen bzw. Maxterms selbst. Ohne diese Ordnung können wir aus einer kanonischen konjunktiven oder kanonischen disjunktiven Normalform eine weitere erzeugen, indem wir die Literale innerhalb eines Minterms umordnen. Ebenso können wir die Position der Minterme oder der Maxterme vertauschen, ohne die Normalform-eigenschaft zu verletzen. Mathematisch ausgedrückt handelt es sich bei der KKNF und der KDNF um eine kanonische Darstellung modulo Kommutativität und Assoziativität.

Eine im mathematischen Sinne echte kanonische Darstellung ließe sich erzeugen, indem wir die Literale und Minterme beispielsweise alphabetisch anordnen. Für die meisten Anwendungen ist der Normalformbegriff in der hier eingeführten Form aber völlig ausreichend.

Tabelle 3.4: Konstruktionsschema der kanonischen Normalformen. Die KKNF wird erzeugt, indem zunächst für jede Belegung I mit $I \neq F$ ein Maxterm erzeugt wird. Eine Variable wird unverändert in den Maxterm aufgenommen, wenn sie von I mit 0 belegt wird, andernfalls wird sie negiert aufgenommen. Anschließend werden alle Maxterme miteinander konjunktiv verknüpft. Die Konstruktion der KDNF verläuft analog, indem zunächst für jede Belegung I mit $I \models F$ ein Minterm erzeugt wird. Jetzt wird eine Variable unverändert in den Minterm aufgenommen, wenn sie von I mit 1 belegt wird, andernfalls wird sie negiert aufgenommen. Anschließend werden alle Minterme miteinander disjunktiv verknüpft.

	A	B	C	D	F
$A \vee B \vee C \vee D$	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	1
	0	0	1	1	1
$A \vee \bar{B} \vee C \vee D$	0	1	0	0	0
	0	1	0	1	1
$A \vee \bar{B} \vee \bar{C} \vee D$	0	1	1	0	0
$A \vee \bar{B} \vee \bar{C} \vee \bar{D}$	0	1	1	1	0
$\bar{A} \vee B \vee C \vee D$	1	0	0	0	0
$\bar{A} \vee B \vee C \vee \bar{D}$	1	0	0	1	0
	1	0	1	0	1
	1	0	1	1	0
$\bar{A} \vee B \vee \bar{C} \vee D$	1	1	0	0	1
	1	1	0	1	1
	1	1	1	0	1
	1	1	1	1	0
$\bar{A} \vee \bar{B} \vee \bar{C} \vee \bar{D}$	1	1	1	1	0

- KKNF: $(A \vee B \vee C \vee D) \wedge (A \vee \neg B \vee C \vee D) \wedge (A \vee \neg B \vee \neg C \vee D) \wedge (A \vee \neg B \vee \neg C \vee \neg D) \wedge (\neg A \vee B \vee C \vee D) \wedge (\neg A \vee B \vee C \vee \neg D) \wedge (\neg A \vee B \vee \neg C \vee \neg D) \wedge (\neg A \vee \neg B \vee C \vee \neg D)$
- KDNF: $(\neg A \wedge \neg B \wedge \neg C \wedge D) \vee (\neg A \wedge \neg B \wedge C \wedge \neg D) \vee (\neg A \wedge \neg B \wedge C \wedge D) \vee (\neg A \wedge B \wedge \neg C \wedge D) \vee (A \wedge \neg B \wedge C \wedge \neg D) \vee (A \wedge \neg B \wedge \neg C \wedge \neg D) \vee (A \wedge B \wedge \neg C \wedge D) \vee (A \wedge B \wedge C \wedge \neg D)$

Abbildung 3.10: Kanonische Normalformen für die Funktion aus Tabelle 3.4

In den folgenden Betrachtungen ist die Eigenschaft der Eindeutigkeit nicht von Bedeutung. Aus diesem Grund werden wir auf eine kompaktere Darstellung zurückgreifen, die die zweistufige Grundstruktur der Normalform nicht zerstört. Die neue Darstellung basiert auf der Beobachtung, dass wir zwei Min- bzw. Maxterme immer dann zu einem gemeinsamen Term verschmelzen können, wenn sich diese im Vorzeichen eines einzigen Literals unterscheiden. Für unsere Beispielfunktion sind unter anderem die folgenden Vereinfachungen möglich:

$$\begin{aligned}
 & (A \wedge B \wedge \neg C \wedge \neg D) \vee (A \wedge B \wedge \neg C \wedge D) \\
 & \equiv (A \wedge B \wedge \neg C) \wedge (\neg D \vee D) \\
 & \equiv (A \wedge B \wedge \neg C)
 \end{aligned}$$

Wenden wir das Vereinfachungsschema durchgängig an, so können wir die 8 Teilausdrücke der KKNF und der KDNF auf jeweils 4 Teilausdrücke reduzieren. Beachten Sie, dass wir die Eigenschaft der Eindeutigkeit durch die Reduktion verlieren. Wie in Abbildung 3.11 gezeigt, existieren zwei verschiedene Möglichkeiten, die ursprüngliche Funktion kompakt darzustellen. Folgerichtig sprechen wir nicht mehr länger von einer *kanonischen Normalform*, sondern nur noch von einer *Normalform* (KNF oder DNF).

Eine konjunktive bzw. disjunktive *Minimalform* liegt vor, wenn die Funktion mit der kleinstmöglichen Anzahl von Literalen dargestellt wird, d. h., wenn keine andere konjunktive bzw. disjunktive Form existiert, die mit weniger Literalen auskommt.



Definition 3.9 (Konjunktive und disjunktive Minimalform)

Sei F eine aussagenlogische Formel.

- F ist in *konjunktiver Normalform* (KNF), wenn sie eine Konjunktion von Disjunktionen von Literalen ist.
- Eine konjunktive Form heißt *minimal*, wenn es keine äquivalente KNF gibt, die mit weniger Literalen auskommt.
- F ist in *disjunktiver Normalform* (DNF), wenn sie eine Disjunktion von Konjunktionen von Literalen ist.
- Eine disjunktive Form heißt *minimal*, wenn es keine äquivalente DNF gibt, die mit weniger Literalen auskommt.

Die Erzeugung einer konjunktiven oder einer disjunktiven Minimalform ist ein gut untersuchtes Teilgebiet der technischen Informatik. Unter dem Begriff der *zweistelligen Logikminimierung* wurde eine Vielzahl von Verfahren entwickelt, mit deren Hilfe sich die Minimalform effizient erzeugen bzw. annähern lässt [49]. Unter anderem kann mit diesen Verfahren gezeigt werden, dass es sich bei den Formeln aus Abbildung 3.11 tatsächlich um Minimalformen handelt.

Für die konjunktive Normalform einer aussagenlogischen Formel F existiert mit der *Klauseldarstellung* eine eigene Notation, von der wir in Abschnitt 3.1.3.2 im Zusammenhang mit dem Resolutionskalkül umfassend Gebrauch machen werden.



Definition 3.10 (Klauseldarstellung)

Eine *Klausel* ist eine Menge von Literalen. Die Menge

$$\{(\neg)A_1, \dots, (\neg)A_i\}, \dots, \{(\neg)B_1, \dots, (\neg)B_j\}$$

steht stellvertretend für die Formel

$$((\neg)A_1 \vee \dots \vee (\neg)A_i) \wedge \dots \wedge ((\neg)B_1 \vee \dots \vee (\neg)B_j).$$

Die *leere Klausel* \square repräsentiert den Wahrheitswert 0.

■ Reduzierte KNF

$$\begin{aligned} F = & (B \vee C \vee D) \wedge \\ & (A \vee \neg B \vee D) \wedge \\ & (\neg B \vee \neg C \vee \neg D) \wedge \\ & (\neg A \vee B \vee \neg D) \end{aligned}$$

$$\begin{aligned} F = & (A \vee C \vee D) \wedge \\ & (A \vee \neg B \vee \neg C) \wedge \\ & (\neg A \vee \neg C \vee \neg D) \wedge \\ & (\neg A \vee B \vee C) \end{aligned}$$

■ Reduzierte DNF

$$\begin{aligned} F = & (\neg B \wedge C \wedge \neg D) \vee \\ & (A \wedge B \wedge \neg D) \vee \\ & (B \wedge \neg C \wedge D) \vee \\ & (\neg A \wedge \neg B \wedge D) \end{aligned}$$

$$\begin{aligned} F = & (\neg A \wedge \neg C \wedge D) \vee \\ & (\neg A \wedge \neg B \wedge C) \vee \\ & (A \wedge C \wedge \neg D) \vee \\ & (A \wedge B \wedge \neg C) \end{aligned}$$

Abbildung 3.11: Die betrachtete Beispielfunktion ist so strukturiert, dass sich je zwei Min- bzw. Maxterme zu einem gemeinsamen Term verschmelzen lassen. Die reduzierte Darstellung ist nicht mehr eindeutig, so dass wir die entstehenden Formeln nur noch als Normalformen und nicht mehr als kanonische Normalformen bezeichnen.

■ Kommutativität

$$\begin{array}{ccc} A \vee \neg B & & \neg B \vee A \\ \downarrow & & \downarrow \\ \{A, \neg B\} & & \{A, \neg B\} \end{array}$$

■ Assoziativität

$$\begin{array}{ccc} A \vee (\neg B \vee C) & & (A \vee \neg B) \vee C \\ \downarrow & & \downarrow \\ \{A, \neg B, C\} & & \{A, \neg B, C\} \end{array}$$

■ Idempotenz

$$\begin{array}{ccc} A \vee A & & A \vee A \vee A \\ \downarrow & & \downarrow \\ \{A\} & & \{A\} \end{array}$$

Abbildung 3.12: Zwischen Klauseln und Formeln besteht keine Eins-zu-eins-Beziehung. Jede Formel lässt sich eindeutig einer Klausel zuordnen, aber nicht umgekehrt. In der Klauseldarstellung sind die Eigenschaften der Kommutativität, Assoziativität und Idempotenz implizit vorhanden.

Auch wenn wir Klauseln fast immer wie Formeln behandeln werden, sollten Sie stets daran denken, dass zwischen beiden Darstellungsformen keine Eins-zu-eins-Beziehung besteht. Jede aussagenlogische Formel lässt sich eindeutig in eine Klausel verwandeln, aber nicht umgekehrt (vgl. Abbildung 3.12). Schuld daran ist die Mengendarstellung, in der zum einen die Information über die Reihenfolge der Elemente verloren geht und zum anderen jeder Term nur einmal aufgenommen werden kann. In einigen Anwendungsfällen ist dies gewollt. So ist die Klauseldarstellung der Formeldarstellung immer dann überlegen, wenn die Kommutativität ($F \vee G \equiv G \vee F$), die Assoziativität ($F \vee (G \vee H) \equiv (F \vee G) \vee H$) und die Idempotenz ($F \vee F \equiv F$) keine Rolle spielen.

3.1.3 Beweistheorie

In Abschnitt 3.1.1 haben wir die Semantik der Aussagenlogik über die Modellrelation \models festgelegt und darauf aufbauend den Begriff der *allgemeingültigen Formel* definiert.

In diesem Abschnitt werden wir eine Reihe von Beweissystemen einführen, mit deren Hilfe sich die Allgemeingültigkeit einer Formel formal beweisen lässt. Auch wenn die vorgestellten Systeme äußerlich betrachtet sehr unterschiedlich wirken, folgen sie alle dem gleichen Ansatz: Die Allgemeingültigkeit wird mit einem Regelsystem bewiesen, das auf der symbolischen Manipulation von Zeichenketten beruht. Da ein Beweis vollständig auf der syntaktischen Ebene durchgeführt wird, benötigen wir keinerlei Wissen über Interpretationen, Modelle oder andere Begriffe, die sich mit den semantischen Eigenschaften von Formeln beschäftigen. Ein solches Regelsystem bezeichnen wir fortan als *Kalkül*.

Jeder Kalkül definiert eine *Ableitungsrelation* \vdash , die über die folgenden Beziehungen mit der Modellrelation \models verbunden ist:



Definition 3.11 (Korrektheit, Vollständigkeit)

Sei K ein aussagenlogischer Kalkül. Ist eine Formel F innerhalb des Kalküls ableitbar, so schreiben wir $\vdash_K F$.

- K heißt *korrekt*, wenn aus $\vdash_K F$ stets $\models F$ folgt.
- K heißt *vollständig*, wenn aus $\models F$ stets $\vdash_K F$ folgt.

Ein korrekter Kalkül besitzt demnach die Eigenschaft, dass ausschließlich wahre Aussagen ableitbar sind. Vollständig ist ein Kalkül genau dann, wenn sich *alle* wahren Aussagen innerhalb des Kalküls als solche beweisen lassen. Geht aus dem Kontext hervor, auf welchen Kalkül wir uns beziehen, so schreiben wir nur noch $\vdash F$ anstelle von $\vdash_K F$.

Neben den geschilderten Kalkülen existieren sogenannte *Widerspruchskalküle*, in denen die Allgemeingültigkeit einer Formel F nicht konstruktiv bewiesen wird. Stattdessen wird die Erfüllbarkeit der negierten Formel $\neg F$ unterstellt und die Annahme zu einem Widerspruch geführt ($\neg F \models \square$). Aus der Unerfüllbarkeit von $\neg F$ ergibt sich dann sofort die Allgemeingültigkeit von F .

Die Abkehr von der konstruktiven Beweisführung macht es erforderlich, die weiter oben eingeführten Begriffe der Korrektheit und Vollständigkeit für Widerspruchskalküle geringfügig umzuformulieren:



Definition 3.12 (Korrektheit, Vollständigkeit)

Sei K ein Widerspruchskalkül. Lässt sich aus einer Formel F ein Widerspruch herleiten, so schreiben wir $F \vdash_K \square$.

- K heißt *korrekt*, wenn aus $\neg F \vdash_K \square$ stets $\models F$ folgt.
- K heißt *vollständig*, wenn aus $\models F$ stets $\neg F \vdash_K \square$ folgt.

Ein Widerspruchskalkül ist demnach korrekt, wenn ein Widerspruch nur aus der Negation einer allgemeingültigen Formel abgeleitet werden kann. Er ist vollständig, wenn sich aus der Negation einer allgemeingültigen Formel immer ein Widerspruch ableiten lässt. Auch hier schreiben wir \vdash anstelle von \vdash_K , wenn aus dem Kontext hervorgeht, auf welchen Kalkül wir uns beziehen.

In den nächsten Abschnitten werden Sie drei Kalküle kennen lernen, die sich in ihrer äußereren Form und der Art der Beweisführung erheblich voneinander unterscheiden. Der erste fällt in die Klasse der *Hilbert-Kalküle*. Diese Kalküle bilden weitgehend das Prinzip des mathematischen Schließens nach und bestechen vor allem durch ihre Transparenz. Auf der negativen Seite machen sie es vergleichsweise schwer, Beweise zu finden. In der Praxis wird aus diesem Grund meist auf *Resolutionskalküle* oder *Tableaukalküle* zurückgegriffen, die wir in den Abschnitten 3.1.3.2 und 3.2.3.2 besprechen werden.

■ Axiome

Abschwächung (A1)
$\frac{\{\}}{F \rightarrow (G \rightarrow F)}$
Distributivität (A2)
$\frac{\{\}}{(F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))}$
Kontraposition (A3)
$\frac{\{\}}{(\neg F \rightarrow \neg G) \rightarrow (G \rightarrow F)}$

■ Schlussregeln

Modus ponens (MP)
$\frac{F, F \rightarrow G}{G}$

Tabelle 3.5: Axiome und Schlussregeln des betrachteten Hilbert-Kalküls

3.1.3.1 Hilbert-Kalkül

Hilbert-Kalküle zeichnen sich dadurch aus, dass wahre Aussagen aus einer Menge von Axiomen durch die sukzessive Anwendung fest definierter Schlussregeln abgeleitet werden. In diesen Systemen ist ein Beweis eine Folge von Tautologien, an deren Ende die zu zeigende Behauptung steht. Alle anderen Formeln in der Beweiskette sind entweder ein Axiom oder eine Tautologie, die mit einer der Schlussregeln aus den vorher erzeugten Formeln abgeleitet wurde.

Im Folgenden werden wir die Schlussregeln in der Notation

$$\frac{F_1, F_2, \dots, F_n}{G}$$

angeben. Die über dem Mittelstrich notierten Formeln bilden zusammen die *Prämissen* und beschreiben die Voraussetzungen, die vor der Anwendung der Regel erfüllt sein müssen. Die unter dem Mittelstrich notierte Formel ist die *Konklusion*, d. h. die Schlussfolgerung, die aus der Prämisse abgeleitet werden kann. Das Notationsschema ist stark genug, um auch Axiome darzustellen, da wir diese ganz einfach als spezielle Schlussregeln mit leerer Prämisse auffassen können.

Im Folgenden betrachten wir den Hilbert-Kalkül, der durch die Axiome und Schlussregeln aus Tabelle 3.5 gegeben ist. Das erste Axiom wird als *Abschwächungsregel* bezeichnet und besagt, dass aus F stets auch $G \rightarrow F$ folgt. Das zweite Axiom drückt die *Distributivitätseigenschaft* des Implikationsoperators aus. Das dritte und letzte Axiom ist die logische *Kontraposition* – ein Schlussprinzip, das wir täglich einsetzen. Es postuliert, dass wir die logische Schlussrichtung umdrehen können, wenn wir die Argumente verneinen („Wenn es regnet, dann ist die Straße nass“ ist gleichbedeutend mit „Wenn die Straße nicht nass ist, dann regnet es nicht“). Innerhalb des Kalküls existiert mit dem *Modus ponens* eine einzige Schlussregel, mit der neue Sätze abgeleitet werden können. Diese Regel ist uns intuitiv vertraut; sie besagt, dass G wahr sein muss, wenn wir wissen, dass F wahr ist und G aus F gefolgt werden kann.

Auf den ersten Blick wirkt der Kalkül ungewöhnlich spartanisch. Zum einen mag es den einen oder anderen Leser verwundern, dass er mit nur drei Axiomen und einer einzigen Schlussregel auskommt. Zum anderen scheint er nur eine begrenzte Formelklasse zu beschreiben, da die Axiome und Schlussregeln vollständig ohne die Elementaroperatoren \wedge und \vee formuliert sind. Der Ausschluss dieser Verknüpfungen ist jedoch keine Beschränkung im eigentlichen Sinne, da wir in Abschnitt 3.1.1 gezeigt haben, dass die Menge $\{\neg, \rightarrow\}$ ein vollständiges Operatoren-System bildet. Folgerichtig können wir jede aussagenlogische Formel

so umformen, dass sie nur noch Negations- und Implikationsoperatoren enthält.

Beachten Sie bei der Betrachtung der Axiome und Regeln, dass die Variablen lediglich Platzhalter sind, die durch beliebige aussagenlogische Ausdrücke substituiert werden können. So lassen sich aus dem Distributivitätsaxiom unter anderem die folgenden *Instanzen* bilden:

- Substitution: $[F \leftarrow B, G \leftarrow C, H \leftarrow A]$
 $\vdash B \rightarrow (C \rightarrow A) \rightarrow ((B \rightarrow C) \rightarrow (B \rightarrow A))$
- Substitution: $[F \leftarrow A, G \leftarrow A, H \leftarrow A]$
 $\vdash A \rightarrow (A \rightarrow A) \rightarrow ((A \rightarrow A) \rightarrow (A \rightarrow A))$
- Substitution: $[F \leftarrow A, G \leftarrow (A \rightarrow A), H \leftarrow A]$
 $\vdash (A \rightarrow ((A \rightarrow A) \rightarrow A)) \rightarrow ((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A))$

Das folgende Beispiel zeigt, wie die Tautologie $A \rightarrow A$ innerhalb des Hilbert-Kalküls bewiesen werden kann:

- 1: $\vdash (A \rightarrow ((A \rightarrow A) \rightarrow A)) \rightarrow ((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A))$ (A2)
- 2: $\vdash (A \rightarrow ((A \rightarrow A) \rightarrow A))$ (A1)
- 3: $\vdash (A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)$ (MP 1,2)
- 4: $\vdash (A \rightarrow (A \rightarrow A))$ (A1)
- 5: $\vdash (A \rightarrow A)$ (MP 3,4)

Das erste und zweite Glied der Beweiskette sind Instanzen des Distributivitätsaxioms und des Abschwächungsaxioms. Das dritte Glied entsteht durch die Anwendung der Schlussregel auf die vorher erzeugten Formeln. Das vierte Glied ist wiederum eine Instanz des Abschwächungsaxioms. Jetzt lässt sich F mithilfe der Schlussregel aus den Tautologien 3 und 4 direkt ableiten.

Nachdem wir die Formel $A \rightarrow A$ als allgemeingültig identifiziert haben, können wir auf sie in allen zukünftigen Beweisen zurückgreifen. Der Hilbert-Kalkül ist bewusst so ausgelegt, dass einmal bewiesene Formeln nicht mehr erneut bewiesen werden müssen. Solche Formeln dürfen wir wie Axiome behandeln und an beliebigen Stellen in einer Beweiskette einfügen. Mit der Zeit entsteht eine sich kontinuierlich vergrößernde Bibliothek von Tautologien, der wir uns frei bedienen können. Damit spiegelt der Hilbert-Kalkül eins zu eins das Vorgehen der klassischen Mathematik wider. Auch hier beweisen wir eine Behauptung, indem wir auf den reichhaltigen Fundus bereits bekannter Sätze zurückgreifen.

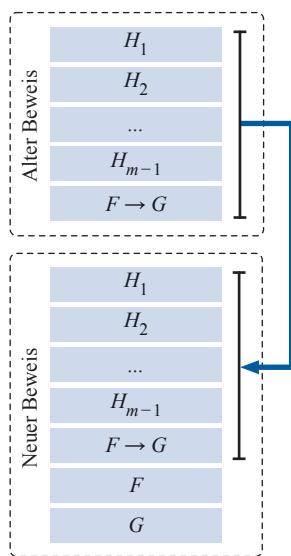


Abbildung 3.13: Beweisschema des Deduktionstheorems (Schlussrichtung von rechts nach links)

Um den Hilbert-Kalkül gewinnbringend einsetzen zu können, wollen wir ihn noch um einen wichtigen Baustein ergänzen. Die Rede ist von *Annahmen*, die in der klassischen Mathematik in den verschiedensten Formen gemacht werden und nicht notwendigerweise selbst wahr sein müssen. Um auch Aussagen der Form „Unter der Annahme, dass F gilt, folgt G “ mithilfe des Hilbert-Kalküls modellieren zu können, erlauben wir, einen Beweis um eine Menge von *Voraussetzungen* zu ergänzen. In diesem erweiterten Kalkül ist ein Beweis eine Kette von Formeln F_1, F_2, \dots, F_n , die nach den folgenden Konstruktionsregeln gebildet wird:

- F_i ist eine Instanz eines Axioms oder
- F_i ist eine Instanz einer Voraussetzung oder
- F_i entsteht aus den vorangegangenen Gliedern der Beweiskette durch die Anwendung einer Schlussregel.

Bezeichnet M die Menge der Voraussetzungen, so schreiben wir $M \vdash F$, falls sich die Formel F mit den beschriebenen Konstruktionsregeln ableiten lässt. Mit dieser Notation können wir den weiter oben eingeführten Ausdruck $\vdash F$ als abkürzende Schreibweise für $\emptyset \vdash F$ auffassen. Beachten Sie, dass für jede Formel F und eine beliebige Formelmenge M die folgende Beziehung gilt:

$$\{F\} \cup M \vdash F \quad (3.4)$$

Die Korrektheit folgt unmittelbar aus den oben genannten Konstruktionsregeln.

Vielleicht haben Sie sich selbst die Frage gestellt, ob eine Erweiterung des Hilbert-Kalküls wirklich notwendig ist, schließlich sind wir in der Lage, beliebige Wenn-dann-Beziehungen mithilfe des Implikationsoperators \rightarrow zu formulieren. Der Unterschied zwischen beiden Konstrukten besteht darin, dass der Operator \rightarrow innerhalb der Logik existiert, während die Folgerungsbeziehung $M \vdash F$ eine Aussage über die Beweisbarkeit der Aussage F macht. Mit anderen Worten: $M \vdash F$ ist eine Metaaussage, die außerhalb der Logik steht. Nichtsdestotrotz existiert zwischen beiden Konstrukten ein enger Zusammenhang, wie das nachstehende Theorem zum Ausdruck bringt:



Satz 3.2 (Deduktionstheorem der Aussagenlogik)

Für beliebige aussagenlogische Formeln F, F_1, \dots, F_n und G gilt:

$$\{F_1, \dots, F_n\} \cup \{F\} \vdash G \Leftrightarrow \{F_1, \dots, F_n\} \vdash F \rightarrow G$$

Das Deduktionstheorem ist in seiner Bedeutung nicht zu unterschätzen. Zum einen erlaubt es uns, zwischen der Logik- und der Metaebene nach Belieben hin- und herzuspringen. Zum anderen wird es uns in die Lage versetzen, Beweise deutlich einfacher zu finden als bisher. Doch bevor wir das Deduktionstheorem produktiv einsetzen können, wollen wir uns zunächst von seiner Richtigkeit überzeugen.

Die Richtung von rechts nach links ist nahezu trivial. Gilt

$$\{F_1, \dots, F_n\} \vdash F \rightarrow G,$$

so existiert ein formaler Beweis, der $F \rightarrow G$ aus $\{F_1, \dots, F_n\}$ ableitet. Die Schlusskette können wir zu einem Beweis verlängern, der G aus $\{F_1, \dots, F_n, F\}$ ableitet. Hierzu setzen wir F zunächst als Instanz ein und leiten G anschließend durch die Modus-Ponens-Schlussregel aus $F \rightarrow G$ und F ab (vgl. Abbildung 3.13).

Die Schlussrichtung von links nach rechts erfordert etwas mehr Aufwand, folgt aber dem gleichen Schema. Ausgehend von einem Beweis für G aus $\{F_1, \dots, F_n\} \cup \{F\}$ werden wir einen Beweis für $F \rightarrow G$ aus $\{F_1, \dots, F_n\}$ konstruieren. Das Grundschema des neuen Beweises ist in Abbildung 3.14 skizziert. Aus der vorhandenen Beweiskette H_1, \dots, H_{m-1}, G erzeugen wir eine neue, in der nacheinander die Formeln $F \rightarrow H_i$ abgeleitet werden und am Ende die zu beweisende Behauptung $F \rightarrow G$ steht. Beachten Sie, dass die Abbildung lediglich die Grobstruktur der Beweiskette festgelegt, da zwischen den Formeln Leerräume gelassen wurden. Die folgenden Ableitungssequenzen zeigen, wie diese genau zu füllen sind. Wir unterscheiden drei Fälle:

- H_i ist ein Axiom oder eine Voraussetzung

$$\begin{aligned} &\vdash H_i \\ &\vdash H_i \rightarrow (F \rightarrow H_i) \quad (\text{A1}) \\ &\vdash (F \rightarrow H_i) \quad (\text{MP}) \end{aligned}$$

- H_i ist die Formel F

$$\begin{aligned} &\vdash (F \rightarrow ((F \rightarrow F) \rightarrow F)) \quad (\text{A1}) \\ &\vdash (F \rightarrow ((F \rightarrow F) \rightarrow F)) \rightarrow ((F \rightarrow (F \rightarrow F)) \rightarrow (F \rightarrow F)) \quad (\text{A2}) \\ &\vdash (F \rightarrow (F \rightarrow F)) \rightarrow (F \rightarrow F) \quad (\text{MP}) \\ &\vdash (F \rightarrow (F \rightarrow F)) \quad (\text{A1}) \\ &\vdash (F \rightarrow F) \quad (\text{MP}) \end{aligned}$$

- H_i wurde durch die Regel (MP) aus H_j und $H_j \rightarrow H_i$ erzeugt.

In diesem Fall wissen wir, dass die Formeln

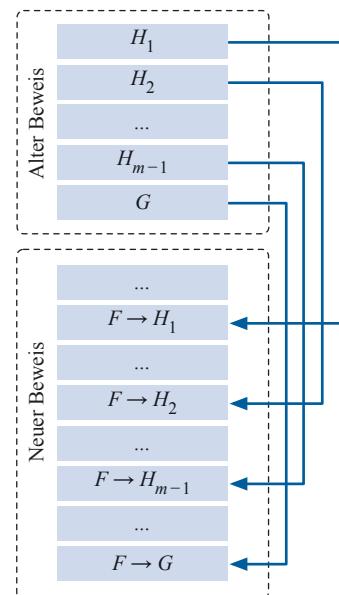


Abbildung 3.14: Beweisschema des Deduktionstheorems (Schlussrichtung von links nach rechts)

Theorem T1	$(F \rightarrow H_j),$ $(F \rightarrow (H_j \rightarrow H_i))$
$(A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$	
Theorem T2	bereits abgeleitet sind. Wir können den Beweis damit wie folgt ergänzen:
$A \rightarrow ((A \rightarrow B) \rightarrow B)$	$\vdash (F \rightarrow (H_j \rightarrow H_i)) \rightarrow ((F \rightarrow H_j) \rightarrow (F \rightarrow H_i))$ (A2) $\vdash (F \rightarrow H_j) \rightarrow (F \rightarrow H_i)$ (MP) $\vdash (F \rightarrow H_i)$ (MP)
Theorem T3	Damit ist das Deduktionstheorem bewiesen.
$\neg\neg A \rightarrow A$	
Theorem T4	Die in Tabelle 3.6 zusammengefassten Formeln stammen aus [72] und sollen uns als Beispiele dienen, wie sich mithilfe des Deduktionstheorems Beweise führen lassen. Die einzelnen Ableitungssequenzen sind in Abbildung 3.15 zusammengefasst. Eine Analyse der Beweisketten zeigt, dass die Theoreme aufeinander aufbauen, d. h., es wird so oft wie möglich auf bereits bewiesene Ergebnisse zurückgegriffen. Welche Abhängigkeiten zwischen den Theoremen im Einzelnen bestehen, wird in Abbildung 3.16 grafisch verdeutlicht.
$A \rightarrow \neg\neg A$	
Theorem T5	Beachten Sie, dass die entwickelten Ableitungssequenzen keine echten Beweise im Sinne des ursprünglichen Hilbert-Kalküls sind. Verantwortlich hierfür ist die Eigenschaft des Deduktionstheorems, eine Metaschlussregel zu sein, die Aussagen <i>über</i> Beweise macht und nicht <i>innerhalb</i> des Kalküls existiert. Mit jeder Anwendung des Deduktionstheorems treten wir gewissermaßen aus dem Kalkül heraus. Dass wir die Ableitungssequenzen trotzdem als Beweis ansehen dürfen, verdanken wir unserer geleisteten Vorausbereitung. Weiter oben haben wir gezeigt, wie sich jede mit (DT) markierte Ableitung durch eine äquivalente Ableitungssequenz ersetzen lässt, die ohne das Deduktionstheorem auskommt. In diesem Sinne können wir die gezeigten Ableitungssequenzen als Bauplan verstehen, aus dem sich systematisch eine <i>echte</i> Beweiskette erzeugen lässt.
$(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$	
Theorem T6	Abschließend wollen wir uns mit der Frage beschäftigen, wie es um die Korrektheit und Vollständigkeit des Hilbert-Kalküls in der hier vorgestellten Form steht. Wie wir in Definition 3.11 formal fixiert haben, ist ein Kalkül genau dann korrekt, wenn sich ausschließlich wahre Aussagen beweisen lassen (aus $\vdash F$ folgt $\models F$). Vollständig ist er genau dann, wenn alle wahren Aussagen innerhalb des Kalküls abgeleitet werden können (aus $\models F$ folgt $\vdash F$).
$A \rightarrow (\neg B \rightarrow \neg(A \rightarrow B))$	
Theorem T7	Die Korrektheit des Hilbert-Kalküls liegt auf der Hand. Zunächst lässt sich zeigen, dass alle Axiome allgemeingültig sind. Im Übungsteil auf
$\neg A \rightarrow (A \rightarrow B)$	
Theorem T8	
$\neg(A \rightarrow B) \rightarrow A$	
Theorem T9	
$\neg(A \rightarrow B) \rightarrow \neg B$	

Tabelle 3.6: Übersicht über die in Abbildung 3.15 bewiesenen Theoreme

$\blacksquare (A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$	(T1)	$\{A \rightarrow B\} \vdash \neg\neg A \rightarrow A$	(T3)
$\{A \rightarrow B, B \rightarrow C, A\} \vdash A$	(3.4)	$\{A \rightarrow B\} \vdash (A \rightarrow B) \rightarrow (\neg\neg A \rightarrow B)$	(T1+MP)
$\{A \rightarrow B, B \rightarrow C, A\} \vdash A \rightarrow B$	(3.4)	$\{A \rightarrow B\} \vdash A \rightarrow B$	(3.4)
$\{A \rightarrow B, B \rightarrow C, A\} \vdash B$	(MP)	$\{A \rightarrow B\} \vdash \neg\neg A \rightarrow B$	(MP)
$\{A \rightarrow B, B \rightarrow C, A\} \vdash B \rightarrow C$	(3.4)	$\{A \rightarrow B\} \vdash B \rightarrow \neg\neg B$	(T4)
$\{A \rightarrow B, B \rightarrow C, A\} \vdash C$	(MP)	$\{A \rightarrow B\} \vdash (B \rightarrow \neg\neg B) \rightarrow (\neg\neg A \rightarrow \neg\neg B)$	(T1+MP)
$\{A \rightarrow B, B \rightarrow C\} \vdash A \rightarrow C$	(DT)	$\{A \rightarrow B\} \vdash \neg\neg A \rightarrow \neg\neg B$	(MP)
$\{A \rightarrow B\} \vdash (B \rightarrow C) \rightarrow (A \rightarrow C)$	(DT)	$\{A \rightarrow B\} \vdash \neg B \rightarrow \neg A$	(MP)
$\vdash (A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$	(DT)	$\vdash (A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$	(DT)
<hr/>			
$\blacksquare A \rightarrow ((A \rightarrow B) \rightarrow B)$	(T2)	$\blacksquare A \rightarrow (\neg B \rightarrow \neg(A \rightarrow B))$	(T6)
$\{A, A \rightarrow B\} \vdash A$	(3.4)	$\{A\} \vdash (A \rightarrow B) \rightarrow B$	(T2+DT)
$\{A, A \rightarrow B\} \vdash A \rightarrow B$	(3.4)	$\{A\} \vdash (A \rightarrow B) \rightarrow B \rightarrow (\neg B \rightarrow \neg(A \rightarrow B))$	(T5)
$\{A, A \rightarrow B\} \vdash B$	(MP)	$\{A\} \vdash \neg B \rightarrow \neg(A \rightarrow B)$	(MP)
$\{A\} \vdash (A \rightarrow B) \rightarrow B$	(DT)	$\{A\} \vdash A \rightarrow (\neg B \rightarrow \neg(A \rightarrow B))$	(DT)
$\vdash A \rightarrow ((A \rightarrow B) \rightarrow B)$	(DT)	<hr/>	
<hr/>			
$\blacksquare \neg\neg A \rightarrow A$	(T3)	$\blacksquare \neg A \rightarrow (A \rightarrow B)$	(T7)
$\vdash \neg\neg A \rightarrow (\neg\neg\neg A \rightarrow \neg\neg A)$	(A1)	$\{\neg A\} \vdash \neg B \rightarrow \neg A$	(A1+DT)
$\{\neg\neg A\} \vdash \neg\neg\neg A \rightarrow \neg\neg A$	(DT)	$\{\neg A\} \vdash (\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$	(A3)
$\{\neg\neg A\} \vdash (\neg\neg\neg A \rightarrow \neg\neg A) \rightarrow (\neg A \rightarrow \neg\neg A)$	(A3)	$\{\neg A\} \vdash (A \rightarrow B)$	(MP)
$\{\neg\neg A\} \vdash \neg A \rightarrow \neg\neg A$	(MP)	$\vdash \neg A \rightarrow (A \rightarrow B)$	(DT)
$\{\neg\neg A\} \vdash (\neg A \rightarrow \neg\neg A) \rightarrow (\neg\neg A \rightarrow A)$	(A3)	<hr/>	
$\{\neg\neg A\} \vdash \neg\neg A \rightarrow A$	(MP)	$\blacksquare \neg(A \rightarrow B) \rightarrow A$	(T8)
$\{\neg\neg A\} \vdash A$	(DT)	$\vdash \neg A \rightarrow (A \rightarrow B)$	(T7)
$\vdash \neg\neg A \rightarrow A$	(DT)	$\vdash (\neg A \rightarrow (A \rightarrow B)) \rightarrow (\neg(A \rightarrow B) \rightarrow \neg\neg A)$	(T5)
<hr/>			
$\blacksquare A \rightarrow \neg\neg A$	(T4)	$\vdash \neg\neg(A \rightarrow B) \rightarrow \neg\neg A$	(MP)
$\vdash \neg\neg A \rightarrow \neg A$	(T3)	$\vdash \neg\neg A \rightarrow A$	(T3)
$\vdash (\neg\neg A \rightarrow \neg A) \rightarrow (A \rightarrow \neg\neg A)$	(A3)	$\vdash ((\neg\neg A \rightarrow A) \rightarrow (\neg(A \rightarrow B) \rightarrow A))$	(T1+MP)
$\vdash A \rightarrow \neg\neg A$	(MP)	$\vdash \neg(A \rightarrow B) \rightarrow A$	(MP)
<hr/>			
$\blacksquare (A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$	(T5)	$\blacksquare \neg(A \rightarrow B) \rightarrow \neg B$	(T9)
$\{A \rightarrow B\} \vdash (\neg\neg A \rightarrow \neg\neg B) \rightarrow (\neg B \rightarrow \neg A)$	(A3)	$\vdash B \rightarrow (A \rightarrow B)$	(A1)
		$\vdash (B \rightarrow (A \rightarrow B)) \rightarrow (\neg(A \rightarrow B) \rightarrow \neg B)$	(T5)
		$\vdash \neg(A \rightarrow B) \rightarrow \neg B$	(MP)

Abbildung 3.15: Beweisführung im Hilbert-Kalkül

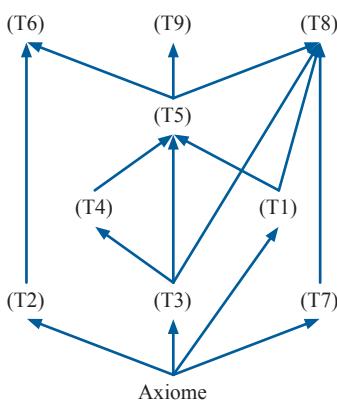


Abbildung 3.16: Die Beweise der Theoreme T1 bis T9 sind so angelegt, dass so oft wie möglich auf bereits bewiesene Ergebnisse zurückgegriffen wird. Der dargestellte Graph macht deutlich, welche Abhängigkeiten zwischen den Einzelbeweisen bestehen.

Seite 153 dürfen Sie sich durch das Aufstellen der Wahrheitstabellen selbst davon überzeugen. Ferner ist leicht nachzuvollziehen, dass die einzige Schlussregel des Kalküls – der Modus ponens – die Tautologie-eigenschaft erhält. Mit anderen Worten: Sind die Prämissen allgemeingültig, so ist es auch die Konklusion. Damit sind alle ableitbaren Formeln Tautologien und der Hilbert-Kalkül im Sinne von Definition 3.11 korrekt.

Die Vollständigkeit des Hilbert-Kalküls ist weit weniger offensichtlich. Sie erfordert einen komplizierten Beweis, den wir hier nur grob in seiner Struktur skizzieren wollen. Er besteht aus zwei Teilen, in denen die folgenden Teilaussagen nacheinander gezeigt werden:

- Teilaussage 1: „Aus $M \not\vdash F$ folgt: $M \cup \{\neg F\}$ ist konsistent.“

Eine Formelmenge M heißt konsistent, wenn für keine Formel F sowohl F als auch $\neg F$ gleichzeitig aus M abgeleitet werden kann.

- Teilaussage 2: „Jede konsistente Menge hat ein Modell“

Die Aussage kann konstruktiv bewiesen werden, indem die Formelmenge iterativ um weitere Elemente angereichert wird. Sobald der Konstruktionsprozess endet, lässt sich aus der Ergebnismenge ein Modell extrahieren.

Zusammen führen beide Teilaussagen zu folgender Überlegung: Aus $M \not\vdash F$ folgt, dass $M \cup \{\neg F\}$ ein Modell besitzt. Dann gilt aber $M \not\models F$ und im Umkehrschluss schließlich $M \models F \Rightarrow M \vdash F$.

3.1.3.2 Resolutionskalkül

Der Hilbert-Kalkül besticht zum einen durch seine einfache Struktur und zum anderen durch seine Vorgehensweise, die sich eng an jene der klassischen Mathematik anlehnt. Für die praktische Anwendung ist er jedoch nur eingeschränkt geeignet, da Beweise oft nur mit Mühe gefunden werden können. Die Suche wird insbesondere dadurch erschwert, dass in vielen Beweisen Instanzen der Axiome nach einem Schema erzeugt werden müssen, das auf den ersten Blick kaum ersichtlich ist.

In diesem Abschnitt werden wir mit dem *Resolutionskalkül* eine deutlich praktikablere Methode kennenlernen. Der Kalkül fällt in die Klasse der Widerspruchskalküle und setzt eine Formel in Klauseldarstellung voraus. Ein Resolutionsbeweis wird geführt, indem die Klauselmenge so lange erweitert wird, bis keine neuen Elemente mehr gebildet werden können oder die leere Klausel \square erzeugt wurde.

Für die Bildung neuer Klauseln steht die in Abbildung 3.17 dargestellte *Resolutionsregel* zur Verfügung. Sie lässt sich auf jedes Klauselpaar anwenden, das eine gemeinsame Variable A_i mit unterschiedlichen Vorzeichen enthält. Die neu erzeugte Klausel heißt *Resolvente* und wird gebildet, indem die Ausgangsklauseln vereinigt und alle Vorkommen von A_i und $\neg A_i$ entfernt werden.

Das Ziel eines Resolutionsbeweises ist es, die leere Klausel \square abzuleiten. Hierzu sind im Allgemeinen mehrere Resolutionsschritte notwendig, die sich übersichtlich in einem *Resolutionsbaum* darstellen lassen (vgl. Abbildung 3.18). Beachten Sie, dass der Kalkül ausschließlich auf Formeln in Klauseldarstellung arbeitet und die Eingabe im Rahmen einer Vorverarbeitung zunächst in eine konjunktive Form gebracht werden muss.

Alles in allem setzt sich ein vollständiger Resolutionsbeweis aus drei Schritten zusammen, die nacheinander durchlaufen werden müssen:

- Die zu beweisende Formel F wird negiert und in die konjunktive Normalform übersetzt.
- Die konjunktive Normalform wird in die Klauseldarstellung überführt.
- Durch die kontinuierliche Bildung von Resolventen wird die leere Klausel \square erzeugt.

Als Beispiel werden wir versuchen, die Allgemeingültigkeit der Formel

$$(A \leftrightarrow B) \vee (A \leftrightarrow C) \vee (B \leftrightarrow C)$$

mithilfe des Resolutionskalküls zu beweisen. Im ersten Schritt müssen wir die Formel negieren und die konjunktive Normalform erzeugen:

$$\begin{aligned} & \neg((A \leftrightarrow B) \vee (A \leftrightarrow C) \vee (B \leftrightarrow C)) \\ & \equiv \neg(A \leftrightarrow B) \wedge \neg(A \leftrightarrow C) \wedge \neg(B \leftrightarrow C) \\ & \equiv (A \leftrightarrow B) \wedge (A \leftrightarrow C) \wedge (B \leftrightarrow C) \\ & \equiv (A \vee B) \wedge (\neg A \vee \neg B) \wedge (A \vee C) \wedge \\ & \quad (\neg A \vee \neg C) \wedge (B \vee C) \wedge (\neg B \vee \neg C) \end{aligned}$$

In Klauselform ausgedrückt erhalten wir die folgende Darstellung:

$$\{A, B\}, \{\neg A, \neg B\}, \{A, C\}, \{\neg A, \neg C\}, \{B, C\}, \{\neg B, \neg C\}$$

■ Resolutionsregel

$$\frac{\{A_i\} \cup M_1 \quad \{\neg A_i\} \cup M_2}{M_1 \cup M_2}$$

■ Beispiel 1

$$\frac{\{A, B\} \quad \{\neg A, \neg C\}}{\{B, \neg C\}}$$

■ Beispiel 2

$$\frac{\{B, \neg C\} \quad \{\neg B, \neg C\}}{\{\neg C\}}$$

■ Beispiel 3

$$\frac{\{C\} \quad \{\neg C\}}{\square}$$

Abbildung 3.17: Aussagenlogische Resolventenbildung

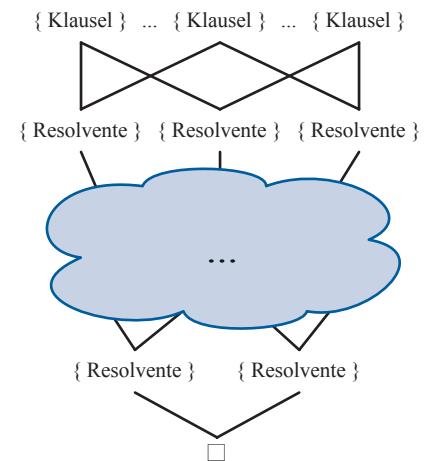


Abbildung 3.18: Allgemeines Schema eines Resolutionsbeweises

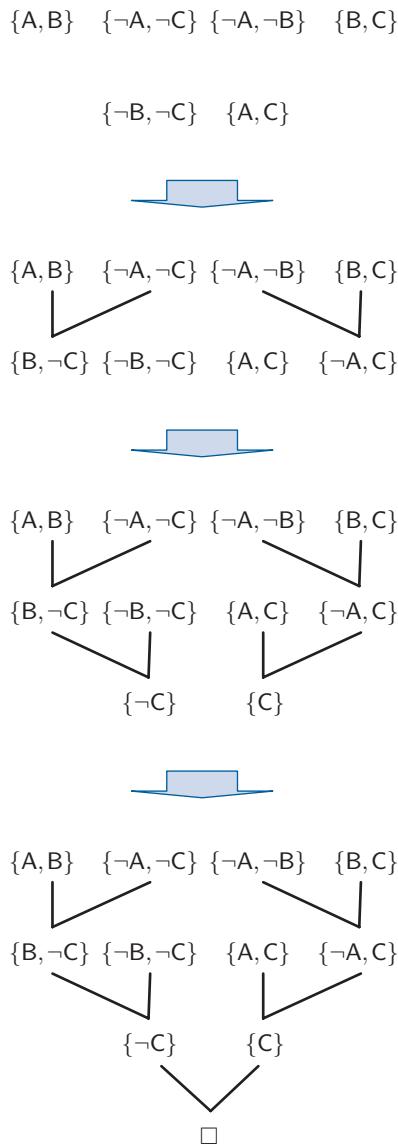


Abbildung 3.19: Schrittweise Konstruktion des Resolutionsbaums

Abbildung 3.19 zeigt schrittweise, wie sich der Resolutionsbaum aus der ursprünglichen Klauselmenge aufbauen lässt. Nach 5 Regelanwendungen ist die leere Klausel \square erzeugt und damit die Unerfüllbarkeit der Klauselmenge bewiesen.

Als Nächstes wollen wir das in Abschnitt 3.1.1 eingeführte Schubfachprinzip mithilfe der aussagenlogischen Resolution beweisen. Ausgangspunkt ist die in Abbildung 3.7 eingeführte Formel für 3 Gegenstände und 2 Schubfächer:

$$F = \left(\begin{array}{l} (A_{11} \vee A_{12}) \wedge \\ (A_{21} \vee A_{22}) \wedge \\ (A_{31} \vee A_{32}) \end{array} \right) \rightarrow \left(\begin{array}{l} A_{11}A_{21} \vee A_{11}A_{31} \vee A_{21}A_{31} \vee \\ A_{12}A_{22} \vee A_{12}A_{32} \vee A_{22}A_{32} \end{array} \right)$$

Im ersten Schritt transformieren wir die negierte Formel $\neg F$ in eine konjunktive Form. Wir erhalten das nachstehende Ergebnis:

$$\begin{aligned} \neg F \equiv & (A_{11} \vee A_{12}) \wedge (A_{21} \vee A_{22}) \wedge (A_{31} \vee A_{32}) \wedge \\ & (\neg A_{11} \vee \neg A_{21}) \wedge (\neg A_{11} \vee \neg A_{31}) \wedge (\neg A_{21} \vee \neg A_{31}) \wedge \\ & (\neg A_{12} \vee \neg A_{22}) \wedge (\neg A_{12} \vee \neg A_{32}) \wedge (\neg A_{22} \vee \neg A_{32}) \end{aligned}$$

In Klauseldarstellung liest sich die Formel so:

$$\begin{array}{lll} \{A_{11}, A_{12}\}, & \{A_{21}, A_{22}\}, & \{A_{31}, A_{32}\} \\ \{\neg A_{11}, \neg A_{21}\}, & \{\neg A_{11}, \neg A_{31}\}, & \{\neg A_{21}, \neg A_{31}\} \\ \{\neg A_{12}, \neg A_{22}\}, & \{\neg A_{12}, \neg A_{32}\}, & \{\neg A_{22}, \neg A_{32}\} \end{array}$$

Abbildung 3.20 zeigt, wie sich aus der konstruierten Menge die leere Klausel \square ableiten lässt. Damit ist die Gültigkeit des Dirichlet'schen Schubfachprinzips formal bewiesen.

Die beiden Beispiele geben einen ersten Eindruck, wie sich die Allgemeingültigkeit von aussagenlogischen Formeln im Resolutionskalkül beweisen lässt. Damit ist es an der Zeit, dass wir uns über die Korrektheit und Vollständigkeit des Kalküls Gedanken machen. Die Korrektheit der Resolutionsmethode ist leicht einzusehen. Ist M eine Klauselmenge und F eine ableitbare Resolvente, so besitzen M und $M \cup \{F\}$ die gleichen Modelle. Ist es möglich, M um die leere Klausel \square zu ergänzen, so kann M kein Modell besitzen, da jedes Modell für F auch ein Modell für $M \cup \{\square\}$ und damit auch ein Modell für \square sein müsste. Die leere Klausel \square steht jedoch stellvertretend für den Wahrheitswert 0 und damit für eine unerfüllbare Formel. Genau wie im Falle des Hilbert-Kalküls lässt sich zeigen, dass der Resolutionskalkül vollständig ist, d. h., für jede unerfüllbare Menge sind wir in der Lage, die leere Klausel \square auch wirklich abzuleiten.

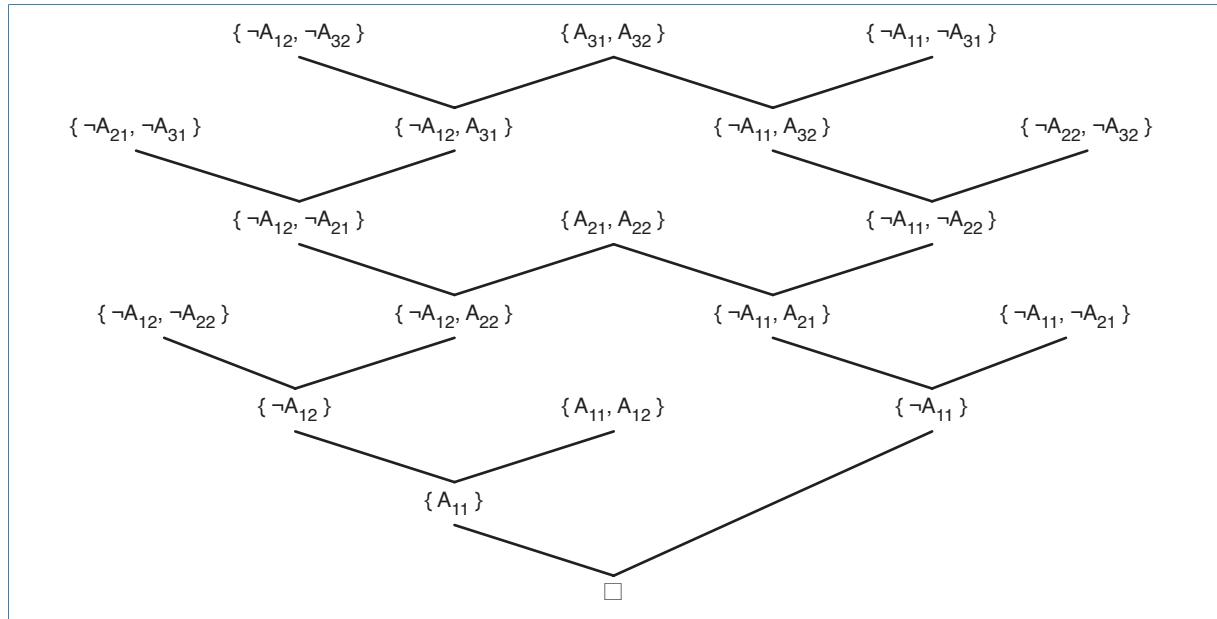


Abbildung 3.20: Formaler Beweis des Schubfachprinzips

Der Resolutionskalkül besticht vor allem durch seine einfache Anwendbarkeit, schließlich ist eine einzige Regel ausreichend, um den kompletten Beweis zu führen. Ferner müssen keine Formelinstanzen aus dem Nichts generiert werden, wie es im Hilbert-Kalkül der Fall ist. Die Achillesferse des Resolutionskalküls ist die Vorverarbeitung. Liegt eine Formel nicht in konjunktiver Form vor, so muss diese vorab erzeugt werden. Eine entsprechende Umformung ist zwar immer möglich, allerdings nimmt die Anzahl der resultierenden Klauseln für viele aussagenlogische Ausdrücke exponentiell mit der Formellänge zu. Eindrucksvoll lässt sich das Phänomen am Beispiel der n -stelligen Paritätsfunktion

$$P_n := A_1 \leftrightarrow A_2 \leftrightarrow \dots \leftrightarrow A_n$$

demonstrieren (vgl. Tabelle 3.7). Für die Praxis bedeutet dieses Ergebnis, dass die konjunktive Form vieler realer Formeln nicht mehr darstellbar ist. An den Einsatz des Resolutionskalküls ist in diesen Fällen nicht zu denken, da der Beweis bereits an der Vorverarbeitung scheitert. Trotzdem gibt es keinen Grund, vorschnell die Waffen zu strecken. Im nächsten Abschnitt werden wir mit dem semantischen Tableau einen Kalkül kennen lernen, der die geschilderte Limitierung beseitigt und auf jegliche Vorverarbeitungen der zu beweisenden Formel verzichtet.

Tabelle 3.7: Im Allgemeinen wächst die Klauseldarstellung exponentiell mit der Formelgröße an, hier demonstriert am Beispiel der Paritätsfunktion $P_n = A_1 \leftrightarrow A_2 \leftrightarrow \dots \leftrightarrow A_n$

3.1.3.3 Tableaukalkül

Der Tableaukalkül gehört wie der Resolutionskalkül zur Gruppe der Widerspruchskalküle, d. h., die Allgemeingültigkeit einer Formel wird über die Unerfüllbarkeit ihrer Negation bewiesen. Trotz dieser Gemeinsamkeit ist die Art der Beweisführung eine völlig andere. Im Tableaukalkül wird eine Formel schrittweise in eine Baumstruktur – das *Tableau* – transformiert, aus dem sich die zu beweisenden Formeleigenschaften ablesen lassen. Das Verfahren besticht durch zwei wesentliche Merkmale. Zum einen lässt sich das Tableau einer Formel direkt aus deren Teilformeln aufbauen, so dass eine Vorverarbeitung, wie sie im Resolutionskalkül notwendig ist, vollständig entfällt. Zum anderen lässt sich der Kalkül einsetzen, um Modelle für erfüllbare Formeln zu erzeugen. Schlägt ein Beweis fehl, so kann aus dem konstruierten Tableau eine erfüllende Variablenbelegung abgelesen werden. Jede solche Variablenbelegung spielt die Rolle eines *Gegenbeispiels*, das die Unerfüllbarkeitsannahme widerlegt.

Ein aussagenlogisches Tableau für eine Formel F ist durch zwei Eigenschaften charakterisiert:

- Jeder Knoten ist mit einer Teilformel oder einer negierten Teilformel von F beschriftet.
- Für jedes Modell I von F existiert ein Pfad, so dass I ein gemeinsames Modell für alle Formeln dieses Pfads ist.

Als Beispiel ist in Abbildung 3.21 ein Tableau für die aussagenlogische Formel $F := A \wedge (A \rightarrow B)$ dargestellt. Innerhalb des Tableaus existieren zwei Pfade, die potenzielle Modelle von F repräsentieren. Der linke Pfad enthält mit A und $\neg A$ zwei Formeln, die kein gemeinsames Modell besitzen. Ein Pfad mit dieser Eigenschaft heißt *geschlossen* oder *widersprüchlich*. Der rechte Pfad enthält kein widersprüchliches Variablenpaar und lässt sich zur Konstruktion eines Modells I verwenden. Setzen wir $I(A) = 1$ und $I(B) = 1$, so sind alle Formeln des rechten Pfade und auch F selbst erfüllt. Ein Pfad mit dieser Eigenschaft heißt *offen* oder *widerspruchsfrei*.

Verallgemeinert gilt: Ist ein aussagenlogisches Tableau einer Formel F vorhanden, so können wir die Modelle von F an den offenen Paden ablesen. Kommt eine Variable auf dem Pfad positiv vor (A), so wird sie mit dem Wahrheitswert 1 belegt. Kommt sie dagegen negativ vor ($\neg A$), so wird ihr der Wahrheitswert 0 zugewiesen. Alle anderen Variablen dürfen beliebige Werte besitzen, so dass ein offener Pfad im Allgemeinen mehrere Modelle gleichzeitig repräsentiert.

■ Tableau

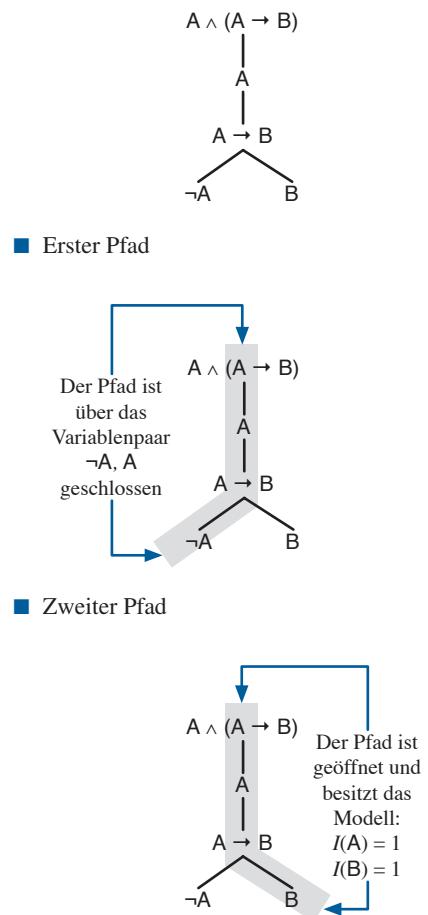


Abbildung 3.21: Aussagenlogisches Tableau für die Formel $A \wedge (A \rightarrow B)$

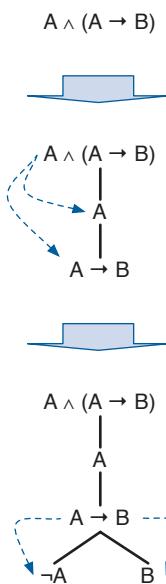


Abbildung 3.22: Schrittweise Konstruktion des aussagenlogischen Tableaus für die Formel $A \wedge (A \rightarrow B)$

Es bleibt zu klären, wie wir für eine aussagenlogische Formel F ein entsprechendes Tableau systematisch konstruieren können. Für unsere Beispiefunktion F lässt es sich ganz einfach über die folgende Überleitung herleiten:

- Im ersten Schritt wird ein Tableau erzeugt, das über einen einzigen, mit der Formel $A \wedge (A \rightarrow B)$ beschrifteten Knoten verfügt (vgl. Abbildung 3.22 oben).
- Auf der obersten Ebene besitzt F die Form einer Konjunktion, so dass alle Modelle von F auch Modelle der beiden Teilausdrücke A und $A \rightarrow B$ sein müssen. Damit können wir den initialen Pfad unseres Tableaus um die Formeln A und $A \rightarrow B$ erweitern, ohne die oben formulierte Modelleigenschaft zu verletzen: Für jedes Modell I von F existiert weiterhin ein Pfad, so dass I ein gemeinsames Modell für alle Formeln dieses Pfads ist (vgl. Abbildung 3.22 Mitte).
- Das expandierte Tableau enthält mit $A \rightarrow B$ eine neue Teilformel, die weiter zerlegt werden kann. Entsprechend der Semantik des Implikationsoperators ist $A \rightarrow B$ genau dann wahr, wenn $\neg A$ wahr oder B wahr ist. Die versteckte Disjunktion zwingt uns zu einer Fallunterscheidung, die innerhalb des Tableaus durch eine Verzweigung dargestellt wird (vgl. Abbildung 3.22 unten). Der linke Zweig entspricht dem Fall $I(A) = 0$ ($\neg A$ ist wahr), der rechte Zweig dem Fall $I(B) = 1$ (B ist wahr).

Die Beispielkonstruktion verdeutlicht die Grundprinzipien, nach denen Tableaus für beliebige Formeln erzeugt werden können. Ausgehend von der initialen Formel wird diese sukzessive in ihre Bestandteile zerlegt und das Tableau schrittweise expandiert. Ein Zweig, auf dem ein komplementäres Variablenpaar entsteht, ist geschlossen und bedarf keiner weiteren Bearbeitung. Offene Pfade werden dagegen so lange erweitert, bis sämtliche Teilformeln zerlegt wurden. Ein solcher Pfad heißt *vollständig*.

Bisher haben wir stets von geschlossenen, offenen und vollständigen *Pfaden* gesprochen. Die Begriffe lassen sich in intuitiver Weise auf vollständige Tableaus übertragen. Wir nennen ein Tableau *geschlossen*, wenn alle seine Pfade geschlossen sind. Folgerichtig ist ein Tableau *offen* bzw. *widerspruchsfrei*, wenn mindestens ein offener Pfad existiert. Ein Tableau heißt *vollständig*, wenn alle offenen Pfade vollständig sind.

Damit lässt sich die Beweisführung im Tableaukalkül wie folgt zusammenfassen: Um die Allgemeingültigkeit einer Formel F zu zeigen, konstruieren wir ein *vollständiges* Tableau für $\neg F$. Ist es geschlossen, so

besitzt $\neg F$ kein Modell und die Formel F ist als Tautologie identifiziert. Ist das Tableau offen, so existiert mindestens ein offener Pfad. Diesen können wir verwenden, um eine erfüllbare Belegung für $\neg F$ abzulesen. Die Variablenbelegung ist ein Gegenbeispiel, das die Allgemeingültigkeit von F widerlegt.

In Abbildung 3.23 sind die Konstruktionsregeln für alle aussagenlogischen Operatoren zusammengefasst. Da sie in direkter Weise die Operatorensemantik nachbilden, können sie mit denselben Überlegungen hergeleitet werden, die wir für die Konstruktion unseres Beispieltabeaus angestellt haben.

Betrachten wir die Regeln von einem abstrakteren Standpunkt, so lassen sich diese in zwei Gruppen einteilen. Die Vertreter der ersten Gruppe (α -Regeln) besitzen einen konjunktiven Charakter und bewirken die Verlängerung eines Pfads. Die Vertreter der zweiten Gruppe (β -Regeln) besitzen einen disjunktiven Charakter und führen zu einer Verzweigung. In Abhängigkeit des angewendeten Regeltyps sprechen wir folgerichtig von einer α - oder einer β -Expansion eines Tableaus.

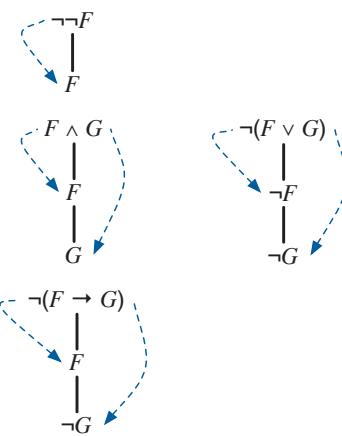
Plakativ ausgedrückt sind α -Expansionen gutmütiger als β -Expansionen, da sie die Anzahl der Pfade innerhalb des Tableaus nicht vergrößern. Welche Konsequenzen sich hieraus ergeben, machen die beiden in Abbildung 3.24 dargestellten Tableaus für die Formel

$$\neg((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$$

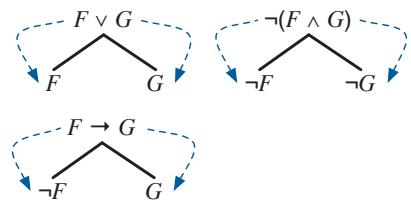
deutlich. Während das linke Tableau 6 Pfade besitzt, kommt die rechte Variante mit 4 Pfaden aus. Ein Blick auf die Reihenfolge der expandierten Formeln bringt den Grund für die unterschiedliche Größenentwicklung zum Vorschein. Im linken Tableau wurden zuerst die β -Expansionen und anschließend die α -Expansionen ausgeführt. Hierdurch entstehen frühzeitige Verzweigungen, die das Tableau merklich anwachsen lassen. Im rechten Tableau wurde stattdessen mit den α -Regeln begonnen. Hierdurch wird die Verzweigung verzögert und ein insgesamt kleineres Tableau erzeugt. Für die praktische Beweisführung ist es immer sinnvoll, die α -Regeln vor den β -Regeln anzuwenden.

In Abbildung 3.25 ist als weiteres Beispiel der formale Beweis des Dirichlet'schen Schubfachprinzips dargestellt. Das entstehende Tableau enthält insgesamt 12 Pfade, die allesamt geschlossen sind. Damit ist auch das Tableau selbst geschlossen und das Schubfachprinzip formal bewiesen. Im Gegensatz zum Resolutionsbeweis aus Abbildung 3.20 bedarf die ursprüngliche Formel keinerlei Vorverarbeitung.

■ α -Expansionen



■ β -Expansionen



■ Kombinierte α/β -Expansionen

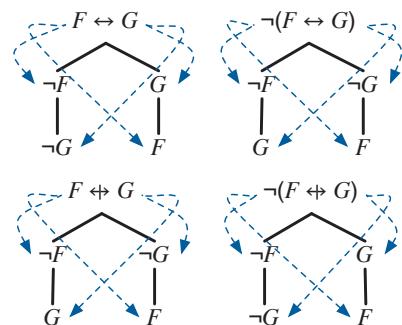


Abbildung 3.23: Expansionsregeln des aussagenlogischen Tableaukalküls

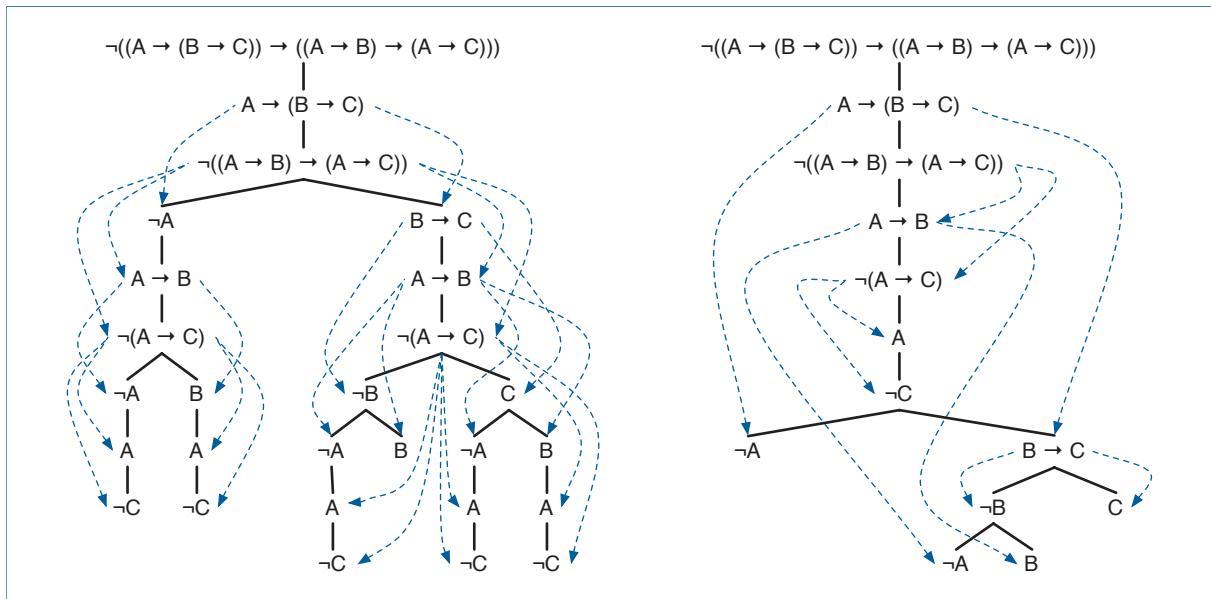


Abbildung 3.24: Die Reihenfolge, in der α - und β -Regeln angewendet werden, beeinflusst die Größe eines Tableaus.

3.1.4 Anwendung: Hardware-Entwurf

Weiter oben haben wir gleich an mehreren Stellen die Bedeutung der Aussagenlogik für den digitalen Schaltungsentwurf angesprochen. In diesem Abschnitt wollen wir den Zusammenhang zwischen aussagenlogischen Formeln auf der einen Seite und Hardware-Schaltungen auf der anderen Seite genauer beleuchten. Konkret werden wir zeigen, wie sich Hardware-Schaltungen mit den Mitteln der Aussagenlogik systematisch entwerfen lassen.

Als Beispiel betrachten wir die in Abbildung 3.26 dargestellte Addition zweier Binärzahlen. Wie die Beispielrechnung zeigt, werden Zahlen im Binärsystem nach dem gleichen Schema addiert, das wir aus dem vertrauten Dezimalsystem kennen. Wir müssen lediglich darauf achten, einen Übertrag zu generieren, sobald die Summe zweier Ziffern größer als 1 ist. Im unteren Teil von Abbildung 3.26 ist das allgemeine Additionschema dargestellt.

■ Beispiel: $92 + 106$

$$\begin{array}{r}
 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
 + & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
 \hline
 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
 \hline
 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0
 \end{array}$$

■ Allgemeines Additionsschema

$$\begin{array}{r}
 & A_n & \dots & A_3 & A_2 & A_1 & A_0 \\
 + & B_n & \dots & B_3 & B_2 & B_1 & B_0 \\
 & C_n & \dots & C_3 & C_2 & C_1 & \\
 \hline
 C_{n+1} & Z_n & \dots & Z_3 & Z_2 & Z_1 & Z_0
 \end{array}$$

Abbildung 3.26: Addition mehrstelliger Binärzahlen

Um die Addition mithilfe von Logikformeln abbilden zu können, codieren wir das i -te Bit der beiden Summanden mit den Variablen A_i und B_i . Die Variable Z_i beschreibt das i -te Summenbit und die Variable C_i den potenziellen Übertrag, der an der i -ten Bitstelle entsteht.

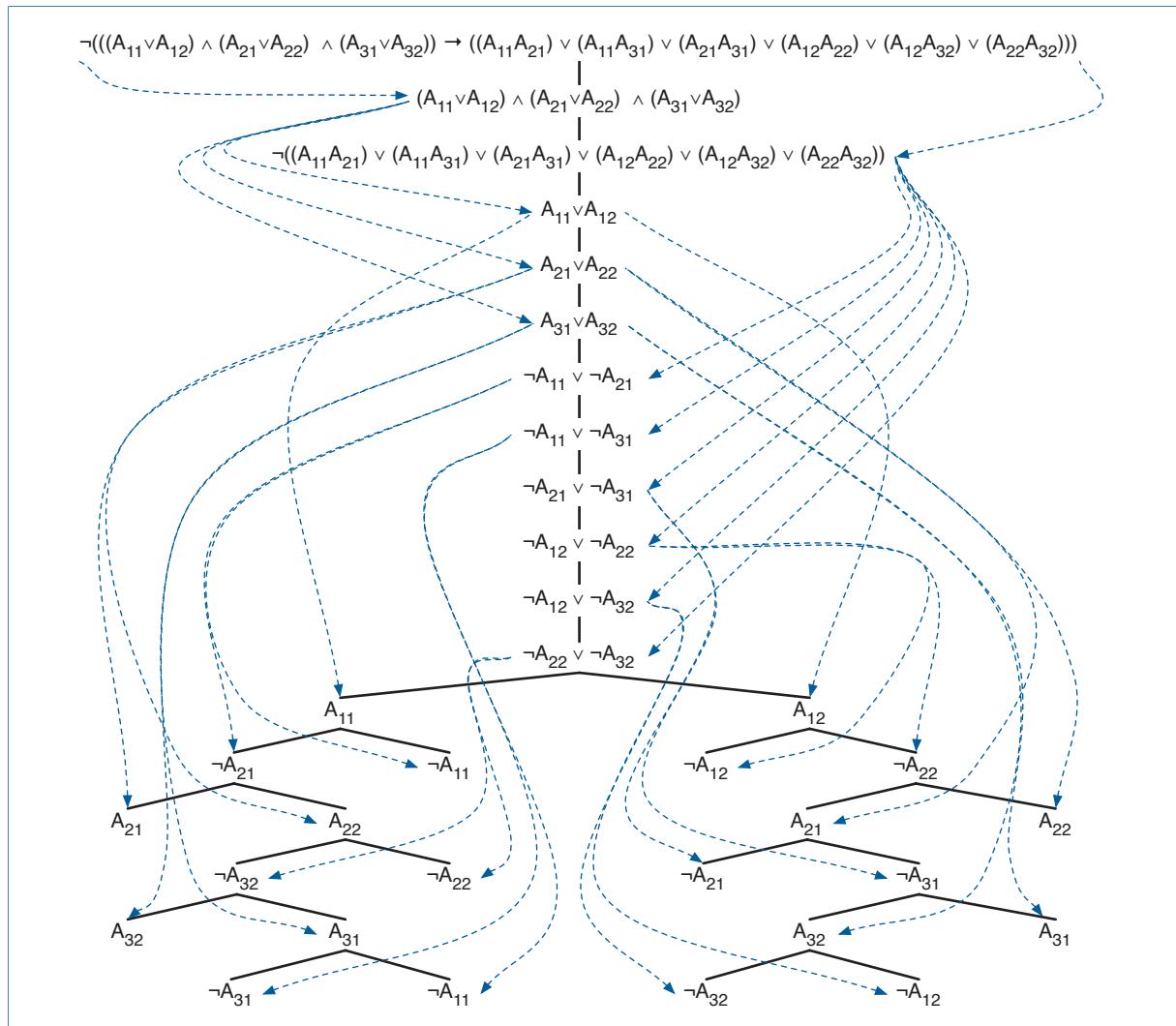


Abbildung 3.25: Formaler Beweis des Schubfachprinzips im aussagenlogischen Tableaukalkül

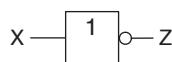
Aus dem Additionsschema können wir sofort die folgenden Beziehungen ableiten:

$$Z_i \equiv A_i \leftrightarrow B_i \leftrightarrow C_i \quad (3.5)$$

$$C_{i+1} \equiv A_i B_i \vee A_i C_i \vee B_i C_i \quad (3.6)$$

$$\equiv A_i B_i \vee C_i (A_i \leftrightarrow B_i) \quad (3.7)$$

■ NOT-Gatter



X	Z
0	1
1	0

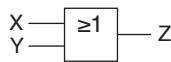
$$Z \equiv \neg X$$

■ AND-Gatter, OR-Gatter



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

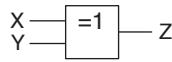
$$Z \equiv X \wedge Y$$



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

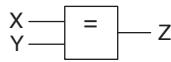
$$Z \equiv X \vee Y$$

■ XOR-Gatter, NOXOR-Gatter



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

$$Z \equiv X \leftrightarrow Y$$



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

$$Z \equiv X \leftrightarrow Y$$

Abbildung 3.27: Logikgatter

Der letzte Umformungsschritt ist nicht unmittelbar einsichtig und wird im Übungsteil auf Seite 151 bewiesen.

Um die weiter unten entwickelten Gleichungen übersichtlich zu halten, vereinbaren wir die folgenden Abkürzungen:

$$P_i := A_i \leftrightarrow B_i$$

$$G_i := A_i B_i$$

Jetzt lesen sich die Formeln (3.5) und (3.7) wie folgt:

$$Z_i \equiv P_i \leftrightarrow C_i \quad (3.8)$$

$$C_{i+1} \equiv G_i \vee C_i P_i \quad (3.9)$$

Aus den beiden Gleichungen können wir ohne Umwege die Hardware-Implementierung eines Addierwerks ableiten. Hierzu identifizieren wir jede Variable mit einem Leitungssegment und erzeugen für jeden aussagenlogischen Operator ein physikalisches Logikgatter. In Abbildung 3.27 sind die Symbole sowie die berechneten Funktionen der wichtigsten Gatter in einer Übersicht zusammengefasst.

Beschränken wir uns auf die Verarbeitung von Zahlen der Bitbreite 4, so erhalten wir die in Abbildung 3.28 dargestellte Hardware-Schaltung. Auf den ersten Blick mögen die durcheinanderlaufenden Leitungen für mehr Verwirrung als Klärung sorgen. Nehmen Sie sich deshalb ein wenig Zeit und versuchen Sie, die Signalwege von den Ausgängen (Z_0, \dots, Z_3, C_4) bis zu den Eingängen ($A_0, \dots, A_3, B_0, \dots, B_3$) zurückzuverfolgen. Sie werden erkennen, dass die Formeln (3.8) und (3.9) eins zu eins durch die Hardware-Schaltung nachgebildet werden.

Die erzeugte Schaltung reicht das Übertragsbit (*carry bit*) von rechts nach links durch die gesamte Schaltung hindurch und wird aufgrund dieser Besonderheit als *Carry-ripple-Addierer* bezeichnet. Für große Bitbreiten wird das Addierwerk durch dieses Verhalten deutlich verlangsamt, da das i -te Summenbit erst berechnet werden kann, wenn das vorangegangene Übertragsbit vorliegt. Dieses kann wiederum erst dann berechnet werden, wenn das seinerseits vorangegangene Übertragsbit vorliegt und so fort. Der Carry-ripple-Addierer ist damit ausschließlich für Anwendungen geeignet, in denen die Kompaktheit der Schaltung wichtiger ist als die Geschwindigkeit.

Glücklicherweise sind wir mithilfe der Aussagenlogik in der Lage, die Hardware-Schaltung zu optimieren. Hierzu rollen wir die rekursiv definierte Formel

$$C_{i+1} \equiv G_i \vee C_i P_i$$

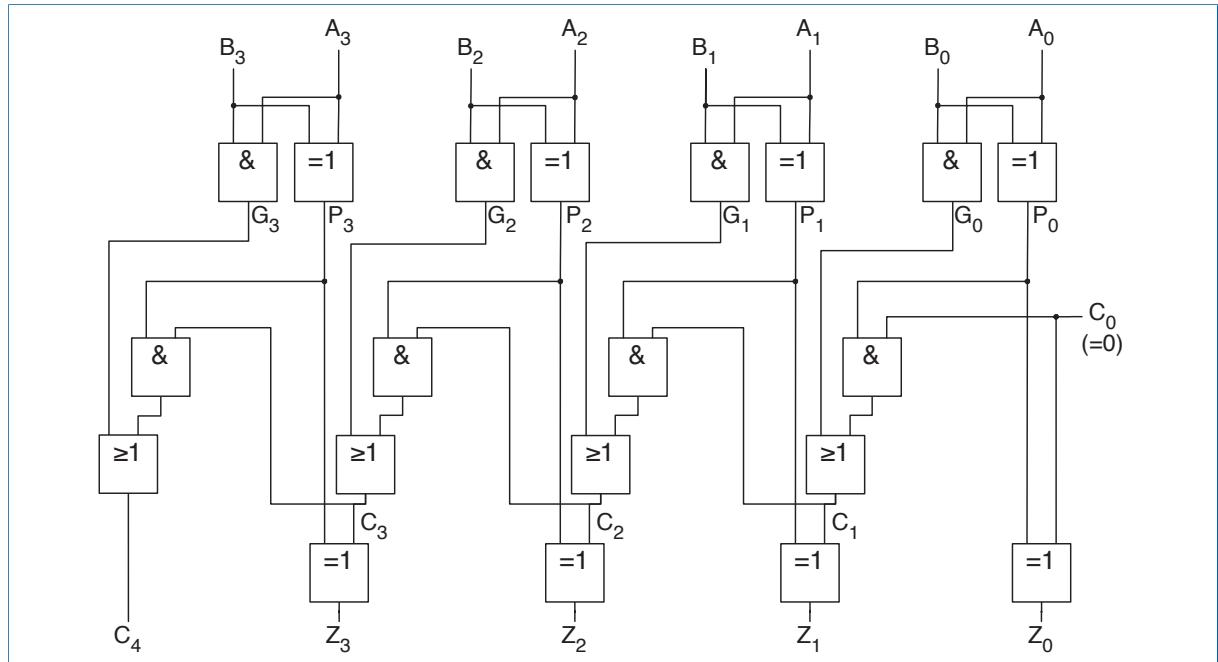


Abbildung 3.28: Vollständig aufgebauter 4-Bit-Carry-ripple-Addierer

so lange aus, bis sämtliche Vorkommen von C_i mit $i \neq 0$ aus der Formel verschwinden. Wir erhalten das folgende Ergebnis:

$$C_1 \equiv G_0 \vee C_0 P_0 \quad (3.10)$$

$$\begin{aligned} C_2 &\equiv G_1 \vee C_1 P_1 \\ &\equiv G_1 \vee (G_0 \vee C_0 P_0) P_1 \\ &\equiv G_1 \vee G_0 P_1 \vee C_0 P_0 P_1 \end{aligned} \quad (3.11)$$

$$\begin{aligned} C_3 &\equiv G_2 \vee C_2 P_2 \\ &\equiv G_2 \vee (G_1 \vee G_0 P_1 \vee C_0 P_0 P_1) P_2 \\ &\equiv G_2 \vee G_1 P_2 \vee G_0 P_1 P_2 \vee C_0 P_0 P_1 P_2 \end{aligned} \quad (3.12)$$

$$\begin{aligned} C_4 &\equiv G_3 \vee C_3 P_3 \\ &\equiv G_3 \vee (G_2 \vee G_1 P_2 \vee G_0 P_1 P_2 \vee C_0 P_0 P_1 P_2) P_3 \\ &\equiv G_3 \vee G_2 P_3 \vee G_1 P_2 P_3 \vee G_0 P_1 P_2 P_3 \vee C_0 P_0 P_1 P_2 P_3 \end{aligned} \quad (3.13)$$

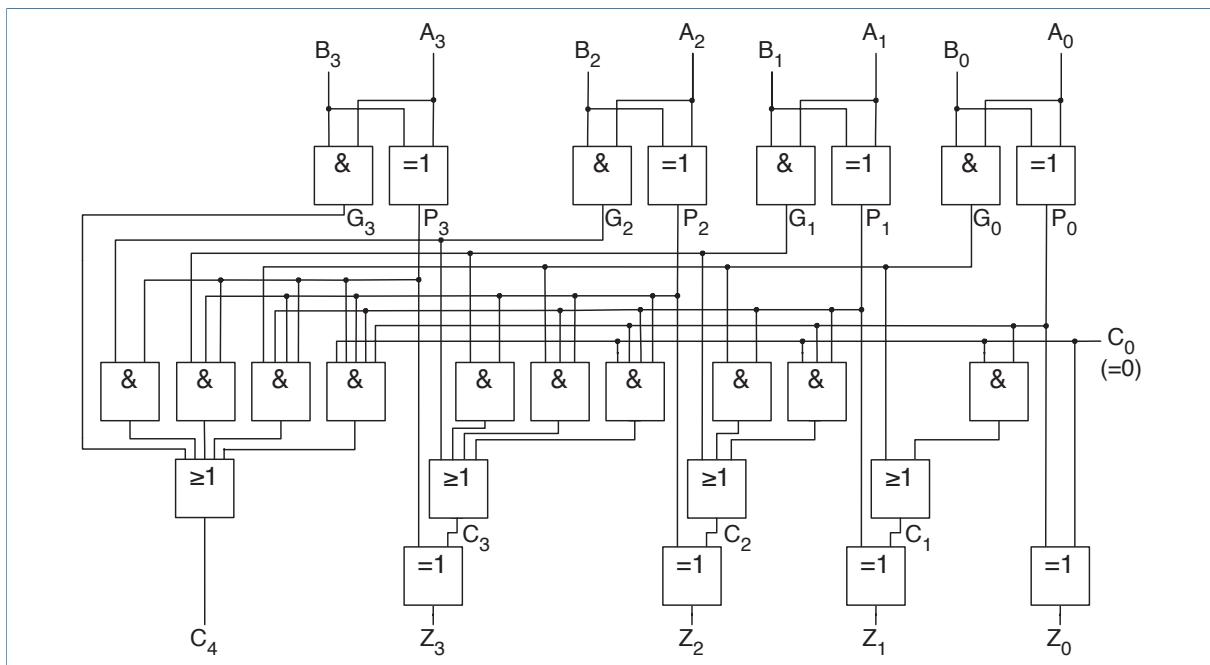


Abbildung 3.29: Vollständig aufgebauter 4-Bit-Carry-look-ahead-Addierer

Setzen wir die Formeln (3.10) bis (3.13) in eine Hardware-Schaltung um, so entsteht das in Abbildung 3.29 dargestellte Schaltnetz. Durch die getätigten Umformungen ist die *Tiefe* der Schaltung jetzt konstant, d.h., unabhängig von der Bitbreite durchläuft ein Signal von den Eingängen zu den Ausgängen eine konstante Anzahl an Logikgattern. Im Gegensatz zum Carry-ripple-Addierer werden die Übertragsbits jetzt direkt aus den Eingabebits erzeugt, d.h., sie werden vorausschauend berechnet. Dieser Eigenschaft verdankt die Schaltung ihren Namen; sie wird in der Literatur als *Carry-look-ahead-Addierer* bezeichnet.

Vergleichen wir die Strukturbilder beider Addierer, so wird der Nachteil der geschwindigkeitsoptimierten Variante unmittelbar deutlich. Die Anzahl der Gatter, die für die Vorausberechnung eines Übertragsbits benötigt wird, vergrößert sich mit jeder Bitstelle. Hierdurch steigt der Flächenbedarf der Schaltung stark an. Wie stark die Größenzunahme wirklich ausfällt, werden wir im Detail klären, sobald in Kapitel 7 die formalen Grundlagen der Komplexitätstheorie gelegt wurden. Dort kommen wir im Übungsteil auf Seite 388 auf die spezielle Struktur des Carry-look-ahead-Addierers zurück.

3.2 Prädikatenlogik

In den vorhergehenden Abschnitten haben Sie gelernt, wie sich elementare Aussagen durch aussagenlogische Variablen repräsentieren lassen und diese zu komplexen Gebilden verknüpft werden können. Auch wenn sich viele Sachverhalte auf diese Weise beschreiben lassen, sind die Ausdrucksmöglichkeiten bei weitem nicht stark genug, um alle gängigen Aspekte des logischen Schließens abzubilden. Die Limitierungen, mit denen wir an dieser Stelle zu kämpfen haben, werden durch das folgende, in der Literatur häufig bemühte Beispiel deutlich:

1. Sokrates ist ein Mensch
2. Alle Menschen sind sterblich
3. Dann ist auch Sokrates sterblich

Die Eigenschaft, ein Mensch zu sein, ist eine parametrisierte Aussage der Form $\text{Mensch}(x)$, die in Abhängigkeit des Arguments wahr oder falsch wird. Setzen wir für x das Individuum *Sokrates* ein, so ist die Aussage wahr, für andere Interpretationen von x ist sie sicherlich falsch. Eine atomare Aussage, deren Wahrheitswert durch ein oder mehrere eingesetzte Individuen bestimmt wird, bezeichnen wir als *Prädikat*.

Die zweite Aussage macht eine quantitative Aussage über Individuen. Sie besagt, dass *alle* Menschen sterblich sind. Im Jargon der Logiker lässt sich die Aussage wie folgt ausdrücken: Für alle x gilt: Ist $\text{Mensch}(x)$ wahr, dann ist auch $\text{Sterblich}(x)$ wahr.

Erweitern wir die Aussagenlogik um mehrstellige Prädikate sowie um die Möglichkeit, quantifizierende Aussagen zu formulieren, so gelangen wir auf direktem Wege zur *Prädikatenlogik erster Stufe*, kurz PL1. In ihr lässt sich das diskutierte Beispiel wie folgt formalisieren:

1. $\text{Mensch}(\text{Sokrates})$
2. $\forall x (\text{Mensch}(x) \rightarrow \text{Sterblich}(x))$
3. $\models \text{Sterblich}(\text{Sokrates})$

Genau wie im Falle der Aussagenlogik werden wir Beweissysteme entwickeln, mit denen sich die Allgemeingültigkeit der soeben formulierten Aussage beweisen lässt. Um die Details solcher Kalküle richtig verstehen zu können, sind aber noch einige Vorarbeiten zu leisten. Zunächst wollen wir formal klären, was wir unter einem prädikatenlogischen Ausdruck genau zu verstehen haben.



Achten Sie darauf, aussagenlogische Variablen und prädikatenlogische Variablen nicht zu verwechseln! In der Prädikatenlogik ist eine Variable ein Platzhalter für ein beliebiges Element einer festgelegten Grundmenge. Beispielsweise kann die Variable x in der Formel $\text{Mensch}(x)$ stellvertretend für das Individuum *Sokrates* oder ein anderes Element der Grundmenge stehen. Erst die konkrete Wahl der Variablen x macht die Formel $\text{Mensch}(x)$ zu einer wahren oder einer falschen Aussage. In der Aussagenlogik stehen Variablen dagegen für atomare Aussagen, die wahr oder falsch sein können. Damit sind sie in Wirklichkeit 0-stellige Prädikate und haben mit den prädikatenlogischen Variablen nur den Namen gemeinsam.

■ Signatur Σ

$$\Sigma = (V_\Sigma, F_\Sigma, P_\Sigma) \text{ mit}$$

$$\begin{aligned} V_\Sigma &= \{x, y\} \\ F_\Sigma &= \{f \text{ (2-stellig)}\} \\ P_\Sigma &= \{P \text{ (2-stellig)}\} \end{aligned}$$

■ Terme über Σ

$$\begin{aligned} &x \\ &y \\ &f(x, x) \\ &f(x, y) \\ &f(f(x, y), x) \\ &f(x, f(x, y)) \\ &f(f(x, x), f(x, y)) \\ &\dots \end{aligned}$$

■ Atomare Formeln über Σ

$$\begin{aligned} &P(x, x) \\ &P(x, y) \\ &P(f(x, y), x) \\ &P(x, f(x, y)) \\ &P(x, f(f(x, y), x)) \\ &P(f(f(x, y), x), y) \\ &\dots \end{aligned}$$

■ Formeln über Σ

$$\begin{aligned} &\forall x P(x, x) \\ &\exists x P(x, x) \\ &P(f(x, x), x) \leftrightarrow P(y, y) \\ &\forall y \exists x (P(f(x, x), x) \leftrightarrow P(y, y)) \\ &\exists y \forall x (P(f(x, x), x) \leftrightarrow P(y, y)) \\ &\dots \end{aligned}$$

3.2.1 Syntax und Semantik

Die Syntaxdefinition der Prädikatenlogik erfolgt in drei Schritten. Zunächst führen wir den Begriff der *prädikatenlogischen Signatur* ein. Darauf aufbauend definieren wir den Begriff des *prädikatenlogischen Terms* und erweitern diesen anschließend zum Begriff der *prädikatenlogischen Formel*.



Definition 3.13 (Prädikatenlogische Signatur)

Eine prädikatenlogische Signatur Σ ist ein Tripel $(V_\Sigma, F_\Sigma, P_\Sigma)$. Sie besteht aus

- einer Menge $V_\Sigma = \{x_1, x_2, \dots\}$ von *Variablen*,
- einer Menge $F_\Sigma = \{f_1, f_2, \dots\}$ von *Funktionssymbolen*,
- einer Menge $P_\Sigma = \{P_1, P_2, \dots\}$ von *Prädikaten*.

Jede Funktion und jedes Prädikat besitzt eine feste Stelligkeit ≥ 0 .

Grob gesprochen definiert eine prädikatenlogische Signatur den Vorrat an elementaren Symbolen, aus denen eine Formel aufgebaut ist. Genau wie in der Aussagenlogik werden wir auch hier den Symbolvorrat von Fall zu Fall anpassen und z. B. x, y, z für x_1, x_2, x_3 , f, g, h für f_1, f_2, f_3 und P, Q, R für P_1, P_2, P_3 schreiben.



Definition 3.14 (Prädikatenlogischer Term)

Sei $\Sigma = (V_\Sigma, F_\Sigma, P_\Sigma)$ eine prädikatenlogische Signatur. Die Menge der *prädikatenlogischen Terme* ist induktiv definiert:

- Jede Variable aus der Menge V_Σ ist ein Term.
- Jedes 0-stellige Funktionssymbol aus der Menge F_Σ ist ein Term.
- Sind t_1, \dots, t_n Terme und ist $f \in F_\Sigma$ ein n -stelliges Funktionssymbol, so ist $f(t_1, \dots, t_n)$ ein Term.

Abbildung 3.30: Schrittweise Konstruktion von prädikatenlogischer Ausdrücke

Wie in Abbildung 3.30 (oben) demonstriert, lassen sich aus dem Symbolvorrat einer prädikatenlogischen Signatur im Allgemeinen unendlich viele Terme erzeugen. Eine besondere Bedeutung fällt dabei den 0-stelligen Funktionssymbolen zu. Diese besitzen keine Parameter und

spielen die Rolle von *Konstanten*. In unserem Eingangsbeispiel haben wir mit der Konstanten *Sokrates* bereits ein solches Funktionssymbol verwendet.

Mit den eingeführten Begriffen sind wir in der Lage, die Menge der *prädikatenlogischen Formeln* präzise zu definieren:

Definition 3.15 (Syntax der Prädikatenlogik)

Sei Σ eine prädikatenlogische Signatur. Die Menge der *atomaren prädikatenlogischen Formeln* definieren wir wie folgt:

- Sind t_1, \dots, t_n Terme und P ein n -stelliges Prädikat, so ist $P(t_1, \dots, t_n)$ eine atomare Formel.

Die *prädikatenlogischen Formeln* sind induktiv definiert:

- Jede atomare Formel ist eine Formel.
- Ist $x \in V_\Sigma$ und sind F und G Formeln, dann sind es auch $0, 1, (\neg F), (F \wedge G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G), (F \Leftrightarrow G), \forall x F, \exists x F$.

Abbildung 3.30 (unten) zeigt eine kleine Auswahl erzeugbarer Formeln. Beachten Sie, dass nicht alle Variablen zwangsläufig im Wirkungsbereich eines Quantors stehen müssen. So kommt die Variable x in der Formel $P(x, x)$ *frei* bzw. *ungebunden*, in der Formel $\forall x P(x)$ dagegen *gebunden* vor. Das Beispiel in Abbildung 3.31 demonstriert, dass eine Variable in der gleichen Formel sowohl frei als auch gebunden vorkommen kann. Im Folgenden werden wir fast ausschließlich Formeln betrachten, in denen alle Variablenvorkommen durch Quantoren gebunden sind. Solche Formeln heißen *geschlossen*.

Der Umgang mit prädikatenlogischen Ausdrücken wird erheblich erleichtert, wenn die Variablen in zwei voneinander unabhängigen Teilausdrücken unterschiedlich benannt werden. Sind die quantifizierten Variablen einer *geschlossenen Formel* F paarweise verschieden, so sprechen wir von einer *bereinigten Formel*. Abbildung 3.32 zeigt, wie eine Formel F durch die Umbenennung mehrfach verwendeter Variablen in eine bereinigte Form gebracht werden kann. Da die umbenannten Variablen durch einen Quantor gebunden sind, sprechen wir auch von einer *gebundenen Ersetzung*.

Werden in einer Formel F *freie* Vorkommen einer Variablen x ersetzt, so sprechen wir von einer *Substitution* und schreiben $F[x \leftarrow t]$. Beachten

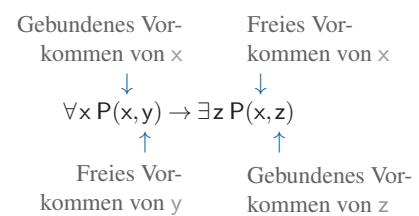


Abbildung 3.31: Steht eine Variable im Wirkungsbereich eines Quantors, so sprechen wir von einem *gebundenen* Vorkommen, ansonsten von einem *freien* Vorkommen. Formeln, in denen alle Variablenvorkommen gebunden sind, heißen *geschlossen*.

■ Beispiel 1

$$\exists y \forall x P(x, y) \rightarrow \forall x \exists y P(x, y)$$



$$\exists y \forall x P(x, y) \rightarrow \forall v \exists w P(v, w)$$

■ Beispiel 2

$$\exists x (P(x) \wedge \forall x Q(x))$$



$$\exists x (P(x) \wedge \forall y Q(y))$$

Abbildung 3.32: Eine geschlossene Formel wird *bereinigt*, indem mehrfach quantifizierte Variablen umbenannt werden. In der bereinigten Formel sind die quantifizierten Variablen paarweise verschieden.

■ Grundsubstitutionen

$$\sigma_1 = [x \leftarrow a]$$

$$\sigma_2 = [y \leftarrow b]$$

$$\sigma_3 = [x \leftarrow a, y \leftarrow b]$$

■ Substitutionen

$$\sigma_4 = [x \leftarrow y]$$

$$\sigma_5 = [y \leftarrow f(y)]$$

$$\sigma_6 = [x \leftarrow y, y \leftarrow f(y)]$$

■ Beispiele (ohne Quantoren)

$$\sigma_1 P(x, y) = P(a, y)$$

$$\sigma_2 P(x, y) = P(x, b)$$

$$\sigma_3 P(x, y) = P(a, b)$$

$$\sigma_4 P(x, y) = P(y, y)$$

$$\sigma_5 P(x, y) = P(x, f(y))$$

$$\sigma_6 P(x, y) = P(y, f(y))$$

■ Beispiele (mit Quantoren)

$$\sigma_6 (\forall x P(x, y)) = \forall x P(x, f(y))$$

$$\sigma_6 (\forall y P(x, y)) = \forall y P(y, y))$$

$$\sigma_6 (\forall x \forall y P(x, y)) = \forall x \forall y P(x, y)$$

Sie, dass eine Substitution keine gebundenen Variablen ersetzt; enthält F also Teilterme der Form $\exists x F'$ oder $\forall x F'$, so bleiben diese unangetastet (vgl. Abbildung 3.33). Enthalten die eingesetzten Terme selbst keine Variablen, so sprechen wir von einer *Grundsubstitution*. Der Substitutionsbegriff spielt in der Prädikatenlogik eine prominente Rolle und wird uns weiter unten erneut begegnen, wenn wir die verschiedenen Kalküle der Prädikatenlogik besprechen.

Wir wollen nun daran gehen, die Semantik der Prädikatenlogik zu definieren. Genau wie in der Aussagenlogik legen wir die Semantik über eine *Modellrelation* \models fest. Um diese zu definieren, müssen wir zunächst den Begriff der Interpretation auf prädikatenlogische Formeln erweitern:



Definition 3.16 (Prädikatenlogische Interpretation)

Sei $\Sigma = (V_\Sigma, F_\Sigma, P_\Sigma)$ eine prädikatenlogische Signatur. Eine *Interpretation* (U, I) über Σ besteht aus einer beliebigen nichtleeren Menge U und einer Abbildung I , die

- jedem n -stetlichen Funktionssymbol $f \in F_\Sigma$ eine Funktion

$$I(f) : U^n \rightarrow U \quad \text{und}$$

- jedem n -stetlichen Prädikatsymbol $P \in P_\Sigma$ eine Relation

$$I(P) \subseteq U^n \quad \text{zuordnet.}$$

Abbildung 3.33: Eine Variablensubstitution der Form $[x \leftarrow t]$ ersetzt alle freien Vorkommen von x durch t . Alle gebundenen Vorkommen bleiben dagegen unangetastet.

Die Menge U wird in der Literatur als *Universum*, *Individuenbereich* oder *Grundmenge* bezeichnet. Beachten Sie, dass die getätigte Festlegung auch 0-stetige Funktions- und Prädikatsymbole mit einschließt. Einem 0-stetigen Funktionssymbol wird dann formal eine Funktion $U^0 \rightarrow U$ zugeordnet, hinter der sich ein einzelnes Element aus dem Individuenbereich und damit eine Konstante verbirgt. 0-stetige Prädikatsymbole werden einer Relation U^0 zugeordnet. Sie sind atomare Aussagen, die entweder wahr oder falsch sein können, und entsprechen damit eins zu eins den altbekannten aussagenlogischen Variablen.

Die Abbildung I , die jedem Funktionssymbol f eine Funktion $I(f)$ zuordnet, lässt sich in naheliegender Weise auf variablenfreie Terme übertragen. Hierzu erweitern wir I nach dem folgenden induktiven Schema:

$$I(f(t_1, \dots, t_n)) := I(f)(I(t_1), \dots, I(t_n))$$

Abbildung 3.34 demonstriert den Interpretationsbegriff anhand zweier Beispiele. Beide assoziieren das Funktionszeichen f mit der gewöhnlichen Addition und das Prädikatsymbol P mit der Nullmenge, d. h., $P(x)$ ist genau dann wahr, wenn x als die Null interpretiert wird. Unterschiedlich gewählt sind die zugrunde liegenden Individuenmengen. Die erste Interpretation schöpft aus dem Bereich der ganzen Zahlen, während die zweite Interpretation nur natürliche Zahlen in Betracht zieht. Unter diesen Voraussetzungen liest sich die Beispielformel $\forall x \exists y P(f(x,y))$ so: „Für alle x existiert ein y mit $x + y = 0$.“

Für die Menge der ganzen Zahlen ist die Aussage offensichtlich erfüllt, für die Teilmenge der nichtnegativen Zahlen dagegen nicht. Jetzt ist es an der Zeit, die informellen Überlegungen zu präzisieren und die Modellrelation \models formal einzuführen.



Definition 3.17 (Semantik der Prädikatenlogik)

F und G seien geschlossene prädikatenlogische Formeln und (U, I) eine Interpretation. Die Semantik der Prädikatenlogik ist durch die *Modellrelation* \models gegeben, die induktiv über dem Formelaufbau definiert ist:

$$(U, I) \models 1$$

$$(U, I) \not\models 0$$

$$(U, I) \models P(t_1, \dots, t_n) :\Leftrightarrow (I(t_1), \dots, I(t_n)) \in I(P)$$

$$(U, I) \models (\neg F) :\Leftrightarrow (U, I) \not\models F$$

$$(U, I) \models (F \wedge G) :\Leftrightarrow (U, I) \models F \text{ und } (U, I) \models G$$

$$(U, I) \models (F \vee G) :\Leftrightarrow (U, I) \models F \text{ oder } (U, I) \models G$$

$$(U, I) \models (F \rightarrow G) :\Leftrightarrow (U, I) \not\models F \text{ oder } (U, I) \models G$$

$$(U, I) \models (F \leftrightarrow G) :\Leftrightarrow (U, I) \models F \text{ genau dann, wenn } (U, I) \models G$$

$$(U, I) \models (F \Leftrightarrow G) :\Leftrightarrow (U, I) \not\models (F \leftrightarrow G)$$

$$(U, I) \models \forall x F :\Leftrightarrow \text{Für alle } u \in U \text{ ist } (U, I_{[x/u]}) \models F$$

$$(U, I) \models \exists x F :\Leftrightarrow \text{Es gibt ein } u \in U \text{ mit } (U, I_{[x/u]}) \models F$$

Eine Interpretation (U, I) mit $(U, I) \models F$ heißt *Modell* für F .

In dieser Definition wird erstmals der Ausdruck $I_{[x/u]}$ verwendet. Ist (U, I) eine prädikatenlogische Interpretation, so ist mit $(U, I_{[x/u]})$ jene Interpretation gemeint, die x wie eine Konstante behandelt, der das Individuumelement u zugeordnet ist, und sonst mit I übereinstimmt.

■ Formel

$$F := \forall x \exists y P(f(x,y))$$

■ Signatur

$$V_\Sigma = \{x, y\}$$

$$F_\Sigma = \{f\}$$

$$P_\Sigma = \{P\}$$

■ Erste Interpretation

$$U := \mathbb{Z}$$

$$I(f) := (x, y) \mapsto x + y$$

$$I(P) := \{0\}$$

■ Zweite Interpretation

$$U := \mathbb{N}$$

$$I(f) := (x, y) \mapsto x + y$$

$$I(P) := \{0\}$$

Abbildung 3.34: Zwei Interpretationen für die Formel $\forall x \exists y P(f(x,y))$

■ Elimination von \leftrightarrow und \Leftrightarrow

$$(F \leftrightarrow G) \equiv FG \vee \neg F \neg G$$

$$(F \Leftrightarrow G) \equiv F \neg G \vee \neg FG$$

■ Negationsnormalform

$$\begin{aligned}\neg\neg F &\equiv F \\ \neg(F \wedge G) &\equiv \neg F \vee \neg G \\ \neg(F \vee G) &\equiv \neg F \wedge \neg G \\ \neg(F \rightarrow G) &\equiv F \wedge \neg G \\ \neg(F \leftrightarrow G) &\equiv F \neg G \vee \neg FG \\ \neg(F \Leftrightarrow G) &\equiv FG \vee \neg F \neg G \\ \neg\forall x F &\equiv \exists x \neg F \\ \neg\exists x F &\equiv \forall x \neg F\end{aligned}$$

■ Pränex-Form

$$\begin{aligned}F \wedge (\exists x G) &\equiv \exists x (F \wedge G), \quad x \notin F \\ (\exists x F) \wedge G &\equiv \exists x (F \wedge G), \quad x \notin G \\ F \wedge (\forall x G) &\equiv \forall x (F \wedge G), \quad x \notin F \\ (\forall x F) \wedge G &\equiv \forall x (F \wedge G), \quad x \notin G \\ F \vee (\exists x G) &\equiv \exists x (F \vee G), \quad x \notin F \\ (\exists x F) \vee G &\equiv \exists x (F \vee G), \quad x \notin G \\ F \vee (\forall x G) &\equiv \forall x (F \vee G), \quad x \notin F \\ (\forall x F) \vee G &\equiv \forall x (F \vee G), \quad x \notin G \\ F \rightarrow (\exists x G) &\equiv \exists x (F \rightarrow G), \quad x \notin F \\ (\exists x F) \rightarrow G &\equiv \forall x (F \rightarrow G), \quad x \notin G \\ F \rightarrow (\forall x G) &\equiv \forall x (F \rightarrow G), \quad x \notin F \\ (\forall x F) \rightarrow G &\equiv \exists x (F \rightarrow G), \quad x \notin G\end{aligned}$$

Abbildung 3.35: Umformungsregeln für die Erzeugung der Negationsnormalform (oben) und der Pränex-Form (unten)

Genau wie im Falle der Aussagenlogik bezeichnen wir eine Formel F in der Prädikatenlogik als *erfüllbar*, wenn sie (mindestens) ein Modell besitzt. Ist jedes Modell einer Formelmenge M auch Modell einer Formel F , so schreiben wir $M \models F$ („Aus M folgt F “). Ist ausnahmslos jede Interpretation ein Modell von F , gilt also $\emptyset \models F$, so ist F *allgemeingültig*. Wie gewohnt bezeichnen wir F in diesem Fall als *Tautologie* und verwenden die gekürzte Schreibweise $\models F$.

3.2.2 Normalformen

Weiter oben haben wir herausgearbeitet, dass bestimmte Beweisverfahren der Aussagenlogik eine Vorverarbeitung der zu prüfenden Formeln benötigen. Das Gleiche gilt in der Prädikatenlogik. In diesem Abschnitt werden wir die wichtigsten Darstellungsformen für prädikatenlogische Formeln einführen und damit eine notwendige Vorbereitung für die in Abschnitt 3.2.3 diskutierten Beweisverfahren leisten. Wir beginnen mit der einfachsten Normalform für prädikatenlogische Ausdrücke:



Definition 3.18 (Negationsnormalform)

Eine Formel F liegt in *Negationsnormalform* vor, wenn alle Negationszeichen vor atomaren Teilformeln stehen.

Die Negationsnormalform einer Formel F ist vergleichsweise einfach zu erzeugen. Die Grundlage hierfür bilden die in Abbildung 3.35 (oben) dargestellten Umformungsregeln.

Als Nächstes betrachten wir die *Pränex-Form*. Diese verfolgt die Idee, Quantoren aus tiefer liegenden Teiltermen in die oberste Formelebene zu verschieben.



Definition 3.19 (Pränex-Form)

Seien $Q_i \in \{\forall, \exists\}$ prädikatenlogische Quantoren. Eine Formel F liegt in *Pränex-Form* vor, wenn sie die Form $Q_1 x_1 Q_2 x_2 \dots Q_n x_n F^*$ besitzt und die Teilformel F^* keine weiteren Quantoren mehr enthält. F^* heißt die *Matrix* von F .

Die Pränex-Form lässt sich aus der Negationsnormalform erzeugen. Im ersten Schritt wird die Formel durch die Umbenennung mehrfach verwendeter Variablen vereinfacht. Anschließend wird der Wirkungsbereich

der Quantoren durch die Anwendung elementarer Umformungsregeln schrittweise vergrößert (vgl. Abbildung 3.35 unten). Ist keine Umformungsregel mehr anwendbar, so stehen die Quantoren bereits alle auf der linken Seite und die (bereinigte) Pränex-Form ist hergestellt.

Im nächsten Schritt wollen wir versuchen, die Existenzquantoren aus einer Formel zu eliminieren. Wir beginnen mit dem einfachsten Fall und betrachten eine Formel der Bauart $\exists x F$. Informell ausgedrückt ist die Formel genau dann wahr, wenn eine Belegung für x existiert, die F wahr werden lässt. Der Wert von x ist nicht davon abhängig, wie wir die anderen in F vorkommenden Variablen belegen. Daher können wir x ruhigen Gewissens durch ein neues Konstantensymbol $u \notin F$ ersetzen und erhalten mit $F[x \leftarrow u]$ eine Formel, die genau dann erfüllbar ist, wenn die ursprüngliche Formel $\exists x F$ erfüllbar ist. Mit diesem Trick ist es uns gelungen, die ursprüngliche Formel in eine *erfüllbarkeitsäquivalente* Formel zu übersetzen, die keinen Existenzquantor mehr besitzt.

In einem anderen Licht stellt sich die Situation für Formeln der Bauart $\forall x_1 \dots \forall x_n \exists x F$ dar. Informell ist eine solche Formel genau dann wahr, wenn für alle Wertekombinationen von x_1, \dots, x_n eine Belegung für x existiert, die F wahr werden lässt. Anders als im ersten Beispiel können wir x nicht einfach durch ein neues Konstantensymbol ersetzen, da die Belegung für x von der konkreten Belegung von x_1, \dots, x_n abhängt. Wenn aber für jede Wertekombination von x_1, \dots, x_n eine Belegung für x existiert, dann lässt sich der Wert von x durch eine n -stellige Funktion g berechnen. Folgerichtig können wir eine erfüllbarkeitsäquivalente Formel erzeugen, indem wir den Existenzquantor eliminieren und die Vorkommen von x durch den prädikatenlogischen Term $g(x_1, \dots, x_n)$ mit $g \notin F$ ersetzen. Die Stelligkeit des neuen Funktionssymbols g entspricht der Anzahl der Allquantoren, die links des eliminierten Existenzquantors stehen.

Damit haben wir alle Bausteine zusammengestellt, um sämtliche Existenzquantoren einer Formel F zu eliminieren. Über die Negationsnormalform erzeugen wir die bereinigte Pränex-Form und eliminieren danach sämtliche Existenzquantoren nach dem gezeigten Schema. Die entstehende Formel heißt die *Skolem-Form* von F .



Definition 3.20 (Skolem-Form)

Eine prädikatenlogische Formel F liegt in *Skolem-Form* vor, wenn sie die Form $\forall x_1 \forall x_2 \dots \forall x_n F^*$ besitzt und die Teilformel F^* keine weiteren Quantoren enthält.

$$\neg(\exists x \forall y P(f(x), y) \vee \exists x \forall y \neg Q(y, x))$$

(Ausgangsformel)



$$\neg(\exists x \forall y P(f(x), y) \vee \exists x \forall y \neg Q(y, x))$$

$$\equiv \neg \exists x \forall y P(f(x), y) \wedge \neg \exists x \forall y \neg Q(y, x)$$

$$\equiv \forall x \neg \forall y P(f(x), y) \wedge \forall x \neg \forall y \neg Q(y, x)$$

$$\equiv \forall x \exists y \neg P(f(x), y) \wedge \forall x \exists y \neg \neg Q(y, x)$$

$$\equiv \forall x \exists y \neg P(f(x), y) \wedge \forall x \exists y Q(y, x)$$

(Negationsnormalform)



$$\forall x \exists y \neg P(f(x), y) \wedge \forall x \exists y Q(y, x)$$

$$\equiv \forall x \exists y \neg P(f(x), y) \wedge \forall z \exists w Q(w, z)$$

(Bereinigte Negationsnormalform)



$$\forall x \exists y \neg P(f(x), y) \wedge \forall z \exists w Q(w, z)$$

$$\equiv \forall x \exists y \forall z \exists w (\neg P(f(x), y) \wedge Q(w, z))$$

(Pränex-Form)



$$\forall x \exists y \forall z \exists w (\neg P(f(x), y) \wedge Q(w, z))$$

$$\equiv_E \forall x \forall z \exists w (\neg P(f(x), g(x)) \wedge Q(w, z))$$

$$\equiv_E \forall x \forall z (\neg P(f(x), g(x)) \wedge Q(h(x, z), z))$$

(Skolem-Form)

Abbildung 3.36: Schrittweise Erzeugung der Skolem-Normalform

$$\forall x \exists y \neg P(f(x), y) \wedge (\forall z \exists w Q(w, z))$$

(Bereinigte Negationsnormalform)



$$(\forall x \exists y \neg P(f(x), y)) \wedge (\forall z \exists w Q(w, z))$$

$$\equiv \forall z \exists w ((\forall x \exists y \neg P(f(x), y)) \wedge Q(w, z))$$

$$\equiv \forall z \exists w \forall x \exists y (\neg P(f(x), y) \wedge Q(w, z))$$

(Pränex-Form)



$$\forall z \exists w \forall x \exists y (\neg P(f(x), y) \wedge Q(w, z))$$

$$\equiv_E \forall z \forall x \exists y (\neg P(f(x), y) \wedge Q(h(z), z))$$

$$\equiv_E \forall z \forall x (\neg P(f(x), g(x, z)) \wedge Q(h(z), z))$$

(Skolem-Form)

Abbildung 3.37: Alternative Umformung.
Im Allgemeinen sind die Pränex- und die Skolem-Form nicht eindeutig.

Abbildung 3.36 demonstriert die vollständige Umformungskette anhand eines konkreten Beispiels. Beachten Sie, dass die erzeugte Pränex-Form immer noch äquivalent zur ursprünglichen Formel ist, d. h. beide die gleichen Modelle besitzen. Im Gegensatz hierzu ist die Skolem-Form von F lediglich *erfüllbarkeitsäquivalent* zu F , geschrieben als $F \equiv_E \text{Skolem}(F)$. Mit anderen Worten: Aus der Erfüllbarkeit von F folgt die Erfüllbarkeit ihrer Skolem-Form und umgekehrt; trotzdem besitzen beide unterschiedliche Modelle. Obgleich die Erfüllbarkeitsäquivalenz eine schwächere Eigenschaft als die Äquivalenz ist, reicht sie aus, um z. B. die Allgemeingültigkeit einer Formel F zu beweisen. Können wir zeigen, dass die Skolem-Form von $\neg F$ unerfüllbar ist, so ist auch $\neg F$ unerfüllbar und F im Umkehrschluss als Tautologie identifiziert. In Abschnitt 3.2.3.1 werden wir die Skolem-Form im Zusammenhang mit der prädikatenlogischen Resolution wieder aufgreifen.

Halten Sie stets im Gedächtnis, dass sowohl die Pränex-Form als auch die Skolem-Form im Allgemeinen nicht eindeutig sind. Warum dies so ist, verdeutlicht Abbildung 3.37. Durch eine geänderte Abfolge der Regelanwendungen entsteht eine zweite Pränex-Form, in der die Quantoren anders angeordnet sind. Die geänderte Reihenfolge schlägt sich in direkter Weise auf die neu hinzugefügten Terme in der Skolem-Form nieder.

3.2.3 Beweistheorie

In diesem Abschnitt beschäftigen wir uns mit der Frage, wie die Allgemeingültigkeit einer prädikatenlogischen Formel systematisch bewiesen werden kann. Genau wie im Fall der Aussagenlogik werden wir verschiedene Kalküle definieren, die eine formale Ableitbarkeit wahrer Aussagen erlauben.

Um eine prädikatenlogische Formel F als allgemeingültig zu identifizieren, müssen wir zeigen, dass *jede* Interpretation ein Modell von F ist. Auf den ersten Blick mag es aussichtslos erscheinen, die Allgemeingültigkeit mithilfe eines mechanisch arbeitenden Kalküls zu beweisen. Schuld daran ist der in Definition 3.16 eingeführte Interpretationsbegriff, der uns mit einer wahren Flut an potenziellen Modellen überschüttet. Die große Anzahl kommt dadurch zustande, dass die Individuenmenge U einer Interpretation beliebig gewählt werden darf. Beispielsweise kann U die Menge der natürlichen Zahlen sein, alle Einwohner von Paris umfassen oder der Menge aller gleichschenkligen Dreiecke entsprechen. So unterschiedlich diese Interpretationen auch sind: Für ausnahmslos jede müssen wir zeigen, dass sie ein Modell von F ist.

In Wirklichkeit ist die Situation weniger aussichtslos, als sie an dieser Stelle wirken mag. Der französische Mathematiker Jacques Herbrand (Abbildung 3.38) konnte zeigen, dass es gar nicht nötig ist, sämtliche in Frage kommenden Interpretationen zu untersuchen. Um eine Formel als allgemeingültig zu entlarven, ist es völlig ausreichend, die Suche auf die Klasse der *Herbrand-Interpretationen* zu beschränken. Um zu verstehen, wie eine solche Interpretation aufgebaut ist, führen wir zunächst den Begriff des *Herbrand-Universums* ein.



Definition 3.21 (Herbrand-Universum)

Sei F eine prädikatenlogische Formel über der Signatur Σ . Das *Herbrand-Universum* $HU(F)$ ist die Menge aller variablenfreien Terme, die mit Funktionssymbolen aus Σ gebildet werden können.

Aus der Signatur $\Sigma = (V_\Sigma, F_\Sigma, P_\Sigma)$ einer prädikatenlogischen Formel F lässt sich das Herbrand-Universum rekursiv erzeugen:

- Alle Konstanten aus F_Σ gehören zu $HU(F)$.
- Ist $f \in F_\Sigma$ ein n -stelliges Funktionszeichen und gehören t_1, \dots, t_n zu $HU(F)$, so gehört auch der Term $f(t_1, \dots, t_n)$ zu $HU(F)$.

Nicht selten führt der Begriff des Herbrand-Universums zu Verständnisproblemen, da eine Vermischung zwischen der syntaktischen und der semantischen Ebene stattfindet. Da die Grundmenge aus den Elementen von Σ erzeugt wird, beeinflusst die Syntax von F direkt den Individuenbereich, über dem die einzelnen Formelbestandteile interpretiert werden. Auch wenn die Konstruktion trickreich ist, spricht nichts gegen sie, da wir die Individuenmenge völlig frei wählen dürfen.

Die beiden nachstehenden Beispiele verdeutlichen die Konstruktion:

- Beispiel 1: $F := \forall x P(f(c, g(x)))$
 $HU(F) = \{c, g(c), f(c, c), g(f(c, c)), f(g(c), c), f(g(c), g(c)), \dots\}$
- Beispiel 2: $F := \exists x P(f(x))$
 $HU(F) = \{a, f(a), f(f(a)), f(f(f(a))), f(f(f(f(a)))), \dots\}$

Das zweite Beispiel macht auf eine Besonderheit aufmerksam, die wir nicht vorschnell übergehen dürfen. Enthält Σ keine 0-stelligen Funktionssymbole, so lassen sich zunächst keine variablenfreien Terme bilden



Jacques Herbrand
(1908 – 1931)

Abbildung 3.38: Jacques Herbrand wurde am 12. Februar 1908 in Paris geboren und bereits in seiner frühen Jugend erkannten die Eltern sein mathematisches Talent. Im Alter von 17 Jahren meisterte er mit Bravour die Aufnahmeprüfung der École Normale und begann sehr früh, ein ausgeprägtes Interesse für die Grundlagen der Mathematik zu entwickeln. 1929 promovierte Herbrand in mathematischer Logik und absolvierte anschließend seinen Militärdienst. Im Jahr 1931 eröffnete ihm ein Rockefeller-Stipendium die Möglichkeit, verschiedene Universitäten im Ausland zu besuchen. Unter anderen führte ihn seine Reise nach Berlin und Göttingen, wo er mit John von Neumann und Emmy Noether zusammentraf. Das Bild zeigt Jacques Herbrand in jungen Jahren. Später existieren nicht, da sein Leben im Alter von 23 Jahren ein frühes Ende fand. Vor seiner Rückkehr von Göttingen nach Frankreich kam er bei einem Wanderunfall in den Alpen zu Tode. Trotz seines frühen Ablebens ist Herbrands wissenschaftliches Vermächtnis immens. Seine Arbeiten auf dem Gebiet der mathematischen Logik bilden das theoretische Grundgerüst, auf dem die gesamte Beweistheorie der Prädikatenlogik beruht.

■ Formel

$$F := \forall x \exists y P(f(x, y))$$

■ Herbrand-Universum

$$\begin{aligned} HU(F) = & \{a, \\ & f(a, a), \\ & f(a, f(a, a)), \\ & f(f(a, a), f(a, a)), \\ & \dots \\ & \} \end{aligned}$$

■ Beispielinterpretation 1

$$U = HU(F)$$

$$I(P) = \emptyset$$

■ Beispielinterpretation 2

$$U = HU(F)$$

$$I(P) = \{f(t, a) \mid t \in HU(F)\}$$

■ Beispielinterpretation 3

$$U = HU(F)$$

$$I(P) = HU(F)$$

Abbildung 3.39: Verschiedene Herbrand-Interpretationen für die prädikatenlogische Formel $\forall x \exists y P(f(x, y))$

und das Herbrand-Universum würde zur leeren Menge degradieren. In diesem Fall lösen wir das Problem, indem wir der Menge F_Σ schlicht ein neues Konstantensymbol a hinzufügen. Die Änderung der Signatur hat keinen Einfluss auf die weiter unten erarbeiteten Ergebnisse.

Aufbauend auf dem Begriff des Herbrand-Universums führen wir den Begriff der *Herbrand-Interpretation* ein:



Definition 3.22 (Herbrand-Interpretation)

Sei F eine prädikatenlogische Formel über der Signatur Σ . Eine Interpretation (U, I) von F heißt *Herbrand-Interpretation*, falls

- $U = HU(F)$ und
- $I(t) = t$ für alle variablenfreien Terme t .

Eine Herbrand-Interpretation einer Formel F erfüllt demnach zwei charakteristische Eigenschaften. Zum einen entspricht die Grundmenge dem Herbrand-Universum $HU(F)$, d.h., der Individuenbereich ist die Menge der variablenfreien Terme, die aus den Funktionssymbolen von F gebildet werden können. Zum anderen ist die Zuordnung I so gestaltet, dass jeder variablenfreie Term t durch sich selbst interpretiert wird. Die Selbstinterpretation ist möglich, da als Grundmenge das Herbrand-Universum gewählt wurde und dieses alle variablenfreien Terme umfasst. In der Konsequenz unterscheiden sich Herbrand-Interpretationen ausschließlich im Umgang mit den Prädikatsymbolen (vgl. Abbildung 3.39).

Herbrand konnte zeigen, dass eine Formel F genau dann erfüllbar ist, wenn sie ein *Herbrand-Modell* besitzt. Hieraus folgt, dass F genau dann allgemeingültig ist, wenn die negierte Formel $\neg F$ kein Herbrand-Modell besitzt. Der Zusammenhang hat weitreichende Konsequenzen für die Beweisführung, da wir die Suche nach Modellen auf die spezielle Menge der Herbrand-Interpretationen eingrenzen können. Mit einem Schlag gerät die automatisierte Beweisführung in greifbare Nähe. Den Durchbruch erzielte Herbrand mit dem folgenden Satz, den wir an dieser Stelle ohne Beweis akzeptieren wollen [43]:



Satz 3.3 (Satz von Herbrand)

Eine Formel F in Skolem-Form besitzt genau dann ein Herbrand-Modell, wenn alle endlichen Teilmengen der *Grundinstanzen* von F im aussagenlogischen Sinne erfüllbar sind.

Im Umkehrschluss besagt der Satz, dass eine Formel F in Skolem-Form genau dann unerfüllbar ist, wenn eine endliche Teilmenge der Grundinstanzen existiert, die im aussagenlogischen Sinne widersprüchlich ist. Die Menge der Grundinstanzen von F wird erzeugt, indem alle Variablen durch Terme aus $\text{HU}(F)$ und somit durch variablenfreie Terme ersetzt werden. Formal ist die Menge wie folgt definiert:

Definition 3.23 (Grundinstanzen)

Sei Σ eine prädikatenlogische Signatur und F eine zu Σ passende Formel in Skolem-Form, d. h., es gilt:

$$F = \forall x_1 \dots \forall x_n F^*$$

Die Menge der *Grundinstanzen* von F , kurz $G(F)$, ist definiert als

$$G(F) := \{ F^*[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n] \mid t_1, \dots, t_n \in \text{HU}(F) \}$$

Für die praktische Beweisführung bedeutet der Satz von Herbrand das Folgende: Um die Allgemeingültigkeit einer Formel F zu beweisen, bilden wir zunächst die Negation $\neg F$ und wählen eine passende Menge von Grundinstanzen aus. Jede atomare Formel behandeln wir als eigenständige aussagenlogische Variable. Können wir mit dem Instrumentarium der Aussagenlogik die Unerfüllbarkeit der konstruierten Menge nachweisen, so ist gezeigt, dass $\neg F$ kein Herbrand-Modell besitzt. Das bedeutet, dass $\neg F$ überhaupt keine Modelle besitzt und F deshalb allgemeingültig sein muss.

Als Beispiel betrachten wir die prädikatenlogische Formel

$$F := \exists x \forall y P(x, y) \rightarrow \forall y \exists x P(x, y)$$

Um die Allgemeingültigkeit von F zu zeigen, bilden wir zunächst die Skolem-Form von $\neg F$:

$$\begin{aligned} \neg F &\equiv \neg(\exists x \forall y P(x, y) \rightarrow \forall y \exists x P(x, y)) \\ &\equiv \exists x \forall y P(x, y) \wedge \neg \forall y \exists x P(x, y) \\ &\equiv \exists x \forall y P(x, y) \wedge \exists y \forall x \neg P(x, y) \\ &\equiv_E \forall y P(a, y) \wedge \forall x \neg P(x, b) \\ &\equiv \forall y \forall x (P(a, y) \wedge \neg P(x, b)) \end{aligned}$$

Aus der Skolem-Form können wir die Grundinstanzen von F ableiten:

$$G(F) := \{ P(a, a) \wedge \neg P(a, b), P(a, a) \wedge \neg P(b, b), \\ P(a, b) \wedge \neg P(a, b), P(a, b) \wedge \neg P(b, b) \}$$

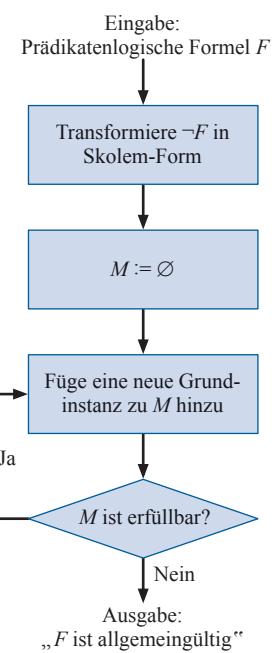


Abbildung 3.40: Algorithmus von Gilmore

$$F = \exists x \forall y P(x, y) \rightarrow \forall y \exists x P(x, y)$$



$$\neg F \equiv \exists x \forall y P(x, y) \wedge \exists y \forall x \neg P(x, y)$$

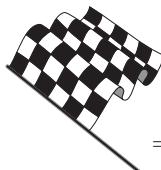


$$\text{Skolem}(F) = \forall y \forall x (P(a, y) \wedge \neg P(x, b))$$

i	Grundinstanzen
0	\emptyset
1	$P(a, a) \wedge \neg P(a, b)$
2	$P(a, a) \wedge \neg P(a, b), P(a, a) \wedge \neg P(b, b)$
3	$P(a, a) \wedge \neg P(a, b), P(a, a) \wedge \neg P(b, b), P(a, b) \wedge \neg P(a, b)$

Die gefundene Menge ist unerfüllbar.

Der Algorithmus terminiert.



$\neg F$ ist unerfüllbar
 $\Rightarrow F$ ist allgemeingültig

Abbildung 3.41: Algorithmus von Gilmore, angewendet auf die prädikatenlogische Formel $\exists x \forall y P(x, y) \rightarrow \forall y \exists x P(x, y)$

Ganz offensichtlich ist die Teilmenge

$$\{P(a, b) \wedge \neg P(a, b)\}$$

unerfüllbar und damit die Allgemeingültigkeit von F bewiesen.

Aus den angestellten Überlegungen lässt sich der *Algorithmus von Gilmore* ableiten. Hierbei handelt es sich um ein konstruktives Verfahren, mit dem die Allgemeingültigkeit einer prädikatenlogischen Formel F systematisch bewiesen werden kann. Im Rahmen der Vorverarbeitung wird F negiert und in Skolem-Form überführt. Anschließend werden alle endlichen Teilmengen von $G(F)$ gebildet. Dies kann in einem iterativen Prozess geschehen, der von der leeren Menge ausgeht und diese sukzessive um neue Formeln aus $G(F)$ erweitert. Am Ende jedes Schritts wird die Menge mithilfe eines beliebigen aussagenlogischen Verfahrens auf Erfüllbarkeit geprüft. Ist die Menge unerfüllbar, so folgt aus dem Satz von Herbrand, dass die Formel $\neg F$ kein Modell besitzt und F im Umkehrschluss eine allgemeingültige Formel sein muss. Abbildung 3.40 fasst die Arbeitsweise des Algorithmus von Gilmore grafisch zusammen.

In Abbildung 3.41 sind die Iterationsschritte dargestellt, die der Algorithmus von Gilmore für die diskutierte Beispieldformel durchläuft. Ausgehend von der leeren Menge wird in jedem Schritt eine neue Grundinstanz hinzugefügt und die entstandene Menge auf Erfüllbarkeit geprüft. Nach 3 Iterationen ist eine unerfüllbare Teilmenge gefunden und die Ausgangsformel als allgemeingültig identifiziert.

Beachten Sie, dass der Algorithmus für unser Beispiel nach 4 Iterationen ebenfalls terminiert hätte, da $G(F)$ in diesem speziellen Fall endlich ist. Die Endlichkeit von $G(F)$ geht auf die Eigenschaft von F zurück, keine mehrstelligen Funktionssymbole zu enthalten. Kommt in einer Formel F mindestens ein n -stelliges Funktionssymbol mit $n \geq 1$ vor, so lassen sich unendlich viele Grundinstanzen bilden und der Algorithmus terminiert nur dann, wenn eine unerfüllbare Teilmenge gefunden wird. Ist die bearbeitete Formel F nicht allgemeingültig, so sind alle generierten Teilmengen erfüllbar und es entsteht eine Endlosschleife.

Anhand der folgenden Formel wollen wir dieses Verhalten genauer untersuchen:

$$F := \forall y \exists x P(x, y) \rightarrow \exists x \forall y P(x, y)$$

Die Formel unterscheidet sich von unserem ersten Beispiel lediglich in der Schlussrichtung der logischen Implikation.

Wie gewohnt bilden wir die Skolem-Form von $\neg F$:

$$\begin{aligned}\neg F &\equiv \neg(\forall y \exists x P(x,y) \rightarrow \exists x \forall y P(x,y)) \\ &\equiv \forall y \exists x P(x,y) \wedge \neg \exists x \forall y P(x,y) \\ &\equiv \forall y \exists x P(x,y) \wedge \forall x \exists y \neg P(x,y) \\ &\equiv_E \forall y P(f(y),y) \wedge \forall x \neg P(x,g(x)) \\ &\equiv \forall y \forall x (P(f(y),y) \wedge \neg P(x,g(x)))\end{aligned}$$

Die Grundinstanzen berechnen sich wie folgt:

$$\begin{aligned}G(F) &:= \{P(f(a),a) \wedge \neg P(a,g(a)), \\ &\quad P(f(f(a)),f(a)) \wedge \neg P(a,g(a)), \\ &\quad P(f(a),a) \wedge \neg P(f(a),g(f(a))), \dots\}\end{aligned}$$

Wie in Abbildung 3.42 angedeutet, lassen sich für diese Formel immer wieder neue Grundinstanzen bilden, so dass der Algorithmus von Gilmore nicht terminiert.

Die Eigenschaft des Algorithmus, für gewisse Formeln F in eine Endlosschleife zu geraten, ist kein Schönheitsfehler. Sie ist der Tatsache geschuldet, dass das Erfüllbarkeitsproblem der Prädikatenlogik in die Klasse der *unentscheidbaren Probleme* fällt. Anders als in der Aussagenlogik sind wir in der Prädikatenlogik nicht in der Lage, für jede prädikatenlogische Formel zweifelsfrei zu bestimmen, ob sie allgemeingültig ist oder nicht. Konkret hat dies zur Folge, dass wir den Beweis nur für den positiven Fall erfolgreich zu Ende bringen können. Ist F nicht allgemeingültig, so können wir zu keinem Zeitpunkt entscheiden, ob der Algorithmus unendlich lange läuft oder nicht doch im nächsten Schritt terminieren wird. Da der Entscheidungsprozess jetzt nur noch in eine Richtung funktioniert, sprechen wir in diesem Zusammenhang auch von einem *Semi-Entscheidungsverfahren*. Für den Moment wollen wir uns mit der etwas informellen Darstellung des Sachverhalts zufriedengeben. In Abschnitt 6.3 werden wir die Begriffe der *Entscheidbarkeit* und der *Semi-Entscheidbarkeit* erneut aufgreifen und in aller Ausführlichkeit behandeln.

In den nächsten beiden Abschnitten werden wir zeigen, wie sich die aussagenlogischen Resolutions- und Tableaukalküle zu prädikatenlogischen Beweisverfahren ausbauen lassen. Prädikatenlogische Hilbert-Kalküle existieren ebenfalls, haben aber nur eine geringe Praxisbedeutung. Wie schon im aussagenlogischen Fall müssen geeignete Formelinstanzen geraten werden. Da die Anzahl möglicher Kandidaten jedoch ungleich größer ist, wird die Beweisführung zusätzlich erschwert. Für eine ausführliche Darstellung prädikatenlogischer Hilbert-Kalküle sei der interessierte Leser auf [71] oder [50] verwiesen.

$$F = \forall y \exists x P(x,y) \rightarrow \exists x \forall y P(x,y)$$



$$\neg F \equiv \forall y \exists x P(x,y) \wedge \forall x \exists y \neg P(x,y)$$



$$\begin{aligned}\text{Skolem}(F) &= \\ &\forall y \forall x (P(f(y),y) \wedge \neg P(x,g(x)))\end{aligned}$$

i	Grundinstanzen
0	\emptyset
1	$P(f(a),a) \wedge \neg P(a,g(a))$
2	$P(f(a),a) \wedge \neg P(a,g(a)),$ $P(f(f(a)),f(a)) \wedge \neg P(a,g(a))$
3	$P(f(a),a) \wedge \neg P(a,g(a)),$ $P(f(f(a)),f(a)) \wedge \neg P(a,g(a)),$ $P(f(a),a) \wedge \neg P(f(a),g(f(a)))$
	...



Abbildung 3.42: Algorithmus von Gilmore, angewendet auf die prädikatenlogische Formel $\forall y \exists x P(x,y) \rightarrow \exists x \forall y P(x,y)$

■ Beispiel 1

$$F_1 := P(f(y), y)$$

$$F_2 := P(x, g(x))$$



F_1 und F_2 sind nicht unifizierbar. Es müsste gleichzeitig gelten:

$$[x \leftarrow f(y)] \text{ und } [y \leftarrow g(x)]$$

■ Beispiel 2

$$F_1 := P(x, f(y))$$

$$F_2 := P(f(y), x)$$



Unifikatoren:

$$\sigma_1 = [x \leftarrow f(y)]$$

$$\sigma_2 = [x \leftarrow f(f(z)), y \leftarrow f(z)]$$

$$\sigma_3 = [x \leftarrow f(f(f(z))), y \leftarrow f(f(z))]$$

...

Abbildung 3.43: Beispiele zur Verdeutlichung des Unifikationsbegriffs

3.2.3.1 Resolutionskalkül

Um den prädikatenlogischen Resolutionskalkül zu verstehen, werfen wir erneut einen Blick auf Abbildung 3.41. Für die gezeigte Beispielformel

$$F := \exists x \forall y P(x, y) \rightarrow \forall y \exists x P(x, y)$$

bricht der Algorithmus von Gilmore ab, sobald die widersprüchliche Grundinstanz $P(a, b) \wedge \neg P(a, b)$ gefunden wurde. In der Klauseldarstellung schreiben wir diese Formel als $\{P(a, b)\}, \{\neg P(a, b)\}$.

Betrachten wir die Skolem-Form von $\neg F$ in Klauseldarstellung, so erhalten wir ein sehr ähnlich aussehendes Ergebnis:

$$\{P(a, y)\}, \{\neg P(x, b)\}$$

Die widersprüchliche Grundinstanz $\{P(a, b)\}, \{\neg P(a, b)\}$ können wir sofort erhalten, indem wir eine Substitution σ konstruieren, die das Literal $P(a, y)$ mit dem (unnegierten) Literal $P(x, b)$ in Übereinstimmung bringt. Konkret können wir die Gleichheit herstellen, indem wir x durch a und y durch b ersetzen. In der Terminologie der Logik werden $P(a, y)$ und $P(x, b)$ durch die Substitution σ *unifiziert* und der gesamte Konstruktionsvorgang als *Unifikation* bezeichnet.



Definition 3.24 (Unifikation)

Zwei prädikatenlogische Formeln F und G sind *unifizierbar*, falls eine Substitution σ existiert mit $\sigma F = \sigma G$. Eine Substitution mit dieser Eigenschaft heißt *Unifikator*.

Der Unifikationsbegriff lässt sich intuitiv auf Formelmengen mit einer beliebigen Anzahl an Elementen erweitern. Die Menge $\{F_1, \dots, F_n\}$ wird durch die Substitution σ unifiziert, falls $\sigma F_1 = \dots = \sigma F_n$ gilt.

Auch wenn die angegebene Substitution in unserem Beispiel die einzige Möglichkeit ist, beide Literale zu unifizieren, ist die Situation im Allgemeinen ein wenig komplizierter. Wie das obere Beispiel in Abbildung 3.43 zeigt, sind nicht alle Formelpaare unifizierbar. Gäbe es einen Unifikator σ für die Formeln $P(f(y), y)$ und $P(x, g(x))$, so müsste dieser x durch $f(y)$ und gleichzeitig y durch $g(x)$ ersetzen. Eine solche Substitution kann aufgrund des entstehenden Selbstbezugs nicht existieren. Wie das untere Beispiel in Abbildung 3.43 zeigt, existieren für andere Formelpaare mehrere, ja sogar unendlich viele Möglichkeiten,

um die Gleichheit herzustellen. σ_1 spielt in diesem Beispiel eine besondere Rolle, da wir alle anderen Unifikatoren durch die Anwendung einer weiteren Substitution aus σ_1 erzeugen können. Eine Substitution mit dieser Eigenschaft bezeichnen wir als *allgemeinsten Unifikator*.

Wir wollen uns nun mit der Frage beschäftigen, wie sich der allgemeinsten Unifikator systematisch erzeugen lässt. Die Antwort liefert uns der in Abbildung 3.44 skizzierte *Algorithmus von Robinson*. Intern arbeitet dieser auf der Menge M und einer Substitution σ . Zu Beginn wird M mit der zu unifizierenden Formelmenge und σ mit der identischen Abbildung $\text{id} : x \mapsto x$ initialisiert. Anschließend prüft der Algorithmus, ob M nur ein einziges Element enthält. In diesem Fall unifiziert σ die Eingabemenge und wird ausgegeben. Enthält M zwei oder mehr Elemente, so werden die Formeln Zeichen für Zeichen von links nach rechts miteinander verglichen. An der ersten Position, an der sich mindestens zwei Formeln unterscheiden, überprüft der Algorithmus, ob die Gleichheit durch Substitution hergestellt werden kann. Dies ist immer dann der Fall, wenn eine Menge von Variablen x_1, \dots, x_m und ein Term t mit $x_1, \dots, x_m \notin t$ vorgefunden werden, nicht aber zwei oder mehr unterschiedliche Terme aufeinandertreffen. Im ersten Fall wird σ mit der Substitution $\mu = [x_1 \leftarrow t, \dots, x_m \leftarrow t]$ verketzt und der Vorgang auf der Menge μM wiederholt. Andernfalls terminiert der Algorithmus mit der Ausgabe „nicht unifizierbar“. Tabelle 3.8 fasst die schrittweise Berechnung eines allgemeinsten Unifikators für das Formelpaar

$$\begin{aligned} F_1 &:= P(f(g(x_2, x_3)), g(x_1, g(a, b))) \\ F_2 &:= P(f(x_4), g(h(a, x_5), g(x_2, x_3))) \end{aligned}$$

zusammen.

Nachdem wir Unifikatoren jetzt systematisch berechnen können, ist es an der Zeit, uns der prädikatenlogischen Resolutionsregel zuzuwenden. Sie lautet wie folgt:

$$\frac{M_1 \cup \{L_1\} \quad M_2 \cup \{\neg L_2\}}{\sigma(M_1 \cup M_2)} \quad \sigma L_1 = \sigma L_2$$

Die Substitution σ bezeichnet den *allgemeinsten Unifikator* von L_1 und L_2 . In Worten liest sich die Resolutionsregel wie folgt: Zwei Klauseln lassen sich genau dann zu einer Resolvente verschmelzen, wenn diese ein komplementäres Formelpaar aufweisen, das mit einer Substitution σ unifiziert werden kann. Um die Resolvente zu bilden, wird der Unifikator σ auf M_1 und M_2 angewendet, das komplementäre Formelpaar entfernt und die restlichen Formeln zu einer gemeinsamen Klausel vereint.

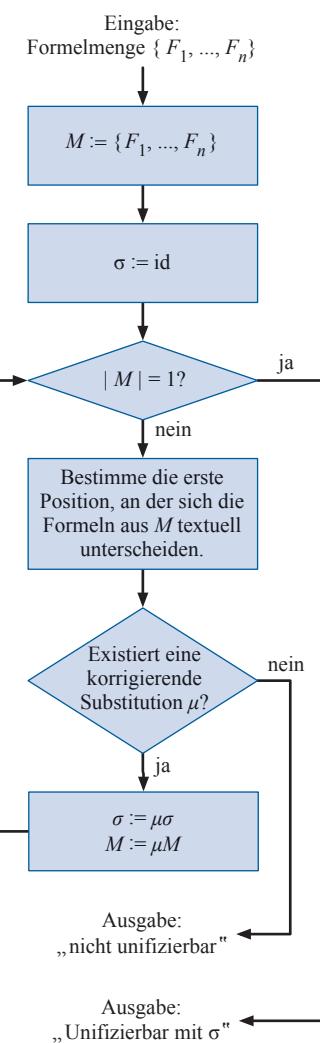


Abbildung 3.44: Unifikationsalgorithmus von Robinson

M	μ	σ
$P(f(g(x_2, x_3)), g(x_1, g(a, b)))$ $P(f(x_4), g(h(a, x_5), g(x_2, x_3)))$		id
$P(f(g(x_2, x_3)), g(x_1, g(a, b)))$ $P(f(x_4), g(h(a, x_5), g(x_2, x_3)))$	$x_4 \leftarrow g(x_2, x_3)$	$x_4 \leftarrow g(x_2, x_3)$
$P(f(g(x_2, x_3)), g(x_1, g(a, b)))$ $P(f(g(x_2, x_3)), g(h(a, x_5), g(x_2, x_3)))$	$x_1 \leftarrow h(a, x_5)$	$x_4 \leftarrow g(x_2, x_3),$ $x_1 \leftarrow h(a, x_5)$
$P(f(g(x_2, x_3)), g(h(a, x_5), g(a, b)))$ $P(f(g(x_2, x_3)), g(h(a, x_5), g(x_2, x_3)))$	$x_2 \leftarrow a$	$x_4 \leftarrow g(a, x_3),$ $x_1 \leftarrow h(a, x_5),$ $x_2 \leftarrow a$
$P(f(g(a, x_3)), g(h(a, x_5), g(a, b)))$ $P(f(g(a, x_3)), g(h(a, x_5), g(a, x_3)))$	$x_3 \leftarrow b$	$x_4 \leftarrow g(a, b),$ $x_1 \leftarrow h(a, x_5),$ $x_2 \leftarrow a,$ $x_3 \leftarrow b$
$P(f(g(a, b)), g(h(a, x_5), g(a, b)))$ $P(f(g(a, b)), g(h(a, x_5), g(a, b)))$		$x_4 \leftarrow g(a, b),$ $x_1 \leftarrow h(a, x_5),$ $x_2 \leftarrow a,$ $x_3 \leftarrow b$

Tabelle 3.8: Schrittweise Abarbeitung des Algorithmus von Robinson

Um Namenskollisionen zu vermeiden, darf die Resolutionsregel nur dann angewendet werden, wenn die beiden Prämissen keine gemeinsamen Variablen aufweisen. Hierbei handelt es sich um keine problematische Beschränkung, da wir die Variablen einer Klausel vor der Resolventenbildung nach Belieben umbenennen können, ohne die Korrektheit des Verfahrens zu gefährden.

Erinnern Sie sich noch an das Eingangsbeispiel, in dem wir Sokrates als Mensch und die Sterblichkeit als universelle Eigenschaft eines jeden Menschen beschrieben haben? Mithilfe der prädikatenlogischen Reso-

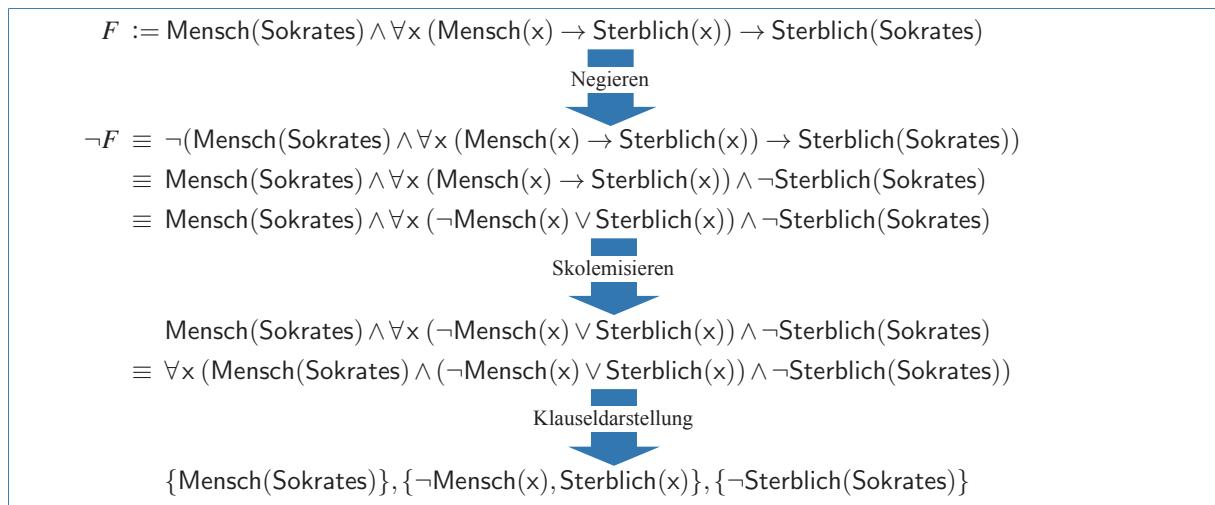


Abbildung 3.45: Um die Sterblichkeit von Sokrates im Resolutionskalkül zu beweisen, muss die Ausgangsformel zunächst in Klauseldarstellung überführt werden.

lution sind wir jetzt in der Lage, die Sterblichkeit von Sokrates formal zu beweisen. Hierzu gilt es, die Allgemeingültigkeit der folgenden Aussage zu zeigen:

$$\begin{aligned} & \text{Mensch}(\text{Sokrates}) \wedge \forall x (\text{Mensch}(x) \rightarrow \text{Sterblich}(x)) \\ & \quad \rightarrow \text{Sterblich}(\text{Sokrates}) \end{aligned}$$

Um die Behauptung im Resolutionskalkül zu beweisen, müssen wir die negierte Formel zunächst in die Skolem-Form und anschließend in die Klauseldarstellung transformieren. Führen wir die in Abbildung 3.45 zusammengefassten Umformungsschritte durch, so erhalten wir als Ergebnis drei Klauseln, auf die wir den Resolutionskalkül direkt anwenden können. Der eigentliche Resolutionsbeweis erweist sich nach der geleisteten Vorbereitung als denkbar einfach. Wie in Abbildung 3.46 gezeigt, lässt sich die leere Klausel in nur zwei Schritten ableiten.

Wir wollen die prädikatenlogische Resolution jetzt von einem abstrakteren Standpunkt betrachten und der Frage nachgehen, ob wir ein korrektes und vollständiges Kalkül vor uns haben. Die Korrektheit der prädikatenlogischen Resolutionsregel ist leicht einzusehen. Sie folgt aus der Korrektheit der aussagenlogischen Regel sowie der Tatsache, dass der angewendete Unifikator die Aussagen der Prämissen unverändert lässt oder einschränkt, jedoch in keinem Fall verallgemeinert.

Dagegen ist der Kalkül in der vorgestellten Form nicht mehr vollständig. Warum diese Eigenschaft verloren geht, wollen wir am Beispiel der

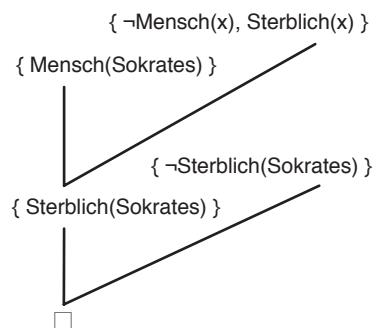
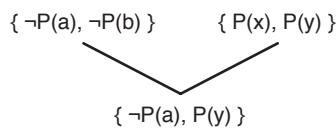
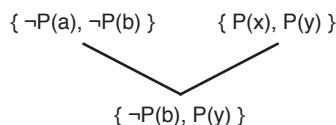


Abbildung 3.46: Sokrates ist sterblich! Formal bewiesen im Resolutionskalkül.

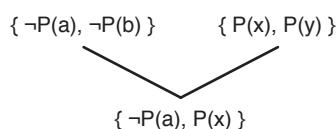
■ Erste Resolvente



■ Zweite Resolvente



■ Dritte Resolvente



■ Vierte Resolvente

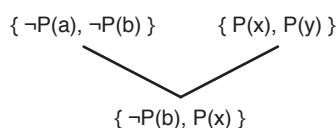


Abbildung 3.47: Obwohl die Ausgangsmenge unerfüllbar ist, lässt sich die leere Klausel \square nicht ableiten.

folgenden Klauselmenge herausarbeiten:

$$\{P(x), P(y)\}, \{\neg P(a), \neg P(b)\}$$

Die erste Klausel besagt, dass die Eigenschaft P für alle Elemente der Individuenmenge erfüllt ist, die zweite, dass P für die Elemente a und b nicht gilt. Obwohl die Menge unerfüllbar ist, kann die leere Klausel nicht mit der eingeführten Resolutionsregel abgeleitet werden. Warum dies so ist, macht Abbildung 3.47 deutlich: Es lassen sich ausschließlich Resolventen bilden, die wiederum 2 Elemente besitzen.

Der Verursacher des Problems ist die Klausel $\{P(x), P(y)\}$. Sie entspricht dem folgenden Ausdruck:

$$\forall x \forall y (P(x) \vee P(y))$$

Die Formel lässt sich zu

$$\forall x P(x) \vee \forall y P(y)$$

umformen und ist logisch äquivalent zu

$$\forall x P(x)$$

Wären wir in der Lage, aus der Klausel $\{P(x), P(y)\}$ die Klausel $\{P(x)\}$ zu erzeugen, so ließe sich auch die leere Klausel ableiten. Genau dies leistet die *Faktorisierungsregel*, die wir im prädikatenlogischen Fall als zusätzliche Schlussregel in den Kalkül integrieren müssen:

$$\frac{\{L_1, \dots, L_{n-1}, L_n\}}{\{\sigma L_1, \dots, \sigma L_{n-1}\}} \quad \sigma L_{n-1} = \sigma L_n$$

Abbildung 3.48 zeigt, wie sich die Unerfüllbarkeit der Klauselmenge unter Einbeziehung der Faktorisierungsregel beweisen lässt. Zunächst wird aus $\{P(x), P(y)\}$ die Klausel $\{P(x)\}$ erzeugt und anschließend die leere Klausel durch zweifache Resolventenbildung hergeleitet.

Zusammen mit der Faktorisierungsregel wird der Resolutionskalkül tatsächlich vollständig, d. h., wir können jede allgemeingültige prädikatenlogische Formel innerhalb des Kalküls als solche beweisen. Der formale Beweis ist kompliziert und soll an dieser Stelle nicht geführt werden. Der interessierte Leser sei auf [88] oder [72] verwiesen. Beachten Sie, dass dieses Ergebnis nicht im Widerspruch zur Semi-Entscheidbarkeit steht, die wir im Zusammenhang mit dem Algorithmus von Gilmore besprochen haben. Ist eine Formel F nicht allgemeingültig, so lassen sich normalerweise unendlich viele Resolventen bilden und wir wissen zu keinem Zeitpunkt, ob die leere Klausel irgendwann darunter sein wird oder nicht.

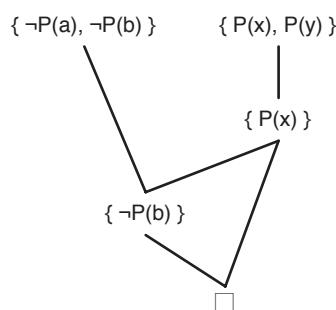


Abbildung 3.48: Durch die Hinzunahme der Faktorisierungsregel wird der Resolutionskalkül vollständig. Auch die Unerfüllbarkeit unserer Beispieldformel lässt sich jetzt beweisen.

3.2.3.2 Tableaukalkül

Weiter oben haben wir herausgearbeitet, wie die aussagenlogische Resolution mithilfe der Unifikation zu einem prädikatenlogischen Kalkül ausgebaut werden kann. Mit dem gleichen Ansatz können wir auch den Tableaukalkül auf die Prädikatenlogik ausdehnen. Die grundlegende Vorgehensweise bleibt dabei unverändert: Um die Allgemeingültigkeit einer Formel F zu zeigen, gehen wir von der negierten Formel $\neg F$ aus und versuchen, diese zu einem geschlossenen Tableau zu erweitern.

Um prädikatenlogische Formeln mit dem Tableaukalkül verarbeiten zu können, müssen wir zwei Erweiterungen vornehmen: Zum einen benötigen wir Schlussregeln für die beiden Quantoren \forall und \exists , zum anderen eine angepasste Regel, um Pfade zu schließen. Der Umgang mit den prädikatenlogischen Quantoren wird möglich, indem die bereits vorhandenen α - und β -Regeln durch zwei neue, in Abbildung 3.49 zusammengefasste Regelgruppen ergänzt werden. Die neu hinzugekommenen γ -Regeln gestatten uns, einen Allquantor ($\forall x F$) bzw. einen negierten Existenzquantor ($\neg \exists x F$) zu eliminieren, indem x durch eine neue Variable y substituiert wird. Die Ersetzung muss *kollisionsfrei* erfolgen, d.h., y darf nicht an anderer Stelle des Tableaus als freie Variable auftauchen. Die δ -Regeln legen den Umgang mit Existenzaussagen ($\exists x F, \neg \forall x F$) fest. In diesem Fall wird die Variable x skolemisiert, indem sie durch einen Term der Form $f(x_1, \dots, x_n)$ ersetzt wird. Hierbei ist f ein neues Funktionssymbol und x_1, \dots, x_n sind die freien Variablen von F .

Auch das Schließen eines Pfades erfordert mehr Sorgfalt als bisher. Mussten wir im aussagenlogischen Fall lediglich darauf warten, dass im Rahmen der Expansion irgendwann ein komplementäres Formelpaar erzeugt wird, so müssen wir ein solches Paar im prädikatenlogischen Fall fast immer aktiv erzeugen. Hierzu werden auf dem betrachteten Pfad zwei Formeln F_1 und $\neg F_2$ gewählt und F_1 und F_2 miteinander unifiziert. Gelingt die Unifikation, so kann der Pfad geschlossen werden, indem der allgemeinste Unifikator von F_1 und F_2 auf sämtliche Tableauformeln angewendet wird.

Damit haben wir alle Bausteine zusammen, um ein prädikatenlogisches Tableau zu erzeugen. In jedem Konstruktionsschritt können wir zwischen zwei möglichen Aktionen wählen:

■ Erweitern des Tableaus

Das Tableau wird durch die Anwendung einer α -, β -, γ - oder δ -Regel erweitert. Genau wie im aussagenlogischen Fall wirkt sich die

■ γ -Expansionen



■ δ -Expansionen

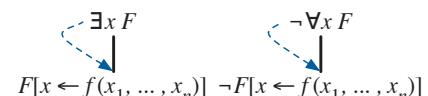


Abbildung 3.49: Zusätzliche Expansionsregeln des prädikatenlogischen Tableaukalküls. Die γ -Regeln führen eine neue Variable y ein, die an keiner anderen Stelle im Tableau als freie Variable vorkommen darf. Die δ -Regeln ersetzen die quantifizierte Variable durch einen Term der Form $f(x_1, \dots, x_n)$. f ist ein neues Funktionssymbol und x_1, \dots, x_n sind die freien Variablen der Formel F .

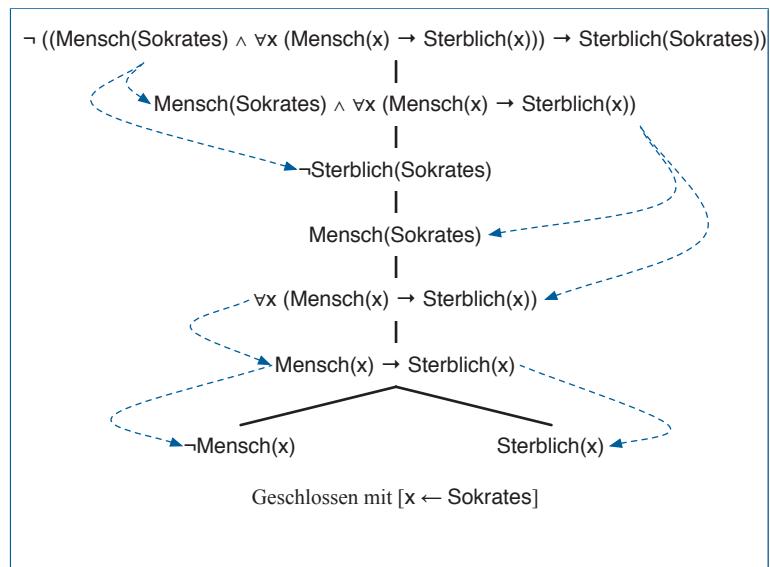


Abbildung 3.50: Sokrates ist sterblich!
Diesmal bewiesen im Tableaukalkül.

Reihenfolge der Regelanwendungen auf die Größe des entstehenden Tableaus aus. Um die Anzahl der Verzweigungen so gering wie möglich zu halten, sind α -Regeln immer vor β -Regeln anzuwenden. Ferner ist es unnötig, die gleiche Formel mehrmals mit einer α -, β - oder δ -Regel zu bearbeiten. Zusätzliche Expansionen vergrößern das Tableau, generieren aber niemals neue Abschlussmöglichkeiten. Anders verhält es sich mit den γ -Regeln, da ein prädikatenlogisches Tableau in vielen Fällen nur dann geschlossen werden kann, wenn Formeln des γ -Typs mehrmals expandiert wurden. Wie häufig die γ -Regeln angewendet werden müssen, bevor sich ein Tableau schließen lässt, kann nicht vorhergesagt werden. Hierdurch wird der Tableaukalkül zu einem Semi-Entscheidungsverfahren, das uns nur für allgemeingültige Formeln eine sichere Aussage erlaubt.

■ Schließen eines Pfads

Ein Pfad kann geschlossen werden, wenn er zwei Formeln F_1 und $\neg F_2$ enthält und F_1 und F_2 unifizierbar sind. Um den Abschluss durchzuführen, berechnen wir zunächst den allgemeinsten Unifikator σ von F_1 und F_2 und wenden diesen anschließend auf alle bisher erzeugten Formeln an. Danach kann das Tableau weiter expandiert oder ein anderer Pfad geschlossen werden. Beachten Sie, dass der Unifikator σ auf das gesamte Tableau angewendet wird und nicht nur auf die Formeln des aktuell zu schließenden Pfads.

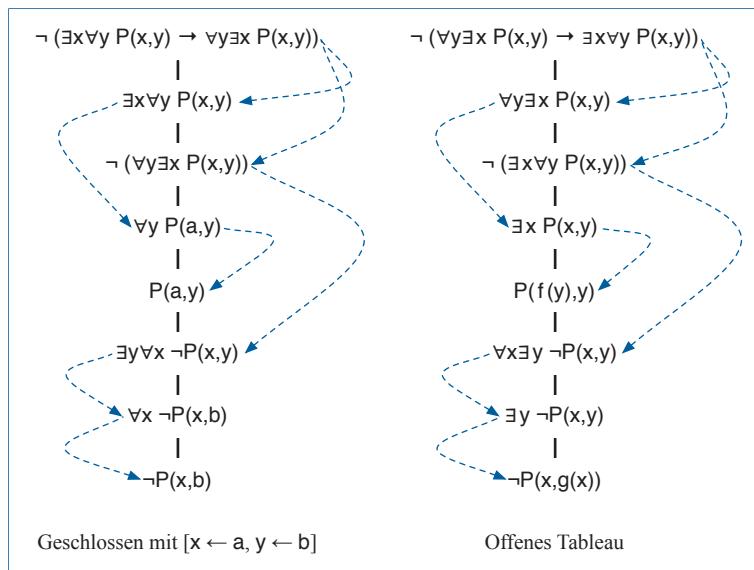


Abbildung 3.51: Zwei prädikatenlogische Tableaus. Das linke Tableau ist geschlossen und die Ausgangsformel als unerfüllbar identifiziert. Um das rechte Tableau zu schließen, müssten wir x mit $f(y)$ und gleichzeitig y mit $g(x)$ ersetzen. Eine solche Substitution existiert nicht und das Tableau kann nicht geschlossen werden.

Abbildung 3.50 demonstriert, wie die Sterblichkeit von Sokrates mithilfe des Tableaukalküls formal bewiesen werden kann. Zunächst wird die Eingabeformel mit den beiden α -Regeln in ihre konjunktiven Bestandteile zerlegt. Anschließend wird der Allquantor mithilfe der γ -Regel eliminiert und der verbleibende Implikationsoperator mit der β -Regel aufgelöst. Als Ergebnis entsteht ein Tableau, das mit der Substitution $\sigma = [x \leftarrow \text{Sokrates}]$ geschlossen werden kann.

Die Anwendung des Tableaukalküls wollen wir an zwei weiteren Beispielen demonstrieren. Dabei greifen wir auf die beiden Formeln zurück, die uns bereits in der Diskussion über den Resolutionskalkül als fruchtbare Anschauungsobjekte dienten:

$$\begin{aligned} F_1 &:= \exists x \forall y P(x,y) \rightarrow \forall y \exists x P(x,y) \\ F_2 &:= \forall y \exists x P(x,y) \rightarrow \exists x \forall y P(x,y) \end{aligned}$$

F_1 ist allgemeingültig und konnte bereits erfolgreich als eine solche Formel identifiziert werden. F_2 ist nicht allgemeingültig, da auf der linken Seite der Implikation eine stärkere Aussage steht als auf der rechten. Abbildung 3.51 zeigt die expandierten Tableaus für die Formeln F_1 und F_2 . Das linke Tableau kann mit der Substitution $\sigma = [x \leftarrow a, y \leftarrow b]$ geschlossen werden. Wenden wir σ auf das Tableau an, so entsteht auf dem einzigen vorhandenen Pfad das komplementäre Formelpaar $P(a,b), \neg P(a,b)$. Im Gegensatz hierzu lässt sich das Tableau für die Formel F_2 nicht schließen, da mit $P(f(y),y)$ und $P(x,g(x))$ genau die-

jenigen Terme entstehen, die wir in Abbildung 3.43 als nicht unifizierbar erkannt haben. Weitere γ -Regelanwendungen können die Situation nicht ändern, da wir das Tableau lediglich mit weiteren Instanzen der gleichen Form anreichern und keine neuen Abschlussmöglichkeiten erzeugen würden.

3.2.4 Anwendung: Logische Programmierung

War die Prädikatenlogik ursprünglich ein Teilbereich der reinen Mathematik, so erhielt sie im Zuge der aufkeimenden Computertechnik eine ganz praktische Bedeutung: Sie ist die theoretische Grundlage der Programmiersprache Prolog (*PRO*gramming in *LOG*ic). Prolog wurde Anfang der Siebzigerjahre entwickelt und ist der bekannteste Vertreter des *deklarativen Programmierparadigmas* [63]. Deklarative Programmiersprachen stellen die Problemformulierung und nicht die Lösungsstrategie in den Vordergrund; sie sind von der Idee getrieben, der Computer möge die Lösung aus der Beschreibung selbstständig deduzieren. Die Vorgehensweise unterscheidet sich damit fundamental von jener der imperativen oder der objektorientierten Programmiersprachen. Dort wird der Lösungsweg detailliert durch den Entwickler vorgegeben.

Ein Prolog-Programm ist aus *Fakten* und *Regeln* aufgebaut. Was sich hinter diesen Begriffen genau verbirgt, wollen wir am Beispiel des Stammbaums aus Abbildung 3.52 herausarbeiten. Dargestellt sind die Verwandtschaftsverhältnisse einiger Charaktere aus J. R. R. Tolkiens Fantasy-Epos *Herr der Ringe*.

Ein Blick auf den Stammbaum reicht aus, um beispielsweise die folgenden Verwandtschaftsbeziehungen zu erkennen:

- „Elboron ist Eowyns Sohn“
- „Eowyn ist Eomunds Tochter“

Beide Sachverhalte werden innerhalb eines Prolog-Programms mithilfe zweier Fakten formuliert:

- `sohn(elboron,eowyn).`
- `tochter(eowyn,eomund).`

Die Prolog-Syntax deutet unmissverständlich an: Mit Fakten werden Prädikate beschrieben. Konkret legen die beiden Anweisungen fest,

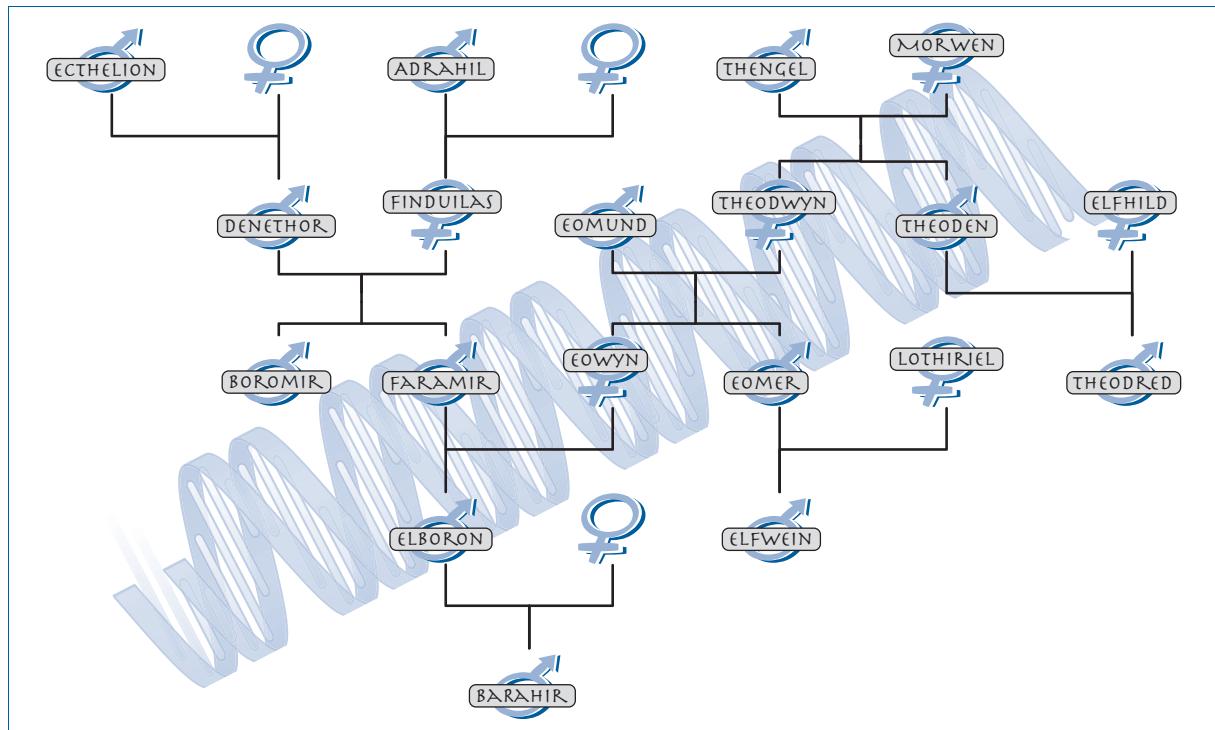


Abbildung 3.52: Kleiner Ausschnitt aus dem Stammbaum der Charaktere der Fantasy-Trilogie *Herr der Ringe*

dass das Prädikat `sohn` für die Kombination (`elboron,eowyn`) und das Prädikat `tochter` für die Kombination (`eowyn, eomund`) wahr ist.

Regeln werden in Prolog eingesetzt, um Prädikate miteinander in Beziehung zu setzen. So können wir die Eigenschaft, ein Kind zu sein, in direkter Weise auf die Sohn-Tochter-Beziehung zurückführen: X ist ein Kind von Y , falls X ein Sohn von Y ist oder X eine Tochter von Y ist. In Prolog können wir die Beziehung wie folgt beschreiben:

- `kind(X,Y) :- sohn(X,Y).`
- `kind(X,Y) :- tochter(X,Y).`

Beachten Sie die Groß- und Kleinschreibung! Prädikate und Individuen beginnen in Prolog immer mit einem Kleinbuchstaben, Variablen mit einem Großbuchstaben. Aufweichungen dieser Regel existieren nicht, da Prolog über keine andere Möglichkeit verfügt, Variablen von Konstanten zu unterscheiden.

stammbaum.pl

```

sohn(denethor,ecthelion).
sohn(boromir,denethor).
sohn(boromir,finduilas).
sohn(faramir,denethor).
sohn(faramir,finduilas).
sohn(elboron,faramir).
sohn(elboron,eowyn).
sohn(barahir,elboron).
sohn(eomer,eomund).
sohn(eomer,theodwyn).
sohn(elfwein,eomer).
sohn(elfwein,lothiriel).
sohn(theoden,thengel).
sohn(theoden,morwen).
sohn(theodred,theoden).
sohn(theodred,elfhild).
tochter(finduilas,adrahil).
tochter(eowyn,eomund).
tochter(eowyn,theodwyn).
tochter(theodwyn,thengel).
tochter(theodwyn,morwen).

kind(X,Y) :- sohn(X,Y).
kind(X,Y) :- tochter(X,Y).

nachfahre(X,Y) :- kind(X,Y).
nachfahre(X,Y) :- kind(X,Z),
                 nachfahre(Z,Y).

```

Abbildung 3.53: Prolog-Programm zur Beschreibung des Stammbaums aus Abbildung 3.52

Die Kindbeziehung können wir nutzen, um weitere Prädikate zu definieren. Beispielsweise lässt sich die Eigenschaft, ein Nachfahre einer anderen Person zu sein, wie folgt charakterisieren: X ist ein Nachfahre einer Person Y , wenn X das Kind von Y oder das Kind einer anderen Person ist, die ihrerseits ein Nachfahre von Y ist. In Prolog drücken wir diesen Zusammenhang folgendermaßen aus:

- 1 6 ■ nachfahre(X,Y) :- kind(X,Y).
- 2 7 ■ nachfahre(X,Y) :- kind(X,Z), nachfahre(Z,Y).
- 3 8
- 4 9
- 5 10
- 6 11

Prolog-Programme lassen sich mit einem gewöhnlichen Texteditor erstellen. Anschließend wird der Interpreter gestartet und das Programm mit dem vordefinierten Befehl `consult('...')` eingelesen. Die folgenden Versuche wurden mit dem freien Interpreter GNU Prolog durchgeführt:

```

17     GNU Prolog 1.3.0
18     By Daniel Diaz
19     Copyright (C) 1999–2007 Daniel Diaz
20     | ?- consult('stammbaum.pl').
21     compiling stammbaum.pl for byte code...
22     stammbaum.pl compiled,
23     28 lines read – 2938 bytes written, 11 ms
24     yes
25     | ?-
26
27
28

```

Der Interpreter übersetzt die eingelesenen Fakten und Regeln zunächst in Byte-Code und fügt sie anschließend in die Wissensbasis ein. Der Quelltext des eingelesenen Programms ist in Abbildung 3.53 dargestellt und enthält eine komplette Beschreibung des Beispielstammbaums aus Abbildung 3.52 in Prolog-codierter Form.

Nachdem das Programm erfolgreich eingelesen wurde, ist der Interpreter bereit, Anfragen entgegenzunehmen. Beispielsweise können wir die Frage stellen, ob Elboron ein Nachfahre von Denethor ist.

```

:- nachfahre(elboron,denethor).
yes

```

Wie zu erwarten, wird die Anfrage bejaht. Anders verhält sich der Interpreter in dem folgenden Fall:

```

:- nachfahre(eowyn, ecthelion).
no

```

Dass die Anfrage verneint wird, überrascht nicht, schließlich ist Eowyn kein Nachfahre von Ecthelion.

Die Fähigkeiten von Prolog gehen weit über die Generierung von Ja-nein-Antworten hinaus. Indem wir Anfragen mit Variablen versehen, können wir Prolog dazu bewegen, mit konkreten Instanzen zu antworten. Beispielsweise können wir mit der nachstehenden Anfrage alle Nachfahren von Denethor berechnen:

```
:– nachfahre(X,denethor).
```

Der Prolog-Interpreter erzeugt die folgende Antwort:

```
X = boromir ?
```

In der Tat ist Boromir ein Nachfahre von Denethor; aber sind Faramir, Elboron und Barahir nicht ebenfalls Nachfahren? Die Antwort lautet „Ja“ und Prolog ist imstande, diese ebenfalls auszugeben. Wir müssen den Interpreter lediglich anweisen, *alle* Lösungen zu berechnen. Hierzu genügt es, nach dem Fragezeichen den Buchstaben 'a' einzugeben. Prolog reagiert darauf mit der folgenden Ausgabe:

```
X = faramir
X = elboron
X = barahir
(1 ms) no
```

Das Wort „no“ in der letzten Zeile deutet an, dass keine weiteren Lösungen existieren.

Die Beispiele in Abbildung 3.54 zeigen, dass wir die Anfrage auch umgekehrt stellen können, indem die Variable *X* an die zweite Position gerückt wird. In diesem Fall berechnet der Prolog-Interpreter alle *Vorfahren* der im ersten Argument spezifizierten Person.

Wir können sogar noch einen Schritt weiter gehen und den Interpreter mit der Anfrage *nachfahre(X,Y)* aktivieren. In diesem Fall werden alle Personenpaare (*X,Y*) erzeugt, in denen *X* ein Nachfahre von *Y* ist (vgl. Abbildung 3.55). Anfragen lassen sich auch kombinieren, wie das Beispiel in Abbildung 3.56 demonstriert. Durch die zusätzliche Angabe des Prädikats *tochter(X,_)* wird die Ergebnisliste auf alle weiblichen Nachfahren eingeschränkt. Der spezielle Bezeichner *_* steht für eine willkürliche Variable, für deren Wert wir uns nicht interessieren.

■ Beispiel 1

Prolog-Console

```
?– nachfahre(boromir,X).
1
2
3
4
5
6
7
8
9
```

X = denethor
X = finduilas
X = ecthelion
X = adrahil
(1 ms) no

■ Beispiel 2

Prolog-Console

```
?– nachfahre(elfwein,X).
1
2
3
4
5
6
7
8
9
```

X = eomer
X = lothiriel
X = eomund
X = theodwyn
X = thengel
X = morwen
(1 ms) no

■ Beispiel 3

Prolog-Console

```
?– nachfahre(elfhild,X).
1
2
3
4
5
6
7
8
9
```

no

Abbildung 3.54: Weitere Anfragen an den Prolog-Interpreter

Prolog-Console

```
?- nachfahre(X,Y).
X = denethor
Y = ecthelion

X = boromir
Y = denethor

X = boromir
Y = finduilas

X = faramir
Y = denethor

X = faramir
Y = finduilas

X = elboron
Y = faramir

X = elboron
Y = eowyn

X = barahir
Y = elboron

X = eomer
Y = eomund

X = eomer
Y = theodwyn

X = elfwein
Y = eomer

X = elfwein
Y = lothiriel

...

```

Interne Arbeitsweise von Prolog

Nachdem wir einen ersten Eindruck über die Leistungsfähigkeit von Prolog erhalten haben, wollen wir der internen Arbeitsweise dieser Sprache etwas genauer auf den Grund gehen.

Intern behandelt der Prolog-Interpreter Fakten und Regeln als prädikatenlogische Formeln, die gemeinsam die Wissensbasis bilden. Die wahre Gestalt dieser Formeln wird sofort ersichtlich, wenn wir die Prolog-Notation für den Moment beiseitelassen und auf die vertrauten Logiksymbole zurückgreifen. Die weiter oben definierten Logikformeln lesen sich dann wie folgt:

- sohn(elboron, eowyn)
- tochter(eowyn, eomund)
- ...
- $\forall X \forall Y (\text{sohn}(X, Y) \rightarrow \text{kind}(X, Y))$
- $\forall X \forall Y (\text{tochter}(X, Y) \rightarrow \text{kind}(X, Y))$
- $\forall X \forall Y (\text{kind}(X, Y) \rightarrow \text{nachfahre}(X, Y))$
- $\forall X \forall Y (\text{kind}(X, Z) \wedge \text{nachfahre}(Z, Y) \rightarrow \text{nachfahre}(X, Y))$

Auf den ersten Blick wirken die Ausdrücke wie ganz normale Formeln; auf den zweiten Blick wird deutlich, dass sie über eine spezielle Struktur verfügen. Um diese sichtbar zu machen, transformieren wir die Formeln zunächst in Klauseldarstellung. Wenigen Umformungsschritte reichen hierfür aus:

- $\{\text{sohn}(\text{elboron}, \text{eowyn})\}$
- $\{\text{tochter}(\text{eowyn}, \text{eomund})\}$
- ...
- $\{\neg\text{sohn}(X, Y), \text{kind}(X, Y)\}$
- $\{\neg\text{tochter}(X, Y), \text{kind}(X, Y)\}$
- $\{\neg\text{kind}(X, Y), \text{nachfahre}(X, Y)\}$
- $\{\neg\text{kind}(X, Z), \neg\text{nachfahre}(Z, Y), \text{nachfahre}(X, Y)\}$

Abbildung 3.55: Auf die gestellte Anfrage antwortet der Prolog-Interpreter mit allen Personenpaaren, in denen X ein Nachfahre von Y ist.

Ein Blick auf die generierte Menge macht deutlich, dass keine Klausel mehr als ein positives Literal besitzt. Formeln mit dieser Eigenschaft wurden Anfang der Fünfzigerjahre durch den US-amerikanischen Logiker Alfred Horn ausführlich untersucht und tragen heute seinen Namen.



Definition 3.25 (Horn-Formel, Horn-Klausel)

Eine aussagenlogische oder prädikatenlogische Formel F ist eine *Horn-Formel*, wenn sie die folgende Klauseldarstellung besitzt:

$$\{(\neg)L_1, \neg L_2, \neg L_3, \dots, \neg L_n\}$$

Die Menge der Horn-Klauseln besitzt mehrere interessante Eigenschaften. Zum einen ist sie bezüglich Resolventenbildung abgeschlossen, d. h., die Resolvente zweier Horn-Klauseln ist wiederum eine Horn-Klausel. Darüber hinaus lassen sich Resolutionsbeweise besonders einfach finden, da sich der Ableitungsgraph in Form einer linearen Kette entwickeln lässt.

Am Beispiel der Anfrage

`:– nachfahre(elboron,denethor)`

wollen wir die Arbeitsweise des Interpreters demonstrieren. Bezeichnen wir die Wissensbasis mit M , so versucht Prolog, die Allgemeingültigkeit der Formel

$$M \rightarrow \text{nachfahre}(\text{elboron}, \text{denethor})$$

mithilfe der prädikatenlogischen Resolution zu beweisen. Hierzu erzeugt der Interpreter zunächst die negierte Formel

$$\begin{aligned} & \neg(M \rightarrow \text{nachfahre}(\text{elboron}, \text{denethor})) \\ & \equiv M \wedge \neg \text{nachfahre}(\text{elboron}, \text{denethor}) \end{aligned}$$

und übersetzt diese anschließend in Klauselform:

$$M \cup \{\neg \text{nachfahre}(\text{elboron}, \text{denethor})\}$$

Auf dieser Menge führt Prolog einen prädikatenlogischen Resolutionsbeweis durch. Die Anfrage war erfolgreich, wenn es dem Interpreter gelingt, die leere Klausel abzuleiten. In diesem Fall antwortet das System mit yes und gibt die im Rahmen der Resolventenbildungen angewendeten Variablensubstitutionen aus. Kann die leere Klausel nicht abgeleitet werden, so antwortet der Interpreter mit no.

Abbildung 3.57 zeigt den entstehenden Resolutionsgraphen für unsere Beispieldanfrage. Die erste Resolvente wird immer mit der Anfrageklausel gebildet und das Ergebnis anschließend so lange weiter resolviert, bis die leere Klausel abgeleitet werden konnte oder die Ableitung in

Prolog-Console

?– nachfahre(X,Y),	1
tochter(X,_).	2
X = finduilas	3
Y = adrahil	4
X = eowyn	5
Y = eomund	6
X = eowyn	7
Y = eomund	8
X = eowyn	9
Y = theodwyn	10
X = eowyn	11
Y = theodwyn	12
X = eowyn	13
Y = theodwyn	14
X = eowyn	15
Y = theodwyn	16
X = theodwyn	17
Y = thengel	18
X = theodwyn	19
Y = thengel	20
X = theodwyn	21
Y = thengel	22
X = theodwyn	23
Y = morwen	24
X = theodwyn	25
Y = morwen	26
X = theodwyn	27
Y = morwen	28
X = eowyn	29
Y = thengel	30
X = eowyn	31
Y = thengel	32
X = eowyn	33
Y = thengel	34
X = eowyn	35
Y = morwen	36
X = eowyn	37
Y = morwen	38
X = eowyn	39
Y = morwen	40
X = eowyn	41

Abbildung 3.56: Auf die gestellte Anfrage antwortet der Prolog-Interpreter mit allen Personenpaaren, in denen X ein weiblicher Nachfahre von Y ist.

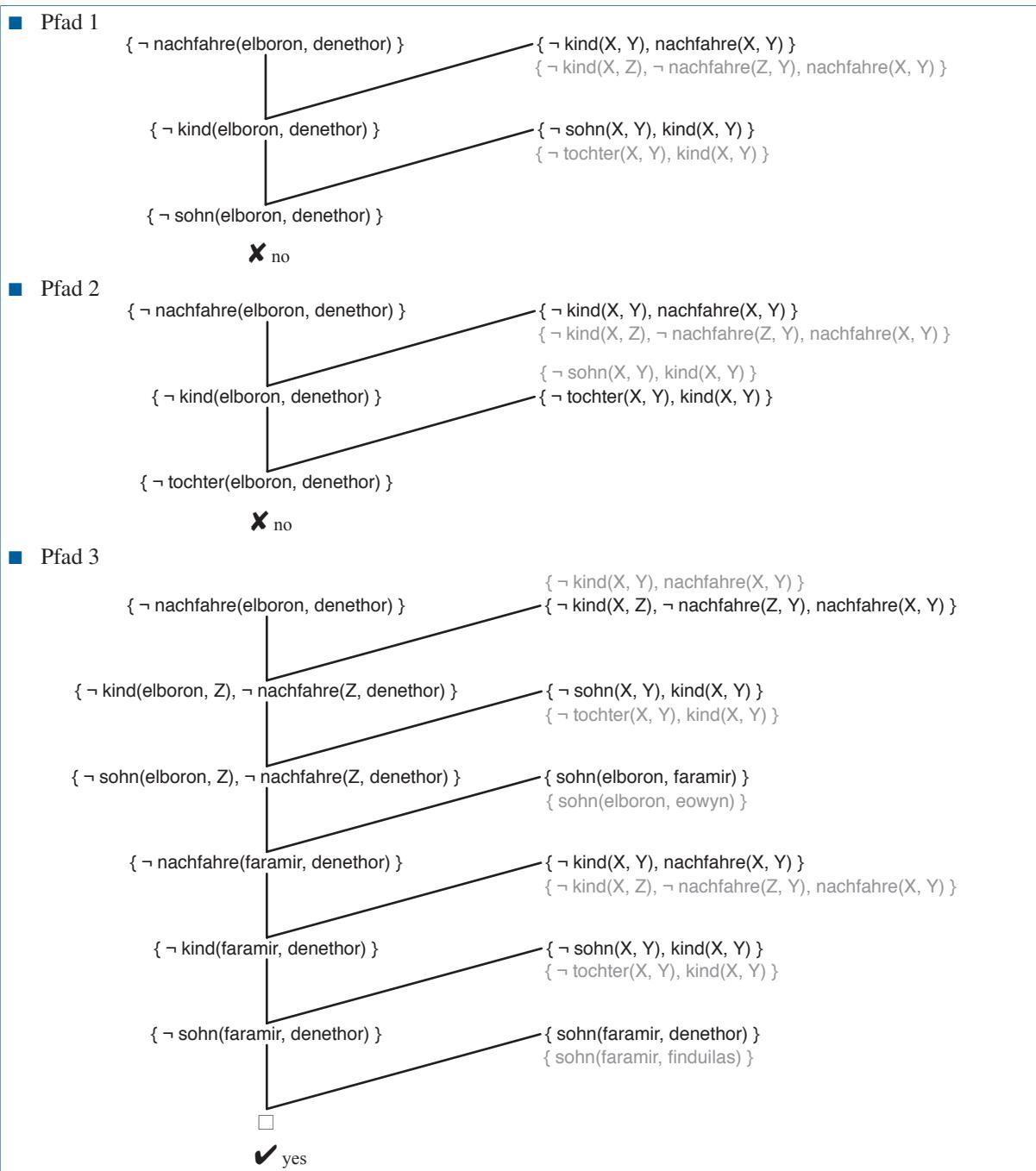


Abbildung 3.57: Intern führt der Prolog-Interpreter einen Resolutionsbeweis durch.

eine Sackgasse gerät. Während der Ableitung existieren an verschiedenen Stellen Entscheidungspunkte, an denen die Resolventenbildung ohne weitere Vorgaben indeterministisch wäre. Zum einen muss der Interpreter entscheiden, welches Literal als Nächstes zu eliminierten ist; zum anderen, mit welcher Klausel der Wissensbasis die neue Resolvente gebildet werden soll. Prolog legt die Vorgehensweise wie folgt fest: Zum einen wird immer das am weitesten links stehende Literal in der Anfrage bzw. in den resolvierten Nachfolgern eliminiert. Zum anderen werden die Regeln der Wissensbasis in derselben Reihenfolge berücksichtigt, wie sie im Programmtext angeordnet sind.

Mit diesen Vorgaben wird die Resolventenbildung zu einem deterministischen Prozess, der für die Anfrage `nachfahre(elboron,denethor)` zunächst den in Abbildung 3.57 (oben) gezeigten Resolutionsbaum hervorbringt. Nach zwei Schritten gerät die Ableitung in eine Sackgasse, da sich die Klausel `{¬sohn(elboron,denethor)}` nicht weiter resolvieren lässt. In diesem Fall kehrt Prolog an den letzten Entscheidungspunkt zurück und revidiert die getroffene Auswahl. In der Terminologie der logischen Programmierung wird dieser Vorgang als *Backtracking* bezeichnet. In unserem Beispiel bildet der Interpreter die zweite Resolvente mit der Klausel `{¬tochter(X,Y), kind(X,Y)}` (vgl. Abbildung 3.57 Mitte), gerät jedoch wiederum in eine Sackgasse. Nach zweimaligem Backtracking wird der Beweis im dritten Anlauf schließlich gefunden. Der Interpreter kann die leere Klausel in 6 Resolutionsschritten erfolgreich ableiten und beantwortet die Anfrage mit 'yes' (vgl. Abbildung 3.57 unten).

Die Leistungsfähigkeit von Prolog geht weit über die hier vorgestellten Möglichkeiten hinaus. Heute existieren professionelle Entwicklungsumgebungen, die ein komfortables Arbeiten ermöglichen und den Programmierer durch eine Vielzahl vorgefertigter Bibliotheken unterstützen. Hiermit lassen sich mit wenigen Zeilen Code grafische Applikationen entwickeln, deren Herkunft von außen nicht mehr sichtbar ist.

Trotzdem ist der logischen Programmierung der große Durchbruch verwehrt geblieben. Wurde Prolog in den Achtzigerjahren noch als zukunftsweisende Programmiersprache der „fünften Generation“ bezeichnet, ist der damalige Enthusiasmus heute weitgehend abgeebbt. In der Praxis funktioniert die Idee, das Problem und nicht den Lösungsweg zu beschreiben, weniger gut als in der Theorie erhofft. Trotzdem konnte Prolog in einigen Disziplinen wie z. B. der künstlichen Intelligenz seinen Platz behaupten. In diesem Bereich wird die Sprache seit vielen Jahren mit Erfolg eingesetzt.

3.3 Logikerweiterungen

Zum Ende dieses Kapitels wollen wir in einem knappen Rundumschlag skizzieren, wie sich die Prädikatenlogik erster Stufe in verschiedene Richtungen erweitern lässt. Als erstes diskutieren wir in Abschnitt 3.3.1 die Prädikatenlogik mit Gleichheit. Anschließend werden wir die uns vertraute PL1 zu einer *Logik höherer Stufe* erweitern, indem wir explizit erlauben, dass die Quantoren \forall und \exists auch auf Prädikate und Funktionen angewendet werden dürfen. Auf diesem Weg werden wir zu einer Logik geführt, die uns in vielerlei Hinsicht positiv, aber auch in mancher Hinsicht negativ überraschen wird. Die Logiken höherer Stufe bilden eine ganz eigene Welt, doch wir werden die Tür dorthin nur einen Spalt weit öffnen können. Für eine ausführlichere Diskussion sei der Leser auf [50] verwiesen.

■ Formel $F_{\geq n}$

$$\exists x_1 \dots \exists x_n \bigwedge_{\substack{1 \leq i, j \leq n \\ i \neq j}} \neg x_i = x_j$$

„Der Individuenbereich umfasst mindestens n Elemente“



$$|U| \geq n$$

■ Formel $F_{\leq n}$

$$\exists x_1 \dots \exists x_n \forall y \bigvee_{i=1}^n y = x_i$$

„Der Individuenbereich umfasst höchstens n Elemente“



$$|U| \leq n$$

■ Formel $F_{=n}$

$$\exists x_1 \dots \exists x_n \bigwedge_{\substack{1 \leq i, j \leq n \\ i \neq j}} \neg x_i = x_j$$

$$\wedge \exists x_1 \dots \exists x_n \forall y \bigvee_{i=1}^n y = x_i$$

„Der Individuenbereich umfasst exakt n Elemente“



$$|U| = n$$

Abbildung 3.58: Die Formeln $F_{\geq n}$ und $F_{\leq n}$ sind genau dann wahr, wenn der Individuenbereich mindestens n bzw. höchstens n umfasst. Folgerichtig ist die Formel $F_{=n}$ unter genau jenen Interpretationen wahr, deren Individuenbereich genau n Elemente enthält.

3.3.1 Prädikatenlogik mit Gleichheit

In den vorangegangenen Abschnitten haben wir die Prädikatenlogik erster Stufe (PL1) als leistungsfähiges Instrument kennen gelernt, um logische Sachverhalte formal zu beschreiben. Dass diese Logik auch Schwächen hat, werden wir nun an einem konkreten Beispiel herausarbeiten. Wir wollen versuchen, eine prädikatenlogische Formel $F_{=3}$ zu konstruieren, die genau unter jenen Interpretationen wahr ist, die einen Individuenbereich mit genau drei Elementen aufweisen:

$$(U, I) \models F_{=3} : \Leftrightarrow |U| = 3 \quad (3.14)$$

Wir wollen uns an das Problem schrittweise herantasten und zunächst nach Formeln $F_{\geq 3}$ und $F_{\leq 3}$ suchen, die genau dann wahr sind, wenn der Individuenbereich U mindestens 3 bzw. höchstens 3 Elemente enthält. Sobald wir solche Formeln in Händen halten, können wir $F_{=3}$ ganz einfach so formulieren:

$$F_{=3} := F_{\leq 3} \wedge F_{\geq 3}$$

Nehmen Sie sich ruhig etwas Zeit und versuchen Sie, Formeln mit den gewünschten Eigenschaften zu finden. Sie werden bemerken, dass die Niederschrift einer passenden Formel nur dann zu gelingen scheint, wenn wir auf einen Baustein zurückgreifen, der in der Prädikatenlogik zunächst nicht zur Verfügung steht: die Gleichheit.

Integrieren wir den Gleichheitsbeziehung in Form eines speziellen Prädikatzeichens $\dot{=}$ in die Prädikatenlogik, so können wir $F_{\geq 3}$ und $F_{\leq 3}$ folgendermaßen formulieren (vgl. Abbildung 3.58):

$$\begin{aligned} F_{\geq 3} &:= \exists x_1 \exists x_2 \exists x_3 (\neg x_1 \dot{=} x_2 \wedge \neg x_1 \dot{=} x_3 \wedge \neg x_2 \dot{=} x_3) \\ F_{\leq 3} &:= \exists x_1 \exists x_2 \exists x_3 \forall y (y \dot{=} x_1 \vee y \dot{=} x_2 \vee y \dot{=} x_3) \end{aligned}$$

Um die Prädikatenlogik mit Gleichheit formal zu definieren, müssen wir lediglich zwei Definitionen aus Abschnitt 3.2.1 geringfügig anpassen. Die erste Änderung betrifft den Aufbau atomarer Formeln. Um das Gleichheitszeichen syntaktisch in der Logik zu verankern, erweitern wir Definition 3.15 um die Zeile

- Sind t_1 und t_2 Terme, so ist $(t_1 \dot{=} t_2)$ eine atomare Formel.

Ferner fügen wir in Definition 3.17 eine Zeile ein, die dem Gleichheitsoperator $\dot{=}$ seine intendierte Bedeutung verleiht:

$$(U, I) \models (t_1 \dot{=} t_2) : \Leftrightarrow I(t_1) = I(t_2)$$

Vielleicht haben Sie sich die Frage gestellt, wie wichtig die Integration der Gleichheit ist. Handelt es sich dabei lediglich um ein Mittel des Komforts, mit dem sich einige Formeln eleganter niederschreiben lassen? Die Antwort ist Nein! Es ist ein wichtiges Metarésultat der Prädikatenlogik, dass eine Formel erster Stufe mit der Eigenschaft (3.14) nicht existieren kann; Sie hatten also gar keine Chance eine solche zu finden [50]. Das bedeutet, dass die Integration des Gleichheitszeichens mehr ist als eine Frage der Ästhetik; sie führt uns auf direkten Weg zu einer Logik, die ausdrucksstärker ist als die Prädikatenlogik erster Stufe aus Abschnitt 3.2.

3.3.2 Logiken höherer Stufe

Die Formeln der Prädikatenlogik erster Stufe sind so aufgebaut, dass wir mit den Quantoren \exists und \forall ausschließlich über die Elemente der Individuenmenge U quantifizieren können, nicht aber über Prädikate und Funktionen. Heben wir diese Beschränkung auf, so gelangen wir auf direktem Weg zur *Prädikatenlogik zweiter Stufe (second-order logic)*, kurz PL2. Neben den uns bereits bekannten Variablen existieren dort zwei weitere Variablenarten. Die Variablen des ersten Typs bezeichnen wir als *Prädikatvariablen* (P, Q, R, \dots), die des zweiten als *Funktionsvariablen* (f, g, h, \dots). Jede dieser Variablen besitzt eine festgelegte Stelligkeit und darf überall dort eingesetzt werden, wo in prädikatenlogischen Ausdrücken erster Stufe ein Prädikatzeichen bzw. ein Funktionssymbol mit der gleichen Stelligkeit auftauchen darf.

Mithilfe der neuen Variablen sind wir auf natürliche Weise in der Lage, über Prädikate und Funktionen zu quantifizieren:

$\forall P \dots \hat{=} „Für alle Prädikate gilt: …“$

$\exists P \dots \hat{=} „Es existiert ein Prädikat, für das gilt: …“$

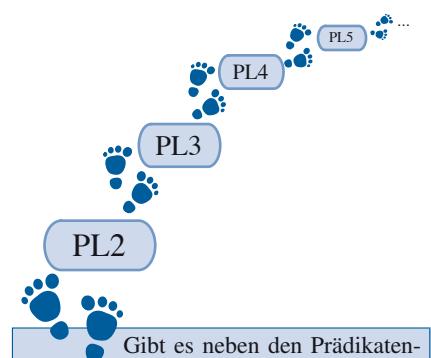
$\forall f \dots \hat{=} „Für alle Funktionen gilt: …“$

$\exists f \dots \hat{=} „Es existiert eine Funktion, für die gilt: …“$

Durch die gewonnene Bewegungsfreiheit haben wir jetzt keinerlei Probleme mehr, um die weiter oben gesuchte Formel $F_{\geq 3}$ zu definieren:

$$\exists x_1 \exists x_2 \exists x_3 \exists P \exists Q (P(x_1) \wedge \neg P(x_2) \wedge \neg P(x_3) \wedge Q(x_2) \wedge \neg Q(x_3))$$

Der Ausdruck $P(x_1) \wedge \neg P(x_2) \wedge \neg P(x_3)$ garantiert, dass x_1 von x_2 und x_3 verschieden sein muss und $Q(x_2) \wedge \neg Q(x_3)$ stellt sicher, dass x_2 und x_3 ebenfalls unterschiedliche Elemente repräsentieren.



Gibt es neben den Prädikatenlogiken erster und zweiter Stufe noch weitere Logiken, z. B. eine Prädikatenlogik dritter Stufe, kurz PL3? Die Antwort ist Ja! Warum dies so sein muss, ist leicht einzusehen, wenn wir Prädikate von einem mengentheoretischen Standpunkt aus betrachten. Ein einstelliges Prädikatzichen P repräsentiert dann eine spezielle Teilmenge des Individuenbereichs; es ist jene Teilmenge, die genau jene Elemente enthält, für die P wahr ist. Das bedeutet im Umkehrschluss, dass wir in der Prädikatenlogik zweiter Stufe über den Individuenbereich und Teilmengen des Individuenbereichs quantifizieren können. Erlauben wir, auch über Teilmengen von Teilmengen des Individuenbereichs zu quantifizieren, so erhalten wir die Prädikatenlogik dritter Stufe. Gehen wir den eingeschlagenen Weg weiter, so erreichen wir die PL4, die Prädikatenlogik vierter Stufe, und so setzt sich die Reise fort.

Die hohe Ausdrucksstärke der Prädikatenlogik zweiter Stufe ermöglicht uns, viele Sachverhalte elegant zu charakterisieren, die in der Prädikatenlogik erster Stufe nur umständlich oder überhaupt nicht beschrieben werden können. Mit der Gleichheit und den natürlichen Zahlen haben wir zwei prominente Beispiele bereits kennen gelernt. Tatsächlich zahlen wir dafür einen hohen Preis. Alle Logiken höherer Stufe verbindet der gemeinsame Makel, dass sie im Gegensatz zur PL1 oder der PL2 mit Gleichheit nicht mehr vollständig sind. Das bedeutet, dass es unmöglich ist, ein formales System zu konstruieren, in dem alle allgemeingültigen PL2-Formeln – und nur diese – aus den Axiomen abgeleitet werden können. Die Unvollständigkeit ist tatsächlich eine Konsequenz aus der hohen Ausdruckskraft. Sie folgt direkt aus dem ersten Gödel'schen Unvollständigkeitssatz und der Tatsache, dass sich die natürlichen Zahlen innerhalb der Prädikatenlogiken höherer Stufe eindeutig axiomatisieren lassen [50] (vgl. Abbildung 3.59).

In der Prädikatenlogik zweiter Stufe ist es auch gar nicht mehr nötig, die Gleichheitsrelation in Form eines speziellen Operators in die Logik zu integrieren. Die PL2 ist ausdrucksstark genug, um die Gleichheit *innerhalb* der Logik zu definieren. Das bedeutet, dass wir eine Formel $F_=_$ finden können, die genau unter jenen Interpretationen wahr ist, die ein bestimmtes Prädikatzeichen, z. B. P, als die Gleichheitsrelation interpretieren. Den Aufbau dieser Formel können wir über die folgenden Überlegung herleiten: Werden die Variablen x und y als zwei verschiedene Individuenelemente interpretiert, so existiert eine Relation, die beide voneinander separiert. Verbirgt sich hinter x und y dagegen das gleiche Element, so können sie durch keine Relation unterschieden werden. Jetzt ist klar, wie wir $F_=_$ aufschreiben können:

$$F_=_ := \forall x \forall y (P(x,y) \leftrightarrow \forall R (R(x) \leftrightarrow R(y)))$$

Diese Formel ist genau dann wahr, wenn die zweistellige Prädikatvariable P als die Gleichheitsrelation interpretiert wird.

Die PL2 ist nicht nur ausdrucksstärker als die PL1, sondern auch ausdrucksstärker als die PL1 mit Gleichheit. Erst die PL2 verfügt über die nötigen Mittel, um die Menge der natürlichen Zahlen N eindeutig zu charakterisieren. Verantwortlich hierfür ist das fünfte Peano-Axiom, das Axiom der vollständigen Induktion. In Abschnitt 2.4.1 haben wir die vollständige Induktion neben dem direkten Deduktionsbeweis und dem indirekten Widerspruchsbeweis als dritte grundlegende Beweistechnik der Mathematik kennen gelernt. Dort haben wir herausgearbeitet, dass sich die Induktionstechnik auf alle Aussagen anwenden lässt, die von einem natürlichezähligen Parameter n abhängen und für alle n gezeigt werden sollen. Um eine solche Aussage zu beweisen, sind wir in drei Schritten vorgegangen. Im Induktionsanfang haben wir die Behauptung zunächst für den Fall $n = 0$ überprüft. Anschließend nahmen wir an, dass die Aussage für einen beliebigen Wert $n \in \mathbb{N}$ gilt, und versuchten zu zeigen, dass sich die Gültigkeit der Behauptung auf den Fall $n + 1$ überträgt. Gelingt dieser Beweis, so garantiert uns das fünfte Peano-Axiom, dass die Aussage für ausnahmslos alle natürlichen Zahlen erfüllt ist. In der Prädikatenlogik zweiter Stufe lässt sich das Axiom auf natürliche Weise beschreiben:

$$\forall P ((P(0) \wedge \forall n (P(n) \rightarrow P(succ(n)))) \rightarrow \forall n P(n))$$

Da der erste Quantor auf ein Prädikat angewendet wird, lässt sich die Formel nicht mit den Mitteln der traditionellen Prädikatenlogik formulieren.

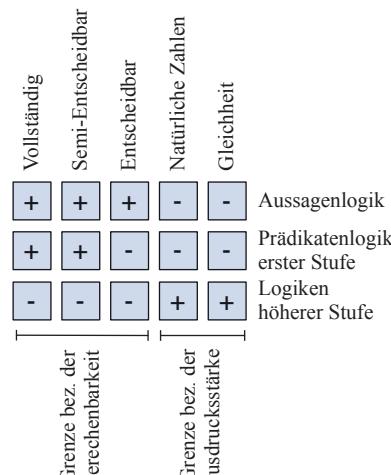


Abbildung 3.59: Berechenbarkeitseigenschaften und Ausdrucksstärke der vorgestellten Logiken

3.3.3 Typentheorie

In der Informatik werden Logiken höherer Stufe (*higher-order logics*) im Bereich der Software-Verifikation eingesetzt. Hier wird gerne auf Varianten zurückgegriffen, die auf dem typisierten Lambda-Kalkül basieren [20]. Diese Logiken zeichnen sich dadurch aus, dass keine strikte Trennung zwischen Termen und Prädikaten mehr vorgenommen wird; es existieren nur noch Terme, die nach gewissen Regeln zu komplexeren Termen kombiniert werden können. Die Verschmelzung beider Begriffe führt unter anderem dazu, dass jedes Prädikat als Argument von Funktionen und anderer Prädikate verwendet werden darf.

Im Bereich der Software-Verifikation wird das Ziel verfolgt, die funktionale Korrektheit eines Software-Systems formal zu beweisen. Hierzu sind ein Programm – die Implementierung – sowie eine formale Spezifikation in Form einer Logikformel vorgegeben. Durch die formale Anwendung mathematischer Ableitungsregeln muss nun gezeigt werden, dass die Implementierung die Spezifikation erfüllt. Um die praktische Verwendung einer solchen Logik zu verdeutlichen, ist in Abbildung 3.60 eine einfache C-Funktion zusammen mit ihrer formalen Spezifikation dargestellt. Das Quellprogramm definiert eine Funktion `is_prime`, die für die Integer-Variablen `nr` entscheidet, ob sie eine Primzahl enthält oder nicht. Rechts daneben ist die Spezifikation dargestellt, formuliert in der Syntax des Theorembeweisers HOL [38]. Die Spezifikation definiert zunächst zwei Hilfsprädikate `divides` und `prime`, die zum einen die Teilbarkeitseigenschaft und zum anderen die Prim-Eigenschaft einer Zahl beschreiben. Das Verhalten der C-Funktion wird in Form eines mathematischen Theorems spezifiziert, das den Ergebniswert der Funktion `is_prime` mit dem Prädikat `prime` verknüpft.

Die hohe Ausdrucksstärke von Logiken höherer Stufe fordert ihren Tribut in Form einer komplexen Beweisführung. Die heute zur Verfügung stehenden Kalküle verfügen nur über ein geringes Automatisierungspotenzial, so dass große Teile eines Korrektheitsbeweises manuell durchgeführt werden müssen. Dies ist einer der Gründe, weshalb sich die praktische Anwendung der formalen Software-Verifikation heute auf ausgewählte Programmabschnitte in besonders sicherheitskritischen Software-Systemen beschränkt. Auch die Berechenbarkeitstheorie zeigt uns an dieser Stelle unüberwindbare Grenzen auf. So folgt aus dem Gödel'schen Unvollständigkeitssatz, dass es für Logiken höherer Stufe keine vollständigen Kalküle mehr geben kann. Das bedeutet, dass wir keine Garantie dafür haben, ein korrektes Software-System immer auch als solches identifizieren zu können.

is_prime.c

```

1 int is_prime(unsigned nr)
2 {
3     int i;
4
5     if (nr <= 1)
6         return 0;
7
8     for (i=2; i<nr; i++) {
9         if (nr % i == 0) {
10             return 0;
11         }
12     }
13
14     return 1;
15 }
```

is_prime.ml

```

1 val divides = Define
2   'divides a b =
3     ?x. b = a * x';
4 val prime = Define
5   'prime p =
6     ~(p=0) /\ ~(p=1) /\ 
7     !x. x divides p ==>
8       (x=1) /\ (x=p)';
9
10  |- !x.
11    (prime(x) ==>
12      (is_prime(x)=1) /\ 
13      !prime(x) ==>
14        (is_prime(x)=0))
```

Abbildung 3.60: Formale Spezifikation eines C-Programms mithilfe einer Logik höherer Stufe. Auf der linken Seite ist die Implementierung, auf der rechten die formale Spezifikation in der Sprache des Theorembeweisers HOL zu sehen.

Aufgabe 3.1**Webcode
3112**

3.4 Übungsaufgaben

Die formale Logik hat ihre Wurzeln im antiken Griechenland. Bereits im vierten Jahrhundert vor Christus begründete der griechische Philosoph Aristoteles die *Syllogistik*.

Gegenstand dieser Lehre sind die *Syllogismen*. Hierbei handelt es sich um logische Schlussfiguren, die alle nach dem gleichen Schema aufgebaut sind. Sie bestehen aus jeweils zwei Prämissen und einer Konklusion. Die erste (allgemeinere) Prämisse wird als Obersatz und die zweite (speziellere) Prämisse als Untersatz bezeichnet.

Der *Modus barbara* gehört zu den bekanntesten Schlussfiguren des Aristoteles. Er besagt das Folgende:

■ Modus Barbara

$$\begin{array}{l} \text{Alle } A \text{ sind } C \\ \text{Alle } B \text{ sind } A \\ \hline \text{Alle } B \text{ sind } C \end{array}$$

■ Beispiel

$$\begin{array}{l} \text{Alle Vögel können fliegen} \\ \text{Alle Tauben sind Vögel} \\ \hline \text{Alle Tauben können fliegen} \end{array}$$



Nachstehend finden Sie eine Auswahl weiterer Schlussfiguren vor, die nach dem gleichen Schema aufgebaut sind. Welche davon sind wahre Syllogismen? Welche sind logisch falsch?

- | | | |
|---|--|---|
| a) Alle <i>A</i> sind <i>B</i>
Einige <i>C</i> sind <i>A</i>
<hr/> Einige <i>C</i> sind <i>B</i> | e) Alle <i>B</i> sind <i>A</i>
Kein <i>B</i> ist <i>C</i>
<hr/> Kein <i>C</i> ist <i>B</i> | i) Kein <i>A</i> ist <i>B</i>
Einige <i>A</i> sind <i>C</i>
<hr/> Einige <i>C</i> sind nicht <i>B</i> |
| b) Kein <i>A</i> ist <i>B</i>
Einige <i>C</i> sind <i>A</i>
<hr/> Einige <i>C</i> sind nicht <i>B</i> | f) Alle <i>A</i> sind <i>B</i>
Einige <i>C</i> sind nicht <i>B</i>
<hr/> Einige <i>C</i> sind nicht <i>A</i> | j) Kein <i>C</i> ist <i>B</i>
Alle <i>A</i> sind <i>B</i>
<hr/> Kein <i>C</i> ist <i>A</i> |
| c) Kein <i>C</i> ist <i>B</i>
Alle <i>B</i> sind <i>A</i>
<hr/> Kein <i>C</i> ist <i>A</i> | g) Alle <i>A</i> sind <i>B</i>
Alle <i>C</i> sind <i>A</i>
<hr/> Einige <i>C</i> sind <i>B</i> | k) Kein <i>A</i> ist <i>B</i>
Alle <i>C</i> sind <i>A</i>
<hr/> Einige <i>C</i> sind nicht <i>B</i> |
| d) Kein <i>A</i> ist <i>B</i>
Alle <i>C</i> sind <i>A</i>
<hr/> Kein <i>C</i> ist <i>B</i> | h) Kein <i>A</i> ist <i>B</i>
Einige <i>C</i> sind <i>B</i>
<hr/> Einige <i>C</i> sind nicht <i>A</i> | l) Kein <i>A</i> ist <i>B</i>
Einige <i>B</i> sind <i>C</i>
<hr/> Einige <i>C</i> sind nicht <i>A</i> |

Gegeben seien die folgenden beiden aussagenlogischen Formeln:

$$F := AB \vee AC \vee BC$$

$$G := AB \vee C(A \leftrightarrow B)$$

Aufgabe 3.2

Webcode
3378

Zeigen Sie, dass F und G äquivalent sind, indem Sie

- für beide Formeln eine Wahrheitstabelle aufstellen.
- G durch die Anwendung der aussagenlogischen Umformungsregeln in F überführen.

Gegeben sei die folgende aussagenlogische Formel:

$$F := \left(\bigvee_{i=1}^n A_i \right) \wedge \left(\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^n \overline{A_i \wedge A_j} \right)$$

Aufgabe 3.3

Webcode
3561

Vervollständigen Sie die nachstehende Wahrheitstabelle für den Fall $n = 4$:

	A_4	A_3	A_2	A_1	F
0	0	0	0	0	
1	0	0	0	1	
2	0	0	1	0	
3	0	0	1	1	
4	0	1	0	0	
5	0	1	0	1	
6	0	1	1	0	
7	0	1	1	1	

	A_4	A_3	A_2	A_1	F
8	1	0	0	0	
9	1	0	0	1	
10	1	0	1	0	
11	1	0	1	1	
12	1	1	0	0	
13	1	1	0	1	
14	1	1	1	0	
15	1	1	1	1	

Beschreiben Sie in Worten, welche Eigenschaft eine Variablenbelegung erfüllen muss, damit F wahr wird.

Eine n -stellige boolesche Funktion hat die Form $\{0, 1\}^n \rightarrow \{0, 1\}$.

Aufgabe 3.4

Webcode
3444

- Listen Sie alle vierstelligen booleschen Funktionen auf und beschreiben Sie jede mit einem booleschen Ausdruck, der ausschließlich die Operatoren \neg , \wedge und \vee enthält.
- Wie viele Funktionen der Stelligkeit n gibt es insgesamt?

Aufgabe 3.5**Webcode
3484**

Mit F sei eine aussagenlogische Formel in disjunktiver Normalform gegeben:

$$F = ((\neg)A_1 \wedge \dots \wedge (\neg)A_i) \vee \dots \vee ((\neg)B_1 \wedge \dots \wedge (\neg)B_j).$$

- a) Geben Sie ein Verfahren an, mit dem sich die Erfüllbarkeit einer solchen Formel effizient überprüfen lässt.
- b) Funktioniert Ihr Verfahren auch für Formeln in konjunktiver Normalform?

Als nächstes sei mit G eine aussagenlogische Formel in konjunktiver Normalform gegeben:

$$G = ((\neg)A_1 \vee \dots \vee (\neg)A_i) \wedge \dots \wedge ((\neg)B_1 \vee \dots \vee (\neg)B_j).$$

- a) Geben Sie ein Verfahren an, mit dem sich die Allgemeingültigkeit einer solchen Formel effizient überprüfen lässt.
- b) Funktioniert Ihr Verfahren auch für Formeln in disjunktiver Normalform?

Aufgabe 3.6**Webcode
3798**

Mit F sei eine variablenfreie aussagenlogische Formel gegeben, die neben 0 und 1 ausschließlich den Äquivalenzoperator \leftrightarrow enthält.

- a) Ist die Formel $(1 \leftrightarrow 0) \leftrightarrow ((1 \leftrightarrow 0) \leftrightarrow (0 \leftrightarrow 1))$ eine Tautologie?
- b) Ist die Formel $(1 \leftrightarrow 0) \leftrightarrow ((0 \leftrightarrow 0) \leftrightarrow (1 \leftrightarrow 0))$ eine Tautologie?
- c) Zeigen oder widerlegen Sie die folgende Behauptung: F ist genau dann eine Tautologie, wenn die Teilformel 0 geradzahlig oft in F vorkommt.

Aufgabe 3.7**Webcode
3192**

Die booleschen Operatoren $\bar{\wedge}$ (NAND) und $\bar{\vee}$ (NOR) sind folgendermaßen definiert:

	A	B	$A \bar{\wedge} B$		A	B	$A \bar{\vee} B$
0	0	0	1	0	0	0	1
1	0	1	1	1	0	1	0
2	1	0	1	2	1	0	0
3	1	1	0	3	1	1	0

- a) Drücken Sie $\bar{\wedge}$ und $\bar{\vee}$ mithilfe der Elementaroperatoren \neg , \wedge und \vee aus.
- b) Zeigen Sie, dass die Mengen $\{\bar{\wedge}\}$ und $\{\bar{\vee}\}$ vollständige Operatorenysteme sind.

Die aussagenlogischen Existenz- und Allquantoren \exists und \forall seien wie folgt definiert:

$$\begin{aligned}\exists A_i F \equiv 1 &\Leftrightarrow F \equiv 1 \text{ für mindestens eine Belegung der Variablen } A_i \\ (\forall A_i F) \equiv 1 &\Leftrightarrow F \equiv 1 \text{ für alle Belegungen der Variablen } A_i\end{aligned}$$

Ferner vereinbaren wir die folgende Schreibweise:

$$\begin{aligned}\exists A_1, A_2, \dots, A_n F &:= \exists A_1 (\exists A_2, \dots, A_n F) \\ \forall A_1, A_2, \dots, A_n F &:= \forall A_1 (\forall A_2, \dots, A_n F)\end{aligned}$$

Beachten Sie, dass die Quantoren auf aussagenlogische Variablen und damit streng genommen auf (nullstellige) Prädikate angewendet werden. Sie unterscheiden sich damit grundsätzlich von ihren prädikatenlogischen Pendants.

Aufgabe 3.8

Webcode
3093

- Lassen sich die Quantoren durch die Elementaroperatoren \neg , \wedge und \vee ausdrücken?
- Seien F und G aussagenlogische Formeln, in denen die Variablen A_1, \dots, A_n vorkommen. Welche bekannten Eigenschaften erfüllen F und G , wenn die Beziehungen $\exists A_1, \dots, A_n G \equiv 1$ bzw. $\forall A_1, \dots, A_n F \equiv 1$ gelten?

In Abschnitt 3.1.3.1 wurde argumentiert, dass der Hilbert-Kalkül korrekt ist. Um den Beweis abzuschließen, muss die Allgemeingültigkeit der Axiome

Aufgabe 3.9

Webcode
3850

- (A1) : $F \rightarrow (G \rightarrow F)$
- (A2) : $(F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$
- (A3) : $(\neg F \rightarrow \neg G) \rightarrow (G \rightarrow F)$

gezeigt werden. Führen Sie den Beweis durch die Vervollständigung der nachstehenden Wahrheitstabellen zu Ende:

	F	G	(A1)
0	0	0	
1	0	1	
2	1	0	
3	1	1	

	F	G	H	(A2)
0	0	0	0	
1	0	0	1	
2	0	1	0	
3	0	1	1	
4	1	0	0	
5	1	0	1	
6	1	1	0	
7	1	1	1	

	F	G	(A3)
0	0	0	
1	0	1	
2	1	0	
3	1	1	

Aufgabe 3.10**Webcode
3659**

In diesem Kapitel haben Sie gelernt, aussagenlogische Formeln mithilfe des Hilbert-Kalküls zu beweisen. Vervollständigen Sie die folgende Ableitungssequenz, indem Sie für jedes Element der Beweiskette angeben, wie dieses entstanden ist. Beachten Sie, dass der Beweis neben den Axiomen auch auf die bereits bewiesenen Theoreme auf Seite 103 zurückgreift.

- $\vdash A \rightarrow (\neg\neg A \rightarrow \neg(A \rightarrow \neg A))$ ()
- $\{A\} \vdash \neg\neg A \rightarrow \neg(A \rightarrow \neg A)$ ()
- $\{A\} \vdash A \rightarrow \neg\neg A$ ()
- $\{A\} \vdash \neg\neg A$ ()
- $\{A\} \vdash \neg(A \rightarrow \neg A)$ ()
- $\vdash A \rightarrow \neg(A \rightarrow \neg A)$ ()
- $\vdash (A \rightarrow \neg(A \rightarrow \neg A)) \rightarrow (\neg\neg(A \rightarrow \neg A) \rightarrow \neg A)$ ()
- $\vdash \neg\neg(A \rightarrow \neg A) \rightarrow \neg A$ ()
- $\vdash (A \rightarrow \neg A) \rightarrow \neg\neg(A \rightarrow \neg A)$ ()
- $\vdash ((A \rightarrow \neg A) \rightarrow \neg\neg(A \rightarrow \neg A)) \rightarrow ((\neg\neg(A \rightarrow \neg A) \rightarrow \neg A) \rightarrow ((A \rightarrow \neg A) \rightarrow \neg A))$ ()
- $\vdash (\neg\neg(A \rightarrow \neg A) \rightarrow \neg A) \rightarrow ((A \rightarrow \neg A) \rightarrow \neg A)$ ()
- $\vdash (A \rightarrow \neg A) \rightarrow \neg A$ ()

Aufgabe 3.11**Webcode
3234**

Die Funktionen F_1 und F_2 seien wie folgt definiert:

$$F_1 := (A_1 \vee A_2 \vee A_3 \vee A_4)$$

$$F_2 := (A_1 \vee A_2 \vee B) \wedge (\neg B \vee A_3 \vee A_4)$$

Zeigen Sie, dass die Formeln F_1 und F_2 erfüllbarkeitsäquivalent sind, d. h., F_1 ist genau dann erfüllbar, wenn F_2 erfüllbar ist.

Aufgabe 3.12**Webcode
3235**

Beweisen oder widerlegen Sie die Unerfüllbarkeit der folgenden Klauselmenge mithilfe der aussagenlogischen Resolution:

$$\begin{array}{lll} \{ A, B, C \} & \{ \neg A, \neg B \} & \{ \neg A, \neg C \} \\ \{ \neg B, \neg A \} & \{ \neg B, \neg C \} & \{ \neg C, \neg A \} \\ \{ \neg C, \neg B \} & \{ A, \neg B \} & \{ B, \neg C \} \end{array}$$

In diesem Kapitel haben Sie gelernt, wie aussagenlogische Formeln im Resolutionskalkül bewiesen werden können. Da die entstehenden Beweise oft von beträchtlicher Länge sind, entsteht der Wunsch, mehrere Resolutionsschritte auf einmal durchzuführen. Hierzu wollen wir den bestehenden Kalkül um die folgende Resolutionsregel erweitern:

$$\frac{\{A_i, A_j\} \cup M_1 \quad \{\neg A_i, \neg A_j\} \cup M_2}{M_1 \cup M_2}$$

Ist der resultierende Kalkül immer noch korrekt? Begründen Sie Ihre Antwort.

Wir wollen versuchen, die Allgemeingültigkeit der Formel $F := \exists x (\forall y P(y) \vee \neg P(x))$ im Resolutionskalkül zu beweisen. Zunächst erzeugen wir die Skolem-Form von $\neg F$:

$$\begin{aligned} \neg \exists x (\forall y P(y) \vee \neg P(x)) &\equiv \forall x \neg (\forall y P(y) \vee \neg P(x)) \\ &\equiv \forall x (\exists y \neg P(y) \wedge P(x)) \\ &\equiv \forall x \exists y (\neg P(y) \wedge P(x)) \quad \equiv_E \forall x (\neg P(f(x)) \wedge P(x)) \end{aligned}$$

Insgesamt erhalten wir zwei Klauseln:

$$\{\neg P(f(x))\} \quad \{P(x)\}$$

Ganz offensichtlich sind $P(f(x))$ und $P(x)$ nicht unifizierbar, so dass es unmöglich ist, die leere Klausel abzuleiten. Was haben wir falsch gemacht?

Unifizieren Sie die folgenden Formelpaare mit dem Algorithmus von Robinson:

- | | | |
|--------------------------------|-----------------------------|--------------------------------------|
| a) $f(g(x), y), f(g(z), g(a))$ | c) $f(x), f(f(x))$ | e) $f(x, g(y)), f(y, g(x))$ |
| b) $f(x), f(f(f(y)))$ | d) $f(x, g(y)), f(g(y), x)$ | f) $f(f(x), y, g(g(z))), f(x, y, z)$ |

Sei K ein beliebiger Kalkül. Vervollständigen Sie die folgenden Definitionen:

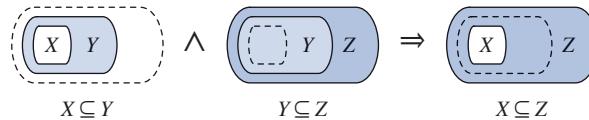
- a) K ist korrekt : \Leftrightarrow
- b) K ist vollständig : \Leftrightarrow

Aufgabe 3.13
Webcode
3021
Aufgabe 3.14
Webcode
3931
Aufgabe 3.15
Webcode
3888
Aufgabe 3.16
Webcode
3046

Aufgabe 3.17

**Webcode
3971**

Aus der elementaren Mengenlehre wissen Sie, dass die Inklusionsbeziehung transitiv ist:
Sind X , Y und Z Mengen, so folgt aus $X \subseteq Y$ und $Y \subseteq Z$ die Beziehung $X \subseteq Z$.



In mathematischer Notation lässt sich die Transitivität wie folgt charakterisieren:

$$\forall x \forall y \forall z (x \subseteq y \wedge y \subseteq z \rightarrow x \subseteq z) \quad (3.15)$$

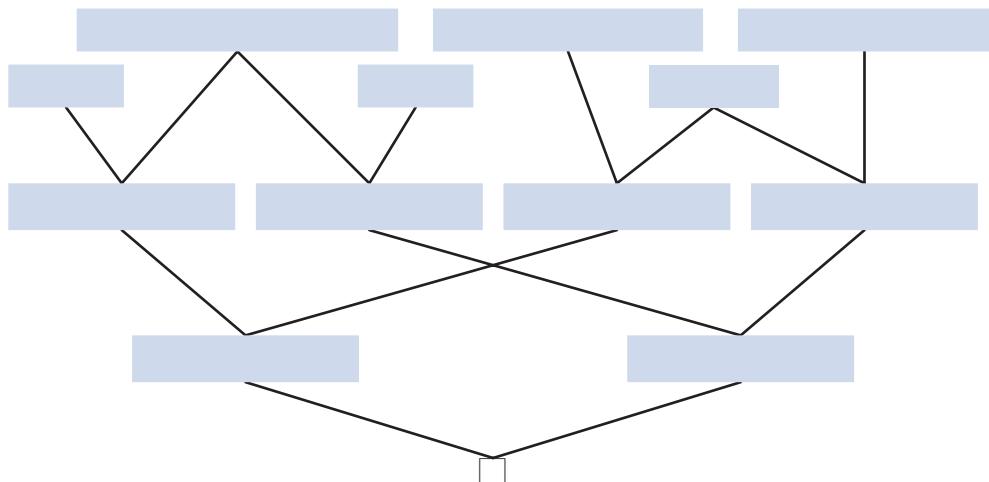
Beschreiben wir die Teilmengenbeziehung über die Formel

$$\forall x \forall y (x \subseteq y \leftrightarrow \forall z (z \in x \rightarrow z \in y)), \quad (3.16)$$

so lässt sich die Transitivität mithilfe der Prädikatenlogik formal beweisen. Hierzu führen wir zunächst die beiden Prädikate M (*member*) und C (*contains*) ein, um die Mengenzugehörigkeit \in und die Inklusionseigenschaft \subseteq zu beschreiben. Die Formeln (3.15) und (3.16) lassen sich dann wie folgt zusammenfassen [72]:

$$\forall x \forall y (C(x, y) \leftrightarrow \forall z (M(z, x) \rightarrow M(z, y))) \rightarrow \forall x \forall y \forall z (C(x, y) \wedge C(y, z) \rightarrow C(x, z)) \quad (3.17)$$

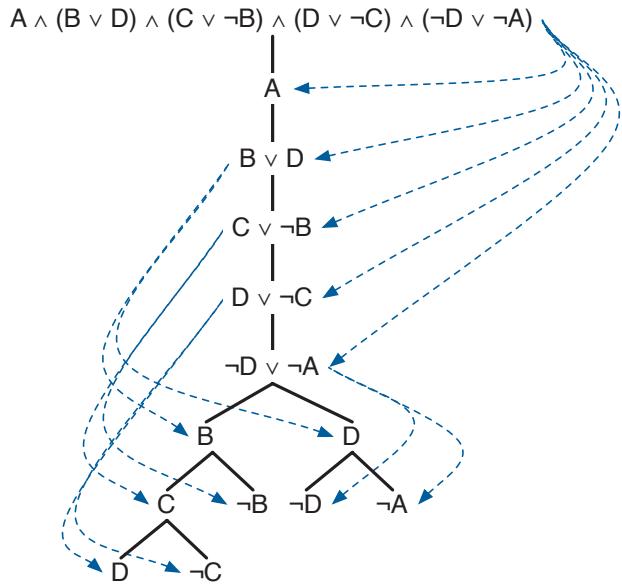
Ihre Aufgabe ist es, die Transitivität mithilfe des Resolutionskalküls zu beweisen. Transformieren Sie die Formel (3.17) hierzu zunächst in Skolem-Form und vervollständigen Sie anschließend den folgenden Resolutionsgraphen:



Gegeben sei das folgende Tableau für die Formel

$$F := A \wedge (B \vee D) \wedge (C \vee \neg B) \wedge (D \vee \neg C) \wedge (\neg D \vee \neg A)$$

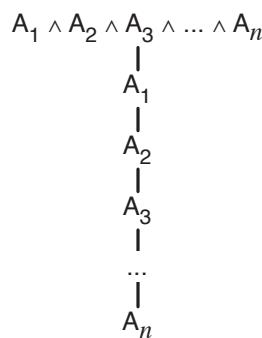
Aufgabe 3.18

Webcode
3369


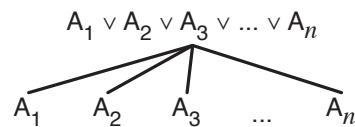
Der linke Pfad ist widerspruchsfrei, so dass die Belegung I mit $I(A) = I(B) = I(C) = I(D) = 1$ ein Modell für F sein muss. Verifizieren Sie diese Behauptung anhand einer Wahrheitstabelle und klären Sie den entstehenden Widerspruch auf.

Welche der folgenden aussagenlogischen Tableauregeln sind korrekt?

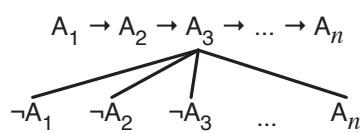
1) Erweiterte Konjunktionsregel



2) Erweiterte Disjunktionsregel



3) Erweiterte Implikationssregel


Aufgabe 3.19

Webcode
3177

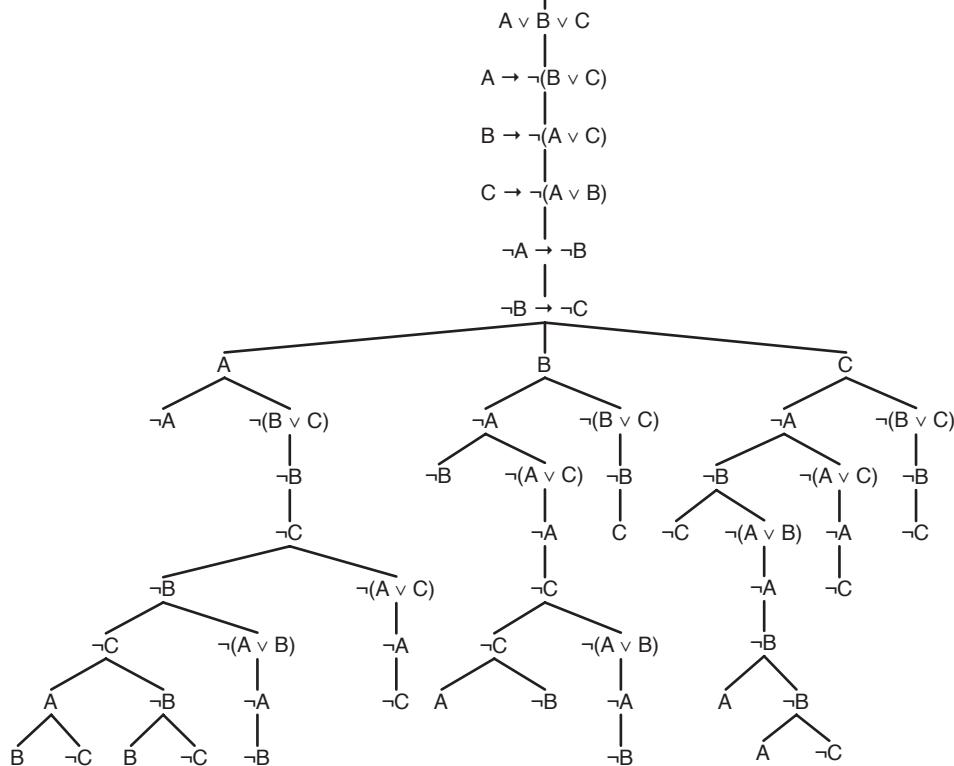
Aufgabe 3.20

Gegeben sei das folgende aussagenlogische Tableau:



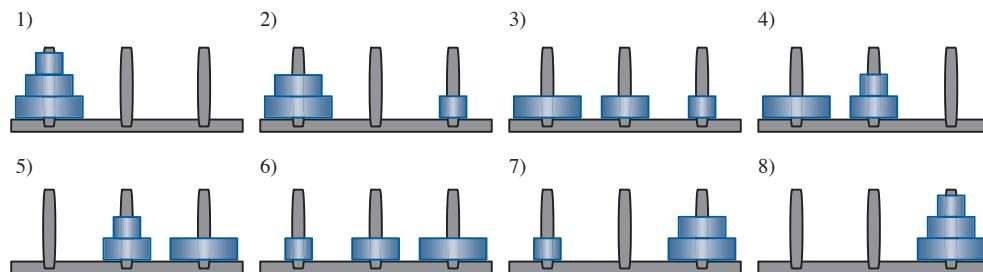
**Webcode
3953**

$$(A \vee B \vee C) \wedge (A \rightarrow \neg(B \vee C)) \wedge (B \rightarrow \neg(A \vee C)) \wedge (C \rightarrow \neg(A \vee B)) \wedge (\neg A \rightarrow \neg B) \wedge (\neg B \rightarrow \neg C)$$



- Geben Sie für jeden widersprüchlichen Pfad an, durch welches Variablenpaar der Abschluss entsteht.
- Extrahieren Sie für jeden widerspruchsfreien Pfad ein Modell, falls solche Pfade überhaupt existieren.
- Ist das Tableau vollständig?
- Ist das Tableau geschlossen?
- Ist die Ausgangsformel eine Tautologie?

Die Türme von Hanoi ist der Name eines bekannten Logikrätsels. Ausgangspunkt sind drei nebeneinander angeordnete Stäbe a, b und c, von denen der mittlere und der rechte leer sind. Auf dem linken Stab befinden sich N Scheiben unterschiedlicher Größe. Die größte Scheibe liegt unten und die kleinste oben. Das Rätsel ist gelöst, wenn es gelingt, den Scheibenstapel vom linken auf den rechten Stab zu verschieben. Dabei ist es nicht erlaubt, mehrere Scheiben gleichzeitig zu bewegen oder eine größere Scheibe auf einer kleineren abzulegen. Die folgende Zugsequenz verdeutlicht, wie ein Stapel der Höhe $N = 3$ unter Beachtung der genannten Spielregeln verschoben werden kann:



Das nebenstehend abgebildete Programm zeigt, wie sich das Rätsel in PROLOG lösen lässt. Insgesamt werden zwei Prädikate definiert. `hanoi(N)` ist das Startprädikat. Um eine Lösungsstrategie für einen Turm mit N Scheiben zu erhalten, müssen wir den PROLOG-Interpreter lediglich mit der Anfrage `hanoi(3)` starten. Die eigentliche Programmlogik ist im Prädikat `move` versteckt. Seine Implementierung basiert auf der Idee, dass wir das Hanoi-Problem mit N Scheiben rekursiv lösen können, indem wir zunächst $N - 1$ Scheiben auf den mittleren Stab verschieben, danach die unterste Scheibe auf den rechten Stab stecken und anschließend die in der Mitte abgelegten Scheiben nach rechts bringen.

```

hanoi.pl
1 move(A, _, C, 1) :- 
2   write(A),
3   write(' nach '),
4   write(C),
5   nl.
6 
7 move(A, B, C, N) :- 
8   M is N-1,
9   move(A, C, B, M),
10  move(A, B, C, 1),
11  move(B, A, C, M).
12 
13 hanoi(N) :- 
14   move(a, b, c, N).

```

Ihre Aufgabe ist es, das Programmverhalten für die Anfrage `hanoi(3)` im Detail zu analysieren. Skizzieren Sie die Schritte, die der PROLOG-Interpreter zur Lösungsfindung durchführt.

Hinweis: Mit `write` und `is` verwendet das Programm zwei vordefinierte Prädikate aus der PROLOG-Bibliothek. `write(X)` ist immer wahr und gibt als Seiteneffekt den Term `X` aus. Über das Prädikat `is` haben wir Zugriff aus die Arithmetikbibliothek. Beispielsweise führt die Auswertung des Ausdrucks `M is 4-1` dazu, dass `M` mit dem Term `3` instanziert wird.

Aufgabe 3.21



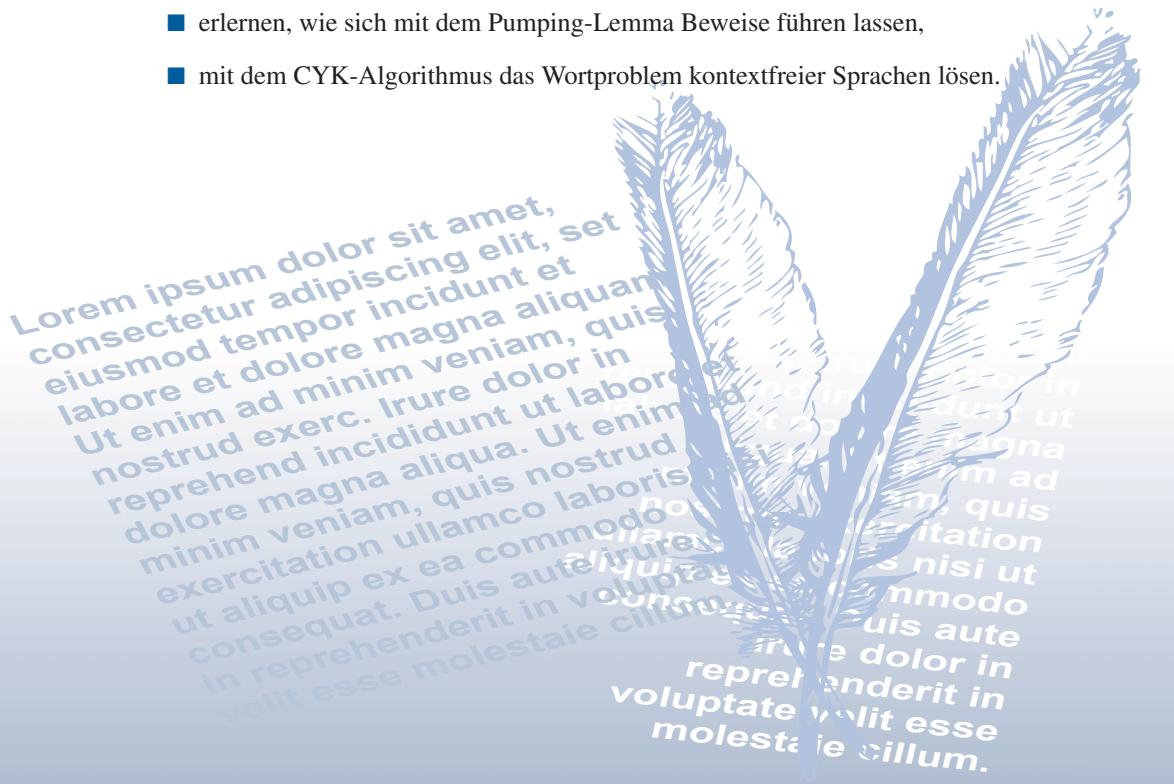
Webcode

3590

4 Formale Sprachen

In diesem Kapitel werden Sie ...

- formale Sprachen mithilfe von Grammatiken erzeugen,
- die Chomsky-Hierarchie verstehen,
- die Besonderheiten regulärer, kontextfreier und kontextsensitiver Grammatiken ergründen,
- die wichtigsten Entscheidungsprobleme im Bereich der formalen Sprachen kennen lernen,
- die Abschlusseigenschaften verschiedener Sprachtypen untersuchen,
- erlernen, wie sich mit dem Pumping-Lemma Beweise führen lassen,
- mit dem CYK-Algorithmus das Wortproblem kontextfreier Sprachen lösen.



■ Definition

$$\begin{aligned}\Sigma^0 &:= \{\varepsilon\} \\ \Sigma^1 &:= \Sigma \\ \Sigma^{n+1} &:= \{xy \mid x \in \Sigma, y \in \Sigma^n\} \\ \Sigma^+ &:= \bigcup_{i=1}^{\infty} \Sigma^i \\ \Sigma^* &:= \bigcup_{i=0}^{\infty} \Sigma^i\end{aligned}$$

■ Beispiel: $\Sigma := \{a, b\}$

$$\begin{aligned}\Sigma^0 &= \{\varepsilon\} \\ \Sigma^1 &= \{a, b\} \\ \Sigma^2 &= \{aa, ab, ba, bb\} \\ &\dots \\ \Sigma^+ &= \{a, b, aa, ab, ba, bb, \dots\} \\ \Sigma^* &= \{\varepsilon, a, b, aa, ab, ba, bb, \dots\}\end{aligned}$$

Abbildung 4.1: Formale Charakterisierung der Wortmengen Σ^i , Σ^* und Σ^+

■ Beispiel: $L := \{ab, ba\}$

$$\begin{aligned}L^0 &= \{\varepsilon\} \\ L^1 &= \{ab, ba\} \\ L^2 &= \{abab, abba, baab, babba\} \\ &\dots \\ L^+ &= \{ab, ba, abab, abba, \\ &\quad baab, babba, \dots\} \\ L^* &= \{\varepsilon, ab, ba, abab, abba, \\ &\quad baab, babba, \dots\}\end{aligned}$$

Abbildung 4.2: Nicht nur Alphabete, sondern beliebige Sprachen lassen sich mit den eingeführten Mengenoperatoren miteinander kombinieren.

4.1 Sprache und Grammatik

Die Theorie der formalen Sprachen beschäftigt sich mit der systematischen Analyse, der Klassifikation und der Konstruktion von Wortmengen, die über einem endlichen *Alphabet* gebildet werden. Bevor wir uns im Detail mit den Eigenschaften formaler Sprachen beschäftigen, wollen wir mit der folgenden Definition die elementaren Begriffe einführen, die uns durch das gesamte Kapitel hinweg auf Schritt und Tritt begleiten werden.



Definition 4.1 (Alphabet, Zeichen, Wort, Sprache)

- Ein *Alphabet* Σ ist eine endliche Menge von Symbolen.
- Jedes Element $\sigma \in \Sigma$ ist ein *Zeichen* des Alphabets.
- Jedes Element $\omega \in \Sigma^*$ wird als *Wort* über Σ bezeichnet.
- Jede Teilmenge $L \subseteq \Sigma^*$ ist eine *formale Sprache* über Σ .

Die Definition fordert ausdrücklich, dass der Zeichenvorrat Σ einer Sprache nur aus endlich vielen Elementen besteht. Die Menge Σ^* wird als *Kleene'sche Hülle* bezeichnet und fasst alle endlichen Symbolsequenzen zusammen, die mit Zeichen aus dem Alphabet Σ aufgebaut werden können. Wie in den Abbildungen 4.1 und 4.2 in mathematischer Notation beschrieben, unterscheidet sie sich von der Menge Σ^+ lediglich dadurch, dass Σ^* auch das leere Wort ε enthält. Anders als die Wörter einer Sprache, die stets eine endliche Länge aufweisen, kann eine Sprache L aus unendlich vielen Wörtern bestehen. Die Beispiele in Abbildung 4.3 verdeutlichen den Sprachbegriff.

Im Bereich der formalen Sprachen sind die folgenden Fragestellungen von Interesse:

- Wortproblem
Gilt für ein Wort $\omega \in \Sigma^*$ und eine Sprache L die Beziehung $\omega \in L$?
- Leerheitsproblem
Enthält eine Sprache L mindestens ein Wort, gilt also $L \neq \emptyset$?
- Endlichkeitsproblem
Besitzt eine Sprache L nur endlich viele Elemente?

■ Äquivalenzproblem

Gilt für zwei Sprachen L_1 und L_2 die Beziehung $L_1 = L_2$?

■ Spracherzeugung

Gibt es für eine Sprache L eine Beschreibung, aus der sich alle Wörter systematisch ableiten lassen?

Die ersten vier Fragestellungen adressieren die *analytischen* Aspekte einer Sprache, auf die wir später im Detail zu sprechen kommen werden. Für den Moment wollen wir unser Augenmerk auf die letzte Fragestellung richten, die sich mit dem *generativen* Aspekt einer Sprache beschäftigt.

Folgt der Aufbau strukturierten Regeln, so lassen sich die Wörter einer Sprache mithilfe einer *Grammatik* erzeugen. Für unsere natürliche Sprache sind wir mit diesem Vorgehen wohlvertraut. Anstatt alle korrekt geformten Sätze nacheinander aufzulisten, wird eine Reihe von Regeln vereinbart, mit denen sich elementare Sprachkonstrukte systematisch zu komplexen Gebilden zusammensetzen lassen. Als Beispiel betrachten wir die folgende Grammatik, die einen kleinen Auszug aus dem deutschen Sprachschatz erzeugt:

$\langle \text{Satz} \rangle$	\rightarrow	$\langle \text{Subjekt} \rangle \langle \text{Prädikat} \rangle \langle \text{Objekt} \rangle$
$\langle \text{Subjekt} \rangle$	\rightarrow	$\langle \text{Artikel} \rangle \langle \text{Adjektiv} \rangle \langle \text{Substantiv} \rangle$
$\langle \text{Artikel} \rangle$	\rightarrow	Der Die Das
$\langle \text{Adjektiv} \rangle$	\rightarrow	kleine süße flinke
$\langle \text{Substantiv} \rangle$	\rightarrow	Eisbär Elch Kröte Maus Nilpferd
$\langle \text{Prädikat} \rangle$	\rightarrow	mag fängt isst
$\langle \text{Objekt} \rangle$	\rightarrow	Kekse Schokolade Käsepizza

In der Terminologie der formalen Sprachen werden die in spitze Klammern gesetzten Platzhalter als *Nonterminale* oder *Nichtterminale* und die nicht weiter ersetzbaren Sprachbestandteile als *Terminale* bezeichnet. Für unsere Beispielgrammatik erhalten wir die nachstehende Einteilung:

■ Nonterminale

$\langle \text{Satz} \rangle, \langle \text{Subjekt} \rangle, \langle \text{Artikel} \rangle, \langle \text{Adjektiv} \rangle, \langle \text{Substantiv} \rangle, \langle \text{Prädikat} \rangle, \langle \text{Objekt} \rangle,$

■ Terminale

Der, Die, Das, kleine, süße, flinke, Eisbär, Elch, Kröte, Maus, Nilpferd, mag, fängt, isst, Kekse, Schokolade, Käsepizza

■ Beispiel 1

Dyck-Sprache D_2	
Σ	$\{((),[],[]\}$
L	Menge der korrekt geklammerten Ausdrücke
$\in L$	$((), [()((())), ()[()]), \dots$
$\notin L$	$(, [()[(())], (x), \dots$

■ Beispiel 2

Primzahlen	
Σ	$\{0, 1, 2, \dots, 9\}$
L	Menge der Ziffernfolgen, die einer Primzahl entsprechen
$\in L$	$2, 3, 5, 7, 11, 13, \dots$
$\notin L$	$0, 1, 4, 6, 8, 9, 10, 12, \dots$

■ Beispiel 3

ABC-Sprache	
Σ	$\{a, b, c\}$
L	Menge aller geordneten Folgen aus a 's, b 's und c 's
$\in L$	$abc, aabbc, aaaabbbcc, \dots$
$\notin L$	$cba, ababc, abcba, \dots$

■ Beispiel 4

Palindromsprache	
Σ	$\{a, b, c, \dots, z\}$
L	Menge aller spiegelbildlich angeordneten Zeichenketten
$\in L$	$aabaa, reittier, anna, otto$
$\notin L$	$abab, aaba, abcab$

Abbildung 4.3: Beispiele formaler Sprachen

Beispiel 1	Beispiel 2
<p><Satz></p> <p>⇒ <Subjekt><Prädikat><Objekt></p> <p>⇒ <Subjekt> fängt <Objekt></p> <p>⇒ <Subjekt> fängt Kekse</p> <p>⇒ <Artikel><Adjektiv><Substantiv> fängt Kekse</p> <p>⇒ Das <Adjektiv><Substantiv> fängt Kekse</p> <p>⇒ Das flinke <Substantiv> fängt Kekse</p> <p>⇒ Das flinke Nilpferd fängt Kekse</p>	<p><Satz></p> <p>⇒ <Subjekt><Prädikat><Objekt></p> <p>⇒ <Subjekt> isst <Objekt></p> <p>⇒ <Subjekt> isst Käsepizza</p> <p>⇒ <Artikel><Adjektiv><Substantiv> isst Käsepizza</p> <p>⇒ Die <Adjektiv><Substantiv> isst Käsepizza</p> <p>⇒ Die kleine <Substantiv> isst Käsepizza</p> <p>⇒ Die kleine Maus isst Käsepizza</p>

Tabelle 4.1: Durch die sukzessive Anwendung der Produktionen einer Grammatik G lassen sich alle Wörter der Sprache $\mathcal{L}(G)$ aus dem Startsymbol ableiten.

Dem Nonterminal <Satz> kommt in unserem Beispiel eine besondere Bedeutung zu. Es ist immer das erste Symbol, mit dem eine Ableitung beginnt, und wird folgerichtig als *Startsymbol* bezeichnet. Die Beispiele in Tabelle 4.1 zeigen, wie sich aus dem Startsymbol einige mehr oder weniger sinnvolle Sätze der deutschen Sprache ableiten lassen.

Mit der geleisteten Vorarbeit sind wir in der Lage, den Begriff der Grammatik formal zu definieren:



Definition 4.2 (Grammatik)

Eine *Grammatik* G ist ein Viertupel (V, Σ, P, S) . Sie besteht aus

- der endlichen *Variablenmenge* V (*Nonterminale*),
- dem endlichen *Terminalalphabet* Σ mit $V \cap \Sigma = \emptyset$,
- der endlichen Menge P von *Produktionen* (*Regeln*) und
- der *Startvariablen* S mit $S \in V$.

Jede Produktion aus P hat die Form $l \rightarrow r$ mit $l \in (V \cup \Sigma)^+$ und $r \in (V \cup \Sigma)^*$.

Durch die Menge der Produktionen definiert jede Grammatik eine Ableitungsrelation \Rightarrow auf der Menge $(V \cup \Sigma)^*$. Haben zwei Wörter $x, y \in$

$(V \cup \Sigma)^*$ die Form $x = lur$ und $y = lvr$ mit $l, r \in (V \cup \Sigma)^*$, so gilt $x \Rightarrow y$ genau dann, wenn die Grammatik eine Produktionsregel der Form $u \rightarrow v$ enthält. Mit \Rightarrow^* bezeichnen wir die reflexiv-transitive Hülle der Ableitungsrelation. Verbal ausgedrückt gilt $x \Rightarrow^* y$ genau dann, wenn das Wort y dem Wort x entspricht oder sich in endlich vielen Schritten aus x ableiten lässt.

Jede Grammatik G erzeugt eine Sprache $\mathcal{L}(G)$. Diese definieren wir als die Menge der Wörter über dem Terminalalphabet Σ , die sich aus dem Startsymbol S ableiten lassen:

$$\mathcal{L}(G) := \{y \in \Sigma^* \mid S \Rightarrow^* y\} \quad (4.1)$$

Ein gezielter Blick auf Gleichung (4.1) zeigt, dass die Wörter der Sprache $\mathcal{L}(G)$ ausschließlich aus Terminalsymbolen bestehen. Nonterminale spielen lediglich die Rolle von Platzhaltern, die nach und nach durch Symbole des Terminalalphabets oder durch weitere Nonterminale ersetzt werden. Erst wenn alle Nonterminale verschwunden sind, haben wir ein Wort der Sprache $\mathcal{L}(G)$ erzeugt.

Als Beispiel betrachten wir die in Abbildung 4.4 dargestellte Grammatik zur Erzeugung der *Dyck-Sprache* D_2 . Allgemein ist die Dyck-Sprache D_n als die Menge der wohlgeformten Wörter definiert, die aus n unterschiedlichen Klammerpaaren aufgebaut sind.

Ein Blick auf die Grammatik zeigt, dass die Menge der Nonterminale $V = \{S\}$, die Menge der Terminalzeichen $\Sigma = \{(., [,])\}$ und das Startsymbol S unmittelbar aus den Ableitungsregeln hervorgehen. In Fällen wie diesem können wir uns daher ruhigen Gewissens auf die Angabe der Produktionen beschränken. Ferner vereinbaren wir eine abkürzende Schreibweise, die mehrere Produktionen mit der gleichen linken Seite zu einer einzigen Regel zusammenfasst. Im oberen Teil von Abbildung 4.5 ist das allgemeine Schema der Schreibweisenverkürzung abgebildet; der untere Teil zeigt, wie sich die Produktionen unserer Beispielgrammatik jetzt wesentlich kompakter formulieren lassen.

Am Beispiel des Dyck-Worts $(.)[().]$ wollen wir die Grammatik in Aktion erleben. Hierzu sind in Tabelle 4.2 drei Möglichkeiten dargestellt, wie sich das Wort aus dem Startsymbol S ableiten lässt.

- Die erste Ableitungssequenz (Tabelle 4.2 links) besitzt die Eigenschaft, dass in jedem Schritt das am weitesten links stehende Nonterminal ersetzt wurde. Eine solche Sequenz heißt *Linksableitung*.
- Die zweite Ableitungssequenz (Tabelle 4.2 Mitte) wurde so konstruiert, dass in jedem Schritt das am weitesten rechts stehende Nonterminal ersetzt wurde. Eine solche Sequenz heißt *Rechtsableitung*.

■ Signatur

$$G = (\{S\}, \{(., [,])\}, P, S)$$

■ Produktionsmenge P

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow SS \\ S &\rightarrow [S] \\ S &\rightarrow (S) \end{aligned}$$

Abbildung 4.4: Grammatik zur Erzeugung der Dyck-Sprache D_2

■ Verkürzte Schreibweise

$$\begin{aligned} l &\rightarrow r_1 \\ &\dots \\ l &\rightarrow r_n \end{aligned}$$

$$l \rightarrow r_1 | \dots | r_n$$

■ Beispiel

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow SS \\ S &\rightarrow [S] \\ S &\rightarrow (S) \end{aligned}$$

$$S \rightarrow \epsilon | SS | [S] | (S)$$

Abbildung 4.5: Zur Verkürzung der Schreibweise dürfen Produktionen mit gleicher linker Seite zu einer einzigen Regel zusammengefasst werden.

Ableitungssequenz 1	Ableitungssequenz 2	Ableitungssequenz 3
$S \Rightarrow SS$ $\Rightarrow (\textcolor{blue}{S})S$ $\Rightarrow ()\textcolor{blue}{S}$ $\Rightarrow ()SS$ $\Rightarrow ()[\textcolor{blue}{S}]S$ $\Rightarrow ()[(\textcolor{blue}{S})]S$ $\Rightarrow ()[()]\textcolor{blue}{S}$ $\Rightarrow ()[()][\textcolor{blue}{S}]$ $\Rightarrow ()[()][()$	$S \Rightarrow SS$ $\Rightarrow S(\textcolor{blue}{S})$ $\Rightarrow \textcolor{blue}{S}()$ $\Rightarrow SS()$ $\Rightarrow S[\textcolor{blue}{S}]()$ $\Rightarrow S[(\textcolor{blue}{S})]()$ $\Rightarrow \textcolor{blue}{S}[()]()$ $\Rightarrow (\textcolor{blue}{S})[()]()$ $\Rightarrow ()[()][()$	$S \Rightarrow SS$ $\Rightarrow \textcolor{blue}{S}SS$ $\Rightarrow (\textcolor{blue}{S})SS$ $\Rightarrow ()\textcolor{blue}{S}S$ $\Rightarrow ()[\textcolor{blue}{S}]S$ $\Rightarrow ()[(\textcolor{blue}{S})]S$ $\Rightarrow ()[()]\textcolor{blue}{S}$ $\Rightarrow ()[()][\textcolor{blue}{S}]$ $\Rightarrow ()[()][()$

Tabelle 4.2: Drei Ableitungssequenzen für das Wort ()[()][()

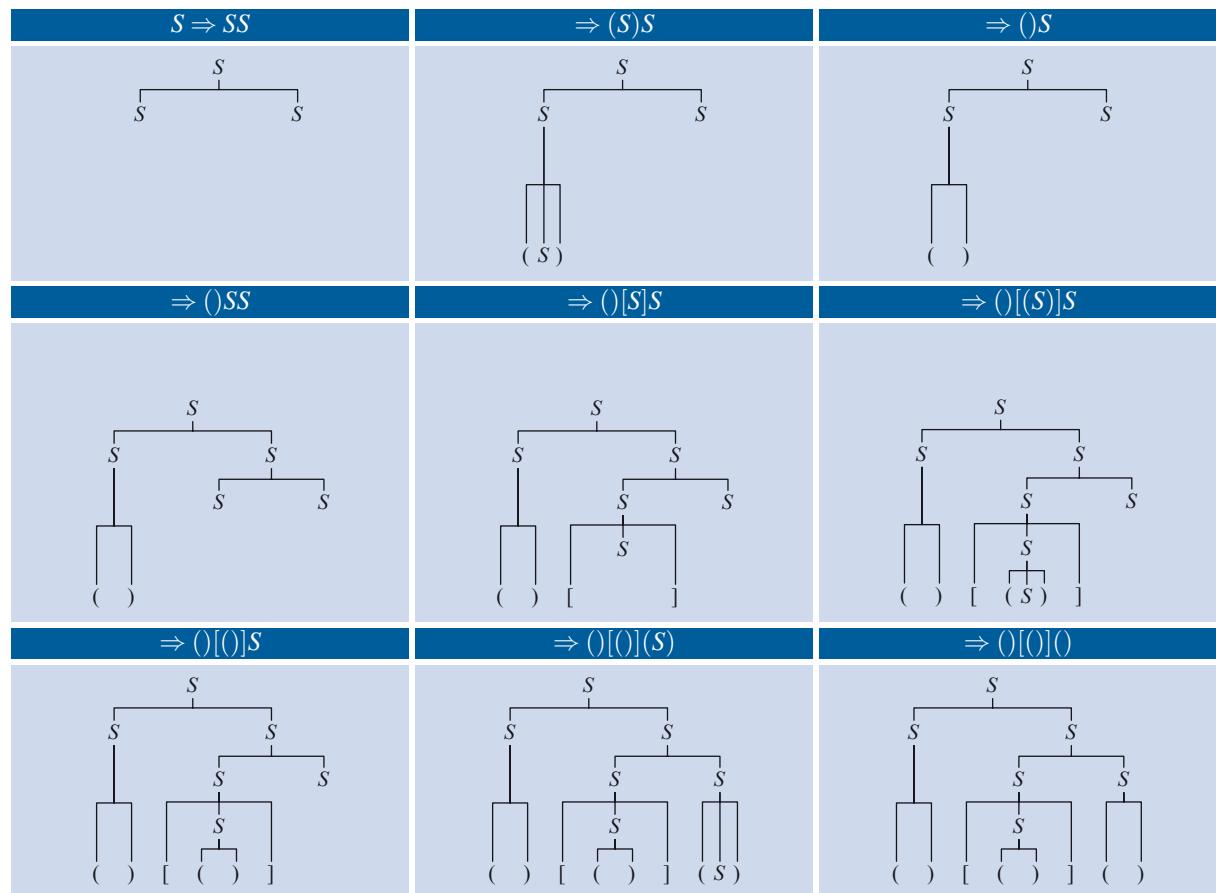


Tabelle 4.3: Schrittweise Konstruktion eines Syntaxbaums für das Dyck-Wort ()[()][()

In diesem Beispiel führt sowohl die Links- als auch die Rechtsableitung zu dem gleichen Wort $()[()()$. Andere Grammatiken erfüllen diese Eigenschaft nicht; dort führen die Links- und die Rechtsableitung zu unterschiedlichen Wörtern.

- Die dritte Ableitungssequenz (Tabelle 4.2 rechts) ist ebenfalls eine Linksableitung, da genau wie im ersten Beispiel in jedem Schritt das am weitesten links stehende Nonterminal ersetzt wird. Der Unterschied zwischen beiden Sequenzen besteht alleine in der Wahl der Produktionen, die auf das entsprechende Nonterminal angewendet werden. Das Beispiel unterstreicht, dass die Links- und die Rechtsableitung im Allgemeinen nicht eindeutig sind.

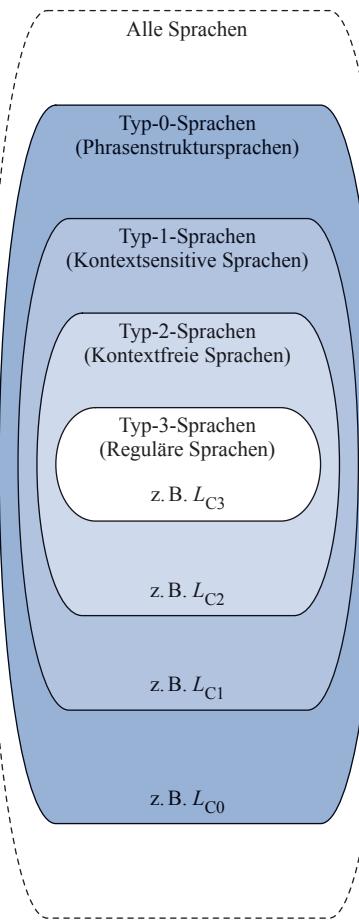
Jede Ableitungssequenz lässt sich in einen *Syntaxbaum* übersetzen. Dieser ist so angelegt, dass jeder innere Knoten einem Nonterminal aus der Ableitungssequenz entspricht und die Blätter aneinandergereiht das abgeleitete Wort ergeben. Für die Konstruktion des Syntaxbaums wird zunächst die Wurzel mit dem Startsymbol markiert. Anschließend werden die Blätter entsprechend den angewendeten Produktionen expandiert oder im Falle einer Regel der Form $S \rightarrow \epsilon$ aus dem Baum entfernt. Tabelle 4.3 zeigt die schrittweise Konstruktion des Syntaxbaums für die erste Ableitungssequenz aus Tabelle 4.2.

Erzeugen wir für jede der eingeführten Ableitungssequenzen einen Syntaxbaum, so erhalten wir die in Tabelle 4.4 dargestellten Ergebnisse. Ein Vergleich der beiden unteren Bäume zeigt, dass zwei verschiedene Ableitungen die gleiche Baumdarstellung ergeben können. Hierfür verantwortlich ist die Eigenschaft von Syntaxbäumen, von der Reihenfolge der angewendeten Regeln zu abstrahieren. Da aber umgekehrt jede Ableitungssequenz zu einem eindeutigen Syntaxbaum führt, müssen zwei strukturell unterschiedliche Syntaxbäume stets aus zwei verschiedenen Ableitungssequenzen entstanden sein.

Eine Grammatik G heißt *eindeutig*, wenn alle Ableitungen eines Worts $\omega \in \mathcal{L}(G)$ immer zu demselben Syntaxbaum führen. Andernfalls bezeichnen wir G als *mehrdeutig*. Beachten Sie, dass die Mehrdeutigkeit die Eigenschaft einer Grammatik und nicht die Eigenschaft einer Sprache ist, da sich eine mehrdeutige Grammatik G häufig in eine eindeutige Grammatik G' mit $\mathcal{L}(G) = \mathcal{L}(G')$ überführen lässt. Nichtsdestotrotz existieren Sprachen, die ausschließlich durch mehrdeutige Grammatiken erzeugt werden. Eine solche Sprache bezeichnen wir als *inhärent mehrdeutig*. Beachten Sie ferner, dass auch in eindeutigen Grammatiken mehrere verschiedene Ableitungen für ein Wort ω existieren können. Das Kriterium der Eindeutigkeit stellt lediglich sicher, dass alle Ableitungen zu ein und demselben Syntaxbaum führen.

Ableitungssequenz 1
$\begin{array}{l} S \Rightarrow SS \\ \Rightarrow (S)S \\ \dots \end{array}$
Ableitungssequenz 2
$\begin{array}{l} S \Rightarrow SS \\ \Rightarrow S(S) \\ \dots \end{array}$
Ableitungssequenz 3
$\begin{array}{l} S \Rightarrow SS \\ \Rightarrow SSS \\ \dots \end{array}$

Tabelle 4.4: Syntaxbäume der drei Ableitungssequenzen aus Tabelle 4.2



4.2 Chomsky-Hierarchie

Formale Grammatiken sind ein mächtiges Werkzeug für die Erzeugung der unterschiedlichsten Sprachen. Die Spanne reicht von einelementigen Wortmengen bis hin zu komplexen Sprachgebilden, die in ihrer Natur dem gesprochenen Wort gleichen. Die Struktur der Produktionen einer Grammatik G hat dabei einen maßgeblichen Einfluss auf die Eigenschaften der erzeugten Sprache $\mathcal{L}(G)$. Im Jahr 1957 veröffentlichte der amerikanische Sprachwissenschaftler Noam Chomsky ein Regelwerk, mit dessen Hilfe sich formale Grammatiken in vier Klassen einteilen lassen [17]:

- **Phrasenstrukturgrammatiken (Typ-0-Grammatiken)**
Jede Grammatik ist per Definition immer auch eine Typ-0-Grammatik. Insbesondere unterliegt die Struktur der Produktionen keinen weiteren als den in Definition 4.2 vereinbarten Einschränkungen.
- **Kontextsensitive Grammatiken (Typ-1-Grammatiken)**
Eine Grammatik heißt *kontextsensitiv*, falls für alle Produktionsregeln $l \rightarrow r$ die Beziehung $|r| \geq |l|$ gilt. In der Konsequenz kann die Anwendung einer Produktion niemals zu einer Verkürzung der abgeleiteten Zeichenkette führen.
- **Kontextfreie Grammatiken (Typ-2-Grammatiken)**
Typ-2-Grammatiken sind dadurch charakterisiert, dass die linke Seite einer Produktionsregel ausschließlich aus einer einzigen Variablen besteht. Für alle Produktionen $l \rightarrow r$ gilt also $l \in V$.
- **Reguläre Grammatiken (Typ-3-Grammatiken)**
Reguläre Grammatiken sind kontextfrei und besitzen die zusätzliche Eigenschaft, dass die rechte Seite einer Produktion entweder aus dem leeren Wort ϵ oder einem Terminalsymbol, gefolgt von einem Nonterminal, besteht. Formal gesprochen besitzt jede Produktion die Form $l \rightarrow r$ mit $l \in V$ und $r \in \{\epsilon\} \cup \Sigma V$.

Abbildung 4.6: Die Chomsky-Hierarchie teilt die Menge der formalen Sprachen in vier Typklassen ein. Eine Sprache L ist eine Typ- n -Sprache, wenn eine Typ- n -Grammatik existiert, die L erzeugt. Zwischen den Sprachklassen besteht eine echte Inklusionsbeziehung, d. h., für alle n mit $0 \leq n < 3$ gilt $\mathcal{L}_n \supset \mathcal{L}_{n+1}$ und $\mathcal{L}_n \neq \mathcal{L}_{n+1}$.

Eine Sprache L bezeichnen wir als Typ- n -Sprache, wenn eine Typ- n -Grammatik G existiert, die L erzeugt. Die Menge aller Typ- n -Sprachen notieren wir mit dem Symbol \mathcal{L}_n .

Zwischen den verschiedenen Sprachklassen besteht die folgende Inklusionsbeziehung:

$$\mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3$$

Die Inklusionsbeziehung ist eine echte, d. h., es gibt zu jeder Klasse \mathcal{L}_n mit $0 \leq n < 3$ eine Sprache L , die in \mathcal{L}_n , jedoch nicht in \mathcal{L}_{n+1} enthalten ist (vgl. Abbildung 4.6). Die folgenden Beispiele geben einen Eindruck über die vier Sprachklassen:

- $L_{C3} := \{(ab)^n \mid n \in \mathbb{N}^+\}$
ist eine Typ-3-Sprache.
- $L_{C2} := \{a^n b^n \mid n \in \mathbb{N}^+\}$
ist eine Typ-2-Sprache, aber keine Typ-3-Sprache.
- $L_{C1} := \{a^n b^n c^n \mid n \in \mathbb{N}^+\}$
ist eine Typ-1-Sprache, aber keine Typ-2-Sprache.
- $L_{C0} := \{\omega \mid \omega \text{ codiert eine terminierende Turing-Maschine}\}$
ist eine Typ-0-Sprache, aber keine Typ-1-Sprache.

Bei der Untersuchung der verschiedenen Sprachklassen spielen insbesondere die Abschlusseigenschaften eine wichtige Rolle. Hierunter verbirgt sich die Frage, ob die Verknüpfung zweier \mathcal{L}_n -Sprachen zu einer Sprache führt, die wiederum in der Sprachklasse \mathcal{L}_n liegt oder aus dieser herausfällt. Die folgenden Verknüpfungen sind in diesem Zusammenhang von Bedeutung:

- Vereinigung
Ist mit $L_1, L_2 \in \mathcal{L}_n$ auch die Sprache $L_1 \cup L_2 \in \mathcal{L}_n$?
- Durchschnitt
Ist mit $L_1, L_2 \in \mathcal{L}_n$ auch die Sprache $L_1 \cap L_2 \in \mathcal{L}_n$?
- Komplement
Ist mit $L \in \mathcal{L}_n$ auch die Sprache $\Sigma^* \setminus L \in \mathcal{L}_n$?
- Konkatenation
Ist mit $L_1, L_2 \in \mathcal{L}_n$ auch die Sprache $L_1 L_2 \in \mathcal{L}_n$?
- Kleene'sche Hülle
Ist mit $L \in \mathcal{L}_n$ auch die Sprache $L^* \in \mathcal{L}_n$?

In den nächsten Abschnitten werden wir die vier eingeführten Sprachklassen genauer untersuchen und dabei zeigen, dass die Beispielsprachen L_{C1} bis L_{C3} tatsächlich in die angegebenen Sprachklassen fallen. Um die ein wenig mysteriös daherkommende Sprache L_{C0} zu verstehen, fehlen uns noch wichtige Grundbegriffe. Doch seien Sie unbesorgt: in Kapitel 6 wird sie sich fast von selbst erklären.

Chomskys Typ-0-Grammatiken sind eng mit den *Semi-Thue-Systemen* verwandt, die der norwegische Mathematiker Axel Thue im Jahr 1914 zur Untersuchung von Ableitungskalkülen ersann [96]. Formal ist ein Semi-Thue-System über einem Alphabet Σ nichts weiter als eine Relation $S \subseteq \Sigma^* \times \Sigma^*$. Die Elemente von S entsprechen den Produktionen einer Grammatik und werden genau wie diese in der Form $u \rightarrow v$ notiert. Damit entpuppen sich Semi-Thue-Systeme als eine primitive Beschreibungsform für Grammatiken, die auf die nötigsten Bestandteile reduziert wurde.

Semi-Thue-Systeme und Grammatiken definieren eine Ableitungsrelation nach dem exakt gleichen Schema. Eine Zeichenkette der Form lur lässt sich in lvr umformen (geschrieben als $lur \Rightarrow lvr$), wenn eine Produktion der Form $u \rightarrow v$ existiert. Eine von Grammatiken bekannte Unterteilung in Terminale und Nonterminale existiert in Semi-Thue-Systemen nicht. Ebenfalls wird auf die Definition eines dedizierten Startsymbols verzichtet. Eine spezielle Variante sind Semi-Thue-Systeme mit einer symmetrischen Ableitungsrelation. Ist mit $u \rightarrow v$ immer auch die Produktion $v \rightarrow u$ enthalten, so sprechen wir von einem *Thue-System*.



Axel Thue (1863 – 1922)

■ Grammatik

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

■ Produktionsmenge P

$$\begin{aligned} S &\rightarrow aB \\ B &\rightarrow bC \\ C &\rightarrow \varepsilon \mid aB \end{aligned}$$

■ Sprache

$$\begin{aligned} \mathcal{L}(G) = \{ &ab, \\ &abab, \\ &ababab, \\ &abababab, \dots \} \end{aligned}$$

Abbildung 4.7: Mit Hilfe einer regulären Grammatik lässt sich die formale Sprache $L_{C3} = \{(ab)^n \mid n \in \mathbb{N}^+\}$ erzeugen.

4.3 Reguläre Sprachen

Die Menge der regulären Sprachen (Typ-3-Sprachen) ist die kleinste Sprachklasse in der Chomsky-Hierarchie. Obwohl sie über eine vergleichsweise einfache Struktur verfügen, nehmen die Sprachen einen prominenten Platz in der Informatik ein. So sind viele Datenformate regulär und die Suchmuster, die uns z. B. auf der Shell-Ebene das Auffinden von Dateien erlauben, sind ebenfalls nichts anderes als reguläre Ausdrücke.

4.3.1 Definition und Eigenschaften

Wie in Abschnitt 4.2 dargelegt, unterliegen die Produktionen einer regulären Grammatik erheblichen Einschränkungen. Nur solche Regeln sind erlaubt, deren linke Seite aus einem Nonterminal und deren rechte Seite entweder aus dem leeren Wort ε oder einem Terminalzeichen, gefolgt von einem Nonterminal, besteht.

Unter Beachtung dieser Einschränkungen lässt sich die Sprache

$$L_{C3} = \{(ab)^n \mid n \in \mathbb{N}^+\}$$

mit der in Abbildung 4.7 dargestellten Grammatik erzeugen. Abbildung 4.8 demonstriert, wie sich die Wörter $abab$ und $ababab$ aus dem Startsymbol ableiten lassen. Ein Blick auf die Ableitungssequenzen entlarvt die Rolle des Nonterminals B . Es tritt immer dann auf, wenn zuletzt ein a erzeugt wurde, und stellt sicher, dass die Zeichenkette mit einem b fortgesetzt wird.

Die einfache Struktur der Produktionen einer regulären Grammatik wirkt sich unmittelbar auf das Erscheinungsbild der entstehenden Syntaxbäume aus. Da die rechte Seite einer Produktion aus maximal zwei Zeichen besteht und das erste immer aus der Menge der Terminalzeichen stammt, weisen die entstehenden Syntaxbäume die Struktur einer linearen Kette auf. Aufgrund dieser Eigenschaft werden reguläre Grammatiken auch als *rechtslineare Grammatiken* bezeichnet.

Die Linearitätseigenschaft sorgt dafür, dass Wörter in regulären Grammatiken immer nach dem gleichen Prinzip erzeugt werden. Ausgehend von dem leeren Wort in Form des Startsymbols fügt jede Produktion der Form $A \rightarrow \sigma B$ das Zeichen σ an und ersetzt das aktuell vorhandene Nonterminal A durch das neue Symbol B . Hierdurch wird die erzeugte Zeichenkette mit jedem Ableitungsschritt um ein einzelnes Zeichen verlängert und nach rechts durch ein wechselndes Nonterminal begrenzt.

Dieses fungiert als Zustandsmerker und reglementiert die Anzahl der im nächsten Schritt anwendbaren Produktionen. Die Epsilon-Regeln der Form $A \rightarrow \epsilon$ spielen ebenfalls eine entscheidende Rolle. Wird eine von ihnen angewendet, so verschwindet das Nonterminal aus der erzeugten Zeichenkette und der Produktionsprozess kommt zum Erliegen.

Für einige Anwendungsfälle ist es wünschenswert, möglichst viele Regeln der Form $l \rightarrow \epsilon$ aus der Menge der Produktionen zu entfernen. In der Tat können wir auf die meisten Epsilon-Regeln verzichten, indem wir auch Produktionen zulassen, deren rechte Seiten aus einem einzigen Terminalzeichen bestehen.

In unserem Beispiel lässt sich die Epsilon-Regel $C \rightarrow \epsilon$ eliminieren, indem die Grammatik durch die zusätzliche Regel $B \rightarrow b$ erweitert wird. Wir erhalten das folgende Ergebnis:

$$\begin{array}{l} S \rightarrow aB \\ B \rightarrow b \\ B \rightarrow bC \\ C \rightarrow aB \end{array}$$

In der entstandenen Form besteht die rechte Seite einer Produktion nur noch aus einem isolierten Terminalzeichen oder einem Terminalzeichen, gefolgt von einem Nonterminal. Würden wir auf der rechten Seite mehr als ein Terminalzeichen zulassen, so ließe sich unsere Beispielgrammatik sogar noch weiter vereinfachen:

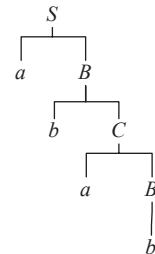
$$\begin{array}{l} S \rightarrow aB \\ B \rightarrow b \\ B \rightarrow baB \end{array}$$

Wir können die gezeigte Umformung auf beliebige Grammatiken anwenden und auf diese Weise fast alle Epsilon-Regeln nach und nach eliminieren. Die einzige Ausnahme bildet die Regel $S \rightarrow \epsilon$. Würden wir diese – falls vorhanden – aus der Menge der Produktionen entfernen, so ließe sich das leere Wort ϵ nicht mehr ableiten.

In Abschnitt 4.1 haben wir mit dem Wortproblem, dem Leerheitsproblem, dem Endlichkeitsproblem und dem Äquivalenzproblem vier wichtige Fragestellungen für die Untersuchung formaler Sprachen eingeführt. Alle vier sind für reguläre Sprachen *entscheidbar*, d. h., es existiert ein Verfahren, das für alle Eingaben feststellt, ob die betreffende Eigenschaft zutrifft oder nicht (vgl. Abbildung 4.9). Die Entscheidbarkeitseigenschaften wollen wir für den Moment ohne Beweis akzeptieren. In Abschnitt 5.4 wird uns deren Gültigkeit im Rahmen der Betrachtungen über endliche Automaten fast von selbst in den Schoß fallen.

■ Ableitung von $abab$

$$\begin{array}{l} S \Rightarrow aB \\ \Rightarrow abC \\ \Rightarrow abaB \\ \Rightarrow ababC \\ \Rightarrow abab \end{array}$$



■ Ableitung von $ababab$

$$\begin{array}{l} S \Rightarrow aB \\ \Rightarrow abC \\ \Rightarrow abaB \\ \Rightarrow ababC \\ \Rightarrow ababaB \\ \Rightarrow abababC \\ \Rightarrow ababab \end{array}$$

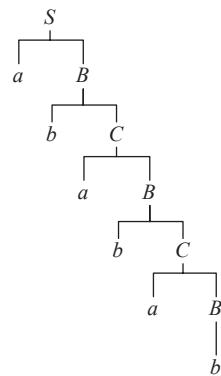


Abbildung 4.8: Von regulären Grammatiken erzeugte Syntaxbäume besitzen die Struktur einer nach rechts unten geneigten linearen Kette.

Entscheidungsprobleme regulärer Sprachen			
Problem	Eingabe	Fragestellung	Entscheidbar?
Wortproblem	Sprache L , Wort $\omega \in \Sigma^*$	Ist $\omega \in L$?	✓ Ja
Leerheitsproblem	Sprache L	Ist $L = \emptyset$?	✓ Ja
Endlichkeitsproblem	Sprache L	Ist $ L < \infty$?	✓ Ja
Äquivalenzproblem	Sprachen L_1 und L_2	Ist $L_1 = L_2$?	✓ Ja

Abschlusseigenschaften regulärer Sprachen			
Operation	Eingabe	Fragestellung	Erfüllt?
Vereinigung	Sprache $L_1, L_2 \in \mathcal{L}_3$	Ist $L_1 \cup L_2 \in \mathcal{L}_3$?	✓ Ja
Schnitt	Sprache $L_1, L_2 \in \mathcal{L}_3$	Ist $L_1 \cap L_2 \in \mathcal{L}_3$?	✓ Ja
Komplement	Sprache $L \in \mathcal{L}_3$	Ist $\Sigma^* \setminus L \in \mathcal{L}_3$?	✓ Ja
Produkt	Sprache $L_1, L_2 \in \mathcal{L}_3$	Ist $L_1 L_2 \in \mathcal{L}_3$?	✓ Ja
Stern	Sprache $L \in \mathcal{L}_3$	Ist $L^* \in \mathcal{L}_3$?	✓ Ja

Abbildung 4.9: Eigenschaften regulärer Sprachen in der Übersicht

4.3.2 Pumping-Lemma für reguläre Sprachen

Wir wollen uns an dieser Stelle ein wenig näher an die Grenzen herantasten, die uns die eingeschränkte Struktur der Produktionen einer regulären Grammatik auferlegt. Weiter oben haben wir dargelegt, wie Wörter in regulären Grammatiken erzeugt werden. Sehen wir von der letzten Regelanwendung ab, so wird in jedem Ableitungsschritt ein Terminalzeichen und ein neues Nonterminal erzeugt. Das Nonterminal steht immer an letzter Stelle und begrenzt die Auswahl der im nächsten Schritt anwendbaren Regeln. In diesem Sinne wirkt es wie ein Zustandsspeicher, der einen Rückschluss auf die bisher erzeugten Zeichen erlaubt. Da die Menge der Nonterminale endlich ist, lassen sich während der Erzeugung eines Wortes nur endlich viele Zustände unterscheiden. Genau hierin liegt eine der grundlegenden Limitierungen regulärer Sprachen verborgen, die wir im Folgenden genauer untersuchen wollen.

Betrachten wir eine Ableitungssequenz, die mehr Ableitungsschritte enthält als Nonterminale zur Verfügung stehen, so muss mindestens ein

Nonterminal mehrfach auftauchen. Bezeichnen wir dieses Nonterminalzeichen mit A , so besitzt die Ableitungssequenz die folgende Form:

$$\begin{aligned} & \dots \\ \Rightarrow & \underbrace{\sigma_1 \sigma_2 \dots \sigma_i}_u A \\ \Rightarrow & \underbrace{\sigma_1 \sigma_2 \dots \sigma_i}_{u} \underbrace{\sigma_{i+1} \sigma_{i+2} \dots \sigma_j}_v A \\ \Rightarrow & \underbrace{\sigma_1 \sigma_2 \dots \sigma_i}_{u} \underbrace{\sigma_{i+1} \sigma_{i+2} \dots \sigma_j}_{v} \underbrace{\sigma_{j+1} \sigma_{j+2} \dots \sigma_k}_w A \end{aligned}$$

Die Ableitungssequenz zeigt, dass es möglich ist, aus dem Nonterminal A die Zeichenkette

$$vA = \sigma_{i+1} \sigma_{i+2} \dots \sigma_j A$$

abzuleiten, die mindestens ein Terminalzeichen enthält ($|v| \geq 1$). Da die erzeugte Sequenz erneut mit dem Nonterminal A endet, lassen sich zusätzlich die Sequenzen

$$\begin{aligned} v^2A &= \sigma_{i+1} \sigma_{i+2} \dots \sigma_j \sigma_{i+1} \sigma_{i+2} \dots \sigma_j A \\ v^3A &= \sigma_{i+1} \sigma_{i+2} \dots \sigma_j \sigma_{i+1} \sigma_{i+2} \dots \sigma_j \sigma_{i+1} \sigma_{i+2} \dots \sigma_j A \\ \dots &= \dots \end{aligned}$$

ableiten (vgl. Abbildung 4.10). Die angestellte Überlegung zeigt zwei erlei: Zum einen lässt sich jedes hinreichend lange Wort einer regulären Sprache in der Form uvw ausdrücken. Zum anderen müssen neben dem Wort uvw auch die Wörter $uv^i w$ für alle $i \in \mathbb{N}$ in der Sprache enthalten sein. Genau dies ist die Kernaussage des *Pumping-Lemmas* für reguläre Sprachen.

Satz 4.1 (Pumping-Lemma für reguläre Sprachen)

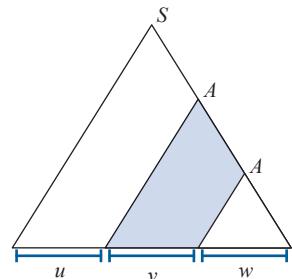
Für jede reguläre Sprache L existiert ein $j \in \mathbb{N}$, so dass sich alle Wörter $\omega \in L$ mit $|\omega| \geq j$ in der folgenden Form darstellen lassen:

$$\omega = uvw \quad \text{mit } |v| \geq 1 \text{ und } |uv| \leq j$$

Dann ist mit ω auch das Wort $uv^i w$ für alle $i \in \mathbb{N}$ in L enthalten.

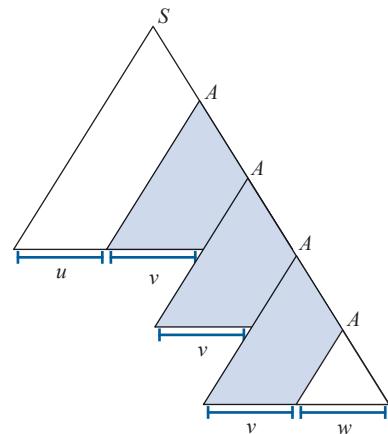
Das Pumping-Lemma gibt uns ein leistungsfähiges Instrument an die Hand, um eine Sprache als nicht regulär zu entlarven. Die Beweisführung folgt dabei immer dem gleichen Muster. Zunächst wird für die

■ Wortstruktur regulärer Sprachen



Überschreitet die Anzahl der Ableitungsschritte eines Worts eine gewisse Grenze j , so muss aufgrund der endlichen Anzahl der Nonterminale mindestens eines davon mehrfach im Syntaxbaum auftauchen (hier das Nonterminal A). Folgerichtig lässt sich jedes hinreichend lange Wort in der Form uvw darstellen mit $|v| \geq 1$ und $|uv| \leq j$.

■ „Aufpumpen“ des Mittelstücks



Die Ableitung des Mittelstücks lässt sich beliebig oft wiederholen. Damit sind neben uvw immer auch die Wörter $uv^i w$ für alle $i \in \mathbb{N}$ in der Sprache enthalten.

Abbildung 4.10: Veranschaulichung des Pumping-Lemmas anhand der Syntaxbäume regulärer Grammatiken

Vergleichen wir die Sprache

$$L_{C2} := \{a^n b^n \mid n \in \mathbb{N}^+\}$$

mit der weiter oben diskutierten Sprache

$$L_{C3} := \{(ab)^n \mid n \in \mathbb{N}^+\},$$

so erscheint der Unterschied auf den ersten Blick nur marginal zu sein. Trotzdem lässt sich L_{C2} , anders als L_{C3} , nicht mithilfe einer regulären Grammatik erzeugen. Schuld daran ist die Anordnung der Symbole a und b . Erzeugen wir die einzelnen Zeichen eines Worts, wie in regulären Grammatiken gefordert, von links nach rechts, so müssen wir uns für alle Wörter der Form $a^n b^n$ zunächst die Anzahl der produzierten a 's merken, um anschließend die richtige Anzahl von b 's hervorbringen zu können. Da der Wert von n nach oben unbeschränkt ist, wären zu diesem Zweck unendlich viele Zustände notwendig. Aufgrund der endlichen Anzahl an Nonterminalen – unseren Zustandsmerkern – ist dies ein unmögliches Unterfangen.

Um die Wörter der Sprache L_{C3} zu erzeugen, ist deutlich weniger logistische Arbeit erforderlich. Hier müssen wir uns lediglich merken, ob ein Terminalsymbol a erzeugt wurde oder nicht. Im ersten Fall müssen wir zunächst ein b produzieren, um ein gültiges Wort zu erhalten. Im zweiten Fall haben wir bereits ein korrektes Wort der Sprache vor uns.

untersuchte Sprache L gezeigt, dass sich ein Wort $\omega \in L$ in der Form uvw darstellen lässt. Dem Pumping-Lemma folgend muss dann auch das Wort $uv^i w$ in der Sprache enthalten sein. Ist dies nicht der Fall, so kann L keine reguläre Sprache sein.

Mithilfe des Pumping-Lemmas können wir z. B. beweisen, dass die in Abschnitt 4.2 eingeführte Sprache

$$L_{C2} := \{a^n b^n \mid n \in \mathbb{N}^+\} \quad (4.2)$$

nicht regulär ist. Wäre L_{C2} eine reguläre Sprache, so würde nach dem Pumping-Lemma ein $j \in \mathbb{N}$ existieren, so dass sich jedes Wort ω mit $|\omega| \geq j$ in der Form uvw darstellen lässt mit $|v| \geq 1$ und $|uv| \leq j$. Für das Wort $a^j b^j$ folgt hieraus, dass der (nichtleere) Mittelteil v nur aus a 's bestehen kann. Mit uvw wäre dann aber auch das Wort

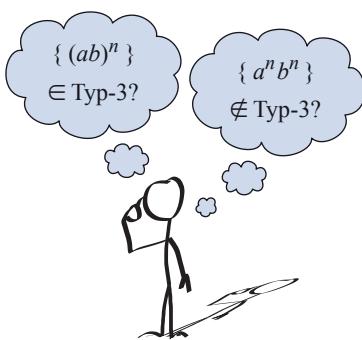
$$uv^2w = a^j a^{|v|} b^j = a^j \underbrace{a \dots a}_{\geq 1} b^j$$

in L enthalten, im Widerspruch zum Aufbau von L_{C2} .

Beachten Sie an dieser Stelle, dass uns das Pumping-Lemma ein Hilfsmittel an die Hand gibt, um nachzuweisen, dass eine Sprache L *nicht* regulär ist. Insbesondere darf die Schlussrichtung von Satz 4.1 nicht umgekehrt werden. Auch wenn die Wörter einer Sprache alle Eigenschaften des Pumping-Lemmas erfüllen, ist diese nicht notwendigerweise regulär.

4.3.3 Satz von Myhill-Nerode

Ende der Sechzigerjahre haben die amerikanischen Mathematiker John Myhill und Anil Nerode ein Kriterium entdeckt, über das sich reguläre Sprachen eindeutig charakterisieren lassen [76, 77]. Im Gegensatz zum Pumping-Lemma handelt es sich dabei um ein notwendiges und zugleich hinreichendes Kriterium, d. h., wir können es sowohl für den Beweis der Regularität als auch für den Beweis der Nichtregularität einer Sprache verwenden. In diesem Abschnitt werden wir die Entdeckung von Myhill und Nerode genauer untersuchen und schicken zu diesem Zweck eine wichtige Begriffsdefinition voraus:





Definition 4.3 (Nerode-Relation)

Sei $L \subseteq \Sigma^*$ eine formale Sprache. Auf der Menge Σ^* definieren wir die *Nerode-Relation* \sim_L über die Beziehung

$$\omega_1 \sim_L \omega_2 \Leftrightarrow (\omega_1 v \in L \Leftrightarrow \omega_2 v \in L \text{ für alle } v \in \Sigma^*)$$

In Worten besagt die Definition das Folgende: Zwei Wörter ω_1 und ω_2 stehen in Relation zueinander, wenn sie beide zu L oder beide nicht zu L gehören und eine Verlängerung um einen beliebigen Suffix $v \in \Sigma^*$ daran nichts ändert.

Wir wollen das Gesagte mit Leben füllen und betrachten erneut die weiter oben eingeführte Sprache

$$L_{C2} := \{a^n b^n \mid n \in \mathbb{N}^+\}$$

In dieser Sprache stehen die beiden Beispielwörter

$$\omega_1 := aab$$

$$\omega_2 := aaabb$$

in Relation zueinander, da sie genau dann zu L gehören, wenn sie um den Suffix b ergänzt werden. Das Wort

$$\omega_3 := aaab$$

steht hingegen weder mit ω_1 noch mit ω_2 in Relation, da die Verlängerung von ω_3 um b noch kein Wort ergibt, das in L enthalten ist.

Es ist leicht einzusehen, dass wir mit \sim_L eine Äquivalenzrelation vor uns haben; die Nerode-Relation ist offensichtlich reflexiv, symmetrisch und transitiv. Das bedeutet, dass die Wörter auf der Menge Σ^* durch \sim_L zu Äquivalenzklassen zusammengefasst werden. Abbildung 4.11 zeigt, welche Klassen für unsere Beispielsprache L_{C2} entstehen. Die Anzahl der Äquivalenzklassen bezeichnen wir als den *Index* von \sim_L .

Unser Begriffsgerüst ist nun weit genug geschärft, um die inhaltliche Aussage des Satzes von Myhill-Nerode zu verstehen:



Satz 4.2 (Satz von Myhill-Nerode)

Für eine formale Sprache $L \subseteq \Sigma^*$ gilt:

$$L \text{ ist regulär} \Leftrightarrow \text{Der Index von } \sim_L \text{ ist endlich}$$

■ $L_{C2} := \{a^n b^n \mid n \in \mathbb{N}^+\}$

Äquivalenzklassen:

$$\{ab, aabb, aaabbb, \dots\},$$

$$\{a, aab, aaabb, aaaabb, \dots\},$$

$$\{aa, aaab, aaaabb, aaaaabb, \dots\},$$

$$\{aaa, aaaaab, aaaaabb, aaaaaabb, \dots\},$$

...

Abbildung 4.11: Durch die Nerode-Relation $\sim_{L_{C2}}$ wird die Menge Σ^* in unendlich viele Äquivalenzklassen aufgeteilt.

- $L_{C3} := \{(ab)^n \mid n \in \mathbb{N}^+\}$

Äquivalenzklassen:

$\{ab, abab, ababab, \dots\}$,
 $\{aba, ababa, abababa, \dots\}$,
 $\{\text{alle anderen Wörter}\}$

Abbildung 4.12: Durch die Nerode-Relation $\sim_{L_{C3}}$ wird die Menge Σ^* in genau drei Äquivalenzklassen aufgeteilt.

Mit Satz 4.2 erhalten wir ein genauso einfaches wie mächtiges Instrument an die Hand, um die Frage nach der Regularität einer Sprache L zu entscheiden. Das einzige, was wir dafür benötigen, ist die Information über die Anzahl der von \sim_L erzeugten Äquivalenzklassen.

Im Handumdrehen können wir mit dem Satz von Myhill-Nerode auch die Frage nach der Regularität von L_{C2} entscheiden. Da \sim_L unendlich viele Äquivalenzklassen erzeugt, kann L_{C2} nicht regulär sein.

Zu einem anderen Ergebnis kommen wir, wenn wir die Sprache

$$L_{C3} := \{(ab)^n \mid n \in \mathbb{N}^+\}$$

betrachten. Wie in Abbildung 4.12 gezeigt, teilt \sim_L die Menge Σ^* in genau 3 Äquivalenzklassen auf. Die Anzahl ist endlich und die Sprache deshalb regulär.

Für den Moment wollen wir uns damit begnügen, den Satz von Myhill-Nerode in seiner inhaltlichen Bedeutung zu verstehen. In Abschnitt 5.4.4 werden wir ihn erneut aufgreifen, nachdem wir den Begriff des minimierten endlichen Automaten eingeführt haben. Der dort aufgezeigte Zusammenhang zwischen regulären Sprachen und endlichen Automaten wird uns im Vorbeigehen die Begründung liefern, warum der Satz von Myhill-Nerode tatsächlich gilt. Haben Sie also noch ein wenig Geduld.

4.3.4 Reguläre Ausdrücke

Reguläre Sprachen lassen sich elegant mithilfe *regulärer Ausdrücke* beschreiben. Formal sind diese folgendermaßen definiert:



Definition 4.4 (Syntax regulärer Ausdrücke)

Mit Σ sei ein beliebiges Alphabet gegeben. Reg_Σ , die Menge der *regulären Ausdrücke* über Σ , wird induktiv durch die folgenden Regeln gebildet:

- $\emptyset, \epsilon \in Reg_\Sigma$
- $\Sigma \subset Reg_\Sigma$
- Mit $r \in Reg_\Sigma$ und $s \in Reg_\Sigma$ sind auch rs und $(r \mid s) \in Reg_\Sigma$
- Mit $r \in Reg_\Sigma$ sind auch (r) und $r^* \in Reg_\Sigma$



Definition 4.5 (Semantik regulärer Ausdrücke)

Sei r ein regulärer Ausdruck über dem Alphabet Σ . Die von r erzeugte Sprache $\mathcal{L}(r)$ ist induktiv definiert:

$$\begin{aligned}\mathcal{L}(\emptyset) &= \emptyset \\ \mathcal{L}(\varepsilon) &= \{\varepsilon\} \\ \mathcal{L}(a \in \Sigma) &= \{a\} \\ \mathcal{L}(rs) &= \mathcal{L}(r)\mathcal{L}(s) \\ \mathcal{L}((r | s)) &= \mathcal{L}(r) \cup \mathcal{L}(s) \\ \mathcal{L}((r)) &= \mathcal{L}(r) \\ \mathcal{L}(r^*) &= \mathcal{L}(r)^*\end{aligned}$$

Kommutativgesetz
$r s = s r$
Idempotenzgesetz
$r r = r$
Distributivgesetze
$r(s t) = rs rt$ $(s t)r = sr tr$
Neutrale Elemente
$r \emptyset = \emptyset r = r$ $r\varepsilon = \varepsilon r = r$

Für reguläre Ausdrücke gelten die in Tabelle 4.5 zusammengefassten Rechenregeln.

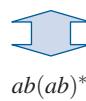
Die Beispiele in Abbildung 4.13 demonstrieren, wie sich Sprachen mithilfe von regulären Ausdrücken beschreiben lassen. Dass wir zu jeder der angegebenen Grammatiken einen regulären Ausdruck finden konnten, der die gleiche Sprache erzeugt, ist bei weitem kein Zufall. In der Tat lässt sich zeigen, dass zu jeder Grammatik ein äquivalenter regulärer Ausdruck existiert und umgekehrt. Zwischen regulären Grammatiken und regulären Ausdrücken besteht somit nur ein äußerlicher Unterschied; beide erzeugen dieselbe Sprachklasse \mathcal{L}_3 . In Kapitel 5 wird dieser elementare Zusammenhang in ein helleres Licht gerückt werden. Dort werden wir den Begriff des endlichen Automaten einführen und zeigen, wie sich reguläre Grammatiken und reguläre Ausdrücke eins zu eins auf den Automatenbegriff reduzieren lassen.

Reguläre Ausdrücke besitzen eine große Bedeutung in der praktischen Informatik. Sie werden von vielen Kommandozeilenwerkzeugen z. B. für die Spezifikation von Suchmustern verwendet und sind damit insbesondere den Benutzern UNIX-ähnlicher Betriebssysteme wohlvertraut. Tabelle 4.6 gibt eine Übersicht über die Syntax, in der reguläre Ausdrücke von typischen UNIX-Werkzeugen wie Grep oder Sed verstanden werden [58]. Die Syntax orientiert sich im Kern an jener aus Definition 4.4, – insbesondere sind die Konkatenation (ab), die Auswahl ($a|b$) und der Kleene-Stern (*) nahezu unverändert vorhanden. Darüber hinaus werden weitere Konstrukte unterstützt, die zu keiner Erweiterung der beschreibbaren Sprachen führen. Die zusätzlichen Syntaxelemente dienen lediglich zur Verkürzung der Schreibweise und lassen sich auf die Kernkonstrukte zurückführen.

Tabelle 4.5: Rechenregeln für reguläre Ausdrücke

■ $L_1 := \{(ab)^n \mid n \in \mathbb{N}^+\}$

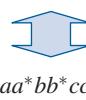
$$\begin{array}{lcl} S & \rightarrow & aB \\ B & \rightarrow & bC \\ C & \rightarrow & \varepsilon \mid aB \end{array}$$



$$ab(ab)^*$$

■ $L_2 := \{a^i b^j c^k \mid i, j, k \in \mathbb{N}^+\}$

$$\begin{array}{lcl} S & \rightarrow & aS \mid aB \\ B & \rightarrow & bB \mid bC \\ C & \rightarrow & cC \mid c \end{array}$$



$$aa^*bb^*cc^*$$

Abbildung 4.13: Reguläre Grammatiken und reguläre Ausdrücke erzeugen die gleiche Sprachklasse \mathcal{L}_3 .

Zeichen	
.	Beliebiges Zeichen außer dem Zeilenumbruch
[...]	Positivliste (jedes Zeichen innerhalb der spezifizierten Liste)
[^...]	Negativliste (jedes Zeichen außerhalb der spezifizierten Liste)
[w]	Klein- oder Großbuchstabe, Unterstrich
[W]	Ziffer, Sonderzeichen, Leerraum
[s]	Leerraum (Whitespace, Tabulator, Carriage return)
[S]	Beliebiges Zeichen außer dem Leerraum
Positionen	
^	Beginn einer Zeile
\$	Ende einer Zeile
<	Beginn eines Worts
>	Ende eines Worts
Kombinationen	
+	Der vorangegangene Ausdruck kommt mindestens einmal vor
?	Der vorangegangene Ausdruck kommt höchstens einmal vor
*	Der vorangegangene Ausdruck kommt gar nicht oder beliebig oft vor
{n,}	Der vorangegangene Ausdruck kommt mindestens n -mal vor
{n,m}	Der vorangegangene Ausdruck kommt mindestens n -mal, aber höchstens m -mal vor
{,m}	Der vorangegangene Ausdruck kommt höchstens m -mal vor
	Alternative (entweder der linke oder der rechte Ausdruck)
()	Gruppierung
Häufig benötigte Zeichen- und Symbolmengen	
:blank:]	Leerzeichen oder Tabulator
:space:]	Leerzeichen, Tabulator, newline, form feed, carriage return
:cntrl:]	Steuerzeichen
:lower:]	Kleinbuchstabe
:upper:]	Großbuchstabe
:alpha:]	Buchstabe (:lower:] oder [:upper:])
:digit:]	Ziffer
:xdigit:]	Hexadezimalziffer
:alnum:]	Alphanumerisches Zeichen (:alpha:] oder [:digit:])
:punct:]	Punktierungszeichen
:graph:]	Grafisches Zeichen (:alpha:] oder [:punct:])
:print:]	Darstellbares Zeichen (:alnum:] oder [:punct:])

Tabelle 4.6: Kommandozeilenwerkzeuge wie Grep oder Sed verwenden reguläre Ausdrücke für die Angabe von Suchmustern.

4.4 Kontextfreie Sprachen

4.4.1 Definition und Eigenschaften

Kontextfreie Grammatiken sind eine Erweiterung der regulären Grammatiken. In beiden sind die Produktionen so gestaltet, dass auf der linken Seite nur ein einzelnes Nonterminal stehen darf. Im Gegensatz zu regulären Grammatiken, die auch die Form der rechten Seite restriktieren, kann diese in kontextfreien Grammatiken aus einer beliebigen Sequenz von Terminal- und Nonterminalzeichen bestehen. Mit anderen Worten: Eine Grammatik ist genau dann kontextfrei, wenn jede Produktion die Form $l \rightarrow r$ mit $l \in V$ und $r \in (\Sigma \cup V)^*$ besitzt.

In Abschnitt 4.3 haben wir mit dem Pumping-Lemma gezeigt, dass die Sprache $\{a^n b^n \mid n \in \mathbb{N}^+\}$ nicht regulär ist und damit von keiner regulären Grammatik erzeugt werden kann. Kontextfreie Sprachen sind hingegen ausdrucksstark genug, um die Sprache zu beschreiben. Abbildung 4.14 fasst die entsprechenden Produktionsregeln zusammen.

Als weiteres Beispiel zeigt Abbildung 4.15 eine Grammatik zur Erzeugung der Sprache $\{a^i b^j a^k \mid i \in \mathbb{N}^+, j, k \in \mathbb{N}\}$. Auch diese ist kontextfrei, da die linken Seiten der Produktionen nur aus Variablen bestehen.

4.4.2 Normalformen

Die Produktionen kontextfreier Grammatiken lassen sich durch geschickte Umformung stark vereinfachen. Was wir hierunter genau zu verstehen haben, werden die nächsten beiden Abschnitte zeigen. Zunächst werden wir in Abschnitt 4.4.2.1 den Begriff der *Chomsky-Normalform* einführen und darlegen, wie sich eine kontextfreie Grammatik äquivalenzerhaltend in eine solche überführen lässt. Anschließend werden wir in Abschnitt 4.4.2.2 zeigen, wie sich kontextfreie Grammatiken mithilfe der *Backus-Naur-Form* beschreiben lassen.

4.4.2.1 Chomsky-Normalform



Definition 4.6 (Chomsky-Normalform)

Eine Grammatik $G = (V, \Sigma, P, S)$ liegt in *Chomsky-Normalform* vor, wenn alle Produktionen die Form $S \rightarrow \epsilon$, $A \rightarrow \sigma$ oder $A \rightarrow BC$ besitzen mit $A \in V$, $B, C \in V \setminus \{S\}$ und $\sigma \in \Sigma$.

Grammatik

$$G := (\{S\}, \{a, b\}, P, S)$$

Produktionsmenge P

$$S \rightarrow aSb \mid ab$$

Ableitung des Worts $aaaabbbb$

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow aaaSbb \\ &\Rightarrow aaaabbb \end{aligned}$$

Abbildung 4.14: Grammatik zur Erzeugung der Sprache $\{a^n b^n \mid n \in \mathbb{N}^+\}$

Grammatik

$$G := (\{S, A, B\}, \{a, b\}, P, S)$$

Produktionsmenge P

$$\begin{aligned} S &\rightarrow AB \mid ABA \\ A &\rightarrow aA \mid a \\ B &\rightarrow Bb \mid \epsilon \end{aligned}$$

Ableitung des Worts $aabbba$

$$\begin{aligned} S &\Rightarrow ABA \\ &\Rightarrow aABA \\ &\Rightarrow aaBA \\ &\Rightarrow aaBbA \\ &\Rightarrow aaBbbA \\ &\Rightarrow aabbA \\ &\Rightarrow aabbaA \\ &\Rightarrow aabbaa \end{aligned}$$

Abbildung 4.15: Grammatik zur Erzeugung der Sprache $\{a^i b^j a^k \mid i \in \mathbb{N}^+, j, k \in \mathbb{N}\}$

Ausgangspunkt	Schritt 1	Schritt 2	Schritt 3	Schritt 4
$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
$S \rightarrow ABA$	$S \rightarrow A$	$S \rightarrow aA$	$S \rightarrow V_a A$	$S \rightarrow V_a A$
$A \rightarrow aA$	$S \rightarrow ABA$	$S \rightarrow a$	$S \rightarrow a$	$S \rightarrow a$
$A \rightarrow a$	$S \rightarrow AA$	$S \rightarrow ABA$	$S \rightarrow ABA$	$S \rightarrow S_2 A$
$B \rightarrow Bb$	$S \rightarrow AA$	$S \rightarrow AA$	$S \rightarrow AA$	$S \rightarrow AA$
$B \rightarrow \epsilon$	$A \rightarrow aA$	$A \rightarrow a$	$A \rightarrow V_a A$	$A \rightarrow V_a A$
	$B \rightarrow Bb$	$A \rightarrow a$	$A \rightarrow a$	$A \rightarrow a$
	$B \rightarrow b$	$B \rightarrow Bb$	$B \rightarrow BV_b$	$A \rightarrow a$
		$B \rightarrow b$	$B \rightarrow b$	$B \rightarrow BV_b$
			$V_a \rightarrow a$	$B \rightarrow b$
			$V_b \rightarrow b$	$V_a \rightarrow a$
				$V_b \rightarrow b$

Tabelle 4.7: Schrittweise Erzeugung der Chomsky-Normalform

Um eine kontextfreie Grammatik G mit $\epsilon \notin \mathcal{L}(G)$ in die Chomsky-Normalform zu überführen, sind vier Schritte zu absolvieren. In Tabelle 4.7 werden diese für das Beispiel aus Abbildung 4.15 durchlaufen.

■ Schritt 1: Elimination der ϵ -Regeln

Alle Regeln der Form $A \rightarrow \epsilon$ werden eliminiert, indem die Ersetzung von A durch ϵ in allen anderen Regel vorweggenommen wird. Da $\epsilon \notin \mathcal{L}(G)$ ist, wird das leere Wort hierdurch vollständig aus den Produktionen entfernt.

■ Schritt 2: Elimination von Kettenregeln

Jede Produktion der Form $A \rightarrow B$ mit $A, B \in V$ wird als *Kettenregel* bezeichnet. Diese tragen nicht zur Produktion von Terminalzeichen bei und lassen sich ebenfalls eliminieren. Hierzu gehen wir wie in Schritt 1 vor und nehmen die Ersetzung der rechten Seite vorweg.

■ Schritt 3: Separation von Terminalzeichen

Jedes Terminalzeichen σ , das in Kombination mit anderen Symbolen auftaucht, wird durch ein neues Nonterminal V_σ ersetzt und die Menge der Produktionen durch die Regel $V_\sigma \rightarrow \sigma$ ergänzt.

■ Schritt 4: Elimination von mehrelementigen Nonterminalketten

Alle Produktionen der Form $A \rightarrow B_1 B_2 \dots B_n$ werden in die Produktionen $A \rightarrow A_{n-1} B_n, A_{n-1} \rightarrow A_{n-2} B_{n-1}, \dots, A_2 \rightarrow B_1 B_2$ zerteilt. Nach der Ersetzung sind alle längeren Nonterminalketten vollständig heruntergebrochen und die Chomsky-Normalform erreicht.

Die spezielle Struktur einer Grammatik in Chomsky-Normalform wirkt sich unmittelbar auf das Erscheinungsbild der entstehenden Syntaxbäume aus. Da jedes Nonterminal entweder durch ein Terminalzeichen oder durch zwei weitere Nonterminale ersetzt wird, entsteht im Innern die Struktur eines Binärbaums. Wie wir in Abschnitt 2.4.2 herausgearbeitet haben, besteht in diesen Bäumen ein enger Zusammenhang zwischen ihrer Tiefe und der Anzahl der Blätter. Ist ein Binärbaum B vollständig, d. h., besitzen alle Blätter die gleiche Tiefe h , so besitzt der Baum exakt 2^h Blätter. Ist der Binärbaum nicht vollständig, so gilt offensichtlich die Beziehung $|B| < 2^h$.

Abbildung 4.16 stellt die entstehenden Syntaxbäume gegenüber, die für das Wort *aabbba* aus der Originalgrammatik (Tabelle 4.7 links) und der generierten Chomsky-Normalform (Tabelle 4.7 rechts) entstehen.

4.4.2.2 Backus-Naur-Form

Kontextfreie Sprachen sind ausdrucksstark genug, um die Syntax der meisten Programmiersprachen zu beschreiben. Bereits Anfang der Sechzigerjahre verwendete der amerikanische Computerpionier John Backus eine kontextfreie Grammatik, um die Syntax der Programmiersprache Algol60 formal zu spezifizieren. Die von Backus eingeführte Notation wird heute als *Backus-Naur-Form* bezeichnet und hat sich zum De-facto-Standard für die Beschreibung von Programmiersprachen entwickelt.

Auf den ersten Blick unterscheidet sich die Backus-Naur-Form von der Produktionssyntax dieses Buches vor allem in der Verwendung des Ableitungssymbols $::=$ anstelle von \rightarrow . Darüber hinaus führte Backus einige Spezialkonstrukte ein, mit denen sich die Produktionen kontextfreier Grammatiken übersichtlich beschreiben lassen. Hierzu gehört unter anderem die weiter oben eingeführte Strichnotation, um Produktionen mit gleicher linker Seite zu einer einzigen Regel zusammenzufassen.

In der *erweiterten Backus-Naur-Form* können Wortfragmente zusätzlich in eckige und geschweifte Klammerpaare eingeschlossen werden. So besagt der Ausdruck

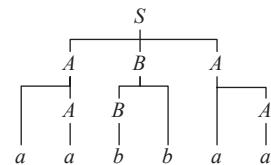
$$A ::= r_1[r_2]r_3,$$

dass zwischen r_1 und r_3 optional das Wort r_2 eingefügt werden darf. Der Ausdruck

$$A ::= r_1\{r_2\}r_3$$

■ Originalgrammatik

$$\begin{aligned} S &\Rightarrow ABA \\ &\Rightarrow aABA \\ &\Rightarrow aaBA \\ &\Rightarrow aaBbA \\ &\Rightarrow aaBbbA \\ &\Rightarrow aabbA \\ &\Rightarrow aabbaA \\ &\Rightarrow aabbaa \end{aligned}$$



■ Chomsky-Normalform

$$\begin{aligned} S &\Rightarrow S_2 A \\ &\Rightarrow ABA \\ &\Rightarrow V_a ABA \\ &\Rightarrow aABA \\ &\Rightarrow aaBA \\ &\Rightarrow aaBV_b A \\ &\Rightarrow aabV_b A \\ &\Rightarrow aabbA \\ &\Rightarrow aabbV_a A \\ &\Rightarrow aabbaA \\ &\Rightarrow aabbaa \end{aligned}$$

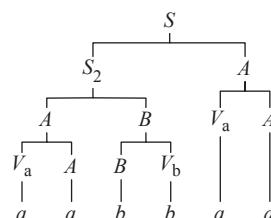


Abbildung 4.16: Die Syntaxbäume von Grammatiken in Chomsky-Normalform haben die Form von Binärbäumen.

■ Auswahl

$$A ::= r_1 \mid \dots \mid r_n$$



$$\begin{array}{l} A \rightarrow r_1 \\ \dots \\ A \rightarrow r_n \end{array}$$

■ Optionales Argument

$$A ::= r_1 [r_2] r_3$$



$$\begin{array}{l} A \rightarrow r_1 r_3 \\ A \rightarrow r_1 r_2 r_3 \end{array}$$

■ Wiederholung

$$A ::= r_1 \{ r_2 \} r_3$$



$$\begin{array}{l} A \rightarrow r_1 B r_3 \\ B \rightarrow B r_2 \\ B \rightarrow \varepsilon \end{array}$$

Abbildung 4.17: Reduktion der Backus-Naur-Form auf gewöhnliche Produktionen.

bedeutet hingegen, dass sich das optionale Wortfragment r_2 beliebig oft wiederholen kann. Keines der Konstrukte führt zu einer Erweiterung der Ausdrucksstärke; beide lassen sich, wie in Abbildung 4.17 gezeigt, auf eine gewöhnliche Produktionenmenge zurückführen. Insgesamt handelt es sich bei der Backus-Naur-Form um eine alternative Beschreibungsform für kontextfreie Grammatiken, die sich lediglich im Aussehen, nicht aber in der Ausdrucksstärke von der bisher verwendeten Notation unterscheidet.

Abbildung 4.18 zeigt einen Auszug aus der Algol60-Grammatik, niedergeschrieben in Backus-Naur-Form. Die Nonterminale sind zur besseren Unterscheidung in spitze Klammern gesetzt und die Schlüsselworte in einfache Hochkommata eingeschlossen.

4.4.3 Pumping-Lemma für kontextfreie Sprachen

Für die Klasse der regulären Sprachen haben wir mit dem Pumping-Lemma ein wertvolles Instrument erhalten, um die Nichtregularität vieler Sprachen zu zeigen. Ein ähnliches Lemma lässt sich auch für die Klasse der kontextfreien Sprachen herleiten.

Für die folgenden Betrachtungen nehmen wir an, dass eine beliebige Grammatik $G = (V, \Sigma, P, S)$ in Chomsky-Normalform gegeben ist. Wir setzen $j := 2^{|V|}$ und wählen ein beliebiges Wort $\omega \in \mathcal{L}(G)$ mit $|\omega| \geq j$, sofern ein solches existiert. Da G in Chomsky-Normalform gegeben ist, besitzt der zugehörige Syntaxbaum im Innern die Form eines Binärbaums mit mindestens $2^{|V|}$ Blättern. Für diesen Baum garantiert uns Satz 2.4, dass wir einen Pfad auswählen können, der mindestens die Länge $|V|$ besitzt. Zusammen mit dem Startsymbol finden wir auf dem gewählten Pfad mindestens $|V| + 1$ Nonterminale wieder, so dass eines davon mehrfach auftauchen muss. Bezeichnen wir dieses Nonterminal mit A , so lässt sich der Syntaxbaum in einer Form darstellen, wie sie im oberen Teil von Abbildung 4.19 skizziert ist. Insgesamt erhalten wir eine Zerlegung des abgeleiteten Wortes in fünf Segmente.

Da G in Chomsky-Normalform vorliegt, kann aus A nur dann ein weiteres A erzeugt werden, wenn mindestens ein Ableitungsschritt der Form $A \rightarrow BC$ durchlaufen wurde. Damit muss mindestens eines der Segmente v oder x ein Zeichen enthalten, d. h., es gilt die Beziehung: $|vx| \geq 1$.

Über die Mindestlänge der Segmente u und y können wir keine Aussage machen; beide können zum leeren Wort degenerieren. Dafür sind wir in der Lage, die Länge der Sequenz vwx nach oben abzuschätzen. Hierzu nehmen wir ohne Beschränkung der Allgemeinheit an, dass die zwei

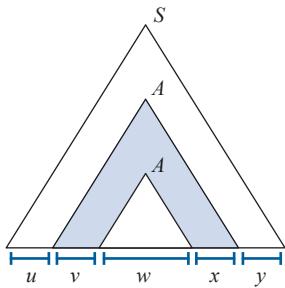
Algol-Grammatik (Auszug)	
<program>	::= <block> <compound statement>
<block>	::= <unlabelled block> <label>: <block>
<unlabelled block>	::= <block head> ; <compound tail>
<block head>	::= 'BEGIN' <declaration> <block head> ; <declaration>
<compound statement>	::= <unlabelled compound> <label>: <compound statement>
<unlabelled compound>	::= 'BEGIN' <compound tail>
<compound tail>	::= <statement> 'END' <statement> ; <compound tail>
<declaration>	::= <type declaration> <array declaration> <switch declaration> <procedure declaration>
<type declaration>	::= <local or own type> <type list>
<local or own type>	::= <type> 'OWN' <type>
<type>	::= 'REAL' 'INTEGER' 'BOOLEAN'
<type list>	::= <simple variable> <simple variable> , <type list>
<array declaration>	::= 'ARRAY' <array list> <local or own type> 'ARRAY' <array list>
<array list>	::= <array segment> <array list> , <array segment>
<array segment>	::= <array identifier> [<bound pair list>] <array identifier> , <array segment>
<array identifier>	::= <identifier>
<bound pair list>	::= <bound pair> <bound pair list> , <bound pair>
<bound pair>	::= <lower bound> : <upper bound>
<upper bound>	::= <arithmetic expression>
<lower bound>	::= <arithmetic expression>

Abbildung 4.18: Auszug aus der Algol60-Syntax, niedergeschrieben in Backus-Naur-Form

Vorkommen der Nonterminale A in Abbildung 4.19 (oben) so gewählt wurden, dass alle weiteren Vorkommen von A , falls diese überhaupt existieren, näher an der Wurzel liegen und alle anderen Nonterminale, die näher an den Blättern liegen, paarweise verschieden sind. Hierdurch ist das obere ausgewählte A maximal $|V|$ Schritte von den Blättern entfernt und der aufgespannte Syntaxbaum kann höchstens $2^{|V|} = j$ Blätter enthalten. Damit gilt die Beziehung $|vwx| \leq j$.

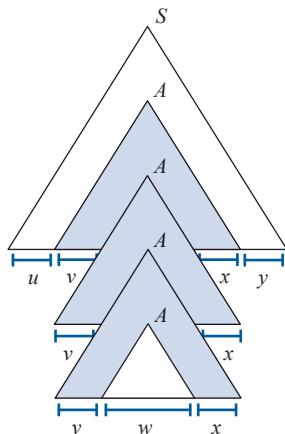
Genau wie im Falle des Pumping-Lemmas für reguläre Sprachen erlaubt uns das Doppelvorkommen von A , das ableitbare Wort „aufzupumpen“, indem wir die Ableitungssequenz von A nach A wiederholen. Hierdurch können wir die Worte uv^2wx^2y , uv^3wx^3y , ... produzieren und eine entsprechende Überlegung zeigt, dass auch das Wort uv^0wx^0y ableitbar ist (vgl. Abbildung 4.19 unten). Fassen wir die erarbeiteten Ergebnisse zusammen, so erhalten wir in direkter Weise das Pumping-Lemma für kontextfreie Sprachen:

■ Wortstruktur kontextfreier Sprachen



Überschreitet die Anzahl der Ableitungsschritte eines Worts eine gewisse Grenze j , so muss aufgrund der endlichen Anzahl der Nonterminale mindestens eines davon mehrfach im Syntaxbaum auftauchen (hier das Nonterminal A). Hierdurch lässt sich jedes hinreichend lange Wort in der Form $uvwxy$ darstellen mit $|vx| \geq 1$ und $|vwx| \leq j$.

■ „Aufpumpen“ des Mittelstücks



Die Ableitung des Mittelstücks lässt sich beliebig oft wiederholen. Neben $uvwxy$ sind somit auch die Wörter $uv^iwx^i y$ für alle $i \in \mathbb{N}$ in der Sprache enthalten.

Abbildung 4.19: Veranschaulichung des Pumping-Lemmas anhand der Syntaxbäume kontextfreier Grammatiken.



Satz 4.3 (Pumping-Lemma für kontextfreie Sprachen)

Für jede kontextfreie Sprache L existiert ein $j \in \mathbb{N}$, so dass sich alle Wörter $\omega \in L$ mit $|\omega| \geq j$ in der folgenden Form darstellen lassen:

$$\omega = uvwxy \quad \text{mit } |vx| \geq 1 \text{ und } |vwx| \leq j$$

Ferner ist mit ω auch das Wort $uv^iwx^i y$ für alle $i \in \mathbb{N}$ in L enthalten.

Das Pumping-Lemma versetzt uns in die Lage, die Sprache $L_{C1} = \{a^n b^n c^n \mid n \in \mathbb{N}^+\}$ als nicht kontextfrei zu entlarven. Für den Beweis nehmen wir an, L_{C1} sei kontextfrei. Dann garantiert uns das Pumping-Lemma, dass ein $j \in \mathbb{N}$ existiert, so dass sich jedes Wort $\omega = a^i b^i c^i$ mit $|\omega| \geq j$ in der Form $uvwxy$ darstellen lässt mit $|vx| \geq 1$ und $|vwx| \leq j$. Wählen wir $i = j$, so kann das Segment vwx aufgrund seiner Längenbeschränkung nicht gleichzeitig a 's und c 's enthalten. Entfernen wir die Segmente v und x aus ω , so entsteht mit uwy ein Wort, das eine ungleiche Anzahl von a 's, b 's und c 's enthält. Nach dem Pumping-Lemma muss das Wort $uwy = uv^0wx^0y$ jedoch in L_{C1} enthalten sein, im Widerspruch zur Definition dieser Sprache.

Das Pumping-Lemma hat sich soeben als wertvolles Hilfsmittel erwiesen, um die Sprache L_{C1} als nicht kontextfrei zu entlarven. Dass es sich um keine Universalwaffe handelt, wollen wir anhand der folgenden Beispieldialekt sprache herausarbeiten [106]:

$$L_{\text{fool}} := \{b^k c^l d^m \mid k, l, m \in \mathbb{N}^+\} \cup \{a^m b^n c^n d^n \mid m, n \in \mathbb{N}^+\}$$

Obwohl L_{fool} nicht kontextfrei ist, erfüllt die Sprache alle innerhalb des Pumping-Lemmas getroffenen Aussagen. Hierzu setzen wir $j = 4$ und wählen für alle Wörter $\omega \in L_{\text{fool}}$ mit $|\omega| \geq j$ die nachstehende Zerlegung:

$$\omega = \left\{ \begin{array}{ll} \underbrace{u}_{b^+} \quad \underbrace{v}_{c^+} \quad \underbrace{w}_{c^+} \quad \underbrace{x}_{d^+} \quad \underbrace{y}_{d^+} & \text{falls } a \notin \omega, bb \in \omega \\ \underbrace{u}_{b^+} \quad \underbrace{v}_{c^+} \quad \underbrace{w}_{c^+} \quad \underbrace{x}_{d^+} \quad \underbrace{y}_{d^+} & \text{falls } a \notin \omega, cc \in \omega \\ \underbrace{b^+ c^+ d^+}_{u} \quad \underbrace{v}_{c^+} \quad \underbrace{w}_{c^+} \quad \underbrace{x}_{d^+} \quad \underbrace{y}_{d^+} & \text{falls } a \notin \omega, dd \in \omega \\ \underbrace{u}_{b^+} \quad \underbrace{v}_{c^+} \quad \underbrace{w}_{c^+} \quad \underbrace{x}_{d^+} \quad \underbrace{y}_{d^+} & \text{falls } a \in \omega \end{array} \right.$$

Mit der getroffenen Wahl von u, v, w, x und y gelten die folgenden Beziehungen:

- $|vx| \geq 1$
- $|vwx| \leq j$
- $uv^iwx^i y \in L$ für alle $i \in \mathbb{N}$

Damit ist es unmöglich, die Sprache L_{fool} mithilfe des Pumping-Lemmas von den kontextfreien Sprachen zu unterscheiden. Der Grund für das Versagen geht auf die Eigenschaft des Pumping-Lemmas zurück, keine Aussage über die Startposition der Teilwörter u, v, w, x und y zu machen. Hierdurch war es uns möglich, den kritischen Teilabschnitt vwx in Wörtern der Form $a^m b^n c^n d^m$ komplett mit a 's zu füllen und damit zu vermeiden, dass die Struktur des nicht kontextfreien Teilworts $b^n c^n d^m$ durch das Aufpumpen von v und x zerstört wird.

Um die Sprache dennoch als nicht kontextfrei zu identifizieren, müssen wir zu stärkeren Waffen greifen. Eine solche gibt uns *Ogdens Lemma* an die Hand – eine Verallgemeinerung des Pumping-Lemmas, das uns die „pumpbaren“ Symbole freier wählen lässt:

Das Pumping-Lemma besitzt viele Namen! Um der zunehmenden Verbreitung von Anglizismen entgegenzuwirken, wurden von einigen deutschsprachigen Autoren die Begriffe *Schleifensatz* oder *Iterationslemma* vorgeschlagen [85]. Trotz einiger Bemühungen konnten sich die Begriffe bisher nicht flächendeckend durchsetzen. Andere Autoren bezeichnen das Pumping-Lemma für reguläre Sprachen schlicht als *uvw*-Theorem und die kontextfreie Variante als *uvwx*-Theorem. Vereinzelt wird es in Ahnlehnung an einen seiner geistigen Väter als *Bar-Hillel-Theorem* bezeichnet, benannt nach dem israelischen Mathematiker Yehoshua Bar-Hillel. Der großen Namensvielfalt zum Trotz halten wir in diesem Buch an dem ursprünglichen Begriff *Pumping-Lemma* fest, da er immer noch am unmissverständlichssten andeutet, um welches Theorem es sich hier handelt.



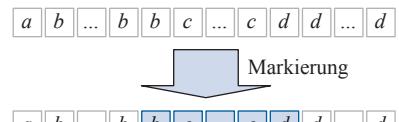
Definition 4.7 (Ogdens Lemma für kontextfreie Sprachen)

Für jede kontextfreie Sprache L existiert ein $j \in \mathbb{N}$, so dass alle Wörter $\omega \in L$ mit $|\omega| \geq j$ die folgende Eigenschaft erfüllen: Markieren wir mindestens j Zeichen in ω , so lässt sich das Wort in der Form $\omega = uvwx$ schreiben, so dass

- mindestens ein Zeichen in vx markiert ist,
- höchstens j Zeichen in vwx markiert sind,
- und für alle $i \in \mathbb{N}$ gilt: $uv^iwx^i y \in L$

Wir wollen das Lemma an dieser Stelle ohne Beweis übernehmen. Eine detaillierte Herleitung findet sich z. B. in [52].

Mithilfe von Ogdens Lemma können wir zeigen, dass L_{fool} keine kontextfreie Sprache sein kann. Hierzu betrachten wir das Wort $\omega = ab^j c^j d^j$, wobei j die Konstante aus Ogdens Lemma ist. Markieren wir die Symbolsequenz $bc^j d$, so muss es für ω eine Zerlegung $uvwx$ geben, so dass vx mindestens einen und vwx höchstens j der markierten Buchstaben enthält. Damit kann die Sequenz vwx und damit auch die Sequenz vx maximal zwei verschiedene Buchstaben aus der Menge $\{b, c, d\}$ enthalten (vgl. Abbildung 4.20). Im Wort uwy kommt die



Nach Ogdens Lemma lässt sich das Ausgangswort in der Form $uvwx$ schreiben, so dass in vx mindestens ein Zeichen und in vwx höchstens j Zeichen markiert sind. Damit ergeben sich für die Sequenz vwx drei Möglichkeiten:

- Möglichkeit 1: $vwx = \underbrace{bc\dots c}_{< j}$
- Möglichkeit 2: $vwx = \underbrace{c\dots c}_{\leq j}$
- Möglichkeit 3: $vwx = \underbrace{c\dots cd}_{< j}$

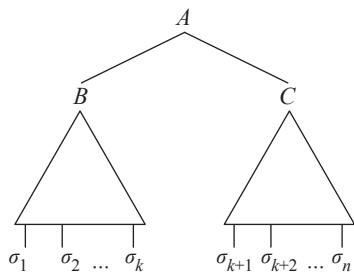
Abbildung 4.20: Ogdens Lemma, angewendet auf das Wort $ab^j c^j d^j$

- Fall 1: $|\omega| = 1$

$$\begin{array}{c} A \\ | \\ \sigma_1 \end{array}$$

Besteht ein ableitbares Wort aus einem einzelnen Terminalzeichen σ_1 , so kann es aus einem Nonterminal A nur durch die Anwendung der Regel $A \rightarrow \sigma_1$ entstanden sein.

- Fall 2: $|\omega| > 1$



Besteht ein ableitbares Wort aus mehreren Terminalzeichen $\sigma_1, \dots, \sigma_n$, so kann es aus einem Nonterminal A nur durch die vorangegangene Anwendung einer Regel $A \rightarrow BC$ entstanden sein.

Abbildung 4.21: Der CYK-Algorithmus arbeitet nach dem Prinzip der dynamischen Programmierung. Er macht sich die spezielle Struktur von Syntaxbäumen zu Nutze, die von Grammatik in Chomsky-Normalform erzeugt werden.

Anzahl an b 's, c 's und d 's hierdurch aus dem Gleichgewicht, so dass es kein Element von L sein kann. Nach Ogdens Lemma müsste uwv aber in L enthalten sein, falls L tatsächlich kontextfrei wäre.

4.4.4 Entscheidungsprobleme

Das Wortproblem für kontextfreie Sprachen ist entscheidbar und lässt sich mit dem Mittel der *dynamischen Programmierung* effizient lösen. Anwenden lässt sich die dynamische Programmierung immer dann, wenn sich ein Problem in kleinere Teile zerlegen lässt und die optimale Lösung des Gesamtproblems aus den optimalen Lösungen der Teilprobleme berechnet werden kann. Von der klassischen Rekursion unterscheidet sich die dynamische Programmierung durch den Einsatz einer Tabelle, in der sämtliche Zwischenergebnisse gespeichert werden. Durch das Vorhalten der bereits berechneten Zwischenlösungen konsumieren die Algorithmen mehr Speicherplatz als ihre rein rekursiv programmierten Pendants, so dass sich die dynamische Programmierung immer dann anbietet, wenn die Laufzeit und nicht der Speicherbedarf eines Algorithmus im Vordergrund steht.

Prominente Algorithmen, die nach dem Prinzip der dynamischen Programmierung arbeiten, sind der *Viterbi-Algorithmus* [102] und der *Floyd-Warshall-Algorithmus* [31, 104]. Auch das bekannte *Rucksackproblem* (vgl. Abschnitt 7.1) lässt sich auf diese Weise effizient lösen [56].

Die Wissenschaftler John Cocke, Daniel Younger und Tadao Kasami erkannten Ende der Sechzigerjahre unabhängig voneinander, dass sich das Wortproblem kontextfreier Sprachen mit dem Mittel der dynamischen Programmierung lösen lässt [22, 55, 110]. Ausgangspunkt für den nach den Anfangsbuchstaben seiner Entdecker benannten *CYK-Algorithmus* ist eine Grammatik G in Chomsky-Normalform. Im Kern basiert der CYK-Algorithmus auf der folgenden Beobachtung:

- Lässt sich aus einem Nonterminal A ein Wort ω ableiten, das aus einem einzelnen Terminalzeichen σ besteht, so muss die Regel $A \rightarrow \sigma$ existieren. Andernfalls würden die Produktionen mindestens ein weiteres Nonterminal und damit auch ein weiteres Terminalzeichen produzieren. Für den Fall $|\omega| = 1$ lässt sich das Wortproblem somit ohne Mühe entscheiden (vgl. Abbildung 4.21 oben).
- Besteht das Wort ω aus mehreren Terminalzeichen $\sigma_1, \dots, \sigma_n$ mit $n \geq 2$, so kann es aus einem Nonterminal A nur durch eine voran-

```

CYK-Algorithmus

// Eingabe: Grammatik  $G = (V, \Sigma, P, S)$ 
//          Wort  $\omega = \sigma_1, \dots, \sigma_n$ 
// Ausgabe: true , wenn  $\omega \in \mathcal{L}(G)$ , false wenn  $\omega \notin \mathcal{L}(G)$ 

boolean cyk(G, ω)
{
    // Berechne die erste Zeile ...
    for (i = 1; i ≤ n; i++) {
        cyk[i][1] = {A | (A → σ_i) ∈ P};
    }

    // Berechne alle restlichen Zeilen ...
    for (j = 2; j ≤ n; j++) {
        for (i = 1; i ≤ n+1-j; i++) {
            cyk[i][j] = ∅;
            for (k = 1; k < j; k++) {
                cyk[i][j] = cyk[i][j] ∪ {A | (A → BC) ∈ P, B ∈ cyk[i][k], C ∈ cyk[i+k][j-k]};
            }
        }
    }
    return S ∈ cyk[1][n];
}

```

Abbildung 4.22: Der CYK-Algorithmus (Pseudo-Code)

gegangene Anwendung einer Regel $A \rightarrow BC$ entstanden sein. Können wir nachweisen, dass ein gewisses k mit $1 \leq k \leq n$ existiert, so dass sich die Anfangssequenz $\sigma_1 \dots \sigma_k$ aus B und die Endesequenz $\sigma_{k+1} \dots \sigma_n$ aus C ableiten lässt, dann lässt sich das Gesamtwort ω aus A ableiten (vgl. Abbildung 4.21 unten).

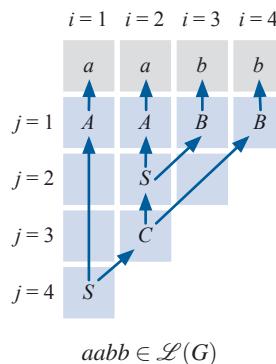
Damit ist es uns gelungen, das Problem für Wörter ω der Länge n auf die Lösung für Wörter der Länge k bzw. $n - k$ zurückzuführen. Auch wenn wir den Wert von k nicht kennen und alle Möglichkeiten in Betracht ziehen müssen, ist sichergestellt, dass die Teilwörter eine kleinere Länge besitzen als ω selbst. Damit sind alle Voraussetzungen für die Anwendbarkeit der dynamischen Programmierung erfüllt.

Um das Wortproblem für $\omega = \sigma_1, \dots, \sigma_n$ zu entscheiden, verwaltet der CYK-Algorithmus intern ein zweidimensionales Array cyk der Größe $n \times n$. Sobald der Algorithmus terminiert, enthält das Feld $cyk[i][j]$ alle Nonterminale, aus denen sich das Teilwort $\sigma_i, \dots, \sigma_{i+j-1}$ ableiten lässt. Offensichtlich gelten die folgenden Eigenschaften:

■ Grammatik

$$\begin{array}{l} S \rightarrow AB \mid AC \\ C \rightarrow SB \\ A \rightarrow a \\ B \rightarrow b \end{array}$$

■ $\omega_1 = aabb$



■ $\omega_2 = abbb$

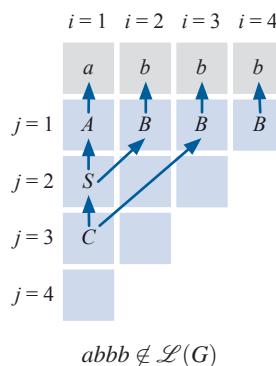


Abbildung 4.23: Der CYK-Algorithmus in Aktion

- Für $i + j > n + 1$ kann $\text{cyk}[i][j]$ niemals Nonterminale enthalten. Effektiv benötigt der CYK-Algorithmus damit nur etwas mehr als die Hälfte der Array-Felder. Alle anderen Felder brauchen nicht berücksichtigt zu werden.

- Das Wort ω lässt sich genau dann ableiten, wenn das Feld $\text{cyk}[1][n]$ das Startsymbol S enthält. Damit reduziert sich das Wortproblem auf einen simplen Inklusionstest, sobald das Array cyk komplett aufgebaut ist.

Die Hauptarbeit des CYK-Algorithmus besteht im Ausfüllen der Array-Felder $\text{cyk}[i][j]$ (vgl. Abbildung 4.22). In zwei verschachtelten Schleifen iteriert der Algorithmus hierzu in gewöhnlicher Leserichtung von links nach rechts und von oben nach unten über die Array-Felder $\text{cyk}[i][j]$. Für jedes Feld wird geprüft, ob eine Regel $A \rightarrow BC$ und ein k existieren, so dass sich das Teilwort $\sigma_i, \dots, \sigma_{i+k-1}$ aus B und das Teilwort $\sigma_{i+k}, \dots, \sigma_{i+j-1}$ aus C erzeugen lässt. Dies ist genau dann der Fall, wenn das Nonterminal B in $\text{cyk}[i,k]$ und das Nonterminal C in $\text{cyk}[i+k,j-k]$ enthalten ist. Fällt der Test erfolgreich aus, so wird dem Element $\text{cyk}[i][j]$ das Nonterminal A hinzugefügt.

Im Kern verbirgt sich hinter dieser Implementierung nichts anderes als ein rekursiver Algorithmus. Der Effizienzgewinn resultiert aus der Eigenschaft, die Zwischenergebnisse tabellarisch zu speichern und den rekursiven Aufruf durch einen simplen Tabellenzugriff zu ersetzen. Während der rein rekursiv arbeitende Algorithmus eine exponentielle Laufzeit benötigen würde, macht es die dynamische Programmierung an dieser Stelle möglich, das Wortproblem in kubischer Laufzeit zu entscheiden. Erkauft wird der Effizienzgewinn durch einen erhöhten Logistikbedarf, der den benötigten Speicherplatz des CYK-Algorithmus quadratisch mit der Länge des Eingabeworts wachsen lässt.

Abbildung 4.23 demonstriert die Anwendung des CYK-Algorithmus anhand zweier Beispiele. Für die erste Zeichensequenz wird das Wortproblem positiv entschieden. Da das Feld $\text{cyk}[1][4]$ das Startsymbol S enthält, ist sichergestellt, dass sich das Wort $aabb$ aus dem Startsymbol ableiten lässt. Für das zweite Beispiel endet der Test negativ. Die Sequenz $abbb$ lässt sich aus keinem Nonterminal und damit erst recht nicht aus dem Startsymbol ableiten.

4.4.5 Abschlusseigenschaften

Für die folgenden Betrachtungen seien mit $G_1 = \{V_1, \Sigma, P_1, S_1\}$ und $G_2 = \{V_2, \Sigma, P_2, S_2\}$ zwei kontextfreie Grammatiken über dem Alpha-

bet Σ gegeben. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass die Variablenmengen V_1 und V_2 keine gemeinsamen Elemente enthalten. Aus G_1 und G_2 lässt sich auf direktem Weg eine Grammatik $G_{1\cup 2}$ definieren, die $\mathcal{L}(G_1) \cup \mathcal{L}(G_2)$ erzeugt. Hierzu fassen wir die Variablenmengen V_1 und V_2 sowie die Produktionenmengen P_1 und P_2 zu einer neuen Variablen- bzw. Produktionenmenge zusammen. Ferner führen wir ein neues Startsymbol S ein, aus dem sich durch eine neu hinzugefügte Produktion sowohl das Startsymbol von G_1 als auch das Startsymbol von G_2 erzeugen lässt. Insgesamt erhalten wir mit

$$G_{1\cup 2} := \{V_1 \cup V_2, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}, S\}$$

eine Grammatik, die alle Wörter aus $\mathcal{L}(G_1)$ und $\mathcal{L}(G_2)$ erzeugt. Damit ist der Abschluss kontextfreier Sprachen bez. der Vereinigungsoperation gezeigt.

Auf ganz ähnliche Weise können wir eine Grammatik $G_{1\cdot 2}$ erzeugen, die alle Wörter aus der Produktmenge $\mathcal{L}(G_1)\mathcal{L}(G_2)$ erzeugt. Auch hier führen wir beide Variablen- und Produktionenmengen zusammen und ersetzen die Startsymbole S_1 und S_2 durch ein frisches Nonterminal S . Der einzige Unterschied betrifft die neu hinzugefügte Ableitungsregel, die S durch S_1S_2 ersetzt:

$$G_{1\cdot 2} := \{V_1 \cup V_2, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}, S\}$$

Die Grammatik erzeugt die Wortmenge $\mathcal{L}(G_1)\mathcal{L}(G_2)$ und beweist den Abschluss kontextfreier Sprachen bez. Produktbildung.

Die Konstruktion der Produktgrammatik weist den Weg für die Konstruktion einer Grammatik, die alle Wörter der Kleene'schen Hülle $\mathcal{L}(G_1)^*$ erzeugt. Hierzu reichern wir die Menge der Produktionen mit neuen Ableitungsregeln an, die aus dem neuen Startsymbol S eine beliebige Anzahl des ursprünglichen Startsymbols S_1 entstehen lassen. Als Ergebnis erhalten wir die folgende Grammatik:

$$G_1^* := \{V_1, \Sigma, P_1 \cup \{S \rightarrow \epsilon \mid SS_1\}, S\}$$

Abbildung 4.24 demonstriert die drei Konstruktionen am Beispiel der Grammatiken $G_1 := (V_1, \Sigma, P_1, S_1)$ und $G_2 := (V_2, \Sigma, P_2, S_2)$ mit $\Sigma := \{a, b, c\}$, $V_1 := \{S_1, A_1, B_1\}$, $V_2 := \{S_2, A_2, B_2\}$ und

$$\begin{aligned} P_1 &:= \{S_1 \rightarrow A_1B_1, A_1 \rightarrow ab \mid aA_1b, B_1 \rightarrow c \mid cB_1\} \\ P_2 &:= \{S_2 \rightarrow A_2B_2, A_2 \rightarrow a \mid aA_2, B_2 \rightarrow bc \mid bB_2c\} \end{aligned}$$

Es bleibt die Untersuchung der Schnitt- und der Komplementoperation. Anhand der eingeführten Beispielgrammatiken G_1 und G_2 können wir

■ Grammatik G_1

$$\begin{aligned} S_1 &\rightarrow A_1B_1 \\ A_1 &\rightarrow ab \mid aA_1b \\ B_1 &\rightarrow c \mid cB_1 \end{aligned}$$

■ Grammatik G_2

$$\begin{aligned} S_2 &\rightarrow A_2B_2 \\ A_2 &\rightarrow a \mid aA_2 \\ B_2 &\rightarrow bc \mid bB_2c \end{aligned}$$

■ Vereinigung

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow A_1B_1 \\ A_1 &\rightarrow ab \mid aA_1b \\ B_1 &\rightarrow c \mid cB_1 \\ S_2 &\rightarrow A_2B_2 \\ A_2 &\rightarrow a \mid aA_2 \\ B_2 &\rightarrow bc \mid bB_2c \end{aligned}$$

■ Produktbildung

$$\begin{aligned} S &\rightarrow S_1S_2 \\ S_1 &\rightarrow A_1B_1 \\ A_1 &\rightarrow ab \mid aA_1b \\ B_1 &\rightarrow c \mid cB_1 \\ S_2 &\rightarrow A_2B_2 \\ A_2 &\rightarrow a \mid aA_2 \\ B_2 &\rightarrow bc \mid bB_2c \end{aligned}$$

■ Kleene'sche Hülle

$$\begin{aligned} S &\rightarrow \epsilon \mid SS_1 \\ S_1 &\rightarrow A_1B_1 \\ A_1 &\rightarrow ab \mid aA_1b \\ B_1 &\rightarrow c \mid cB_1 \end{aligned}$$

Abbildung 4.24: Kontextfreie Grammatiken sind bez. Vereinigung, Produktbildung und Hüllenbildung abgeschlossen.

Entscheidungsprobleme kontextfreier Sprachen			
Problem	Eingabe	Fragestellung	Entscheidbar?
Wortproblem	Sprache L , Wort $\omega \in \Sigma^*$	Ist $\omega \in L$?	✓ Ja
Leerheitsproblem	Sprache L	Ist $L = \emptyset$?	✓ Ja
Endlichkeitsproblem	Sprache L	Ist $ L < \infty$?	✓ Ja
Äquivalenzproblem	Sprachen L_1 und L_2	Ist $L_1 = L_2$?	✗ Nein

Abschlusseigenschaften kontextfreier Sprachen			
Operation	Eingabe	Fragestellung	Erfüllt?
Vereinigung	Sprache $L_1, L_2 \in \mathcal{L}_2$	Ist $L_1 \cup L_2 \in \mathcal{L}_2$?	✓ Ja
Schnitt	Sprache $L_1, L_2 \in \mathcal{L}_2$	Ist $L_1 \cap L_2 \in \mathcal{L}_2$?	✗ Nein
Komplement	Sprache $L \in \mathcal{L}_2$	Ist $\Sigma^* \setminus L \in \mathcal{L}_2$?	✗ Nein
Produkt	Sprache $L_1, L_2 \in \mathcal{L}_2$	Ist $L_1 L_2 \in \mathcal{L}_2$?	✓ Ja
Stern	Sprache $L \in \mathcal{L}_2$	Ist $L^* \in \mathcal{L}_2$?	✓ Ja

Abbildung 4.25: Eigenschaften kontextfreier Sprachen in der Übersicht

zeigen, dass die Menge der kontextfreien Sprachen bez. der Schnittoperation nicht abgeschlossen ist. Es gilt:

$$\begin{aligned}\mathcal{L}(G_1) &= \{a^i b^j c^j \mid i, j \in \mathbb{N}\} \\ \mathcal{L}(G_2) &= \{a^j b^i c^i \mid i, j \in \mathbb{N}\}\end{aligned}$$

Als Schnitt von $\mathcal{L}(G_1)$ und $\mathcal{L}(G_2)$ erhalten wir die Sprache

$$\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \{a^i b^i c^i\},$$

die wir bereits weiter oben als nicht kontextfrei identifiziert haben. Aus dem bisher Bewiesenen folgt unmittelbar, dass die kontextfreien Sprachen auch bez. der Komplementbildung nicht abgeschlossen sein können. Aufgrund der De Morgan'schen Regel gilt die folgende Beziehung:

$$\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \overline{\mathcal{L}(G_1) \cup \mathcal{L}(G_2)}$$

Wären die kontextfreien Sprachen bez. Komplementbildung abgeschlossen, so würde die Abgeschlossenheit der Schnittoperation folgen, im Widerspruch zu den bewiesenen Resultaten. Abbildung 4.25 fasst die herausgearbeiteten Ergebnisse tabellarisch zusammen.

4.5 Kontextsensitive Sprachen

4.5.1 Definition und Eigenschaften

Kontextsensitive Grammatiken sind eine Erweiterung der kontextfreien Grammatiken. Sie zeichnen sich dadurch aus, dass auf der linken Seite einer Produktion eine beliebige Kombination aus Terminal- und Nonterminalzeichen stehen darf. Erst hierdurch wird es möglich, die Ersetzbarkeit eines Nonterminals an die Beschaffenheit seiner Umgebung – den *Kontext* – zu binden. Die einzige Einschränkung, der die Produktionen kontextsensitiver Grammatiken unterliegen, betrifft die Länge der linken und rechten Seiten: Für jede Produktion der Form $l \rightarrow r$ muss die Beziehung $|l| \leq |r|$ gewahrt sein. In kontextsensitiven Grammatiken ist damit sichergestellt, dass die produzierte Zeichensequenz in einem Ableitungsschritt niemals verkürzt wird.

Mithilfe kontextsensitiver Grammatiken sind wir in der Lage, die Sprache

$$L_{C1} = \{a^i b^j c^i \mid i \in \mathbb{N}^+\}$$

zu erzeugen. Für die Konstruktion einer entsprechenden Grammatik gehen wir in drei Schritten vor:

- Neben dem Startsymbol S führen wir drei Nonterminale A , B und C ein, die stellvertretend für jeweils eines der Terminalzeichen a , b und c stehen. Ferner stellen wir durch entsprechende Produktionen sicher, dass wir neben einem a , b und c die Nonterminale A , B und C beliebig oft, aber stets in gleicher Anzahl erzeugen können (obere Regelgruppe in Abbildung 4.26).
- Die bisher eingeführten Produktionen erlauben uns, die benötigte Anzahl A 's, B 's und C 's zu erzeugen. Diese sind jedoch noch ungeordnet und müssen vor der weiteren Bearbeitung zunächst in die richtige Reihenfolge gebracht werden. Um die notwendige Sortierung zu gewährleisten, reichern wir die Grammatik um drei weitere Produktionen an (mittlere Regelgruppe in Abbildung 4.26).
- Zum Schluss benötigen wir Produktionen, die aus den Nonterminalen A , B und C die Terminalzeichen a , b und c erzeugen. Die Ersetzung darf nicht beliebig erfolgen, sondern ausschließlich dann, wenn die Nonterminale in der richtigen Reihenfolge angeordnet sind. Um dies zu erreichen, nutzen wir aus, dass Terminalzeichen in kontextsensitiven Grammatiken auch auf der linken Seite einer Produktion auftauchen dürfen (untere Regelgruppe in Abbildung 4.26).

Grammatik

$$\begin{array}{llll} S & \rightarrow & SABC & \\ S & \rightarrow & abc & \\ \\ CA & \rightarrow & AC & cA \rightarrow Ac \\ CB & \rightarrow & BC & cB \rightarrow Bc \\ BA & \rightarrow & AB & bA \rightarrow Ab \\ \\ aA & \rightarrow & aa & \\ bB & \rightarrow & bb & \\ cC & \rightarrow & cc & \end{array}$$

Ableitung des Worts $aaabbbccc$

$$\begin{aligned} S &\Rightarrow SABC \\ &\Rightarrow SABCABC \\ &\Rightarrow abcABCABC \\ &\Rightarrow abcABACBC \\ &\Rightarrow abcAABCBC \\ &\Rightarrow abcAABBCC \\ &\Rightarrow abAcABBCC \\ &\Rightarrow abAAcBBCC \\ &\Rightarrow abAAbcBCC \\ &\Rightarrow abAABBcCC \\ &\Rightarrow aAbABBcCC \\ &\Rightarrow aAbBBcCC \\ &\Rightarrow aaAbBBcCC \end{aligned}$$

Abbildung 4.26: Erzeugung der Sprache $L_{C1} = \{a^n b^n c^n \mid n \in \mathbb{N}^+\}$ mithilfe einer kontextsensitiven Grammatik

■ Entscheidungsprobleme kontextsensitiver Sprachen

Problem	Eingabe	Fragestellung	Entscheidbar?
Wortproblem	Sprache L , Wort $\omega \in \Sigma^*$	Ist $\omega \in L$?	✓ Ja
Leerheitsproblem	Sprache L	Ist $L = \emptyset$?	✗ Nein
Endlichkeitsproblem	Sprache L	Ist $ L < \infty$?	✗ Nein
Äquivalenzproblem	Sprachen L_1 und L_2	Ist $L_1 = L_2$?	✗ Nein

■ Abschlusseigenschaften kontextsensitiver Sprachen

Operation	Eingabe	Fragestellung	Erfüllt?
Vereinigung	Sprache $L_1, L_2 \in \mathcal{L}_1$	Ist $L_1 \cup L_2 \in \mathcal{L}_1$?	✓ Ja
Schnitt	Sprache $L_1, L_2 \in \mathcal{L}_1$	Ist $L_1 \cap L_2 \in \mathcal{L}_1$?	✓ Ja
Komplement	Sprache $L \in \mathcal{L}_1$	Ist $\Sigma^* \setminus L \in \mathcal{L}_1$?	✓ Ja
Produkt	Sprache $L_1, L_2 \in \mathcal{L}_1$	Ist $L_1 L_2 \in \mathcal{L}_1$?	✓ Ja
Stern	Sprache $L \in \mathcal{L}_1$	Ist $L^* \in \mathcal{L}_1$?	✓ Ja

Abbildung 4.27: Eigenschaften kontextsensitiver Sprachen in der Übersicht

4.5.2 Entscheidungsprobleme

Für alle Produktionen $l \rightarrow r$ einer kontextsensitiven Grammatik gilt $|l| \leq |r|$, d. h., ein Wort kann in einem Ableitungsschritt nicht kürzer werden. Diese Eigenschaft können wir ausnutzen, um das Wortproblem zu entscheiden. Um die Frage $\omega \in L$ für ein Wort mit $|\omega| = n$ zu beantworten, beginnen wir mit einer Menge, die nur das Startsymbol enthält. Anschließend reichern wir sie in einem iterativen Prozess um alle Wörter an, die durch die Anwendung einer Schlussregel entstehen und eine Länge $\leq n$ besitzen. Da nur endlich viele Wörter existieren, deren Länge $\leq n$ ist, wird nach endlich vielen Schritten entweder das gesuchte Wort ω erzeugt oder ein Fixpunkt erreicht. Im ersten Fall ist das Wortproblem positiv, im zweiten Fall negativ entschieden.

Im Gegensatz zum Wortproblem sind das Leerheitsproblem, das Endlichkeitsproblem und das Äquivalenzproblem für kontextfreie Sprachen unentscheidbar, d. h., es gibt kein Verfahren, das eine beliebige Grammatik entgegennimmt und die Fragestellung immer korrekt beantwortet.

4.5.3 Abschlusseigenschaften

Die Menge der kontextsensitiven Sprachen ist bez. Vereinigung, Schnitt, Komplement, Produkt und Kleene'scher Hülle abgeschlossen. Um diese Eigenschaften zu zeigen, führen wir zwei kontextsensitive Grammatiken G_1 und G_2 wie folgt zu einer gemeinsamen Grammatik zusammen:

- Vereinigung

$\mathcal{L}(G_1) \cup \mathcal{L}(G_2)$ wird durch die nachstehende Grammatik erzeugt:

$$G_{1 \cup 2} := \{V_1 \cup V_2, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\}, S\}$$

- Produkt

$\mathcal{L}(G_1)\mathcal{L}(G_2)$ wird durch die nachstehende Grammatik erzeugt:

$$G_{1 \cdot 2} := \{V_1 \cup V_2, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S\}$$

- Kleene'sche Hülle

$\mathcal{L}(G_1)^*$ wird durch die nachstehende Grammatik erzeugt:

$$G_1^* := \{V_1, \Sigma, P_1 \cup \{S \rightarrow \epsilon | SS_1\}, S\}$$

Den Beweis der Abgeschlossenheit bez. Durchschnitt und Komplement wollen wir an dieser Stelle nicht führen. Die Eigenschaften lassen sich am einfachsten mithilfe linear beschränkter Turing-Maschinen zeigen, die wir in Abschnitt 6.4 als äquivalente Beschreibungsform für kontextsensitive Sprachen kennen lernen werden. Abbildung 4.27 fasst die Entscheidbarkeits- und Abschlusseigenschaften für kontextsensitive Sprachen in einer Übersicht zusammen.

4.6 Phrasenstruktursprachen

Jede Sprache, die sich durch eine Grammatik im Sinne von Definition 4.2 erzeugen lässt, nennen wir eine Phrasenstruktursprache. Im Gegensatz zu allen anderen Sprachklassen unterliegen die linke und die rechte Seite einer Produktion $l \rightarrow r$ keinen Restriktionen; beide dürfen aus einer beliebigen Sequenz von Terminal- und Nonterminalzeichen bestehen.

Entscheidungsprobleme von Phrasenstruktursprachen			
Problem	Eingabe	Fragestellung	Entscheidbar?
Wortproblem	Sprache L , Wort $\omega \in \Sigma^*$	Ist $\omega \in L$?	✗ Nein
Leerheitsproblem	Sprache L	Ist $L = \emptyset$?	✗ Nein
Endlichkeitsproblem	Sprache L	Ist $ L < \infty$?	✗ Nein
Äquivalenzproblem	Sprachen L_1 und L_2	Ist $L_1 = L_2$?	✗ Nein

Abschlusseigenschaften von Phrasenstruktursprachen			
Operation	Eingabe	Fragestellung	Erfüllt?
Vereinigung	Sprache $L_1, L_2 \in \mathcal{L}_0$	Ist $L_1 \cup L_2 \in \mathcal{L}_0$?	✓ Ja
Schnitt	Sprache $L_1, L_2 \in \mathcal{L}_0$	Ist $L_1 \cap L_2 \in \mathcal{L}_0$?	✓ Ja
Komplement	Sprache $L \in \mathcal{L}_0$	Ist $\Sigma^* \setminus L \in \mathcal{L}_0$?	✗ Nein
Produkt	Sprache $L_1, L_2 \in \mathcal{L}_0$	Ist $L_1 L_2 \in \mathcal{L}_0$?	✓ Ja
Stern	Sprache $L \in \mathcal{L}_0$	Ist $L^* \in \mathcal{L}_0$?	✓ Ja

Abbildung 4.28: Eigenschaften von Phrasenstruktursprachen in der Übersicht

Typ-0-Grammatiken sind ausdrucksstärker, als es der erste Blick vermuten lässt. In Kapitel 6 werden wir zeigen, dass sich mit Typ-0-Grammatiken alle Sprachen erzeugen lassen, die in irgendeiner Weise algorithmisch berechenbar sind. Dass darüber hinaus Sprachen existieren, die nicht berechenbar sind und damit auch nicht durch eine Grammatik erzeugt werden können, ist ein wichtiges Resultat, das wir in Kapitel 6 ausführlich herausarbeiten werden.

Die große Ausdrucksstärke von Typ-0-Grammatiken führt dazu, dass wir nur noch begrenzte Aussagen über die erzeugten Sprachen machen können. Insbesondere sind das Wortproblem, das Leerheitsproblem, das Endlichkeitsproblem und das Äquivalenzproblem unentscheidbar.

Typ-0-Sprachen sind bez. Vereinigung, Durchschnitt, Konkatenation und Hüllenbildung abgeschlossen, d. h., zwei Grammatiken G_1 und G_2 lassen sich zu einer einzigen Grammatik verschmelzen, die $\mathcal{L}(G_1) \cup \mathcal{L}(G_2)$, $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$, $\mathcal{L}(G_1)\mathcal{L}(G_2)$ oder $\mathcal{L}(G_1)^*$ erzeugt. Einzig die Komplementbildung verletzt die Abschlusseigenschaften, da nicht zu jedem G eine Grammatik existiert, die $\overline{\mathcal{L}(G)}$ erzeugt.

4.7 Übungsaufgaben

Gegeben seien die folgenden Alphabete:

$$\Sigma_1 := \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\Sigma_2 := \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\Sigma_3 := \{A, B, C, D, E, F\}$$

$$\Sigma_4 := \{8, 9\}$$

Aufgabe 4.1



Webcode

4658

Finden Sie umgangssprachliche Beschreibungen für die nachstehenden Sprachen:

- a) $\Sigma_2 \mid \Sigma_1 \Sigma_2^*$
- c) $(\Sigma_2 \setminus \Sigma_4) \mid (\Sigma_1 \setminus \Sigma_4)(\Sigma_2 \setminus \Sigma_4)^*$
- b) $(\Sigma_2 \cup \Sigma_3) \mid (\Sigma_1 \cup \Sigma_3)(\Sigma_2 \cup \Sigma_3)^*$

Gegeben seien die folgenden Mengen:

$$L_1 := \{aa, bb\}$$

$$L_2 := \{a\}^+$$

$$L_3 := \{b\}^+$$

Aufgabe 4.2



Webcode

4194

Erzeugen Sie die nachstehenden Sprachen:

- a) $L_1 \cup L_2$
- b) $L_1 \cup L_3$
- c) $L_1^* \cap L_2$
- d) $L_1^* \cap L_3$

Als Beispiel einer Grammatik haben Sie in diesem Kapitel die folgenden Produktionsregeln für eine Teilmenge der deutschen Sprache kennen gelernt:

- <Satz> → <Subjekt> <Prädikat> <Objekt>
- <Subjekt> → <Artikel> <Adjektiv> <Substantiv>
- <Artikel> → Der | Die | Das
- <Adjektiv> → kleine | süße | flinke
- <Substantiv> → Eisbär | Elch | Kröte | Maus | Nilpferd
- <Prädikat> → mag | fängt | isst
- <Objekt> → Kekse | Schokolade | Käsepizza

Aufgabe 4.3



Webcode

4893

Nicht alle Sätze, die sich aus diesen Produktionen ableiten lassen, sind grammatisch korrekt. Wie das folgende Beispiel zeigt, lassen sich Wortsequenzen ableiten, in denen die Satzteile nicht zusammenpassen: „Das kleine Maus mag Käsepizza“. Schreiben Sie die Grammatik so um, dass nur solche Sätze ableitbar sind, in denen Artikel und Substantiv sprachlich korrekt kombiniert werden.

Aufgabe 4.4**Webcode
4400**

Mit L_1 , L_2 und L_3 seien zwei beliebige Sprachen gegeben. Welche der folgenden Aussagen sind richtig?

- | | |
|---|------------------------------------|
| a) $(L_1 \cup L_2)L_3 = L_1L_3 \cup L_2L_3$ | e) $(L_1^*)^* = L_1^*$ |
| b) $(L_1 \cap L_2)L_3 = L_1L_3 \cap L_2L_3$ | f) $(L_1^+)^+ = L_1^+$ |
| c) $(L_1 \cup L_2)^* = L_1^* \cup L_2^*$ | g) $(L_1L_2)^*L_1 = L_1(L_2L_1)^*$ |
| d) $(L_1 \cap L_2)^* = L_1^* \cap L_2^*$ | h) $(L_1L_2)^+L_1 = L_1(L_2L_1)^+$ |

Aufgabe 4.5**Webcode
4501**

Zeigen Sie mithilfe des Satzes von Myhill-Nerode, dass die Sprache

$$L := \{\omega\omega \mid \omega \in a^*b\}$$

nicht regulär ist.

Aufgabe 4.6**Webcode
4569**

Beschreiben Sie die nachstehenden Sprachen mit regulären Ausdrücken:

- a) $L_1 := \{\omega \in \{a,b,c\}^* \mid \text{In } \omega \text{ ist die Zeichenkette } ab \text{ enthalten.}\}$
- b) $L_2 := \{\omega \in \{a,b,c\}^* \mid \text{In } \omega \text{ ist die Zeichenkette } ab \text{ nicht enthalten.}\}$

Aufgabe 4.7**Webcode
4634**

In der Programmiersprache C werden Variablennamen nach der folgenden Regel gebildet:

„Namen bestehen aus Buchstaben und Ziffern; dabei muss das erste Zeichen ein Buchstabe sein. Der Unterstrich ‘_’ zählt als Buchstabe.“ [57]

Formalisiern Sie die verbale Beschreibung mit einem Ausdruck in Backus-Naur-Form.

Aufgabe 4.8**Webcode
4955**

Die *Palindromsprache* $\mathcal{P}(\Sigma)$ über einem Alphabet Σ ist die Menge der Wörter aus Σ^* , die von links und rechts gelesen die gleiche Zeichensequenz ergeben. Beispielsweise gelten für die Palindromsprache $\mathcal{P}(\{a,b,c\})$ die folgenden Beziehungen:

- | | |
|--|---|
| $aba \in \mathcal{P}(\{a,b,c\})$ | $ab \notin \mathcal{P}(\{a,b,c\})$ |
| $abccba \in \mathcal{P}(\{a,b,c\})$ | $abcabc \notin \mathcal{P}(\{a,b,c\})$ |
| $aabbcbbaa \in \mathcal{P}(\{a,b,c\})$ | $aabbccbaa \notin \mathcal{P}(\{a,b,c\})$ |

-
- a) Geben Sie eine kontextfreie Grammatik G an, die $\mathcal{P}(\{a,b,c\})$ erzeugt.
 b) Zeigen Sie, dass die Palindromsprache $\mathcal{P}(\{a,b,c\})$ nicht regulär ist.

In Abschnitt 4.1 haben Sie die folgende Grammatik zur Erzeugung der Dyck-Sprache D_2 kennen gelernt: $S \rightarrow \epsilon \mid SS \mid [S] \mid (S)$

Aufgabe 4.9**Webcode****4832**

- a) Modifizieren Sie die Grammatik so, dass sich das leere Wort nicht mehr ableiten lässt.
 b) Übersetzen Sie die modifizierte Grammatik in Chomsky-Normalform.

Die Verallgemeinerung der Bildungsregeln regulärer Grammatiken führt uns auf direktem Weg zur *Greibach-Normalform*. Formal liegt eine Grammatik G in Greibach-Normalform vor, wenn alle Produktionen die Form

$$A \rightarrow \sigma \quad \text{oder} \quad A \rightarrow \sigma B_1 \dots B_n$$

besitzen mit $n \in \mathbb{N}$, $A, B_1, \dots, B_n \in V$ und $\sigma \in \Sigma$. Reguläre Grammatiken erhalten wir als Spezialfall für $n = 1$. Es lässt sich zeigen, dass sämtliche von Greibach-Grammatiken erzeugte Sprachen kontextfrei sind. Umgekehrt existiert zu jeder kontextfreien Sprache L mit $\epsilon \notin L$ eine Grammatik in Greibach-Normalform, die L erzeugt (siehe hierzu [52]).

Erzeugen Sie die Greibach-Normalform für die folgenden Grammatiken:

- a) $\begin{array}{ll} S \rightarrow AB & A \rightarrow a \\ S \rightarrow ABA & B \rightarrow Bb \\ A \rightarrow aA & B \rightarrow \epsilon \end{array}$
- b) $\begin{array}{ll} S \rightarrow () \\ S \rightarrow SS \\ S \rightarrow (S) \end{array}$

Die Grammatiken $G_1 := (\{S, E, Z\}, \{a, \dots, j\}, P_1, S)$ und $G_2 := (\{S, E, Z\}, \{a, \dots, j\}, P_2, S)$ seien durch die folgenden Produktionen definiert:

■ Produktionenmenge P_1

$$\begin{array}{l} S \rightarrow a \mid d \mid g \mid j \mid SS \mid EZ \mid ZE \\ E \rightarrow b \mid e \mid h \mid ES \mid ZZ \mid SE \\ Z \rightarrow c \mid f \mid i \mid EE \mid ZS \mid SZ \end{array}$$

Aufgabe 4.11**Webcode****4476**

■ Produktionenmenge P_2

$$\begin{aligned} S &\rightarrow a \mid d \mid g \mid j \mid aS \mid bZ \mid cE \mid dS \mid eZ \mid fE \mid gS \mid hZ \mid iE \mid jS \\ E &\rightarrow b \mid e \mid h \mid aE \mid bS \mid cZ \mid dE \mid eS \mid fZ \mid gE \mid hS \mid iZ \mid jE \\ Z &\rightarrow c \mid f \mid i \mid aZ \mid bE \mid cS \mid dZ \mid eE \mid fS \mid gZ \mid hE \mid iS \mid jZ \end{aligned}$$

- a) Welcher Zusammenhang besteht zwischen G_1 und G_2 ?
- b) Welchem Chomsky-Typ entsprechen $\mathcal{L}(G_1)$ und $\mathcal{L}(G_2)$?
- c) Für welche Werte $n \in \mathbb{N}$ ist das Wort $b(abcdefghij)^n$ in G_1 oder G_2 ableitbar?

Aufgabe 4.12

**Webcode
4878**

Die Grammatiken $G_1 := (\{S, A\}, \{0, \dots, 9\}, P_1, S)$ und $G_2 := (\{S, E, Z\}, \{0, \dots, 9\}, P_2, S)$ seien durch die folgenden Produktionen definiert:

■ Produktionenmenge P_1

$$\begin{aligned} S &\rightarrow A0 \mid A2 \mid A4 \mid A6 \mid A8 \\ A &\rightarrow \varepsilon \mid A0 \mid A1 \mid A2 \mid A3 \mid A4 \mid A5 \mid A6 \mid A7 \mid A8 \mid A9 \end{aligned}$$

■ Produktionenmenge P_2

$$\begin{aligned} S &\rightarrow 0 \mid 3 \mid 6 \mid 9 \mid 1Z \mid 2E \mid 3S \mid 4Z \mid 5E \mid 6S \mid 7Z \mid 8E \mid 9S \\ E &\rightarrow 1 \mid 4 \mid 7 \mid 1S \mid 2Z \mid 3E \mid 4S \mid 5Z \mid 6E \mid 7S \mid 8Z \mid 9E \\ Z &\rightarrow 2 \mid 5 \mid 8 \mid 1E \mid 2S \mid 3Z \mid 4E \mid 5S \mid 6Z \mid 7E \mid 8S \mid 9Z \end{aligned}$$

Welche numerischen Eigenschaften erfüllen die erzeugbaren Ziffernfolgen?

Aufgabe 4.13

**Webcode
4298**

Definieren Sie drei Grammatiken G_1 , G_2 und G_3 mit den folgenden Eigenschaften:

- a) G_1 erzeugt alle Ziffernfolgen, deren Dezimalwert durch 4 teilbar ist.
- b) G_2 erzeugt alle Ziffernfolgen, deren Dezimalwert durch 5 teilbar ist.
- c) G_3 erzeugt alle Ziffernfolgen, deren Dezimalwert durch 6 teilbar ist.

In diesem Kapitel haben Sie mit

$$L_{\text{fool}} := \{b^k c^l d^m \mid k, l, m \in \mathbb{N}\} \cup \{a^m b^n c^n d^n \mid m, n \in \mathbb{N}\}$$

eine Sprache kennengelernt, die nicht kontextfrei ist, aber dennoch alle innerhalb des Pumping-Lemmas getroffenen Aussagen erfüllt. War es wirklich notwendig, die Sprache so kompliziert zu wählen oder hätten wir die gleiche Betrachtung an einer der folgenden Wortmengen durchführen können?

a) $L'_{\text{fool}} := \{a^m b^n c^n d^n \mid m, n \in \mathbb{N}^+\}$

b) $L''_{\text{fool}} := \{a^m b^n c^n d^n \mid m \in \mathbb{N}, n \in \mathbb{N}^+\}$

Für diese Aufgabe ist die Grammatik $G = (\{S, A, B\}, \{0, 1\}, P, S)$ gegeben. Die Menge der Produktionen P lautet wie folgt:

$$\begin{array}{lcl} S & \rightarrow & AB \mid BAB \mid B0 \\ A & \rightarrow & BA \mid 0 \\ B & \rightarrow & 00 \mid 0AB \mid AB0 \mid ABAB \mid 1 \end{array}$$

- a) Lässt sich in G das leere Wort ableiten?
- b) Formen Sie die Grammatik in Chomsky-Normalform um.
- c) Prüfen Sie mithilfe des CYK-Algorithmus nach, ob das Wort 110100 in $\mathcal{L}(G)$ enthalten ist. Füllen Sie hierzu die nebenstehende Tabelle aus.

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
	1	1	0	1	0	0

$j = 1$						
$j = 2$						
$j = 3$						
$j = 4$						
$j = 5$						
$j = 6$						

In diesem Kapitel haben wir herausgearbeitet, dass die Sprache

$$L_{C1} = \{a^n b^n c^n \mid n \in \mathbb{N}^+\}$$

durch die folgende kontextsensitive Grammatik erzeugt wird:

$$\begin{array}{llll} S & \rightarrow & SABC & CA \rightarrow AC \\ S & \rightarrow & abc & CB \rightarrow BC \\ & & & BA \rightarrow AB \\ & & & cA \rightarrow Ac \\ & & & cB \rightarrow Bc \\ & & & bA \rightarrow Ab \\ & & & cC \rightarrow cc \\ & & & aA \rightarrow aa \\ & & & bB \rightarrow bb \\ & & & cC \rightarrow cc \end{array}$$

Erzeugen Sie eine reduzierte Grammatik, die nur 3 Nonterminale besitzt und die gleiche Sprache erzeugt.

Aufgabe 4.14



Webcode

4114

Aufgabe 4.15



Webcode

4411

Aufgabe 4.16



Webcode

4102

Aufgabe 4.17

Gegeben sei die folgende Grammatik:



**Webcode
4703**

$$\begin{array}{rcl} S & \rightarrow & SD \\ SD & \rightarrow & LaD \end{array} \qquad \begin{array}{rcl} aD & \rightarrow & Daa \\ LD & \rightarrow & L \end{array} \qquad \begin{array}{rcl} L & \rightarrow & \epsilon \end{array}$$

- a) Lässt sich das Wort a^2 aus dem Startsymbol S ableiten?
- b) Welchem Chomsky-Typ entspricht diese Grammatik?
- c) Welche Sprache wird durch die Grammatik erzeugt?
- d) Welchem Chomsky-Typ entspricht die erzeugte Sprache?

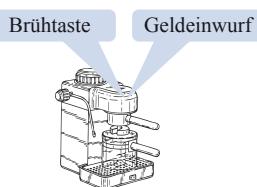
5 Endliche Automaten

In diesem Kapitel werden Sie ...

- den zentralen Begriff des endlichen Automaten kennen lernen,
- den Unterschied zwischen Akzeptoren und Transduktoren verstehen,
- deterministische Automaten um nichtdeterministische Zustandsübergänge anreichern,
- das klassische Automatenmodell zu einer Kellermaschine erweitern,
- den Zusammenhang zwischen Automaten und formalen Sprachen herstellen,
- in Petri-Netzen und zellulären Automaten zwei alternative Automatenmodelle erkennen.



Aufbau



Bevor Kaffee gebrüht werden kann, muss zunächst eine Münze in den Geldschlitz geworfen werden. Anschließend kann die Maschine mit einem Druck auf die Brühtaste gestartet werden.

Zustandsdiagramm

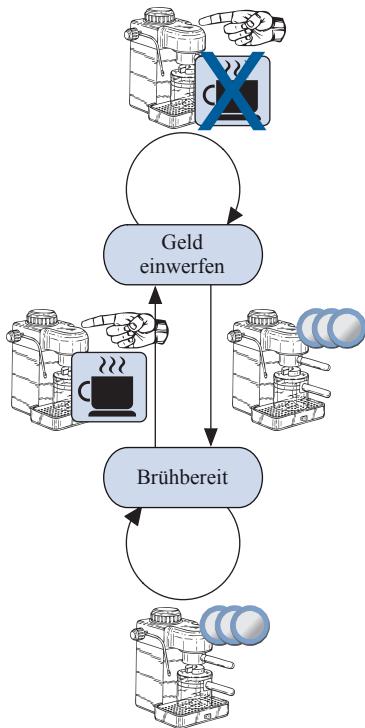


Abbildung 5.1: Aufbau und Funktionsweise einer primitiven Kaffeemaschine

5.1 Begriffsbestimmung

Viele technische Systeme arbeiten ereignisbasiert; sie warten auf das Eintreten eines äußeren Ereignisses (*Ursache*) und reagieren darauf mit einer fest definierten Aktion (*Wirkung*). Wie die Reaktion im Einzelnen aussieht, wird zum einen durch das Ereignis selbst und zum anderen durch den *Zustand* bestimmt, in dem sich das System aktuell befindet. Der Zustand ist das Gedächtnis des Systems und macht es möglich, die Ereignishistorie in der Reaktionsberechnung zu berücksichtigen. Insgesamt hat ein Ereignis damit zwei Auswirkungen: Zum einen veranlasst es das System zu einer nach außen sichtbaren Reaktion, zum anderen kann es zu einem nach außen unsichtbaren Wechsel des internen Zustands führen.

Die Komplexität der entstehenden Interaktionsmuster nimmt mit der wachsenden Anzahl der internen Zustände stark zu und verlangt nach einer adäquaten Beschreibungsform. Eine weit verbreitete Darstellungsmöglichkeit, um das Ein- und Ausgabeverhalten eines Systems übersichtlich zu beschreiben, sind *Zustandsdiagramme*.

Das in Abbildung 5.1 exemplarisch abgebildete Zustandsdiagramm modelliert das Verhalten eines primitiven Kaffeeautomaten, der über einen Münzeinwurfschlitz und einen Knopf zum Starten der Brüheinheit verfügt. Wie der Automat auf das Drücken des Brühenknopfes reagiert, hängt davon ab, in welchem von zwei Zuständen er sich gegenwärtig befindet. Im Zustand „Geld einwerfen“ wird der Knopfdruck ignoriert, während im Zustand „Brühbereit“ die Brüheinheit gestartet wird. Der Zustandswechsel erfolgt ebenfalls ereignisgesteuert. Der Zustand „Brühbereit“ wird erst nach dem Einwurf einer Münze eingenommen und nach Betätigung der Brühtaste wieder verlassen.

Das beschriebene Beispiel ist eines von vielen und lässt sich nahezu beliebig ersetzen. Abstrahieren wir von der konkreten Funktion des modellierten Systems, so finden wir in allen Beispielen stets die gleiche Vorgehensweise wieder: Das Systemverhalten wird durch ein Zustandsmodell beschrieben, das zu jeder Zeit bestimmt, mit welcher Aktion auf ein Ereignis reagiert wird.

Mit den *endlichen Automaten* gibt uns die theoretische Informatik ein Instrument an die Hand, um (endliche) Zustandsmodelle wie dieses formal zu erfassen und systematisch zu analysieren. Die Modellierung technischer Systeme ist dabei eine wichtige, aber nicht die einzige Anwendung. Wie wir später sehen werden, lassen sich auch viele sprachtheoretische Fragestellungen auf diesen Beschreibungsformalismus abbilden.

Endliche Automaten kommen in verschiedenen Spielarten vor, von denen wir eine ganze Reihe in den nächsten Abschnitten detailliert untersuchen werden. Bevor wir damit beginnen, wollen wir versuchen, vorab ein wenig Ordnung in das drohende Chaos zu bringen. Grundsätzlich lassen sich zwei Automatentypen voneinander unterscheiden:

■ Akzeptoren

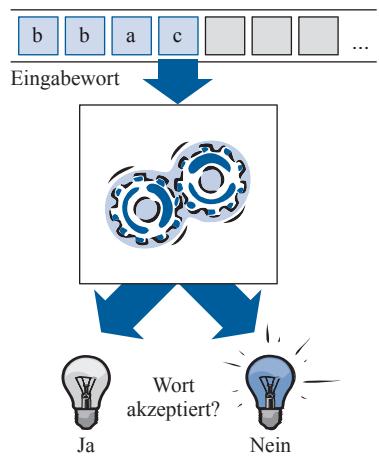
Endliche Automaten dieses Typs nehmen eine Zeichenfolge ω entgegen und entscheiden im Zuge einer Ja-Nein-Entscheidung, ob ω ein gültiges Eingabewort ist (vgl. Abbildung 5.2 oben). Unabhängig von der Länge der Eingabe produzieren Akzeptoren immer eine binäre Antwort; insbesondere wird im Gegensatz zu Transduktoren kein Ausgabewort erzeugt. Die Menge aller Wörter, die von einem endlichen Automaten A mit der Antwort „Ja“ quittiert werden, bildet die von A akzeptierte Sprache $\mathcal{L}(A)$. Wie die nachfolgenden Untersuchungen zeigen werden, sind die von endlichen Automaten akzeptierten Sprachen eng mit den in Kapitel 4 eingeführten Sprachklassen verbunden.

■ Transduktoren

Ein Transduktor ist eine abstrakte Maschine, die eine Zeichenfolge ω von einem Eingabeband liest und daraus eine Folge von Ausgabezeichen generiert. In Abhängigkeit der verwendeten Zustandsübergangsbedingungen werden *Mealy-Automaten* und *Moore-Automaten* unterschieden. Von außen betrachtet folgt ihre Arbeitsweise dem gleichen sequenziellen Schema: Für jedes in einem Berechnungsschritt eingelesene Zeichen wird ein einzelnes Zeichen auf das Ausgabeband geschrieben. Beide sind damit nichts anderes als Übersetzer, die eine Eingabesequenz in eine Ausgabesequenz gleicher Länge transformieren (vgl. Abbildung 5.2 unten). Transduktoren spielen eine gewichtige Rolle im Hardware-Entwurf, da sich jede synchron getaktete Schaltung durch einen Mealy- oder einen Moore-Automaten beschreiben lässt.

Im direkten Vergleich zeigen Akzeptoren einen einfacheren Aufbau als Transduktoren und werden aus diesem Grund in den nachfolgenden Abschnitten zuerst behandelt. In Abschnitt 5.2 werden wir das Konzept des Akzeptors zunächst in seiner Reinform einführen und anschließend in den Abschnitten 5.3 bis 5.5 um nichtdeterministisches Verhalten und einen Kellerspeicher ergänzen. In Abschnitt 5.6 werden wir mit dem Mealy- und dem Moore-Automaten die beiden wichtigsten Transduktorentypen im Detail vorstellen und an einem konkreten Beispiel herausarbeiten, wie sich Automaten auf systematische Weise in digitale Hardware-Schaltungen übersetzen lassen.

■ Akzeptor



■ Transduktor

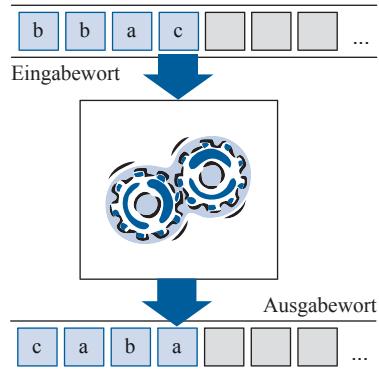
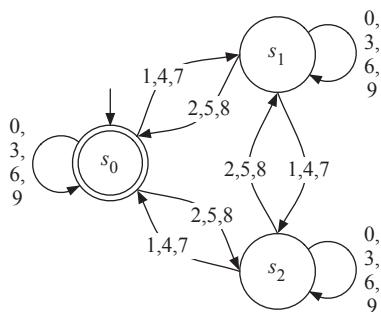


Abbildung 5.2: Akzeptoren und Transduktoren im Vergleich

■ Automat

$$\begin{aligned}
 S &:= \{s_0, s_1, s_2\} \\
 \Sigma &:= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
 \delta(s_0, \sigma) &:= \begin{cases} s_0 & \text{für } \sigma \in \{0, 3, 6, 9\} \\ s_1 & \text{für } \sigma \in \{1, 4, 7\} \\ s_2 & \text{für } \sigma \in \{2, 5, 8\} \end{cases} \\
 \delta(s_1, \sigma) &:= \begin{cases} s_1 & \text{für } \sigma \in \{0, 3, 6, 9\} \\ s_2 & \text{für } \sigma \in \{1, 4, 7\} \\ s_0 & \text{für } \sigma \in \{2, 5, 8\} \end{cases} \\
 \delta(s_2, \sigma) &:= \begin{cases} s_2 & \text{für } \sigma \in \{0, 3, 6, 9\} \\ s_0 & \text{für } \sigma \in \{1, 4, 7\} \\ s_1 & \text{für } \sigma \in \{2, 5, 8\} \end{cases} \\
 E &:= \{s_0\}
 \end{aligned}$$

■ Zustandsdiagramm



■ Legende

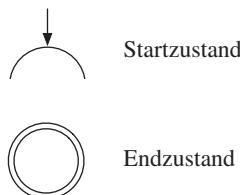


Abbildung 5.3: Deterministischer endlicher Automat mit drei Zuständen. Die eingelesene Ziffernfolge wird genau dann akzeptiert, wenn sie einer durch 3 teilbaren Dezimalzahl entspricht.

5.2 Deterministische Automaten

5.2.1 Definition und Eigenschaften

Wir beginnen mit der Definition des *deterministischen endlichen Akzeptors*, kurz *DEA*.



Definition 5.1 (Deterministischer endlicher Akzeptor)

Ein deterministischer endlicher Akzeptor (*deterministic finite state machine*), kurz *DEA*, ist ein 5-Tupel $(S, \Sigma, \delta, E, s_0)$. Er besteht aus

- der endlichen *Zustandsmenge* S ,
- dem endlichen *Eingabealphabet* Σ ,
- der *Zustandsübergangsfunktion* $\delta : S \times \Sigma \rightarrow S$,
- der Menge der *Endzustände* (*Finalzustände*) $E \subseteq S$ und
- dem *Startzustand* $s_0 \in S$.

Zu Beginn befindet sich jeder Automat in seinem Startzustand s_0 . Wird ein Akzeptor mit dem Eingabewort

$$\omega = \sigma_0 \sigma_1 \sigma_2 \dots \sigma_n \quad (5.1)$$

stimuliert, so durchläuft er nacheinander die Zustände

$$s_0, s_1, s_2, \dots, s_{n+1} \quad \text{mit} \quad s_{i+1} = \delta(s_i, \sigma_i) \quad (5.2)$$

Nachdem das letzte Zeichen σ_n verarbeitet wurde, hält der Automat im Zustand s_{n+1} an. Das Wort ω gilt genau dann als akzeptiert, wenn der zuletzt eingenommene Zustand in der Menge E der Endzustände enthalten ist. Die von einem *DEA* A akzeptierte Sprache $\mathcal{L}(A)$ lässt sich damit wie folgt beschreiben:

$$\mathcal{L}(A) := \{\sigma_0 \sigma_1 \dots \sigma_n \in \Sigma^* \mid \delta(\dots \delta(\delta(s_0, \sigma_0), \sigma_1), \dots, \sigma_n) \in E\} \quad (5.3)$$

Ob die Sprache L das leere Wort ϵ enthält, lässt sich direkt an der Menge der Endzustände ablesen. ϵ wird genau dann akzeptiert, wenn der Startzustand selbst ein Endzustand ist.

Als Beispiel betrachten wir den endlichen Automaten in Abbildung 5.3. Er besteht aus insgesamt drei Zuständen und kann als Eingabe eine

beliebige Folge von Dezimalziffern verarbeiten. Der Automat ist so konstruiert, dass er sich genau dann im Endzustand s_0 befindet, wenn die eingelesene Ziffernfolge einer durch drei teilbaren Dezimalzahl entspricht. Die Funktionsweise basiert auf der zahlentheoretischen Erkenntnis, dass eine Dezimalzahl genau dann durch 3 teilbar ist, wenn es ihre Quersumme ist. Jetzt wird auf einen Schlag die Bedeutung der Zustände s_0 , s_1 und s_2 klar. Der Zustand s_0 wird genau dann eingenommen, wenn die bisher eingelesene Ziffernfolge ohne Rest durch 3 teilbar ist. In den Zuständen s_1 und s_2 befindet sich der Akzeptor genau dann, wenn die Division durch 3 den ganzzahligen Rest 1 bzw. 2 ergibt. Damit entspricht die eingelesene Ziffernfolge genau dann einer durch 3 teilbaren Dezimalzahl, wenn sich der Automat am Ende wieder im Startzustand befindet.

Für die später angestellten Untersuchungen ist es ratsam, das Automatenverhalten ein wenig formaler zu charakterisieren, als wir es mit den Gleichungen (5.1) bis (5.3) bereits getan haben. Den Schlüssel hierzu bilden der Begriff der *Konfiguration* und die darauf definierte Übergangsrelation \rightarrow_A :



Definition 5.2 (Konfiguration (DEA))

Mit $A = (S, \Sigma, \delta, E, s_0)$ sei ein deterministischer endlicher Automat gegeben. Jedes Tupel (s, ω) mit $s \in S$ und $\omega \in \Sigma^*$ heißt *Konfiguration* von A . Die Übergangsrelation \rightarrow_A definieren wir wie folgt:

$$(s_1, \sigma_0 \sigma_1 \dots \sigma_n) \rightarrow_A (s_2, \sigma_1 \dots \sigma_n) : \Leftrightarrow s_2 = \delta(s_1, \sigma_0)$$

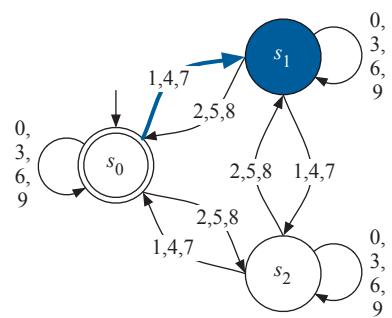
Geht aus dem Kontext hervor, um welchen Automaten es sich handelt, so schreiben wir verkürzend \rightarrow anstelle von \rightarrow_A .

Grob gesagt entspricht eine Konfiguration dem aktuellen Verarbeitungszustand des Automaten, der durch den aktuell eingenommenen Zustand und das einzulesende Restwort vollständig beschrieben wird. Mithilfe des Konfigurationsbegriffs lässt sich die von einem Automaten A akzeptierte Sprache $\mathcal{L}(A)$ auch so charakterisieren:

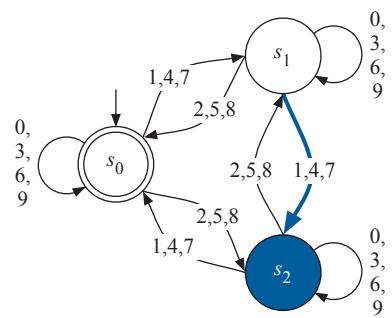
$$\mathcal{L}(A) := \{\omega \in \Sigma^* \mid \text{Für ein } s_e \in E \text{ gilt } (s_0, \omega) \xrightarrow{*} (s_e, \epsilon)\} \quad (5.4)$$

Abbildung 5.4 demonstriert, wie sich die Konfiguration unseres Beispielautomaten während der Verarbeitung des Eingabeworts 147 ändert. Da wir einen deterministischen Automaten vor uns haben, ist der im nächsten Schritt einzunehmende Folgezustand jederzeit eindeutig definiert und der in Gleichung (5.4) vorkommende Endzustand s_e damit ebenfalls eindeutig festgelegt.

■ $(s_0, 147) \rightarrow (s_1, 47)$



■ $(s_1, 47) \rightarrow (s_2, 7)$



■ $(s_2, 7) \rightarrow (s_0, \epsilon)$

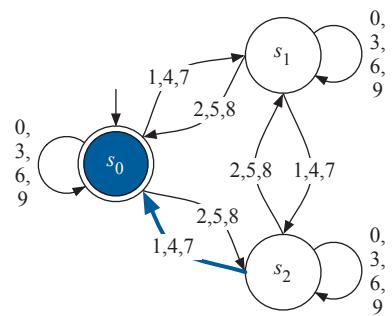


Abbildung 5.4: Konfigurationsübergänge für das Beispielwort 147

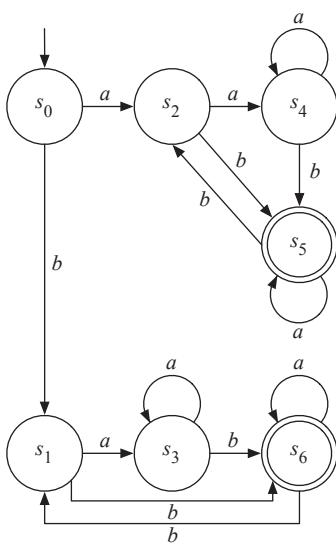


Abbildung 5.5: Ausgangsbeispiel für die Automatenminimierung

5.2.2 Automatenminimierung

Endliche Automaten sind keine kanonische Beschreibungsform für formale Sprachen, d. h., zu jedem Automaten A existieren weitere, die dieselbe Sprache akzeptieren. Zwei Automaten A_1 und A_2 mit $\mathcal{L}(A_1) = \mathcal{L}(A_2)$ bezeichnen wir als *äquivalent*. Äquivalente Automaten können große strukturelle Unterschiede aufweisen und sich insbesondere in der Anzahl der Zustände erheblich unterscheiden. Ein Automat heißt *reduziert*, wenn kein anderer Automat existiert, der die gleiche Sprache akzeptiert und mit weniger Zuständen auskommt. Aufgrund ihrer Minimalität sind reduzierte Automaten von besonderem Interesse und entsprechend groß ist das Verlangen nach einem Verfahren, das einen gegebenen Automaten in eine reduzierte Form überführt.

Die Grundidee der Automatenreduktion besteht darin, *äquivalente Zustände* zu bestimmen. Grob gesprochen sind zwei Zustände s_1 und s_2 genau dann äquivalent, wenn sie von außen nicht unterschieden werden können. Dies ist genau dann der Fall, wenn die Antwort („akzeptiert“ oder „nicht akzeptiert“) für ein Wort ω stets die gleiche ist, egal ob wir in s_1 oder in s_2 starten. Die Zustandsäquivalenz lässt sich auf den Begriff der k -Äquivalenz zurückführen, die das oben Gesagte auf Wörter ω mit $|\omega| \leq k$ beschränkt. Formal definieren wir die umrissenen Äquivalenzbegriffe wie folgt:



Definition 5.3 (Zustandsäquivalenz, Bisimulation)

Sei $A = (S, \Sigma, \delta, E, s_0)$ ein endlicher deterministischer Akzeptor. Die k -Äquivalenz zwischen zwei Zuständen s_1 und s_2 , geschrieben als $s_1 \sim_k s_2$, definieren wir wie folgt:

$$s_1 \sim_0 s_2 \Leftrightarrow s_1, s_2 \text{ sind beide in } E \text{ oder beide nicht in } E$$

$$s_1 \sim_{k+1} s_2 \Leftrightarrow \text{Für alle } \sigma \in \Sigma \text{ gilt } \delta(s_1, \sigma) \sim_k \delta(s_2, \sigma)$$

Gilt $s_1 \sim_k s_2$ für alle $k \in \mathbb{N}$, so heißen s_1 und s_2 *äquivalent*, in Zeichen $s_1 \sim s_2$. Die Relation \sim heißt *Bisimulation*.

Der rekursive Charakter von Definition 5.3 zeigt den Weg auf, wie wir alle bisimulativen Zustände eines gegebenen Automaten berechnen können. Ausgehend von der initialen Zustandsmenge bestimmen wir im ersten Schritt alle 0-äquivalenten Zustände. Als Ergebnis erhalten wir zwei Äquivalenzklassen. In der ersten finden wir alle Finalzustände, in der zweiten die restlichen Zustände wieder. Jetzt werden die Klassen in einem iterativen Prozess weiter unterteilt. Hierzu bestimmen wir

Übergangstabelle							
	s_0	s_1	s_2	s_3	s_4	s_5	s_6
a	s_2	s_3	s_4	s_3	s_4	s_5	s_6
b	s_1	s_6	s_5	s_6	s_5	s_2	s_1

Erste Partition							
	s_0	s_1	s_2	s_3	s_4	s_5	s_6
	P_1				P_2		
a	s_2, P_1	s_3, P_1	s_4, P_1	s_3, P_1	s_4, P_1	s_5, P_2	s_6, P_2
b	s_1, P_1	s_6, P_2	s_5, P_2	s_6, P_2	s_5, P_2	s_2, P_1	s_1, P_1

Zweite Partition							
	s_0	s_1	s_2	s_3	s_4	s_5	s_6
	P_1	P_2			P_3		
a	s_2, P_2	s_3, P_2	s_4, P_2	s_3, P_2	s_4, P_2	s_5, P_3	s_6, P_3
b	s_1, P_2	s_6, P_3	s_5, P_3	s_6, P_3	s_5, P_3	s_2, P_2	s_1, P_2

für jeden Zustand zunächst die Äquivalenzklassen seiner Folgezustände. Zwei Zustände belassen wir nur dann in derselben Äquivalenzklasse, wenn die Äquivalenzklassen ihrer Folgezustände identisch sind. Auf diese Weise haben wir nach der ersten Iteration alle 1-äquivalenten Zustände, nach der zweiten Iteration alle 2-äquivalenten Zustände isoliert und so fort. Nach spätestens $|S|$ Verfeinerungsschritten lassen sich keine neuen Äquivalenzklassen mehr bilden. Jetzt lässt sich aus der berechneten Partition der reduzierte Automat konstruieren, indem wir für jede Äquivalenzklasse einen eigenen Zustand erzeugen und entsprechend der berechnenden Übergangstabelle miteinander verbinden.

Wir wollen das Gesagte für den Automaten aus Abbildung 5.5 in die Tat umsetzen. Um den reduzierten Automaten zu konstruieren, stellen wir zunächst die Übergangstabelle auf (vgl. Abbildung 5.6 obere Tabelle). Anschließend separieren wir die Finalzustände und erhalten im ersten Verfeinerungsschritt die folgenden beiden Äquivalenzklassen:

$$P_1 = \{s_0, s_1, s_2, s_3, s_4\}$$

$$P_2 = \{s_5, s_6\}$$

Die mittlere Tabelle in Abbildung 5.6 macht die gebildeten Äquivalenzklassen sichtbar. Neben den Zuständen sind in der Tabelle zusätzlich

Abbildung 5.6: Schrittweise Reduktion des endlichen Automaten aus Abbildung 5.5. Im ersten Schritt werden die Zustände in zwei Äquivalenzklassen aufgeteilt, so dass sich alle Finalzustände in der einen und die restlichen Zustände in der anderen wiederfinden. Nach diesem Schritt sind alle 0-äquivalenten Zustände bestimmt. Jetzt werden die Klassen schrittweise verfeinert. Nach der ersten Iteration sind alle in einer Klasse verbleibenden Zustände paarweise 1-äquivalent, nach der zweiten Iteration paarweise 2-äquivalent und so fort. Machen wir mit der Aufteilung so lange weiter, bis ein Fixpunkt erreicht ist, so sind die Zustände in den verbleibenden Äquivalenzklassen zueinander äquivalent. Den reduzierten Automaten können wir jetzt sofort konstruieren, indem wir für jede Äquivalenzklasse einen separaten Zustand erzeugen.

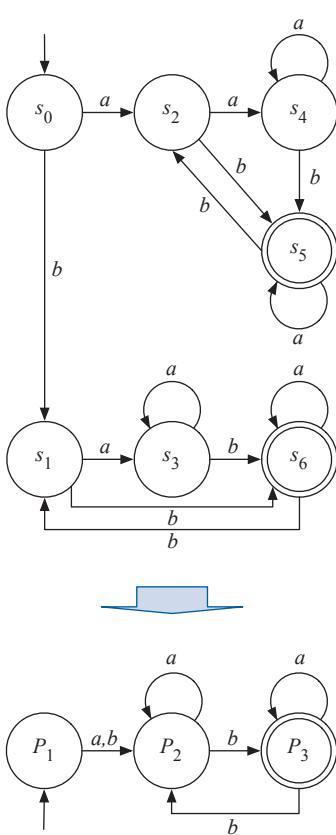


Abbildung 5.7: Minimierter Akzeptor. Die Anzahl der Zustände konnte von 7 auf 3 reduziert werden, ohne die akzeptierte Sprache zu verändern.

die Nummern der Äquivalenzklassen notiert, in die der Automat bei der Verarbeitung eines neuen Eingabezeichens wechselt. An den abgebildeten Nummern lässt sich sofort erkennen, welche Partitionen im nächsten Verfeinerungsschritt gebildet werden müssen. So werden in unserem Beispiel die Zustände in P_1 in zwei weitere Äquivalenzklassen unterteilt, während die Äquivalenzklasse P_2 unverändert bestehen bleibt. Damit erhalten wir die folgende Verfeinerung:

$$\begin{aligned}P_1 &= \{s_0\} \\P_2 &= \{s_1, s_2, s_3, s_4\} \\P_3 &= \{s_5, s_6\}\end{aligned}$$

An dieser Stelle haben wir einen *Fixpunkt* erreicht, da weitere Verfeinerungsschritte keine neuen Äquivalenzklassen hervorbringen. Zusammengefasst kommen wir zu folgendem Ergebnis:

- s_0 ist zu keinem anderen Zustand äquivalent und kann nicht zusammengefasst werden.
- s_1, \dots, s_4 sind paarweise bisimulativ und lassen sich zu einem einzigen Zustand verschmelzen.
- s_5, s_6 sind bisimulativ und werden ebenfalls zu einen einzigen Zustand zusammengefasst.

Wie in Abbildung 5.7 gezeigt, ist es uns mithilfe des vorgestellten Minimierungsverfahrens gelungen, die 7 Zustände des ursprünglichen Automaten auf nur noch 3 zu reduzieren.

5.3 Nichtdeterministische Automaten

5.3.1 Definition und Eigenschaften

Alle bisher betrachteten Automaten waren *deterministisch*, d.h., der Folgezustand war durch das gelesene Eingabezeichen und den jeweils eingenommenen Zustand immer eindeutig bestimmt. Für einige Anwendungsfälle ist es wünschenswert, sich von der Idee des deterministischen Zustandsübergangs zu verabschieden (vgl. Abbildung 5.8). Dies können wir erreichen, indem wir die Übergangsfunktion δ durch eine Berechnungsvorschrift ersetzen, die aus dem gelesenen Eingabezeichen

und dem aktuellen Zustand eine Menge potenzieller Nachfolger berechnet. Formal setzt sich ein solcher *nichtdeterministischer endlicher Akzeptor*, kurz NEA, aus den folgenden Komponenten zusammen:



Definition 5.4 (Nichtdeterministischer endlicher Akzeptor)

Ein nichtdeterministischer endlicher Automat (*nondeterministic finite state machine*), kurz NEA, ist ein 5-Tupel $(S, \Sigma, \delta, E, s_0)$. Er besteht aus

- der endlichen *Zustandsmenge* S ,
- dem endlichen *Eingabealphabet* Σ ,
- der *Zustandsübergangsfunktion* $\delta : S \times \Sigma \rightarrow 2^S$,
- der Menge der *Endzustände* (*Finalzustände*) $E \subseteq S$ und
- dem *Startzustand* $s_0 \in S$.

Genau wie im deterministischen Fall startet der Akzeptor im Zustand s_0 und liest das Eingabewort zeichenweise ein. Mit jedem gelesenen Zeichen σ wechselt er aus dem aktuellen Zustand s_i in einen Folgezustand s_{i+1} aus der Menge $\delta(s_i, \sigma)$. Somit existieren für jede Eingabesequenz

$$\omega = \sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n$$

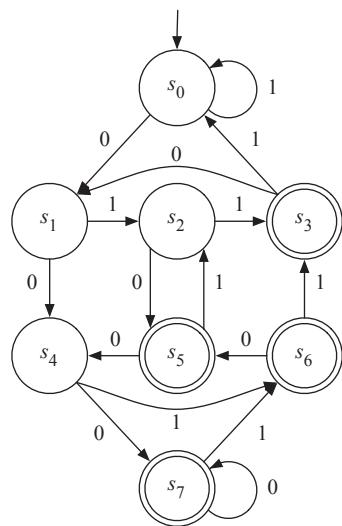
potenziell mehrere Zustandsfolgen

$$s_0, s_1, s_2, \dots, s_{n+1} \quad \text{mit} \quad s_{i+1} \in \delta(s_i, \sigma_i), \quad (5.5)$$

die der Automat durchlaufen kann. Wir vereinbaren, dass ein Wort ω genau dann akzeptiert wird, wenn mindestens eine Zustandsfolge in einem Finalzustand endet. Hinter dieser Festlegung verbirgt sich die Modellvorstellung, dass der Automat eine erfolgreiche Zustandsfolge gewissermaßen erraten kann, d. h., er wählt an jedem Entscheidungspunkt einen Pfad aus, der in einen Finalzustand führt, sofern ein solcher Pfad überhaupt existiert.

Beachten Sie, dass ein NEA unter Umständen überhaupt keine Sequenz der Form (5.5) hervorbringt. Verantwortlich hierfür ist die Menge $\delta(s, \sigma)$, die ausdrücklich auch die leere Menge sein darf. Ist $\delta(s, \sigma) = \emptyset$, so gerät die Abarbeitung im Zustand s ins Stocken, da für das aktuell eingelesene Zeichen σ keine Folgezustände existieren. In diesem Fall gilt das betrachtete Wort per Definition als nicht akzeptiert, unabhängig davon, ob der Zustand s selbst ein Endzustand ist oder nicht.

■ DEA



■ NEA

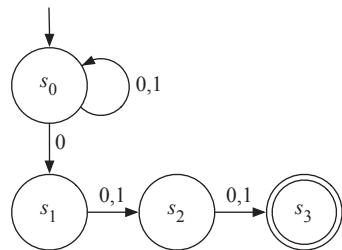
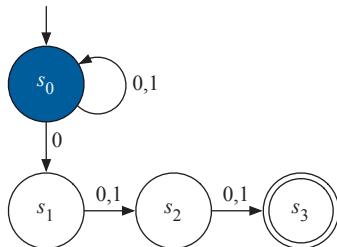
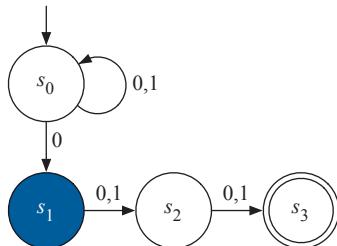


Abbildung 5.8: Einige Sprachen lassen sich mit NEAs erheblich einfacher beschreiben als mit DEAs, hier demonstriert am Beispiel der Sprache aller Bitvektoren, deren drittletzte Ziffer gleich 0 ist. Während der deterministische Automat hierzu 8 Zustände benötigt, kommt die nichtdeterministische Variante mit nur 4 Zuständen aus.

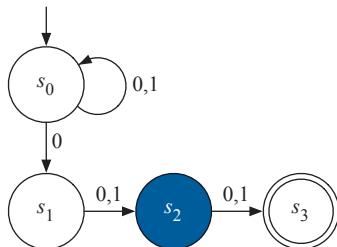
- Möglichkeit 1: $s_0 \rightarrow s_0 \rightarrow s_0 \rightarrow s_0$



- Möglichkeit 2: $s_0 \rightarrow s_0 \rightarrow s_0 \rightarrow s_1$



- Möglichkeit 3: $s_0 \rightarrow s_0 \rightarrow s_1 \rightarrow s_2$



- Möglichkeit 4: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$

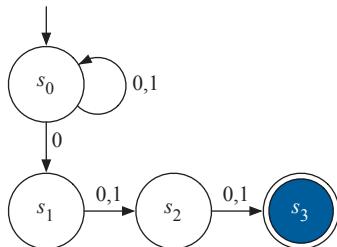


Abbildung 5.9: Die vier möglichen Konfigurationsübergänge für das Eingabewort 000

Um die von nichtdeterministischen Automaten akzeptierten Sprachen formal zu beschreiben, greifen wir, wie schon im Fall der deterministischen Automaten, auf den Begriff der *Konfiguration* zurück.



Definition 5.5 (Konfiguration (NEA))

Mit $A = (S, \Sigma, \delta, E, s_0)$ sei ein nichtdeterministischer endlicher Automat gegeben. Jedes Tupel (s, ω) mit $s \in S$ und $\omega \in \Sigma^*$ heißt eine *Konfiguration* von A . Die Übergangsrelation \rightarrow_A definieren wir wie folgt:

$$(s_1, \sigma_0, \sigma_1, \dots, \sigma_n) \rightarrow (s_2, \sigma_1, \dots, \sigma_n) \Leftrightarrow s_2 \in \delta(s_1, \sigma_0)$$

Die Definition dieses Begriffs ist für beide Automatentypen nahezu identisch. Ein direkter Vergleich mit Definition 5.2 zeigt, dass sich einzige die Definition der Übergangsrelation \rightarrow geringfügig unterscheidet. Anstelle des Gleichheitssymbols taucht in der NEA-Definition das Elementsymbol \in auf.

Mithilfe des Konfigurationsbegriffs lässt sich die von einem nichtdeterministischen Automaten A akzeptierte Sprache $\mathcal{L}(A)$ wie folgt charakterisieren:

$$\mathcal{L}(A) := \{\omega \in \Sigma^* \mid \text{Für ein } s_e \in E \text{ gilt } (s_0, \omega) \xrightarrow{*} (s_e, \varepsilon)\} \quad (5.6)$$

Obwohl die beiden Gleichungen (5.6) und (5.4) im Wortlaut identisch sind, ist der Zustand s_e im Falle von NEAs nicht mehr eindeutig bestimmt. Die Ursache ist in der Definition der Übergangsrelation \rightarrow verborgen, die ein nichtdeterministisches Übergangsverhalten beschreibt.

Als Beispiel sind in Abbildung 5.9 die vier möglichen Konfigurationsübergänge zusammengefasst, die der weiter oben eingeführte Beispielautomat für das Eingabewort 000 durchlaufen kann. Da der vierte Übergang in einem Finalzustand endet, wird das Eingabewort durch den Automaten akzeptiert.

5.3.2 Satz von Rabin, Scott

In Abschnitt 5.3.1 haben wir anhand einer konkreten Sprache herausgearbeitet, dass sich ein akzeptierender Automat deutlich einfacher beschreiben lässt, wenn wir nichtdeterministische Zustandsübergänge erlauben. Die angestellten Überlegungen werfen die Frage auf, ob die

Menge der akzeptierbaren Sprachen durch den hinzugefügten Nichtdeterminismus vergrößert wird. Mit anderen Worten: Gibt es eine Sprache L , die von einem nichtdeterministischen Automaten akzeptiert werden kann, nicht jedoch von einem deterministischen?

Die Antwort auf diese Frage gibt der folgende Satz, der von Michael Oser Rabin und Dana Scott im Jahre 1959 in ihrer berühmten Arbeit „Finite Automata and Their Decision Problems“ bewiesen wurde [83].



Satz 5.1 (Satz von Rabin und Scott)

Zu jedem nichtdeterministischen endlichen Automaten gibt es einen deterministischen endlichen Automaten, der die gleiche Sprache akzeptiert. Es gilt also:

$$\mathcal{L}(\text{NEA}) = \mathcal{L}(\text{DEA})$$

Um den Satz zu beweisen, werden wir darlegen, dass sich der Nichtdeterminismus eines NEAs beseitigen lässt, ohne die akzeptierte Sprache zu verändern. Konkret werden wir zeigen, wie sich ein nichtdeterministischer Akzeptor

$$A_{\text{NEA}} = (S, \Sigma, \delta, E, s_0)$$

in einen deterministischen Akzeptor

$$A_{\text{DEA}} = (S', \Sigma, \delta', E', s'_0)$$

übersetzen lässt, der die gleiche Sprache akzeptiert.

Die Konstruktion basiert auf der Idee, die möglichen Zustände des NEAs gewissermaßen gleichzeitig zu besuchen. Hierzu definieren wir A_{DEA} so, dass seine Zustandsmenge gleich der Potenzmenge der NEA-Zustandsmenge ist. Befindet sich A_{DEA} beispielsweise im Zustand $\{s_4, s_7\}$, so bedeutet dies, dass sich A_{NEA} entweder im Zustand s_4 oder im Zustand s_7 aufhalten kann. Machen wir einen Zustand $\{s_1, \dots, s_n\}$ des DEA genau dann zu einem Endzustand, wenn einer der Zustände s_1, \dots, s_n ein Endzustand des NEA ist, so akzeptieren beide die gleiche Sprache. Zusammengefasst führen die angestellten Überlegungen zu dem folgendem Ergebnis:

$$S' := 2^S$$

$$\delta'(\{s_1, \dots, s_n\}, \sigma) := \bigcup_{i=1}^n \delta(s_i, \sigma)$$

$$E' := \{s \in S' \mid s \cap E \neq \emptyset\}$$

$$s'_0 := \{s_0\}$$

Michael O. Rabin wurde 1931 in Breslau geboren. Nach dem Studium an der Hebrew University in Jerusalem wechselte er 1953 an die Princeton University, die ihm 1956 den Doktorgrad verlieh. Seine akademische Karriere setzte er am neu gegründeten Thomas J. Watson Research Center fort. Im Jahre 1957 arbeitete er zusammen mit Dana Stewart Scott die Arbeit „Finite Automata and Their Decision Problems“ aus. Der ein Jahr jüngere Scott war ein Schüler von Alonzo Church und zu dieser Zeit noch mit seiner Promotion beschäftigt.

Rabin und Scott hatten als Erste die Idee, das Konzept des endlichen Automaten um nichtdeterministische Zustandsübergänge zu erweitern. In ihrer Originalarbeit aus dem Jahre 1959 charakterisieren sie den Begriff wie folgt:

„A nondeterministic automaton is not a probabilistic machine but rather a machine with many choices in its moves. At each stage of its motion across a tape it will be at liberty to choose one of several new internal states. Of course, some sequence of choices will lead either to impossible situations from which no moves are possible or to final states not in the designated class F . We disregard all such failures, however, and agree to let the machine accept a tape if there is at least one winning combination of choices of states leading to a designated final state.“ [83]

Mit ihrer Arbeit gelang Rabin und Scott der große Wurf. Zum einen erwies sich das eingeführte Begriffsgerüst in der Folgezeit als ein leistungsfähiges Instrument für die Beschreibung und die Analyse von Automaten und Sprachen. Zum anderen brachten sie mit dem Indeterminismus eine neuartige Denkrichtung ein, die heute weite Teile der theoretischen Informatik prägt. Im Jahre 1976 wurden Rabin und Scott für ihre richtungsweisende Arbeit mit dem Turing-Award geehrt.

Abbildung 5.10: Jeder NEA lässt sich in einen DEA übersetzen, der die gleiche Sprache akzeptiert. Der Kernidee der Transformation besteht darin, dem DEA eine Zustandsmenge zuzuweisen, die der Potenzmenge der NEA-Zustandsmenge entspricht. Die Zustandsübergangsfunktion definieren wir so, dass sich der aktuell eingenommene DEA-Zustand $\{s_1, \dots, s_n\}$ aus allen Zuständen s_1, \dots, s_n zusammensetzt, die der NEA aufgrund des Nichtdeterminismus potenziell einnehmen kann. Machen wir $\{s_1, \dots, s_n\}$ genau dann zu einem Endzustand, wenn einer der Zustände s_1, \dots, s_n ein Endzustand des NEA ist, so akzeptieren beide Automaten die gleiche Sprache. Angewendet auf den Beispiel-NEA aus Abbildung 5.8 ergibt sich der hier abgebildete Akzeptor.

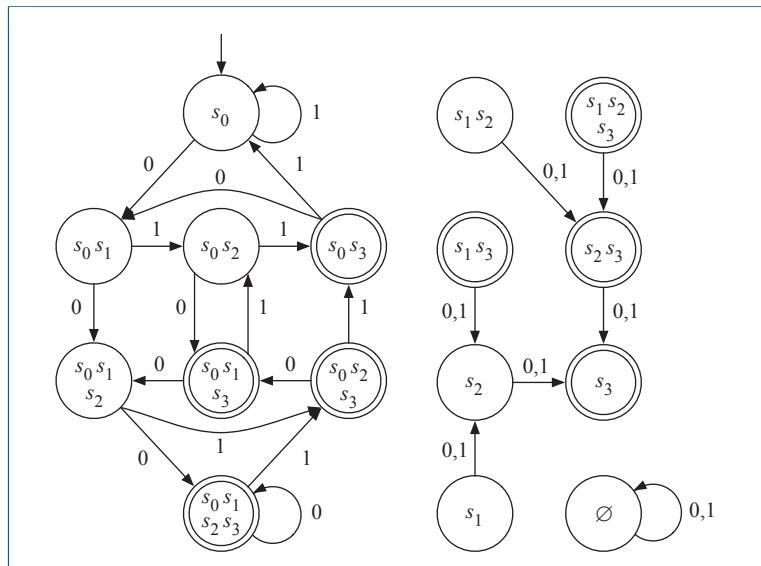


Abbildung 5.10 demonstriert die Automatenkonstruktion am Beispiel des weiter oben eingeführten NEAs. Genau wie der nichtdeterministische Originalautomat akzeptiert der erzeigte DFA alle Bitsequenzen, die an der drittletzten Stelle eine 0 enthalten.

Der Zustandsübergangsgraph macht deutlich, dass die Potenzmengenkonstruktion zu einer Reihe von Zuständen führt, die von dem Startzustand nicht erreichbar sind. Entfernen wir diese aus der Zustandsmenge, so verfügt der konstruierte DFA nur noch über 8 Zustände, ist aber immer noch deutlich größer als die ursprüngliche nichtdeterministische Variante. Ein vergleichender Blick zeigt überdies, dass der konstruierte Automat strukturell dem weiter oben eingeführten DFA aus Abbildung 5.8 entspricht. Zwei Automaten, die sich wie in unserem Beispiel ausschließlich in der Benennung ihrer Zustände unterscheiden, heißen *isomorph*. Offenbar sind zwei isomorphe Automaten immer auch äquivalent, aber nicht umgekehrt.

5.3.3 Epsilon-Übergänge

In Abschnitt 5.3.1 haben wir eine Sprache kennen gelernt, die sich mithilfe deterministischer Automaten nur sehr umständlich beschreiben lässt. Eine erste Verbesserung hat uns die Einführung nichtdeterministischer Zustandsübergänge beschert; hierdurch war es möglich,

die Anzahl der benötigten Zustände erheblich zu reduzieren. In diesem Abschnitt werden wir das Konzept des nichtdeterministischen Automaten um ε -Übergänge anreichern und so zu einer noch handlicheren Beschreibungsform gelangen. Auch hier wird sich zeigen, dass die Erweiterung keinen Einfluss auf die Ausdrucksstärke hat, da sich jeder ε -Automat auf einen äquivalenten NEA oder DEA reduzieren lässt.



Definition 5.6 (Nichtdeterministischer endlicher ε -Akzeptor)

Ein nichtdeterministischer endlicher ε -Akzeptor, kurz ε -NEA, ist ein 5-Tupel $(S, \Sigma, \delta, E, s_0)$. Er besteht aus

- der endlichen Zustandsmenge S ,
- dem endlichen Eingabealphabet Σ mit $\varepsilon \notin \Sigma$,
- der Zustandsübergangsfunktion $\delta : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$,
- der Menge der Endzustände (Finalzustände) $E \subseteq S$ und
- dem Startzustand s_0 .

Die zusätzlich hinzugefügten ε -Übergänge beschreiben eine besondere Art des Zustandsübergangs. Dieser kann spontan erfolgen, d.h., ohne ein neues Eingabezeichen zu konsumieren. Was wir hierunter genau zu verstehen haben, klärt die nachstehende Definition:



Definition 5.7 (Konfiguration (ε -NEA))

Mit $A = (S, \Sigma, \delta, E, s_0)$ sei ein nichtdeterministischer endlicher ε -Akzeptor gegeben. Jedes Tupel (s, ω) mit $s \in S$ und $\omega \in \Sigma^*$ heißt eine *Konfiguration* von A . Die Übergangsrelation \rightarrow definieren wir wie folgt:

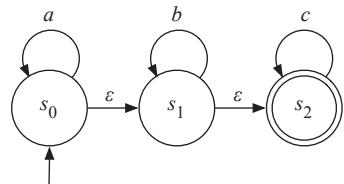
$$(s_1, \omega) \rightarrow (s_2, \omega) :\Leftrightarrow s_2 \in \delta(s_1, \varepsilon)$$

$$(s_1, \sigma\omega) \rightarrow (s_2, \omega) :\Leftrightarrow s_2 \in \delta(s_1, \sigma)$$

Die von einem ε -Automaten A akzeptierte Sprache $\mathcal{L}(A)$ ist wie folgt definiert:

$$\mathcal{L}(A) := \{\omega \in \Sigma^* \mid \text{Für ein } s_e \in E \text{ gilt } (s_0, \omega) \rightarrow (s_e, \varepsilon)\} \quad (5.7)$$

Nichtdeterministischer ε -Akzeptor



Deterministischer Akzeptor

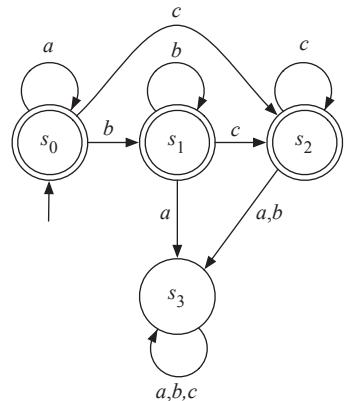
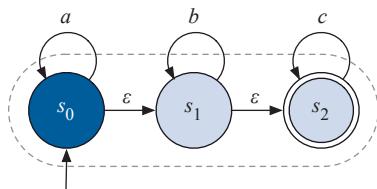
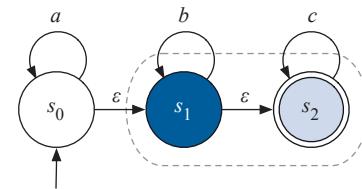


Abbildung 5.11: Viele Sprachen lassen sich mit ε -Automaten kompakter beschreiben als mit deterministischen Akzeptoren. Die dargestellten Automaten demonstrieren diese Eigenschaft am Beispiel der Sprache $L = \{a^i b^j c^k \mid i, j, k \in \mathbb{N}\}$.

- $\|s_0\|_\epsilon = \{s_0, s_1, s_2\}$



- $\|s_1\|_\epsilon = \{s_1, s_2\}$



- $\|s_2\|_\epsilon = \{s_2\}$

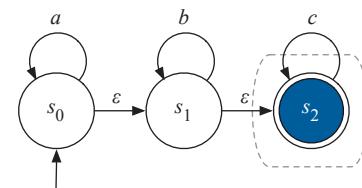


Abbildung 5.12: ϵ -Hüllen unseres Beispielautomats

Abbildung 5.11 (oben) demonstriert die Stärke des ϵ -Prinzips am Beispiel der Sprache

$$L := \{a^i b^j c^k \mid i, j, k \in \mathbb{N}\}$$

Akzeptiert wird auch das leere Wort, obwohl der Startzustand s_0 selbst kein Endzustand ist. Die liebgewonnene Eigenschaft der anderen Automatentypen, dass das leere Wort ϵ genau dann akzeptiert wird, wenn der Startzustand ein Finalzustand ist, wird von ϵ -Automaten somit nicht mehr erfüllt.

Insgesamt erweisen sich ϵ -Übergänge als überaus leistungsfähig, da wir mit ihrer Hilfe Auswahlen modellieren können, die kein Eingabezeichen konsumieren. Trotzdem lassen sich ϵ -Übergänge, wie der DEA in Abbildung 5.11 (unten) bereits vermuten lässt, durch das Hinzufügen neuer Zustände eliminieren. In unserem Beispiel reicht ein weiterer Zustand aus, um die gleiche Sprache mit einem deterministischen Akzeptor ohne ϵ -Übergänge zu akzeptieren.

Im Folgenden wollen wir ein Konstruktionsschema entwickeln, das aus jedem ϵ -NEA systematisch einen DEA erzeugt, der die gleiche Sprache akzeptiert. Das Verfahren ist eine Erweiterung der Teilmengenkonstruktion aus Abschnitt 5.3.2, die ϵ -Übergänge mitberücksichtigt. Eine wichtige Rolle wird der Begriff der ϵ -Hülle eines Zustands s spielen, die wir kurz mit $\|s\|_\epsilon$ bezeichnen:

$$\|s\|_\epsilon := \{s' \mid s \xrightarrow{\epsilon}^* s'\} \quad \text{mit} \quad s \xrightarrow{\epsilon} s' \Leftrightarrow s' \in \delta(s, \epsilon) \quad (5.8)$$

Die eingeführte Relation $\xrightarrow{\epsilon}$ stellt alle Zustände in Beziehung, die über einen ϵ -Übergang direkt miteinander verbunden sind. Die ϵ -Hülle entspricht der reflexiv-transitiven Hülle dieser Relation und enthält damit neben s alle Elemente, die über eine beliebige Anzahl von ϵ -Kanten von s aus erreichbar sind. Die ϵ -Hüllen unseres Beispielautomaten sind in Abbildung 5.12 zusammengefasst. Beachten Sie an dieser Stelle, dass die ϵ -Hüllen keine Äquivalenzrelation ist und damit nicht zu einer Partition der Zustandsmenge führt. Um dies zu erreichen, müsste die eingeführte Relation zusätzlich die Eigenschaft der Symmetrie erfüllen.

Der Begriff der ϵ -Hülle wird in Gleichung (5.8) für einzelne Zustände definiert. Wir wollen ihn in naheliegender Weise auf beliebige Zustandsmengen ausweiten und treffen die folgende Vereinbarung:

$$\|\{s_1, \dots, s_n\}\|_\epsilon := \cup_{i=1}^n \|s_i\|_\epsilon$$

Mit der geleisteten Vorarbeit sind wir in der Lage, die in Abschnitt 5.3.2 eingeführte Potenzmengenkonstruktion fast unverändert auf die Klasse

der ε -Automaten zu übertragen. Für einen beliebigen nichtdeterministischen ε -Akzeptor

$$A_{\varepsilon\text{-NEA}} = (S, \Sigma, \delta, E, s_0)$$

konstruieren wir einen äquivalenten DEA

$$A_{\text{DEA}} = (S', \Sigma, \delta', E', s'_0)$$

wie folgt:

$$S' := 2^S$$

$$\delta'(\{s_1, \dots, s_n\}, \sigma) := \|\cup_{i=1}^n \delta(s_i, \sigma)\|_\varepsilon$$

$$E' := \{s \in S' \mid s \cap E \neq \emptyset\}$$

$$s'_0 := \|s_0\|_\varepsilon$$

Das Konstruktionsverfahren unterscheidet sich lediglich an zwei Stellen von jenem aus Abschnitt 5.3.2. Zum einen werden bei der Berechnung der Folgezustände nicht nur die direkt erreichbaren, sondern auch diejenigen Zustände hinzugenommen, die über einen oder mehrere ε -Übergänge erreichbar sind. Zum anderen definieren wir den neuen Startzustand als die ε -Hülle des ursprünglichen Startzustands s_0 .

In Abbildung 5.13 ist die DEA-Konstruktion für den oben eingeführten ε -Akzeptor dargestellt. Eliminieren wir alle unerreichbaren Teilgraphen, so verbleibt ein DEA mit 4 Zuständen (vgl. Abbildung 5.14). Ein vergleichender Blick auf Abbildung 5.11 zeigt, dass der dort dargestellte DEA und der soeben konstruierte Automat isomorph sind.

Mithilfe des skizzierten Konstruktionsschemas sind wir in der Lage, jeden beliebigen ε -Automaten auf einen äquivalenten DEA abzubilden. Damit haben wir einen konstruktiven Beweis für den folgenden Satz gefunden:



Satz 5.2 (ε -Reduktionstheorem)

Zu jedem nichtdeterministischen endlichen ε -Akzeptor gibt es einen deterministischen endlichen Akzeptor, der die gleiche Sprache akzeptiert. Es gilt also:

$$\mathcal{L}(\varepsilon\text{-NEA}) = \mathcal{L}(\text{DEA})$$

Damit haben wir gezeigt, dass die Hinzunahme von ε -Übergängen zu keiner Erweiterung der Ausdrucksstärke führt. Kurzum: Die (nichtdeterministischen) ε -Akzeptoren begründen exakt dieselbe Sprachklasse wie die weiter oben eingeführten DEAs und NEAs.

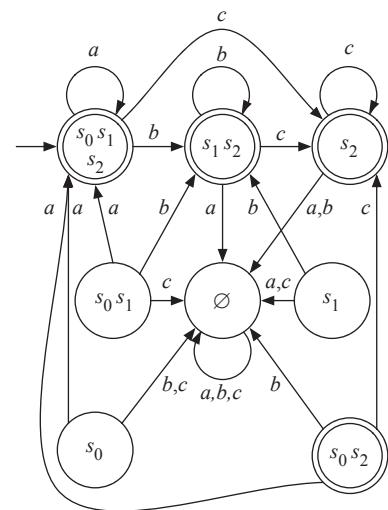


Abbildung 5.13: Übersetzung des ε -Automaten aus Abbildung 5.11 (oben) in einen äquivalenten DEA

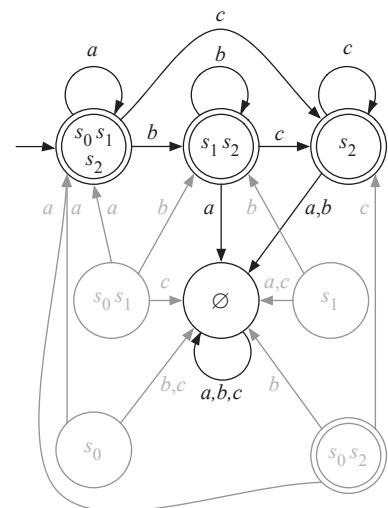
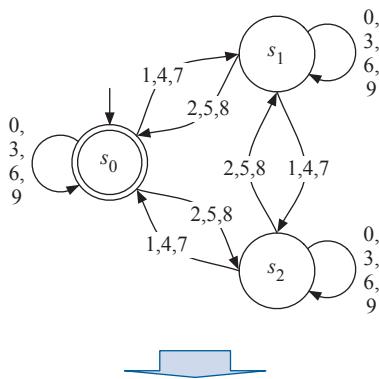


Abbildung 5.14: Eliminieren wir alle nicht erreichbaren Zustände, so bleibt ein DEA mit 4 Zuständen übrig. Der entstandene Automat ist isomorph zu unserem Eingangsbeispiel aus Abbildung 5.11 (unten).



$$\begin{aligned}
 A_0 &\rightarrow 0A_0 \mid 3A_0 \mid 6A_0 \mid 9A_0 \\
 A_0 &\rightarrow 1A_1 \mid 4A_1 \mid 7A_1 \\
 A_0 &\rightarrow 2A_2 \mid 5A_2 \mid 8A_2 \\
 A_1 &\rightarrow 0A_1 \mid 3A_1 \mid 6A_1 \mid 9A_1 \\
 A_1 &\rightarrow 1A_2 \mid 4A_2 \mid 7A_2 \\
 A_1 &\rightarrow 2A_0 \mid 5A_0 \mid 8A_0 \\
 A_2 &\rightarrow 0A_2 \mid 3A_2 \mid 6A_2 \mid 9A_2 \\
 A_2 &\rightarrow 1A_0 \mid 4A_0 \mid 7A_0 \\
 A_2 &\rightarrow 2A_1 \mid 5A_1 \mid 8A_1 \\
 A_0 &\rightarrow \epsilon
 \end{aligned}$$

Abbildung 5.15: Übersetzung eines DEA in eine reguläre Grammatik

$$\begin{aligned}
 A_0 &\rightarrow aA_0 \mid aA_1 \\
 A_1 &\rightarrow bA_1 \mid bA_2 \\
 A_2 &\rightarrow cA_2 \mid cA_3 \\
 A_3 &\rightarrow \epsilon
 \end{aligned}$$

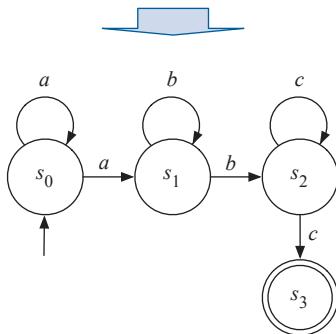


Abbildung 5.16: Übersetzung einer regulären Grammatik in einen NEA

5.4 Automaten und reguläre Sprachen

Zwischen endlichen Automaten und den regulären Sprachen aus Abschnitt 4.3 besteht ein enger Zusammenhang, den wir an dieser Stelle genauer beleuchten wollen. Wir werden zeigen, dass jede von einem Automaten A akzeptierte Sprache regulär ist und zum anderen, dass für jede reguläre Sprache ein endlicher Automat konstruiert werden kann, der sie akzeptiert. Mit anderen Worten: Wir werden zeigen, dass endliche Automaten und reguläre Ausdrücke zwei Beschreibungsformen der gleichen Sprachklasse sind. Dabei spielt es keine Rolle, ob wir das Konzept des DEAs, des NEAs oder des ϵ -NEAs zugrunde legen, schließlich haben wir in den vorherigen Abschnitten herausgearbeitet, wie sich diese äquivalenzhaltend ineinander überführen lassen.

Für die folgenden Betrachtungen sei mit

$$A := (S, \Sigma, \delta, E, s_0)$$

ein beliebiger DEA gegeben. Wir werden zeigen, dass die Sprache $\mathcal{L}(A)$ regulär ist, indem wir eine reguläre Grammatik

$$G := (V, \Sigma, P, S)$$

formulieren mit $\mathcal{L}(G) = \mathcal{L}(A)$. Hierbei verfolgen wir die Grundidee, die Zustände des Automaten als Nonterminale aufzufassen, d. h., für jeden Zustand $s_i \in S$ existiert ein $A_i \in V$ und umgekehrt. Hierdurch lässt sich jeder Zustandsübergang von s_i nach s_j mit $s_j = \delta(s_i, \sigma)$ in direkter Weise in eine Ableitungsregel der Form

$$A_i \Rightarrow \sigma A_j$$

übersetzen. Komplettiert wird die Menge der Produktionen, indem wir für jeden Endzustand $s_e \in E$ eine Produktion der Form

$$A_e \Rightarrow \epsilon$$

hinzufügen. Hierdurch kann die Worterzeugung dann und nur dann abbrechen, wenn sich der zugehörige Automat in einem Endzustand befindet. Verwenden wir dasjenige Nonterminal A_0 als Startsymbol, das dem Startzustand s_0 zugeordnet ist, so erzeugt die konstruierte Grammatik G die Sprache $\mathcal{L}(A)$. Konstruktionsbedingt ist die Grammatik G regulär und damit auch die von A akzeptierte Sprache. Abbildung 5.15 demonstriert das Gesagte anhand unseres Einführungsbeispiels aus Abbildung 5.3.

Wir wollen nun umgekehrt vorgehen und zeigen, dass sich jede reguläre Grammatik G in einen endlichen Automaten A übersetzen lässt.

Mithilfe nichtdeterministischer Akzeptoren gelingt uns die Transformation auf besonders einfache Weise (Abbildung 5.16). Zunächst erzeugen wir für jedes Nonterminal A_i einen separaten Automatenzustand s_i . Anschließend übersetzen wir jede Produktion der Form $A_i \rightarrow \sigma A_j$ in einen Zustandsübergang von s_i nach s_j und markieren die verbindende Kante mit σ . Die Produktionen der Form $A_k \rightarrow \epsilon$ definieren die Endzustände des konstruierten Automaten, d. h., wir setzen $E := \{s_k \mid (A_k \rightarrow \epsilon) \in P\}$. Ein nichtdeterministischer Automat entsteht immer dann, wenn die Grammatik für ein Nonterminal A_i und ein Eingabezeichen σ zwei oder mehr Produktionen der Form $A_i \rightarrow \sigma A_j$ enthält.

5.4.1 Automaten und reguläre Ausdrücke

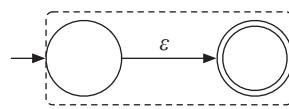
In Abschnitt 4.3.4 haben wir mit den *regulären Ausdrücken* eine alternative Beschreibungsform für reguläre Sprachen eingeführt. Einen Beweis, dass beide die gleiche Sprachklasse beschreiben, sind wir aber bisher schuldig geblieben. Mithilfe der nichtdeterministischen ϵ -Automaten können wir eine der beiden Schlussrichtungen mit Leichtigkeit beweisen. Hierzu sind in Abbildung 5.17 sechs Konstruktionsmuster abgebildet, die zeigen, wie sich ein regulärer Ausdruck R rekursiv in einen ϵ -Automaten transformieren lässt, der $\mathcal{L}(R)$ akzeptiert. Da sich jeder ϵ -NEA in einen DEA und dieser wiederum in eine reguläre Grammatik übersetzen lässt, haben wir gezeigt, dass jede Sprache, die durch einen regulären Ausdruck beschrieben wird, eine Typ-3-Sprache ist.

Die Reduktion eines regulären Ausdrucks auf einen akzeptierenden Automaten ist von hoher praktischer Relevanz. Um z. B. große Datenbestände effizient nach bestimmten Zeichenmustern zu durchsuchen, erzeugen viele Werkzeuge, die reguläre Ausdrücke als Suchmuster verwenden, intern einen endlichen Automaten. Ist dieser konstruiert, so lässt sich in linearer Zeit berechnen, ob eine entsprechende Zeichenkette vorhanden ist bzw. an welcher Stelle sie vorkommt.

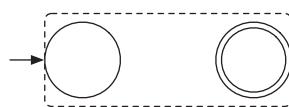
Die bewiesene Schlussrichtung lässt sich umkehren, d. h., für jeden endlichen Automaten A existiert ein regulärer Ausdruck, der $\mathcal{L}(A)$ erzeugt. Für die Praxis ist diese Richtung die unbedeutendere und wir wollen den vergleichsweise komplizierten Beweis an dieser Stelle nicht führen. Eine ausführliche Betrachtung findet sich z. B. in [52].

Insgesamt erhalten wir den in Abbildung 5.18 skizzierten Zusammenhang zwischen den verschiedenen Beschreibungsformen regulärer Sprachen. Wie das Diagramm zeigt, beschreiben nichtdeterministische Akzeptoren, deterministische Akzeptoren, reguläre Grammatiken und reguläre Ausdrücke allesamt die gleiche Sprachklasse.

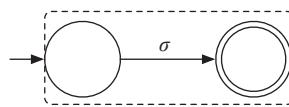
■ Leeres Wort: ϵ



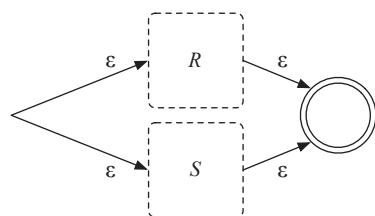
■ Leere Menge: \emptyset



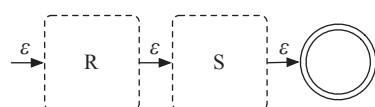
■ Einzelnes Zeichen: σ



■ Auswahl: $R \mid S$



■ Komposition: RS



■ Kleene'sche Hülle: R^*

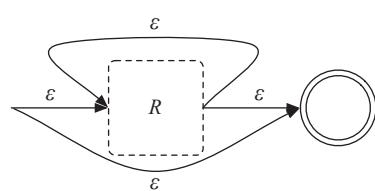
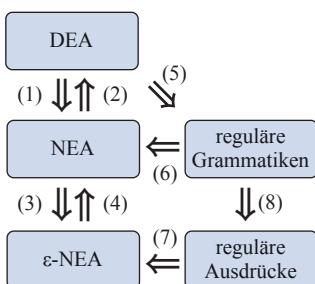


Abbildung 5.17: Jeder reguläre Ausdruck lässt sich rekursiv in einen äquivalenten nichtdeterministischen ϵ -Automaten transformieren.



- (1) jeder DEA ist ein NEA
- (2) Abschnitt 5.3.2
- (3) jeder NEA ist ein ϵ -NEA
- (4) Abschnitt 5.3.3
- (5) Abschnitt 5.4
- (6) Abschnitt 5.4
- (7) Abschnitt 5.4
- (8) siehe z. B. [52]

Abbildung 5.18: Zusammenfassung der herausgearbeiteten Äquivalenzbeziehungen

5.4.2 Abschlusseigenschaften

In Abschnitt 4.3 haben wir informell dargelegt, dass die Menge der regulären Sprachen bez. Vereinigung, Schnitt, Komplement, Produkt und Hüllbildung abgeschlossen ist. Auch hier ebnet unser erworbenes Wissen über die Äquivalenz von regulären Grammatiken und endlichen Automaten den Weg, um die Behauptungen jetzt nachträglich zu beweisen.

Wir beginnen mit der einfachsten Abschlusseigenschaft: dem Komplement. Um zu zeigen, dass für jede reguläre Sprache L auch das Komplement $\bar{L} = \Sigma^* \setminus L$ regulär ist, gehen wir in zwei Schritten vor: Im ersten Schritt konstruieren wir einen endlichen Automaten A mit $\mathcal{L}(A) = L$. Dass ein solcher Automat für jede reguläre Sprache existieren muss, haben wir im vorherigen Abschnitt herausgearbeitet. Im zweiten Schritt konstruieren wir aus A den *Komplementärautomaten* \bar{A} , der die Sprache $\mathcal{L}(\bar{A})$ erzeugt.



Definition 5.8 (Komplementärautomat)

Sei $A = (S, \Sigma, \delta, E, s_0)$ ein deterministischer endlicher Akzeptor.

$$\bar{A} := (S, \Sigma, \delta, S \setminus E, s_0)$$

heißt der *Komplementärautomat* von A .

Dass der Komplementärautomat die Sprache $\mathcal{L}(\bar{A})$ erzeugt, ist leicht einzusehen. Da wir die Menge der Endzustände invertiert haben, wird ein Wort ω genau dann von \bar{A} akzeptiert, wenn es von A zurückgewiesen wird. Mit L wird damit immer auch \bar{L} von einem Automaten akzeptiert, so dass die Menge der regulären Sprachen in Bezug auf das Komplement abgeschlossen ist.

Den Abschluss bez. Schnitt beweisen wir nach dem gleichen Schema. Wir werden zeigen, dass zwei Automaten $A = (S, \Sigma, \delta, E, s_0)$ und $A' = (S', \Sigma, \delta', E', s'_0)$ so miteinander verschmolzen werden können, dass am Ende die Sprache $\mathcal{L}(A) \cap \mathcal{L}(A')$ akzeptiert wird. Der gesuchten Akzeptor heißt *Produktautomat* und basiert auf der Idee, A und A' gewissermaßen parallel auszuführen. Um dies zu erreichen, definieren wir die neue Zustandsmenge als das kartesische Produkt von S und S' . Hierdurch entspricht jeder Zustand des Produktautomaten einem Tupel (s, s') , in das wir die aktuell eingenommenen Zustände von A und A' gleichzeitig hineincodieren können. Ferner legen wir die Zustandsübergangsfunktion so fest, dass der Produktautomat von einem Zustand

(s_1, s'_1) unter Eingabe von σ genau dann in den Zustand (s_2, s'_2) übergeht, wenn A von s_1 nach s_2 und A' von s'_1 nach s'_2 wechselt.

Definition 5.9 (Produktautomat)

Mit $A = (S, \Sigma, \delta, E, s_0)$ und $A' = (S', \Sigma, \delta', E', s'_0)$ seien zwei deterministische endliche Akzeptoren gegeben. Der Automat

$$A \times A' := (S'', \Sigma, \delta'', E'', s''_0) \quad \text{mit}$$

$$S'' := S \times S'$$

$$\delta''((s, s'), \sigma) := (\delta(s, \sigma), \delta'(s', \sigma))$$

$$E'' := \{(s, s') \mid s \in E \text{ und } s' \in E'\}$$

$$s''_0 := \{(s_0, s'_0)\}$$

heißt der *Produktautomat* von A und A' .

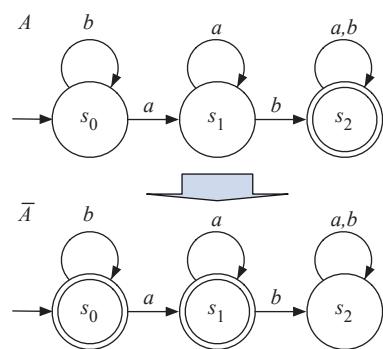


Abbildung 5.19: Der konstruierte Komplementärautomat akzeptiert die Sprache $\overline{\mathcal{L}(A)}$.

In den Abbildungen 5.19 und 5.20 wird die Bildung des Komplementär- und des Produktautomaten anhand konkreter Beispiele demonstriert.

Beachten Sie, dass wir die Menge E in Definition 5.9 so gewählt haben, dass der Produktautomat nur dann einen Finalzustand einnimmt, wenn sich sowohl A als auch A' in Finalzuständen befinden. Hierdurch akzeptiert $A \times A'$ ein Wort genau dann, wenn es von A und A' akzeptiert wird. Mit anderen Worten: Der Produktautomat akzeptiert die Sprache $\mathcal{L}(A) \cap \mathcal{L}(A')$. Ändern wir die Definition von E'' in

$$E'' := \{(s, s') \mid s \in E \text{ oder } s' \in E'\}$$

ab, so erhalten wir einen Akzeptor für die Sprache $\mathcal{L}(A) \cup \mathcal{L}(A')$.

Zwei Automaten A und A' lassen sich auch so zusammenführen, dass die Produktsprache $\mathcal{L}(A)\mathcal{L}(A')$ bzw. die Kleene'sche Hülle $\mathcal{L}(A)^*$ akzeptiert wird. Abbildung 5.21 skizziert die Grundidee für die Konstruktion eines Akzeptors für die Sprache $\mathcal{L}(A)\mathcal{L}(A')$. Die Automaten A und A' werden sequenziell zusammengeschaltet, indem jeder Endzustand von A über eine ϵ -Kante mit dem Startzustand von A' verbunden wird. Setzen wir die Finalzustände des neuen Automaten mit den Finalzuständen von A' gleich, so erhalten wir den gesuchten Akzeptor. Ebenso einfach lässt sich ein Automat A so modifizieren, dass er die Kleene'sche Hülle $\mathcal{L}(A)^*$ akzeptiert. Hierzu erzeugen wir einen neuen Startzustand, der gleichzeitig als Endzustand fungiert, und führen von jedem der alten Endzustände eine zusätzliche ϵ -Kante auf den neuen Startzustand zurück (vgl. Abbildung 5.22).

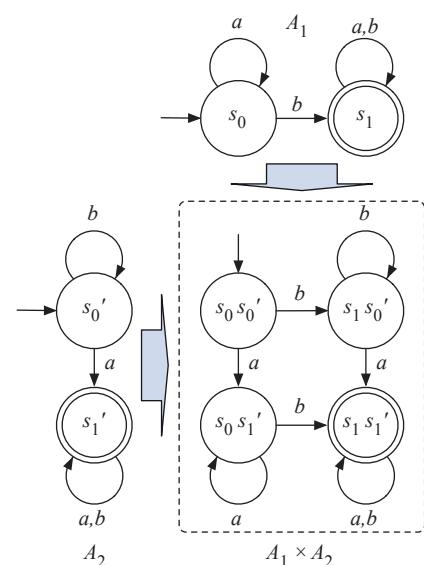


Abbildung 5.20: Der konstruierte Produktautomat akzeptiert die Sprache $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

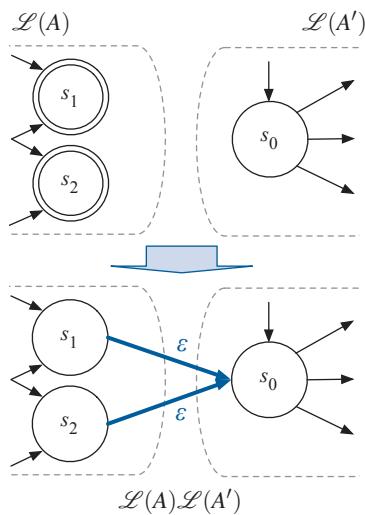


Abbildung 5.21: Zwei Akzeptoren A und A' können zu einem gemeinsamen Automaten verschmolzen werden, der die Sprache $\mathcal{L}(A)\mathcal{L}(A')$ erzeugt.

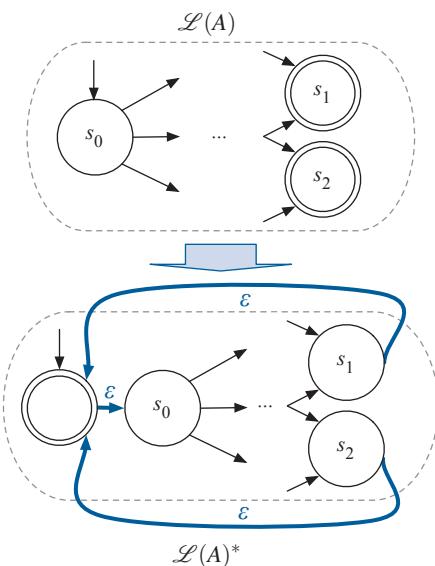


Abbildung 5.22: Jeder Akzeptor A lässt sich zu einem Automaten erweitern, der die Kleene'sche Hülle $\mathcal{L}(A)^*$ akzeptiert.

5.4.3 Entscheidungsprobleme

Im Folgenden wollen wir herausarbeiten, wie sich das Wortproblem, das Leerheitsproblem, das Endlichkeitsproblem und das Äquivalenzproblem für reguläre Sprachen entscheiden lassen. Erneut bezeichne G eine reguläre Grammatik und A einen DEA mit $\mathcal{L}(A) = \mathcal{L}(G)$.

- Das Wortproblem ist entschieden, wenn wir für alle $\omega \in \Sigma^*$ feststellen können, ob ω in $\mathcal{L}(G)$ enthalten ist oder nicht. Mithilfe des Akzeptors A können wir das Wortproblem für G lösen, indem wir A mit dem fraglichen Wort ω schlicht „ausführen“. ω gehört genau dann zu $\mathcal{L}(G)$, wenn sich A nach der Verarbeitung des letzten Eingabezeichens in einem Finalzustand befindet.
- Das Leerheitsproblem lässt sich mithilfe von Graphen-Algorithmen entscheiden. $\mathcal{L}(G)$ entspricht genau dann der leeren Menge, wenn im Zustandsdiagramm von A kein Pfad vom Startzustand zu einem Finalzustand führt.
- Genau wie das Leerheitsproblem lässt sich auch das Endlichkeitsproblem mithilfe von Graphen-Algorithmen entscheiden. $\mathcal{L}(G)$ ist genau dann endlich, wenn kein Pfad existiert, der den Startzustand mit einem Finalzustand verbindet und eine Schleife enthält.
- Hinter dem Äquivalenzproblem verbirgt sich die Frage, ob zwei reguläre Grammatiken G_1 und G_2 die gleiche Sprache erzeugen. Um das Problem zu entscheiden, bestimmen wir zunächst zwei Akzeptoren A_1 und A_2 mit $\mathcal{L}(A_1) = \mathcal{L}(G_1)$ und $\mathcal{L}(A_2) = \mathcal{L}(G_2)$. Jetzt nutzen wir aus, dass reguläre Sprachen bez. Schnitt und Komplement abgeschlossen sind und das Leerheitsproblem entscheidbar ist. Es gilt die folgende Reduktion:

$$\begin{aligned}\mathcal{L}(G_1) = \mathcal{L}(G_2) &\Leftrightarrow \mathcal{L}(G_1) \subseteq \mathcal{L}(G_2) \wedge \mathcal{L}(G_2) \subseteq \mathcal{L}(G_1) \\ \mathcal{L}(G_1) \subseteq \mathcal{L}(G_2) &\Leftrightarrow \mathcal{L}(A_1) \cap \overline{\mathcal{L}(A_2)} = \emptyset \\ \mathcal{L}(G_2) \subseteq \mathcal{L}(G_1) &\Leftrightarrow \mathcal{L}(A_2) \cap \overline{\mathcal{L}(A_1)} = \emptyset\end{aligned}$$

Die Automatenminimierung eröffnet uns eine zweite Möglichkeit, um das Äquivalenzproblem zu entscheiden. Es lässt sich zeigen, dass der Minimierungsalgorithmus Automaten erzeugt, die bis auf Isomorphie eindeutig bestimmt sind. Damit können wir die Äquivalenz überprüfen, indem wir zunächst A_1 und A_2 in eine minimierte Darstellung überführen und die entstandenen Zustandsgraphen anschließend einem Isomorphietest unterziehen. Auch dies können wir erledigen, indem wir auf Standardalgorithmen aus der Graphentheorie zurückgreifen.

5.4.4 Automaten und der Satz von Myhill-Nerode

In Abschnitt 4.3.3 haben wir mit dem Satz von Myhill-Nerode ein leistungsfähiges Instrument an die Hand bekommen, mit dem wir eine formale Sprache L als regulär oder nichtregulär identifizieren können. Inhaltlich besagt der Satz, dass eine Sprache L genau dann regulär ist, wenn die Menge Σ^* durch die Myhill-Relation \sim_L in endlich viele Äquivalenzklassen unterteilt wird. Zwei Wörter ω_1 und ω_2 stehen bezüglich \sim_L genau dann in Relation zueinander, wenn sie beide zu L oder beide nicht zu L gehören und diese Eigenschaft nicht verloren geht, wenn wir beide Wörter durch einen beliebigen Suffix v ergänzen. Bisher haben wir uns ausschließlich damit beschäftigt, wie der Satz von Myhill-Nerode angewendet werden kann, uns aber keinerlei Gedanken darüber gemacht, wie wir seine inhaltliche Aussage absichern können. In diesem Abschnitt wollen wir herausarbeiten, wie sich der Satz mit unserem erworbenen Wissen über endliche Automaten beweisen lässt.

In den nachfolgenden Betrachtungen bezeichnen wir mit A einen DEA und mit L die von A akzeptierte Sprache. Wir gehen davon aus, dass A reduziert ist und somit keine äquivalenten Zustände mehr enthält. Jetzt definieren wir auf der Menge Σ^* die Relation \sim_A :

$$\omega_1 \sim_A \omega_2 : \Leftrightarrow \text{Aus } (s_0, \omega_1) \rightarrow (s, \varepsilon) \text{ folgt } (s_0, \omega_2) \rightarrow (s, \varepsilon)$$

In Worten besagt die Definition, dass zwei Worte $\omega_1, \omega_2 \in \Sigma^*$ genau dann in Relation zueinander stehen, wenn sie A , ausgehend von s_0 , jeweils in denselben Zustand überführen. Den erreichten Zustand nennen wir s . Offenbar ist \sim_A eine Äquivalenzrelation, die für jeden Zustand eine eigene Äquivalenzklasse erzeugt.

Als Beispiel betrachten wir den DEA aus Abbildung 5.23 und die Wörter ε , a , ab und b . Die vier Wörter sind so gewählt, dass der Automat, ausgehend von s_0 , in einen anderen Zustand überführt wird. Somit können wir jedes dieser Wörter als Repräsentant von einer der vier Äquivalenzklassen auffassen, die durch \sim_A erzeugt werden. Im Einzelnen ergeben sich die folgenden Klassen:

$$[\varepsilon] = \{\varepsilon\} \quad (5.9)$$

$$[a] = \{a(ba)^n \mid n \in \mathbb{N}\} \quad (5.10)$$

$$[ab] = \{ab(ab)^n \mid n \in \mathbb{N}\} \quad (5.11)$$

$$[b] = \{\text{alle anderen Wörter}\} \quad (5.12)$$

Die Zustandsrelation \sim_A besitzt eine interessante Abschlusseigenschaft. Oben hatten wir festgelegt, dass $\omega_1 \sim \omega_2$ genau dann gilt, wenn der Automat in den gleichen Zustand übergeht, egal, ob er ω_1 oder ω_2 als

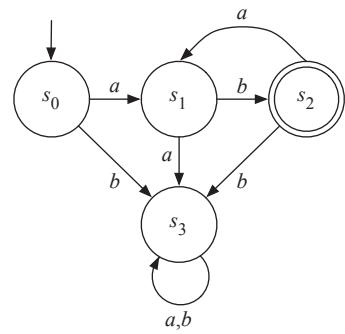


Abbildung 5.23: Akzeptor für die Sprache $\{(ab)^n \mid n \in \mathbb{N}^+\}$

■ Konstruktionsschema

$$S := \{[\omega_1], [\omega_2], \dots, [\omega_n]\}$$

$$\delta([\omega], \sigma) := [\omega\sigma]$$

$$E := \{[\omega] \mid \omega \in L\}$$

$$s_0 := [\varepsilon]$$

■ Beispiel

$$[\varepsilon] = \{\varepsilon\}$$

$$[a] = \{a(ab)^n \mid n \in \mathbb{N}\}$$

$$[ab] = \{ab(ab)^n \mid n \in \mathbb{N}\}$$

$$[b] = \{\text{alle anderen Wörter}\}$$

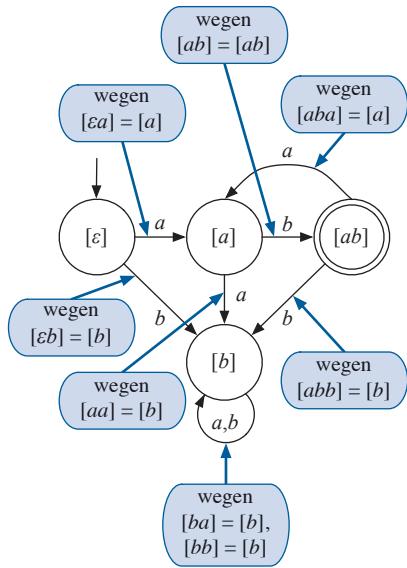


Abbildung 5.24: Unterteilt die Myhill-Relation \sim_L die Menge Σ^* in endlich viele Äquivalenzklassen, so lassen sich diese auf direkte Weise in einen endlichen Automaten übersetzen, der L akzeptiert.

Eingabe entgegennimmt. Die Eingabe weiterer Zeichen könnte daran nichts mehr ändern, denn sobald ein gemeinsamer Zustand erst einmal erreicht ist, läuft die Berechnung immer gleich ab. Das bedeutet, dass zwei Wörter ω_1 und ω_2 mit $\omega_1 \sim_A \omega_2$ auch dann noch in Relation zueinander stehen, wenn wir sie um einen beliebigen Suffix v erweitern. Kurz: Aus $\omega_1 \sim_A \omega_2$ folgt $\omega_1 v \sim_A \omega_2 v$ für alle $v \in \Sigma^*$. Da wir festgelegt haben, dass A reduziert ist, können wir zusätzlich den folgenden Schluss ziehen: Geht der Automat nach der Verarbeitung von ω_1 in einen anderen Zustand über als nach der Verarbeitung von ω_2 , so lassen sich ω_1 und ω_2 so um einen Suffix v ergänzen, dass eines der Wörter $\omega_1 v$ und $\omega_2 v$ von A akzeptiert wird und das andere nicht. Wäre dies nicht der Fall, so wären die erreichten Zustände äquivalent und der Automat, im Gegensatz zu unserer Annahme, nicht minimal.

Jetzt ist es uns gelungen, der Relation \sim_A ihr wahres Gesicht zu entlocken: sie ist mit der Nerode-Relation \sim_L aus Abschnitt 4.3.3 identisch. Der herausgearbeitete Zusammenhang liefert uns zugleich die Begründung für die inhaltliche Aussage des Satzes von Myhill-Nerode. Wir betrachten zunächst die Richtung von links nach rechts. Ist L eine reguläre Sprache, so wissen wir, dass ein DEA A mit $\mathcal{L}(A) = L$ existieren muss. A hat endlich viele Zustände und deshalb ist auch die Anzahl der von \sim_A erzeugten Äquivalenzklassen endlich. \sim_A ist aber nichts anderes als die Nerode-Relation \sim_L . Das bedeutet, dass der Index von \sim_L endlich ist und genau dies besagt der Satz von Myhill-Nerode.

Die Richtung von rechts nach links lässt sich konstruktiv begründen. Hat die Relation \sim_L einen endlichen Index $n \in \mathbb{N}$, so teilt sie die Menge Σ^* in endlich viele Äquivalenzklassen $[\omega_1], [\omega_2], \dots, [\omega_n]$ auf. Einen endlichen Automaten

$$A = (S, \Sigma, \delta, s_0)$$

mit $\mathcal{L}(A) = L$ können wir sehr einfach erhalten, indem wir die Menge der Äquivalenzklassen als Zustandsmenge verwenden.

Abbildung 5.24 verdeutlicht die Konstruktion am Beispiel der weiter oben angegebenen Äquivalenzklassen (5.9) bis (5.12). Als Ergebnis erhalten wir einen Automaten, der mit jenem aus Abbildung 5.23 identisch ist; einzige die Namen der Zustände haben sich geändert. Die durchgeführte Konstruktion besitzt eine weitere nennenswerte Eigenschaft: alle auf diese Weise konstruierten Automaten sind reduziert, d. h., es existieren keine äquivalenten Zustände mehr. Damit sind wir in der Lage, dem Index der Nerode-Relation eine ganz konkrete Bedeutung zu verleihen: er entspricht exakt der Anzahl der Zustände des Minimalautomaten. Ein herrliches Ergebnis!

5.5 Kellerautomaten

5.5.1 Definition und Eigenschaften

In Abschnitt 5.4 haben wir erkannt, dass alle Sprachen, die von endlichen Automaten akzeptiert werden, regulär sind. Das bedeutet, dass für die nichtreguläre Sprache

$$L_{C2} := \{a^n b^n \mid n \in \mathbb{N}^+\} \quad (5.13)$$

kein endlicher Automat existiert, der L akzeptiert. Warum es einen solchen nicht geben kann, demonstrieren die vier Akzeptoren in Abbildung 5.25. Ein Blick auf die Zustandsübergänge zeigt, dass der Automat A_n mit

$$\mathcal{L}(A_n) = \{a^i b^i \mid 1 \leq i \leq n\}$$

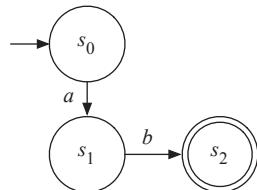
eine endliche Teilmenge von L_{C2} akzeptiert, die sich mit steigendem n kontinuierlich vergrößert. Die Sprache L_{C2} erhalten wir als Approximation der Sprachenfolge $\mathcal{L}(A_n)$ für $n \rightarrow \infty$:

$$L_{C2} = \bigcup_{n=1}^{\infty} \mathcal{L}(A_n)$$

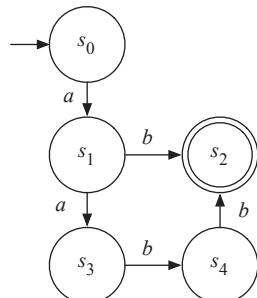
Die Anzahl der Zustände, die wir für den Aufbau des Automaten A_n benötigen, wächst linear mit dem Parameter n . Die Zunahme ist unvermeidbar, da sich der Akzeptor zunächst die Anzahl der gelesenen a 's merken muss, bevor er die darauf folgenden b 's verarbeiten kann. Da endliche Automaten per Definition über keine weiter gehenden Möglichkeiten zur Informationsspeicherung verfügen, bleibt uns als einzige Option übrig, den aktuellen Zählerstand in die Zustandsmenge hineinzucodieren. Folgerichtig muss der Automat A_n mindestens n Zustände besitzen, um n verschiedene Zählerstände zu unterscheiden. Da die Anzahl der a 's und b 's in den Wörtern von L_{C2} nach oben unbeschränkt ist, müsste ein entsprechender Akzeptor unendlich viele Zustände besitzen, im Widerspruch zu seiner Endlichkeit.

Aus dem Gesagten erwächst die folgende Vermutung: Gelingt es, einen endlichen Automaten um einen unendlich großen Gedächtnisspeicher anzureichern, so müssten Sprachen der Form (5.13) problemlos zu erkennen sein. Diese Überlegung führt uns auf direktem Weg zum Begriff des *Kellerautomaten*.

■ Automat A_1



■ Automat A_2



■ Automat A_n

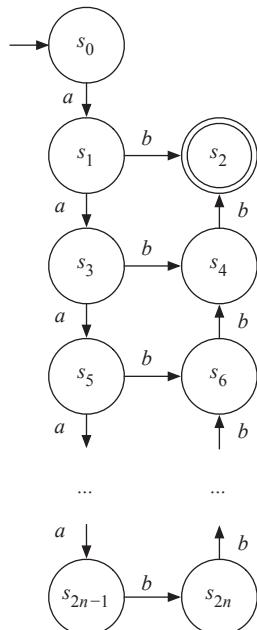


Abbildung 5.25: Endliche Akzeptoren für die Sprachen $L_n = \{a^i b^i \mid 1 \leq i \leq n\}$

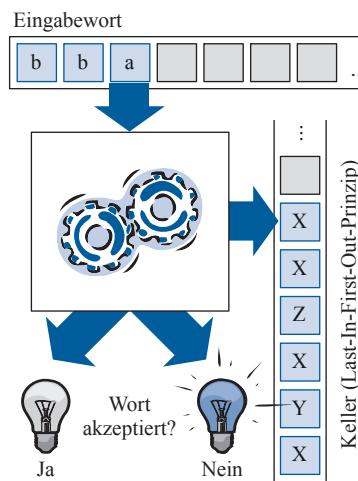


Abbildung 5.26: Kellerautomat

**Definition 5.10 (Kellerautomat)**

Ein Kellerautomat (*pushdown automaton*), kurz PDA, ist ein 5-Tupel $(S, \Sigma, \Gamma, \delta, s_0)$. Er besteht aus

- der endlichen *Zustandsmenge* S ,
- dem endlichen *Eingabealphabet* Σ mit $\epsilon \notin \Sigma$,
- dem endlichen *Kelleralphabet* Γ mit $\perp \in \Gamma$,
- der *Zustandsübergangsfunktion* $\delta : S \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{S \times \Gamma^*}$ mit $|\delta(s, \omega, \gamma)| < \infty$ für alle s, ω, γ .
- dem *Startzustand* s_0 .

Im Wesentlichen entspricht ein Kellerautomat einem nichtdeterministischen ϵ -Akzeptor, der um einen separaten Kellerspeicher mit unendlich großer Kapazität erweitert wurde (vgl. Abbildung 5.26). Der Kellerspeicher ist als *Stapel* (*stack*) organisiert und erlaubt daher nur einen eingeschränkten Zugriff auf die Elemente. Die Funktionsweise ist ähnlich derer eines konventionellen Bücherstapels; hier können wir ein neues Buch entweder oben auf den Stapel packen (*Push-Operation*) oder das oberste Buch wegnehmen (*Pop-Operation*). In einem Stack dürfen Elemente weder von unten noch aus der Mitte entnommen werden, d.h., das zuletzt hinzugefügte Element wird immer als erstes wieder entfernt (*LIFO-Prinzip, Last-In-First-Out*).

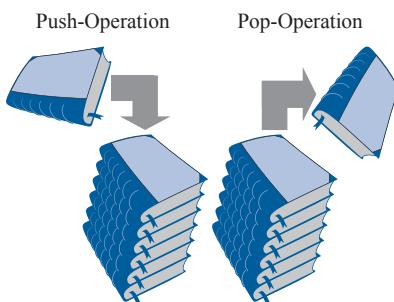


Abbildung 5.27: Stacks (Stapel) arbeiten nach dem Last-In-First-Out-Prinzip. Ähnlich einem gewöhnlichen Bücherstapel werden neue Elemente stets auf die Stapeloberseite gelegt (PUSH-Operation) und von dort entnommen (POP-Operation). Das zuletzt hinzugefügte Element wird also stets als erstes wieder entfernt.

Im Gegensatz zu den bisher betrachteten Akzeptoren besitzen Kellerautomaten zwei verschiedene Alphabete Σ und Γ . Das Eingabealphabet Σ enthält die Zeichen, aus denen sich die Eingabewörter zusammensetzen, und das Kelleralphabet Γ alle Symbole, die in den Kellerspeicher geschrieben werden dürfen. Damit beide Alphabete einfacher unterschieden werden können, halten wir uns an die gängige Konvention, Eingabezeichen mit Kleinbuchstaben und Kellerzeichen mit Großbuchstaben darzustellen.

In gewohnter Weise wird das Eingabewort in einem Kellerautomaten zeichenweise bearbeitet. In jedem Verarbeitungsschritt liest der PDA ein Eingabezeichen σ ein und entfernt das oberste Kellerzeichen γ . In Abhängigkeit von σ , γ und des aktuellen Zustands s geht der Kellerautomat in einen Folgezustand s' über und schreibt mithilfe mehrerer PUSH-Operationen eine Zeichenkette $\gamma'_0 \dots \gamma'_i \in \Gamma^*$ in den Kellerspeicher. Damit im ersten Verarbeitungsschritt überhaupt ein Kellerzeichen entfernt werden kann, befüllen wir den Speicher vorab mit dem dedizierten Element $\perp \in \Gamma$.

Die Zustandsübergänge eines Kellerautomaten entsprechen denen des ε -NEAs aus Abschnitt 5.3.3, so dass es Verarbeitungsschritte geben kann, die kein Eingabezeichen konsumieren (ε -Übergang). Nichtsdestotrotz wird auch in diesem Fall das oberste Kellerzeichen entfernt. Beachten Sie, dass in jedem Schritt mehr als ein Zeichen in den Kellerspeicher zurückgeschrieben werden darf. Hierdurch ist es möglich, die Entfernung des obersten Zeichens rückgängig zu machen, indem wir es als zusätzliches Symbol wieder in den Kellerspeicher legen. Es ist ebenfalls erlaubt, überhaupt kein Zeichen in den Kellerspeicher zurückzuschreiben. In diesem Fall wird die Anzahl der im Keller gespeicherten Elemente um 1 reduziert.

Formal lässt sich das Verhalten eines Kellerautomaten mit dem Begriff der Konfiguration erfassen, den wir in den Abschnitten 5.2 und 5.3 in ähnlicher Form für die Beschreibung von DEAs und NEAs erfolgreich eingesetzt haben.



Definition 5.11 (Konfiguration (PDA))

Mit $K = (S, \Sigma, \Gamma, \delta, s_0)$ sei ein beliebiger Kellerautomat gegeben. Jedes Tripel (s, ω, κ) mit $s \in S$, $\omega \in \Sigma^*$ und $\kappa \in \Gamma^*$ heißt eine *Konfiguration* von K . Die Übergangsrelation \rightarrow definieren wir wie folgt:

$$(s_1, \omega, \gamma\kappa) \rightarrow (s_2, \omega, \kappa'\kappa) \Leftrightarrow (s_2, \kappa') \in \delta(s_1, \varepsilon, \gamma)$$

$$(s_1, \sigma\omega, \gamma\kappa) \rightarrow (s_2, \omega, \kappa'\kappa) \Leftrightarrow (s_2, \kappa') \in \delta(s_1, \sigma, \gamma)$$

Vereinbarungsgemäß akzeptiert ein Kellerautomat $K = (S, \Sigma, \Gamma, \delta, s_0)$ die folgende Sprache:

$$\mathcal{L}(K) := \{\omega \mid (s_0, \omega, \perp) \xrightarrow{*} (s_i, \varepsilon, \varepsilon)\}$$

Wir haben die akzeptierte Sprache in einer Form festgelegt, die gänzlich ohne Endzustände auskommt. Ob das Wort ω akzeptiert wird, hängt ausschließlich von der Beschaffenheit des Kellers ab. Konkret wird ein Wort ω genau dann akzeptiert, wenn es eine Möglichkeit gibt, die Verarbeitung mit einem leeren Keller abzuschließen. Beachten Sie, dass der Kellerautomat nichtdeterministisch arbeitet. Hierdurch ist ein Wort ω bereits dann in der Sprache enthalten, wenn mindestens eine Möglichkeit existiert, die Bearbeitung mit einem leeren Keller zu beenden. Daneben können andere Berechnungspfade existieren, die zu einem nichtleeren Keller führen.

Die angestellten Überlegungen versetzen uns in die Lage, einen Kellerautomaten zu konstruieren, der die nichtreguläre Sprache $L_{C2} = \{a^n b^n \mid n \in \mathbb{N}^+\}$ akzeptiert. Wie in Abbildung 5.28 gezeigt, kommt ein solcher Automat mit

■ Kellerautomat $K = (S, \Sigma, \Gamma, \delta, s_0)$

$$S := \{s_0, s_1\}$$

$$\Sigma := \{a, b\}$$

$$\Gamma := \{A, \perp\}$$

$$\delta(s_0, a, \perp) :=_{(1)} \{(s_0, A\perp)\}$$

$$\delta(s_0, a, A) :=_{(2)} \{(s_0, AA)\}$$

$$\delta(s_0, b, A) :=_{(3)} \{(s_1, \varepsilon)\}$$

$$\delta(s_1, b, A) :=_{(4)} \{(s_1, \varepsilon)\}$$

$$\delta(s_1, \varepsilon, \perp) :=_{(5)} \{(s_1, \varepsilon)\}$$

■ Beispiel 1: $\omega = aabb$

Zustand	ω	Keller
s_0	aabb	\perp
s_0	abb	$A\perp$
s_0	bb	$AA\perp$
s_1	b	$A\perp$
s_1	ε	\perp
s_1	ε	ε

✓ akzeptiert

■ Beispiel 2: $\omega = aaabb$

Zustand	ω	Keller
s_0	aaabb	\perp
s_0	aabb	$A\perp$
s_0	abb	$AA\perp$
s_0	bb	$AAA\perp$
s_1	b	$AA\perp$
s_1	ε	$A\perp$

✗ nicht akzeptiert

Abbildung 5.28: Die nichtreguläre Sprache $L_{C2} = \{a^n b^n \mid n \in \mathbb{N}^+\}$ lässt sich mithilfe des dargestellten Kellerautomaten akzeptieren.

■ Kellerautomat

$$\begin{aligned} S &:= \{s_0, s_1\} \\ \Sigma &:= \{a, b\} \\ \Gamma &:= \{A, B\} \\ \delta(s_0, a, \gamma) &:= (1) \{(s_0, A\gamma)\} \\ \delta(s_0, b, \gamma) &:= (2) \{(s_0, B\gamma)\} \\ \delta(s_0, \epsilon, \gamma) &:= (3) \{(s_1, \gamma)\} \\ \delta(s_1, a, A) &:= (4) \{(s_1, \epsilon)\} \\ \delta(s_1, b, B) &:= (5) \{(s_1, \epsilon)\} \\ \delta(s_1, \epsilon, \perp) &:= (6) \{(s_1, \epsilon)\} \end{aligned}$$

■ Beispiel: $\omega = abbaabba$

	Zustand	ω	Keller
(1)	s_0	abbaabba	\perp
(2)	s_0	bbaabba	$A\perp$
(2)	s_0	baabba	$BA\perp$
(2)	s_0	aabba	$BBA\perp$
(1)	s_0	abba	$ABBA\perp$
(3)	s_1	abba	$ABBA\perp$
(4)	s_1	bba	$BBA\perp$
(5)	s_1	ba	$BA\perp$
(5)	s_1	a	$A\perp$
(4)	s_1	ϵ	\perp
(6)	s_1	ϵ	ϵ

✓ akzeptiert

Abbildung 5.29: Kellerautomat zum Erkennen der Palindromsprache L_{Pal}

zwei Zuständen s_0 und s_1 aus. Das Kelleralphabet Γ besteht aus einem einzigen Symbol A , das wir als Merker für die Anzahl der bereits gelesenen a 's einsetzen. Jedes Mal, wenn ein a eingelesen wird, legt der Automat ein A im Keller ab. Sobald das erste b gelesen wird, wechselt der Automat in den Zustand s_1 und beginnt, den Keller mit jedem gelesenen Symbol zu leeren. Genau dann, wenn die Anzahl der gelesenen a 's gleich der Anzahl der gelesenen b 's ist, kann der Keller vollständig geleert werden und das Eingabewort wird akzeptiert.

In dem diskutierten Beispiel haben wir die volle Leistungsfähigkeit des Kellerautomaten noch gar nicht ausgeschöpft. Dies wollen wir jetzt nachholen und zeigen, wie die *Palindromsprache*

$$L_{\text{Pal}} := \{\epsilon\} \cup \{\sigma_1 \dots \sigma_n \sigma_n \dots \sigma_1 \mid n \in \mathbb{N}^+, \sigma_i \in \{a, b\}\} \quad (5.14)$$

mit einem Kellerautomaten erkannt werden kann. Die Grundidee des Automaten ist simpel: Zunächst werden die Eingabesymbole $\sigma_1, \dots, \sigma_n$ der Reihe nach eingelesen und in den Kellerspeicher geschrieben. Anschließend werden die nächsten n Symbole verarbeitet und Zeichen für Zeichen mit dem Kellerinhalt abgeglichen. Der Kellerautomat fährt mit der Bearbeitung nur dann fort, wenn das aktuell verarbeitete Zeichen mit dem obersten Zeichen des Kellerspeichers übereinstimmt (vgl. Abbildung 5.29).

So weit so gut. Aber wie kann der Automat wissen, wann mit dem Rückbau des Kellerspeichers begonnen werden muss? Da die Länge des Eingabeworts zu Beginn der Verarbeitung nicht bekannt ist, ist es unmöglich, den Wert von n vorab auf deterministischem Weg zu bestimmen. Dass wir die Palindromsprache trotzdem mithilfe eines Kellerautomaten erkennen können, haben wir seinen nichtdeterministischen Eigenschaften zu verdanken. Im Gegensatz zu seinen deterministischen Pendants kann ein nichtdeterministischer Automat die Wortmitte schlicht erraten.

5.5.2 Kellerautomaten und kontextfreie Sprachen

Kellerautomaten sind alles andere als eine willkürliche Erweiterung der endlichen Automaten aus Abschnitt 5.2. Die Architektur ist so angelegt, dass die Menge der von Kellerautomaten akzeptierten Sprachen und die Menge der kontextfreien Sprachen aus Abschnitt 4.4 identisch sind. In diesem Abschnitt werden wir untersuchen, wie dieser Zusammenhang zustande kommt.

Als Erstes wollen wir uns mit der Frage beschäftigen, wie sich eine kontextfreie Grammatik

$$G = (V, \Sigma, P, S)$$

in einen Kellerautomaten

$$K_G := (\{s_0\}, \Sigma, \Gamma, \delta, s_0) \quad (5.15)$$

übersetzen lässt, der exakt die Wörter aus $\mathcal{L}(G)$ akzeptiert. Die Lösung des Problems besteht in der Konstruktion eines Automaten, der die Ableitung eines Wortes ω im Kellerspeicher nachvollzieht. Wie Gleichung (5.15) bereits andeutet, kommt der konstruierte Automat mit nur einem Zustand s_0 aus, d. h., die gesamte Intelligenz ist in den Regeln zur Manipulation des Kellerspeichers verborgen.

Das Kelleralphabet setzen wir wie folgt fest:

$$\Gamma := \Sigma \cup V$$

Die Menge der Produktionen P kann eins zu eins in die Zustandsübergangsfunktion δ hineincodiert werden. Für jedes Nonterminal $A \in V$ übersetzen wir die Menge der Produktionen

$$A \rightarrow \omega_1, \dots, A \rightarrow \omega_n$$

in die folgende Regel:

$$\delta(s_0, \epsilon, A) := \{(s_0, \omega_1), \dots, (s_0, \omega_n)\}$$

Zusätzlich definieren wir Regeln für den Umgang mit Terminalzeichen. Wird aktuell das Terminalzeichen σ eingelesen, so soll der Kellerautomat nur dann mit der Verarbeitung fortfahren, wenn σ gleichzeitig das oberste Kellerzeichen ist. Um das gewünschte Verhalten zu erzielen, definieren wir für jedes Zeichen $\sigma \in \Sigma$ den folgenden Zustandsübergang:

$$\delta(s_0, \sigma, \sigma) := \{(s_0, \epsilon)\}$$

Zu guter Letzt ergänzen wir den Kellerautomaten um eine Regel, die das Symbol \perp durch das Startsymbol S ersetzt und auf diese Weise einen geregelten Start der Worterkennung ermöglicht:

$$\delta(s_0, \epsilon, \perp) := \{(s_0, S)\}$$

Abbildung 5.30 demonstriert die skizzierte Automatenkonstruktion für die in Abschnitt 4.1 eingeführte Dyck-Sprache D_2 . Im unteren Teil der Abbildung ist eine Folge von Zustandsübergängen für das Dyck-Wort

Grammatik

$$S \Rightarrow \epsilon \mid SS \mid [S] \mid (S)$$

Kellerautomat K

$$\begin{aligned} \delta(s_0, \epsilon, S) &:=_{(1)} \{(s_0, \epsilon)\} \\ \delta(s_0, \epsilon, S) &:=_{(2)} \{(s_0, SS)\} \\ \delta(s_0, \epsilon, S) &:=_{(3)} \{(s_0, [S])\} \\ \delta(s_0, \epsilon, S) &:=_{(4)} \{(s_0, (S))\} \\ \delta(s_0, \sigma, \sigma) &:=_{(5)} \{(s_0, \epsilon)\} \\ \delta(s_0, \epsilon, \perp) &:=_{(6)} \{(s_0, S)\} \end{aligned}$$

Beispiel: $\omega = ()[()]()$

Zustand	ω	Keller
s_0	$()[()]()$	\perp
s_0	$()[()]()$	S
s_0	$()[()]()$	SS
s_0	$()[()]()$	$(S)S$
s_0	$)[()]()$	$S)S$
s_0	$)[()]()$	$)S$
s_0	$[()]()$	S
s_0	$[()]()$	SS
s_0	$[()]()$	$[S]S$
s_0	$(]()$	$S]S$
s_0	$(]()$	$(S)]S$
s_0	$] ()$	$S)]S$
s_0	$] ()$	$)]S$
s_0	$] ()$	$]S$
s_0	$()$	S
s_0	$()$	(S)
s_0	$()$	$S)$
s_0	$()$	$)$
s_0	ϵ	ϵ

✓ akzeptiert

Abbildung 5.30: Jede kontextfreie Grammatik G lässt sich in einen Kellerautomaten K übersetzen, der $\mathcal{L}(G)$ akzeptiert. Der in diesem Beispiel konstruierte Automat akzeptiert die Dyck-Sprache D_2 .

■ Kellerautomat

$$\begin{aligned} S &:= \{s_0, s_1\} \\ \Sigma &:= \{a, b, \$\} \\ \Gamma &:= \{A, B\} \\ \delta(s_0, a, \gamma) &:=_{(1)} \{(s_0, A\gamma)\} \\ \delta(s_0, b, \gamma) &:=_{(2)} \{(s_0, B\gamma)\} \\ \delta(s_0, \$, \gamma) &:=_{(3)} \{(s_1, \gamma)\} \\ \delta(s_1, a, A) &:=_{(4)} \{(s_1, \varepsilon)\} \\ \delta(s_1, b, B) &:=_{(5)} \{(s_1, \varepsilon)\} \\ \delta(s_1, \varepsilon, \perp) &:=_{(6)} \{(s_1, \varepsilon)\} \end{aligned}$$

■ Beispiel: $\omega = abba\$abba$

Zustand	ω	Keller
s_0	$abba\$abba$	\perp
(1)	s_0	$bba\$abba$
(2)	s_0	$ba\$abba$
(2)	s_0	$a\$abba$
(1)	s_0	$\$abba$
(3)	s_1	$abba$
(4)	s_1	bba
(5)	s_1	ba
(5)	s_1	a
(4)	s_1	ε
(6)	s_1	ε

✓ akzeptiert

Abbildung 5.31: Deterministischer Kellerautomat zur Erkennung der Palindromsprache $L_{\text{Pal\$}}$

$()()$ dargestellt, die mit einem leeren Keller endet. An diesem Beispiel lässt sich gut erkennen, wie die einzelnen Ableitungsschritte im Kellerspeicher repliziert werden.

Wir haben in diesem Abschnitt herausgearbeitet, wie sich jede kontextfreie Grammatik in einen äquivalenten Kellerautomaten überführen lässt. In der Tat lässt sich auch die Umkehrung beweisen: Jede von einem Kellerautomaten K akzeptierte Sprache $\mathcal{L}(K)$ lässt sich durch eine kontextfreie Grammatik G erzeugen. Den nicht ganz einfachen Beweis wollen wir an dieser Stelle nicht führen; er ist ausführlich in [52] beschrieben. Insgesamt gilt der folgende Satz:



Satz 5.3 (Äquivalenztheorem für Kellerautomaten)

Die Klasse der von Kellerautomaten akzeptierten Sprachen ist mit der Klasse der kontextfreien Sprachen identisch.

Beachten Sie, dass unser Konstruktionsschema einen Kellerautomaten produziert, der genau einen Zustand s_0 besitzt. Damit zeigt Satz 5.3 zugleich, dass sich jeder Kellerautomat auf einen äquivalenten Automaten mit einem einzigen Zustand reduzieren lässt. Einen solchen können wir erzeugen, indem wir aus dem ursprünglichen Automaten zunächst eine kontextfreie Grammatik erzeugen und diese anschließend in einen Kellerautomaten mit nur einem Zustand zurückübersetzen.

Das Ergebnis gibt einen wichtigen Hinweis darauf, woher die Ausdrucksstärke von Kellerautomaten herrührt. Sie wird ausschließlich durch den unendlich großen Kellerspeicher geschaffen und durch die Kombination mit einer endlichen Zustandsmenge nicht erweitert. Trotzdem sind die Zustände nicht sinnlos. Für viele Sprachen lassen sich akzeptierende Kellerautomaten deutlich kompakter und übersichtlicher formulieren, wenn mehr als ein Zustand verwendet wird.

5.5.3 Deterministische Kellerautomaten

Sehen wir von der speziellen Behandlung des Kellerspeichers ab, so arbeitet ein Kellerautomat genau nach dem gleichen Prinzip wie ein nichtdeterministischer ε -Automat. In diesem Abschnitt wollen wir eine spezielle Variante des Kellerautomaten einführen, die die nichtdeterministischen Zustandsübergänge beseitigt.



Definition 5.12 (Deterministischer Kellerautomat)

Ein Kellerautomat $(S, \Sigma, \Gamma, \delta, s_0)$ heißt *deterministisch*, wenn für alle Zustände $s \in S$, Eingabezeichen $\sigma \in \Sigma$ und Kellersymbole $\kappa \in \Gamma$ die folgende Beziehung gilt: $|\delta(s, \sigma, \kappa) \cup \delta(s, \varepsilon, \kappa)| \leq 1$

Demnach existiert in einem deterministischen Kellerautomaten für jedes Eingabezeichen ω und jedes oberste Kellerzeichen κ maximal ein Zustandsübergang. Genau wie bisher wird ein Wort ω akzeptiert, wenn der Automat nach der Verarbeitung des letzten Eingabezeichens einen leeren Keller aufweist. Verbleiben dagegen Symbole im Kellerspeicher oder bleibt der Automat vor der Verarbeitung des letzten Zeichens aufgrund einer fehlenden Übergangsregel stehen, so wird ω abgelehnt. Im Gegensatz zur Architektur des klassischen DEAs aus Abschnitt 5.2 wollen wir die Vereinbarung treffen, dass auch ein deterministischer Kellerautomat nicht in jedem Schritt zwingend ein Eingabezeichen konsumieren muss. ε -Übergänge sollen jedoch nur dann zulässig sein, wenn keine andere Übergangsregel anwendbar ist.

Als Beispiel ist in Abbildung 5.31 ein deterministischer Kellerautomat definiert, der die Sprache

$$L_{\text{Pal\$}} := \{\$\} \cup \{\sigma_1 \dots \sigma_n \$ \sigma_n \dots \sigma_1 \mid n \in \mathbb{N}^+, \sigma_i \in \Sigma \setminus \{\$\}\} \quad (5.16)$$

akzeptiert. Die Sprache $L_{\text{Pal\$}}$ unterscheidet sich von der Palindromsprache L_{Pal} durch ein zusätzliches Terminalzeichen $\$$, das die Mitte des Eingabeworts markiert. Die Verwendung eines zusätzlichen Mittelzeichens ist an dieser Stelle essentiell, da der Automat die Mitte des Worts, anders als im nichtdeterministischen Fall, nicht erraten kann. Verzichten wir auf das Mittelzeichen, so kann die Sprache von keinem deterministischen Kellerautomaten akzeptiert werden.

Tatsächlich stehen wir hier vor einer gänzlich anderen Situation als im Falle des klassischen endlichen Automaten. Dort konnten wir zeigen, dass sich jeder ε -NEA in einen äquivalenten deterministischen Akzeptator überführen lässt. Für Kellerautomaten ist eine solche Reduktion nicht möglich. In der Konsequenz ergibt sich der folgende Satz, den wir an dieser Stelle ohne formalen Beweis akzeptieren wollen:



Satz 5.4

Die Sprachklasse der von deterministischen Kellerautomaten akzeptierten Sprachen ist eine echte Teilmenge der von nichtdeterministischen Kellerautomaten akzeptierten Sprachen.



In der Literatur ist der Begriff des Kellerautomaten nicht eindeutig definiert. In diesem Buch wird ein Kellerautomat als 5-Tupel $(S, \Sigma, \Gamma, \delta, s_0)$ beschrieben, in anderen dagegen als 6-Tupel $(S, \Sigma, \Gamma, \delta, E, s_0)$. Die zusätzlich hinzugefügte Komponente E ist die Menge der Endzustände.

Dass wir vollständig auf Endzustände verzichten konnten, haben wir der gewählten Akzeptanzbedingung zu verdanken. Ein Automat unserer Bauart akzeptiert ein Eingabewort ω genau dann, wenn der Kellerspeicher nach der Verarbeitung des letzten Eingabezeichens leer ist. Dabei ist es unerheblich, in welchem Zustand er sich am Ende befindet.

Automaten mit Endzuständen verwenden eine andere Akzeptanzbedingung. Ähnlich dem klassischen DEA wird ein Wort ω genau dann akzeptiert, wenn nach der Verarbeitung des letzten Eingabezeichens ein Endzustand eingenommen wird. In diesem Fall spielt es keine Rolle, ob der Kellerspeicher zu diesem Zeitpunkt leer ist oder noch Symbole enthält.

Die beiden Akzeptanzbedingungen sind gleichwertig, d.h., für jeden Kellerautomaten K , der eine Sprache L mit leerem Keller akzeptiert, existiert ein Kellerautomat K' , der L per Endzustand akzeptiert, und umgekehrt. Beachten Sie, dass diese Beziehung für deterministische Kellerautomaten nicht gilt! Hier wird die Menge der akzeptierbaren Sprachen durch den Wechsel der Akzeptanzbedingung in der Tat verändert.

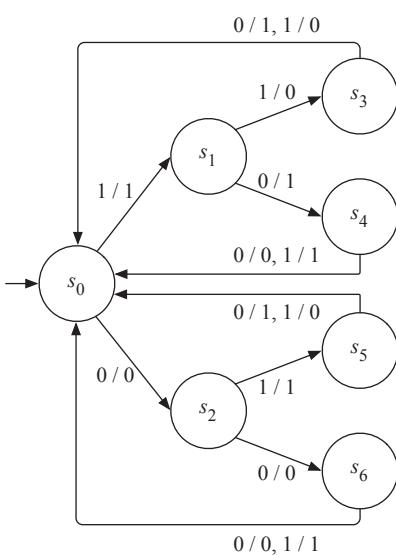


Abbildung 5.32: Seriell arbeitender Binär-Gray-Code-Wandler

5.6 Transduktoren

5.6.1 Definition und Eigenschaften

Alle der bisher betrachteten Automatentypen arbeiten als Akzeptoren. Ein Eingabewort wird Zeichen für Zeichen eingelesen und am Ende entschieden, ob die eingelesene Sequenz akzeptiert oder abgelehnt wird. Im Gegensatz hierzu arbeiten *Transduktoren* als Übersetzer. Anstelle einer Ja-Nein-Ausgabe produzieren diese eine Menge von Ausgabezeichen, die nacheinander auf ein separates Ausgabeband geschrieben werden. Formal definieren wir den Begriff des Transduktors wie folgt:



Definition 5.13 (Endlicher Transduktor)

Ein deterministischer endlicher Transduktor (*deterministic finite state transducer*), kurz DET, ist ein 6-Tupel $(S, \Sigma, \Pi, \delta, \lambda, s_0)$. Er besteht aus

- der endlichen *Zustandsmenge* S ,
- dem endlichen *Eingabealphabet* Σ ,
- dem endlichen *Ausgabealphabet* Π ,
- der *Zustandsübergangsfunktion* $\delta : S \times \Sigma \rightarrow S$,
- der *Ausgabefunktion* $\lambda : S \times \Sigma \rightarrow \Pi$ und
- dem *Startzustand* s_0 .

Zu Beginn befindet sich ein Transduktor in seinem Startzustand s_0 . Wird er mit dem Eingabewort

$$\omega = \sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n$$

stimuliert, so durchläuft er nacheinander die Zustände

$$s_0, s_1, s_2, \dots, s_n \quad \text{mit} \quad s_{i+1} = \delta(s_i, \sigma_i)$$

und produziert die Ausgabe

$$\pi_0, \pi_1, \pi_2, \dots, \pi_n \quad \text{mit} \quad \pi_i = \lambda(s_i, \sigma_i).$$

Transduktoren besitzen per Definition keine Endzustände. Es interessiert einzig das produzierte Ausgabewort.

Um den Begriff mit Leben zu füllen, betrachten wir das Beispiel in Abbildung 5.32. Der dargestellte Transduktor implementiert einen seriellen Code-Wandler, der einen binären Eingabestrom entgegennimmt ($\Sigma = \{0, 1\}$) und in einen ebenfalls binären Ausgabestrom übersetzt ($\Pi = \{0, 1\}$). Der Automat startet im Zustand s_0 und interpretiert jeweils drei kontinuierliche Eingabeziffern als eine zusammengehörige Binärzahl. Diese wird durch die Automatenlogik in den *Gray-Code* umgesetzt und das resultierende Bitmuster auf das Ausgabeband geschrieben. In jedem Verarbeitungsschritt liest und schreibt der Transduktor genau eine Ziffer. Die Bitmuster des Gray-Codes sind in Abbildung 5.33 zusammengefasst.

Beachten Sie die erweiterten Kantenmarkierungen des Transduktors. In der Markierung σ/π steht σ für ein Symbol des Eingabealphabets Σ und π für ein Symbol des Ausgabealphabets Π .

5.6.2 Automatenminimierung

In Abschnitt 5.2 haben wir den Äquivalenzbegriff für endliche Akzeptoren eingeführt. Abstrakt betrachtet sind zwei Automaten genau dann äquivalent, wenn sie nach außen das gleiche Verhalten zeigen. Im Falle von Akzeptoren bedeutet dies nichts anderes, als dass beide die gleiche Sprache akzeptieren. Für übersetzende Automaten können wir den Begriff ganz ähnlich definieren und bezeichnen zwei Transduktoren als äquivalent, wenn jede Eingabesequenz die gleiche Ausgabesequenz erzeugt. Genau wie im Falle des Akzeptors lässt sich die Automatenäquivalenz mit dem Begriff der Bisimulation formal erfassen.



Definition 5.14 (Zustandsäquivalenz, Bisimulation)

Sei $A = (S, \Sigma, \Pi, \delta, \lambda, s_0)$ ein endlicher Transduktor. Die k -Äquivalenz zwischen zwei Zuständen s_1 und s_2 , geschrieben als $s_1 \sim_k s_2$, definieren wir wie folgt:

$$\begin{aligned} s_1 \sim_0 s_2 &:\Leftrightarrow \text{Für alle } \sigma \in \Sigma \text{ gilt } \lambda(s_1, \sigma) = \lambda(s_2, \sigma) \\ s_1 \sim_{k+1} s_2 &:\Leftrightarrow s_1 \sim_0 s_2 \text{ und} \\ &\quad \text{für alle } \sigma \in \Sigma \text{ gilt } \delta(s_1, \sigma) \sim_k \delta(s_2, \sigma) \end{aligned}$$

Gilt $s_1 \sim_k s_2$ für alle $k \in \mathbb{N}$, so heißen s_1 und s_2 äquivalent, in Zeichen $s_1 \sim s_2$. Die Relation \sim wird als *Bisimulation* bezeichnet.

■ Code-Tabelle

Dezimal	Binär	Gray-Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

■ Anwendung

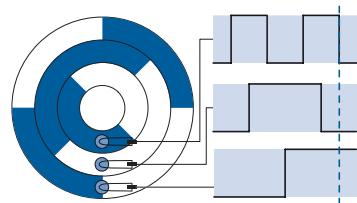


Abbildung 5.33: Der Gray-Code besitzt die Eigenschaft, dass sich die Bitmuster zweier benachbarter Ziffern in genau einem Bit unterscheiden [39]. Codes mit dieser Eigenschaft heißen *einschrittig*. Im Bereich der Automatisierungstechnik wird der Gray-Code unter anderem zur Codierung von Messwerten verwendet, die über mechanische oder optoelektronische Sensoren erfasst werden. Aufgrund der Einschrittigkeit ist er weit weniger anfällig für Messfehler als der konventionelle Binärcode.

Abbildung 5.34: Schrittweise Reduktion des endlichen Transduktors aus Abbildung 5.32. Im ersten Schritt wird die Zustandsmenge so aufgeteilt, dass genau diejenigen Zustände in einer Äquivalenzklasse zusammengefasst sind, die für alle Eingabezeichen σ das gleiche Ausgabezeichen erzeugen. Nach diesem Schritt sind alle 0-äquivalenten Zustände bestimmt. Anschließend werden die Äquivalenzklassen in einem iterativen Prozess weiter aufgeteilt. Nach der ersten Iteration sind alle in einer Klasse verbleibenden Zustände paarweise 1-äquivalent, nach der zweiten Iteration paarweise 2-äquivalent und so fort. Machen wir mit der Aufteilung so lange weiter, bis ein Fixpunkt erreicht ist, so sind die Zustände in den verbleibenden Äquivalenzklassen zueinander äquivalent. Genau wie im Fall der Akzeptorenminimierung können wir den reduzierten Transduktor konstruieren, indem wir für jede Äquivalenzklasse einen separaten Zustand erzeugen.

Übergangstabelle

	s_0	s_1	s_2	s_3	s_4	s_5	s_6
0	$s_2, 0$	$s_4, 1$	$s_6, 0$	$s_0, 1$	$s_0, 0$	$s_0, 1$	$s_0, 0$
1	$s_1, 1$	$s_3, 0$	$s_5, 1$	$s_0, 0$	$s_0, 1$	$s_0, 0$	$s_0, 1$

Erste Partition

	s_0	s_2	s_4	s_6	s_1	s_3	s_5
	P_1				P_2		
0	s_2, P_1	s_6, P_1	s_0, P_1	s_0, P_1	s_4, P_1	s_0, P_1	s_0, P_1
1	s_1, P_2	s_5, P_2	s_0, P_1	s_0, P_1	s_3, P_2	s_0, P_1	s_0, P_1

Zweite Partition

	s_0	s_2	s_4	s_6	s_1	s_3	s_5
	P_1		P_2		P_3		P_4
0	s_2, P_1	s_6, P_2	s_0, P_1	s_0, P_1	s_4, P_2	s_0, P_1	s_0, P_1
1	s_1, P_3	s_5, P_4	s_0, P_1	s_0, P_1	s_3, P_4	s_0, P_1	s_0, P_1

Dritte Partition

	s_0	s_2	s_4	s_6	s_1	s_3	s_5
	P_1	P_2	P_3	P_4	P_4	P_5	
0	s_2, P_2	s_6, P_3	s_0, P_1	s_0, P_1	s_4, P_3	s_0, P_1	s_0, P_1
1	s_1, P_4	s_5, P_5	s_0, P_1	s_0, P_1	s_3, P_5	s_0, P_1	s_0, P_1

Die Äquivalenz zweier Transduktoren ist genau dann gegeben, wenn ihre Startzustände zueinander bisimulativ sind. Für die Konstruktion eines reduzierten Automaten gehen wir wie im Fall des endlichen Akzeptors vor. Im ersten Schritt stellen wir die Übergangstabelle auf und teilen die Zustände anschließend in verschiedene Äquivalenzklassen ein. Zunächst fassen wir diejenigen Zustände zusammen, die für alle Eingabezeichen σ das gleiche Ausgabezeichen erzeugen. Auf diese Weise erhalten wir alle 0-äquivalenten Zustände. Anschließend werden die Äquivalenzklassen so verfeinert, dass im k -ten Iterationsschritt alle Zustände einer Äquivalenzklasse zueinander k -äquivalent sind. Wiederholen wir den Prozess, bis keine neuen Partitionen entstehen, so sind alle äquivalenten Zustände bestimmt. Auch hier konstruieren wir am Ende den reduzierten Automaten, indem wir aus jeder Äquivalenzklasse einen beliebigen Repräsentanten entnehmen und die Zustände gemäß den berechneten Übergängen miteinander verbinden.

Abbildung 5.34 demonstriert die Reduktion am Beispiel des seriellen Gray-Code-Wandlers aus Abbildung 5.32. Nach der dritten Iteration ist ein Fixpunkt erreicht und mit den fünf gefundenen Äquivalenzklassen die finale Partition der Zustandsmenge bestimmt. Das Ergebnis zeigt, dass die Zustände s_4 und s_6 sowie die Zustände s_3 und s_5 zueinander bisimulativ sind und miteinander verschmolzen werden können. Insgesamt ist es uns damit gelungen, die Anzahl der Zustände von 7 auf 5 zu verringern (Abbildung 5.35).

5.6.3 Automatensynthese

Transduktoren sind von großer praktischer Bedeutung, da sie eins zu eins in eine Hardware-Schaltung – ein sogenanntes *Schaltwerk* – übersetzt werden können. Hierzu wird zunächst eine binäre Codierung der Zustände bestimmt, d. h., jedem Zustand $s_i \in S$ wird ein Bitvektor der Länge k zugeordnet, der s_i eindeutig charakterisiert. Da sich mit Bitvektoren der Länge k genau 2^k Zustände unterscheiden lassen, gilt im Umkehrschluss die Beziehung $k = \lceil \log_2 |S| \rceil$. Mit anderen Worten: Die Anzahl der Bits, die für die Codierung benötigt werden, wächst logarithmisch mit der Anzahl der Zustände des Transduktors.

Exemplarisch ist in Abbildung 5.36 eine der vielen Möglichkeiten dargestellt, um die Zustände des minimierten Binär-Gray-Code-Wandlers zu codieren. Auch wenn die Zuordnung der Bitvektoren zu den einzelnen Zuständen im Prinzip willkürlich erfolgen kann, hat sie in der Praxis einen erheblichen Einfluss auf die Eigenschaften der entstehenden Hardware-Schaltung. Genau wie die geschickte Wahl der Zustandscodierung zu einer sehr kompakten Schaltung führen kann, lässt eine ungeschickte Wahl in vielen Fällen eine unnötig komplexe Schaltung entstehen. Das Auffinden einer geeigneten Zustandscodierung ist ein wichtiger Arbeitsschritt im computergestützten Schaltungsentwurf, der in der Praxis mithilfe spezieller Software-Werkzeuge durchgeführt wird.

Ist eine binäre Zustandscodierung bestimmt, werden die Automatenzustände auf die reale Hardware-Schaltung abgebildet. Hierzu wird für jedes Bit der Zustandscodierung ein binäres Speicherelement (*Flipflop* oder *Latch*) in die Schaltung eingesetzt und die Übergangsfunktion δ und die Ausgabefunktion λ werden in zwei konventionelle *Schaltnetze* übersetzt. Das *Übergangsschaltnetz* berechnet aus dem aktuellen Zustand und der aktuellen Eingabe die Steuersignale, mit denen wir die Speicherbausteine beschalten müssen, um sie im nächsten Takt in den korrekten Folgezustand zu bringen. Das *Ausgabeschaltnetz* berechnet die aktuelle Ausgabe des Schaltwerks. Die Ein- und Ausgabe der

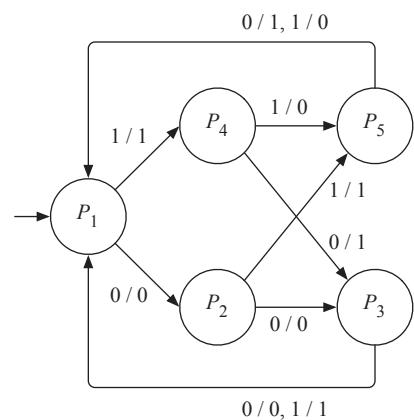


Abbildung 5.35: Serieller Binär-Gray-Code-Wandler. Ergebnis nach der Automatenminimierung.

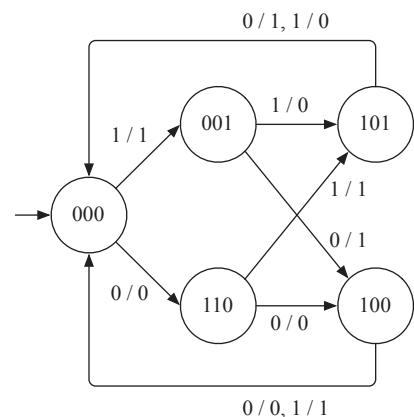


Abbildung 5.36: Zustandscodierter Automat des seriellen Gray-Code-Wandlers.

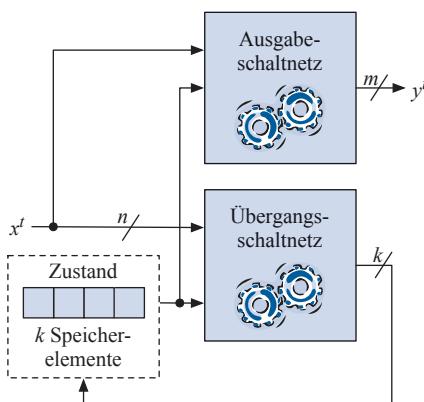


Abbildung 5.37: Jeder Transduktor lässt sich eins zu eins in ein Schaltwerk in Huffman-Normalform übersetzen.

Hardware-Schaltung wird durch Bitvektoren der Länge n bzw. m gebildet. Zwischen einem Schaltwerk mit n Eingangsleitungen und m Ausgangsleitungen und dem binär codierten Automaten besteht damit der folgende Zusammenhang:

$$\Sigma = \{0,1\}^n, \quad \Pi = \{0,1\}^m$$

Führen wir die Ausgänge des Ausgabeschaltnetzes aus der Schaltung heraus und die Ausgänge des Übergangsschaltnetzes zurück auf die Eingänge der Speicherelemente, so entsteht ein Schaltwerk in *Huffman-Normalform*. In Abbildung 5.37 ist dessen Grobstruktur grafisch zusammengefasst.

Abbildung 5.38 zeigt die resultierende Hardware-Schaltung des seriellen Gray-Code-Wandlers in Huffman-Normalform. Eine detaillierte Beschreibung, wie sich das Übergangs- und das Ausgabeschaltnetz aus der codierten Automatenbeschreibung ableiten lässt, wird in [49] gegeben.

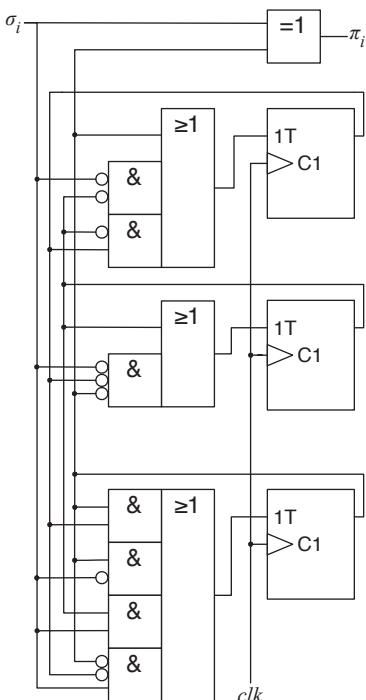


Abbildung 5.38: Implementierung des seriellen Gray-Code-Wandlers in Huffman-Normalform

5.6.4 Mealy- und Moore-Automaten

In Definition 5.13 haben wir festgelegt, dass sowohl der eingenommene Zustand als auch das gelesene Eingabezeichen in die Berechnung des aktuellen Ausgabezeichens $\lambda(s_i, \sigma_i)$ mit einbezogen wird. Einige Automaten nutzen diese Flexibilität nur unvollständig aus und berechnen die aktuelle Ausgabe ausschließlich aus dem gegenwärtig eingenommenen Zustand. Automaten mit dieser Eigenschaft sind in der Praxis von so großer Bedeutung, dass sie in der Informatik einen eigenen Namen erhalten haben.

Definition 5.15 (Mealy- und Moore-Automat)

Gegeben sei ein beliebiger endlicher Automat $(S, \Sigma, \Pi, \delta, \lambda, s_0)$. Geht in die Berechnung des Ausgabezeichens sowohl der aktuelle Zustand als auch das aktuelle Eingabezeichen ein, gilt also

$$\pi_i = \lambda(s_i, \sigma_i),$$

so sprechen wir von einem *Mealy-Automaten*. Ist die Ausgabefunktion stattdessen nur vom aktuellen Zustand abhängig, gilt also

$$\pi_i = \lambda(s_i), \tag{5.17}$$

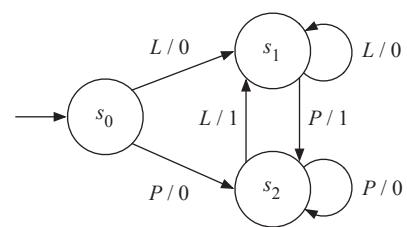
so sprechen wir von einem *Moore-Automaten*.

Abbildung 5.39 demonstriert den Unterschied zwischen Mealy- und Moore-Automaten anhand zweier Beispiele. Die Übergangstabelle des Mealy-Automaten zeigt, dass die Ausgabe in s_1 und s_2 sowohl von dem eingenommenen Zustand als auch von dem aktuell gelesenen Eingabezeichen abhängt. Im Falle des Moore-Automaten spielt das aktuell gelesene Eingabezeichen dagegen keine Rolle; hier wird die Ausgabe ausschließlich durch den eingenommenen Zustand bestimmt. Wegen dieser Eigenschaft werden Moore-Automaten typischerweise in einer Notation angegeben, die das Ausgabezeichen nicht mehr länger den Kanten, sondern den Zuständen zuordnet. Abbildung 5.39 zeigt den Moore-Automaten in beiden Darstellungen. Beachten Sie, dass der Automatentyp nicht zweifelsfrei aus der verwendeten Notation abgeleitet werden kann. Ein Moore-Automat liegt vor, wenn die Ausgabefunktion nur von dem eingenommenen Zustand abhängt, unabhängig davon, ob die Ausgabezeichen in Mealy-typischer Notation an den Kanten oder in Moore-typischer Notation an den Zuständen vermerkt sind.

Liegt ein Schaltwerk in Huffman-Normalform vor, so lässt sich mit einem einzigen Blick erkennen, ob es aus einem Mealy- oder einem Moore-Automaten synthetisiert wurde. Per Definition berechnet sich die Ausgabe eines Mealy-Automaten aus dem aktuellen Zustand und der aktuellen Eingabe des Schaltwerks. In der Huffman-Normalform führen damit neben den Ausgängen der Speicherelemente auch eine oder mehrere Eingangsleitungen in das Schaltnetz zur Ausgabeberechnung (vgl. Abbildung 5.40 links). Im Gegensatz hierzu hängt die Ausgabe bei Moore-Automaten ausschließlich vom aktuellen Zustand des Schaltwerks ab. In der Huffman-Normalform drückt sich die Moore-Eigenschaft dadurch aus, dass keine direkte Verbindung zwischen den Eingangssignalen und den Eingängen des Ausgabeschaltnetzes mehr existiert (vgl. Abbildung 5.40 rechts).

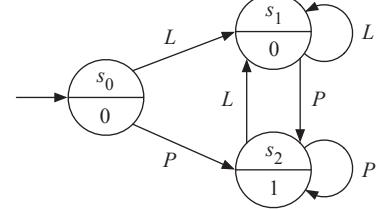
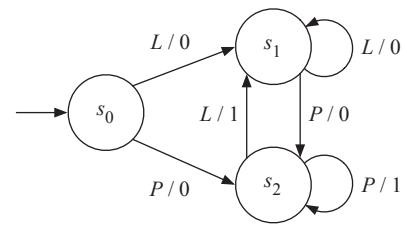
An dieser Stelle drängt sich die Frage auf, ob Mealy- und Moore-Transduktoren ein voneinander verschiedenes Automatenmodell beschreiben oder doch vielleicht zueinander äquivalent sind. Ein kurzer Blick auf die Definitionen 5.13 und 5.15 zeigt, dass wir jeden Moore-Automaten als einen speziellen Mealy-Automaten auffassen können. Die weitaus interessantere Frage ist die, ob wir jeden Mealy-Automaten in einen äquivalenten Moore-Automaten umformen können. In der hier definierten Form ist eine solche Reduktion nicht ohne weiteres möglich, da die Ausgabe eines Mealy-Automaten zum Zeitpunkt i sowohl von dem eingenommenen Zustand s_i als auch der aktuellen Eingabe σ_i abhängt. Ein Moore-Automat kann zur Ausgabeberechnung nur den Zustand σ_i auswerten und hat zum Zeitpunkt i noch kein Wissen von der aktuellen Eingabe σ_i .

■ Mealy-Automat



s	s_0	s_1	s_2
$\delta(s, L)$	s_1	s_1	s_1
$\delta(s, P)$	s_2	s_2	s_2
$\lambda(s, L)$	0	0	1
$\lambda(s, P)$	0	1	0

■ Moore-Automat



s	s_0	s_1	s_2
$\delta(s, L)$	s_1	s_1	s_1
$\delta(s, P)$	s_2	s_2	s_2
$\lambda(s)$	0	0	1

Abbildung 5.39: Mealy- und Moore-Automaten im Vergleich

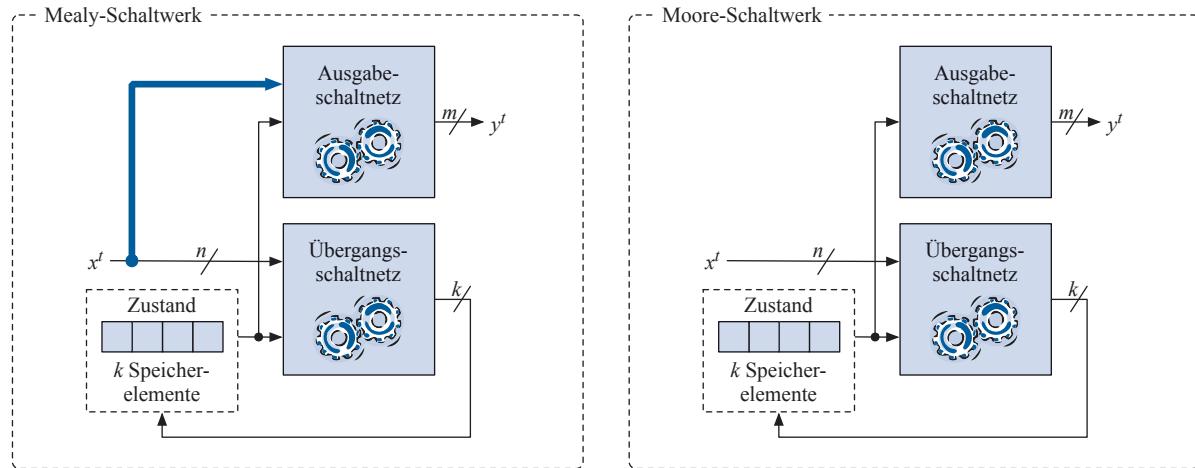


Abbildung 5.40: An der Huffman-Normalform eines Schaltwerks lässt sich besonders einfach erkennen, ob ein Mealy- oder ein Moore-Automat implementiert wird. Während das Ausgabeschaltnetz in Mealy-Schaltwerken zusätzliche Verbindungen zu den Eingangsleitungen aufweist, wird es in Moore-Schaltwerken ausschließlich durch die Leitungen des Speicherblocks gespeist.

Nichtsdestotrotz ist es möglich, für jeden Mealy-Automaten einen Moore-Automaten zu konstruieren, wenn wir das eingeführte Automatenmodell geringfügig ändern. Die Grundidee besteht darin, die produzierte Zeichensequenz um einen Schritt verzögert auf das Ausgabeband zu schreiben. Ein so modifizierter Transduktoren beginnt wie gehabt im Startzustand s_0 . Wird er mit dem Eingabewort

$$\omega = \sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n$$

stimuliert, so durchläuft er die Zustände

$$s_0, s_1, s_2, \dots, s_n \quad \text{mit } s_{i+1} = \delta(s_i, \sigma_i)$$

und produziert die Ausgabe

$$\pi_1, \pi_2, \dots, \pi_{n+1} \quad \text{mit } \pi_i = \lambda(s_i, \sigma_{i-1}). \quad (5.18)$$

Das modifizierte Automatenmodell unterscheidet sich in zwei wesentlichen Punkten von dem bisherigen. Zum einen produziert der Transduktoren zum Zeitpunkt 0 keine Ausgabe; das erste Zeichen wird zum Zeitpunkt 1 auf das Ausgabeband geschrieben, das letzte zum Zeitpunkt $n + 1$. Zum anderen berechnet sich die Ausgabe aus dem eingenommenen Zustand und dem *zuvor* gelesenen Eingabezeichen. Durch diese Verschiebung wird es möglich, das Eingabezeichen in den eingenommenen Zustand hineinzucodieren und die Ausgabefunktion von der Mealy-Form in die Moore-Form zu überführen.

Abbildung 5.41 demonstriert die Umcodierung an einem konkreten Beispiel. Im ersten Schritt werden die Ausgabezeichen aus den Kantenmarkierungen herausgelöst und den Folgezuständen zugeordnet (vgl. Abbildung 5.41 Mitte). Probleme bereiten uns diejenigen Zustände, die über zwei oder mehr eingehende Kanten verfügen, die mit einem unterschiedlichen Ausgabezeichen markiert sind. Im zweiten Schritt werden diese Konflikt aufgelöst, indem die betreffenden Zustände aufgespalten und die Kanten auf diejenigen Zustände mit der passenden Ausgabe umgeleitet werden (vgl. Abbildung 5.41 unten). Die Umwandlung zeigt, dass die vereinfachte Ausgabeberechnung ihren Preis in einer Erhöhung der Zustandsanzahl fordert. Besitzt das Ausgabealphabet n Zeichen, so kann der konstruierte Moore-Automat bis zu n mal mehr Zustände besitzen als der Mealy-Automat.

Unsere Betrachtungen werfen die Frage auf, welches der vorgestellten Automatenmodelle dem anderen überlegen ist. Wie so oft hängt auch hier die Antwort vom Standpunkt des Betrachters ab. Legen wir eine praktische Sicht zugrunde, so scheint das ursprünglich eingeführte Automatenmodell das natürlichere zu sein. In diesem Fall entspricht das Ein- und Ausgabeverhalten eines Mealy- bzw. eines Moore-Automaten exakt dem einer Hardware-Schaltung, so dass wir den Begriff des Transduktors ohne Übertreibung als das theoretische Fundament der digitalen Schaltungstechnik ansehen können. Legen wir stattdessen eine theoretische Sicht zugrunde, so wirkt das modifizierte Ausgabeverhalten nach Gleichung (5.18) als das verführerische. Es besitzt den Vorteil, dass sich Mealy- und Moore-Automaten als äquivalent erweisen, was in der praxisorientierten Variante nicht der Fall ist.

In der Literatur wird der Mealy- und der Moore-Begriff aus den geschilderten Gründen unterschiedlich eingeführt und in vielen Fällen – bewusst oder unbewusst – von der ursprünglichen Definition von George H. Mealy [70] und Edward F. Moore [75] abgewichen. Auch die hier getroffene Definition des Moore-Automaten unterscheidet sich von jener aus der Originalarbeit. Wir folgen hier bewusst der praxisorientierten Variante, die eine direkte Entsprechung im Hardware-Entwurf findet.

Aus dem Gesagten wird eines ganz klar: Die Interpretation der Begriffe Mealy und Moore muss stets mit Bedacht befolgen und kann zu ganz unterschiedlichen Ergebnissen führen. Allen in der Literatur angekommenen Definitionen ist lediglich gemein, dass der Begriff Mealy ein Automatenmodell beschreibt, das die Ausgabe aus Zustand und Eingabe berechnet, während der Begriff Moore ein Automatenmodell meint, das für die Ausgabeberechnung ausschließlich den Zustand in Betracht zieht.

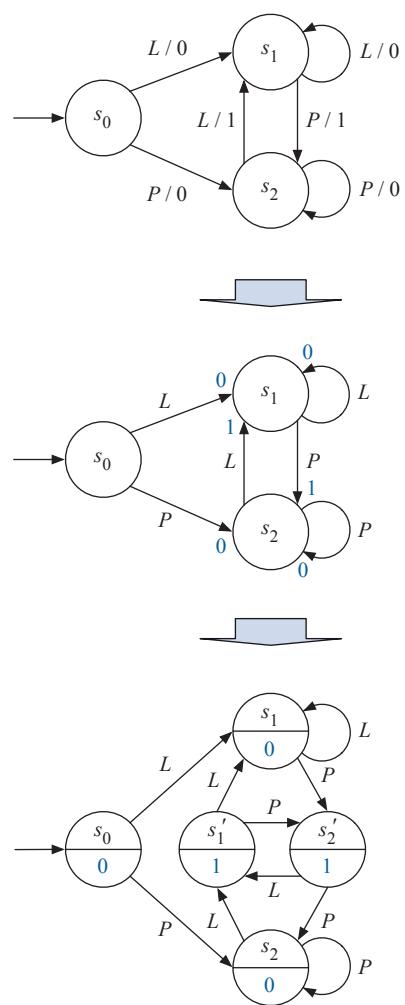
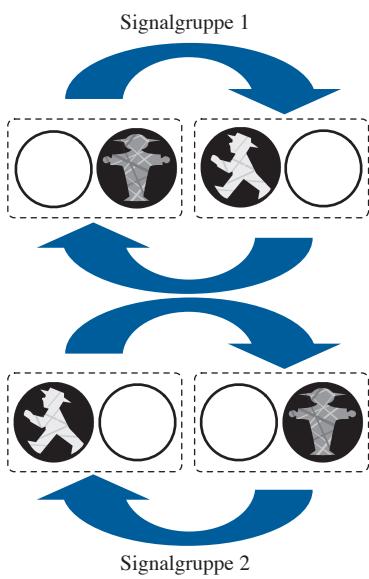


Abbildung 5.41: Moore-Konversion



„Zu keinem Zeitpunkt dürfen beide Signalgruppen ein grünes Licht zeigen.“

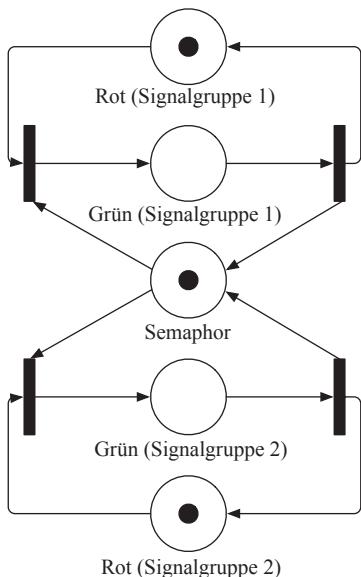


Abbildung 5.42: Modellierung einer Ampelsteuerung mithilfe eines Petri-Netzes

5.7 Petri-Netze

In den vorangegangenen Abschnitten haben wir mit dem endlichen Automaten ein mächtiges Werkzeug zur Modellierung zustandsbasierter Systeme kennen gelernt. Nichtsdestotrotz existieren Anwendungen, die mit den eingeführten Formalismen nur schwer zu erfassen sind. Beispiele sind die Beschreibung kausaler Zusammenhänge sowie die Modellierung von Nebenläufigkeit. Die entstehende Lücke wird durch *Petri-Netze* geschlossen, die genau wie endliche Automaten zustandsbasiert arbeiten, jedoch über deutlich komplexere Übergangsmechanismen verfügen. Der Name Petri-Netz geht auf den deutschen Mathematiker Carl Adam Petri zurück, der diesen Automatentypus Anfang der Sechzigerjahre ausführlich untersuchte [81].

Petri-Netze unterscheiden zwischen *Bedingungen* und *Ereignissen*. Erstere werden durch *Stellen*, letztere durch *Transitionen* beschrieben. Abbildung 5.42 zeigt ein Beispielnetz, dargestellt in der für Petri-Netze typischen Graphennotation. In dieser werden Stellen durch Kreise und Transitionen durch einen Balken, in manchen Fällen auch durch ein ungefülltes Rechteck, repräsentiert. Die Abhängigkeiten, die zwischen Stellen und Transitionen bestehen, werden durch Kanten modelliert. Jede Kante verbindet dabei eine Stelle mit einer Transition oder eine Transition mit einer Stelle, nie jedoch eine Transition mit einer Transition oder eine Stelle mit einer Stelle. Alle Kanten eines Petri-Netzes sind gerichtet, so dass jeder Transition eindeutig eine Eingabe- und eine Ausgabestelle zugeordnet werden kann.

In einem Petri-Netz wird der aktuelle Zustand eines Systems durch *Marken* modelliert. Diese werden den Stellen zugeordnet und in der Graphdarstellung durch Punkte repräsentiert. Die Marken sind gewissermaßen das Elixier, das ein Petri-Netz zum Leben erweckt. Ihre Verteilung bestimmt, ob sich eine Transition in einem *aktivierten*, d. h. schaltbereiten Zustand befindet. Konkret ist eine Transition immer dann aktiviert, wenn alle Eingabestellen mindestens eine Marke enthalten. Beachten Sie, dass eine aktivierte Transition einen Schaltvorgang auslösen *kann*, aber nicht *muss*. Schaltet eine Transition, so wird eine Marke aus jeder Eingabestelle entfernt und jeder Ausgangsstelle eine zusätzliche Marke hinzugefügt. Da die Anzahl der Eingabestellen einer Transition von der Anzahl der Ausgangsstellen abweichen kann, können durch den Schaltvorgang neue Marken entstehen oder bestehende vernichtet werden.

Abbildung 5.42 demonstriert das Gesagte am Beispiel einer primitiven Ampelsteuerung. Das Petri-Netz modelliert zwei Signalgruppen, die

nacheinander eine Rot- und eine Grünphase durchlaufen. Die Umschaltvorgänge der Signalgruppen können asynchron erfolgen, müssen aber jederzeit die Bedingung erfüllen, dass niemals beide Ampeln gleichzeitig Grün zeigen. Im Petri-Netz wird die Abhängigkeit durch einen Semaphor in Form einer separaten Stelle modelliert. Eine Signalgruppe kann nur dann in die Grünphase wechseln, wenn der Semaphor eine Marke enthält. Durch den Schaltvorgang wird sie entfernt und erst beim erneuten Eintreten in die Rotphase zurückgeschrieben. Durch die temporäre Wegnahme ist sichergestellt, dass die zweite Signalgruppe erst dann in die Grünphase eintreten kann, wenn sich die erste wieder in der Rotphase befindet.

Das Petri-Netz zur Ampelsteuerung verwendet typische Grundmuster zur Modellierung des Systemverhaltens. Zum einen wird der periodische Wechsel zwischen der Rot- und der Grünphase durch in Kette geschaltete Stellen realisiert. Zum anderen verwendet das Netz eine zusätzliche Stelle als Semaphor, der die unabhängig voneinander arbeitenden Signalgruppen synchronisiert. Die wichtigsten Grundmuster sind in Abbildung 5.43 in einer Übersicht zusammengefasst.

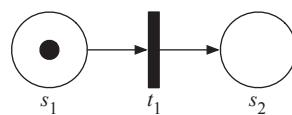
Konfigurationen

Die Anzahl und die Verteilung der Marken eines Petri-Netzes definieren eine *Konfiguration*. Für ein Netz mit n Stellen können wir diese formal als Vektor $\kappa \in \mathbb{N}^n$ auffassen, dessen i -te Komponente angibt, wie viele Marken in der i -ten Stelle vorhanden sind.

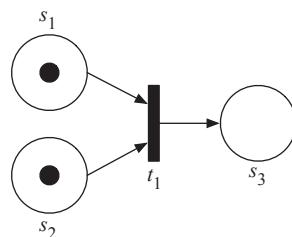
Die Vektordarstellung versetzt uns in die Lage, Petri-Netze mit den Mitteln der linearen Algebra zu modellieren und zu analysieren. Zu diesem Zweck trennen wir uns vorübergehend von der Graphendarstellung und übersetzen die Struktur eines Petri-Netzes in eine sogenannte *Inzidenzmatrix I*. Diese enthält für jede Stelle s_i eine separate Zeile und für jede Transition t_j eine separate Spalte. Der Wert des Elements (i, j) beschreibt, wie sich die Anzahl der Marken in der Stelle s_i ändert, wenn die Transition t_j schaltet. Positive Werte bedeuten, dass Marken hinzugefügt, negative Werte, dass Marken entfernt werden. Bleibt die Anzahl der Marken in einer Stelle unverändert, so enthält die Inzidenzmatrix an der entsprechenden Stelle den Wert 0.

Jede Sequenz von nacheinander schaltenden Transitionen lässt sich ebenfalls in eine Vektordarstellung übersetzen. Hierzu ordnen wir die i -te Vektorzeile der i -ten Transition zu und notieren, wie oft diese in der betrachteten Sequenz schaltet. Der entstehende Vektor wird als *Parikh-Vektor* Ψ bezeichnet. Beachten Sie, dass die Reihenfolge, in der die ein-

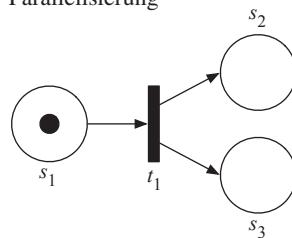
Kettenbildung



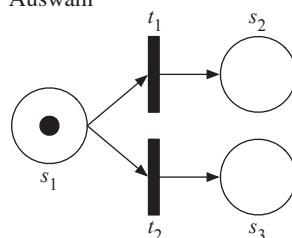
Synchronisation



Parallelisierung



Auswahl



Schleife

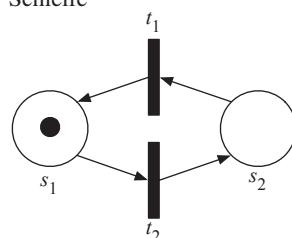
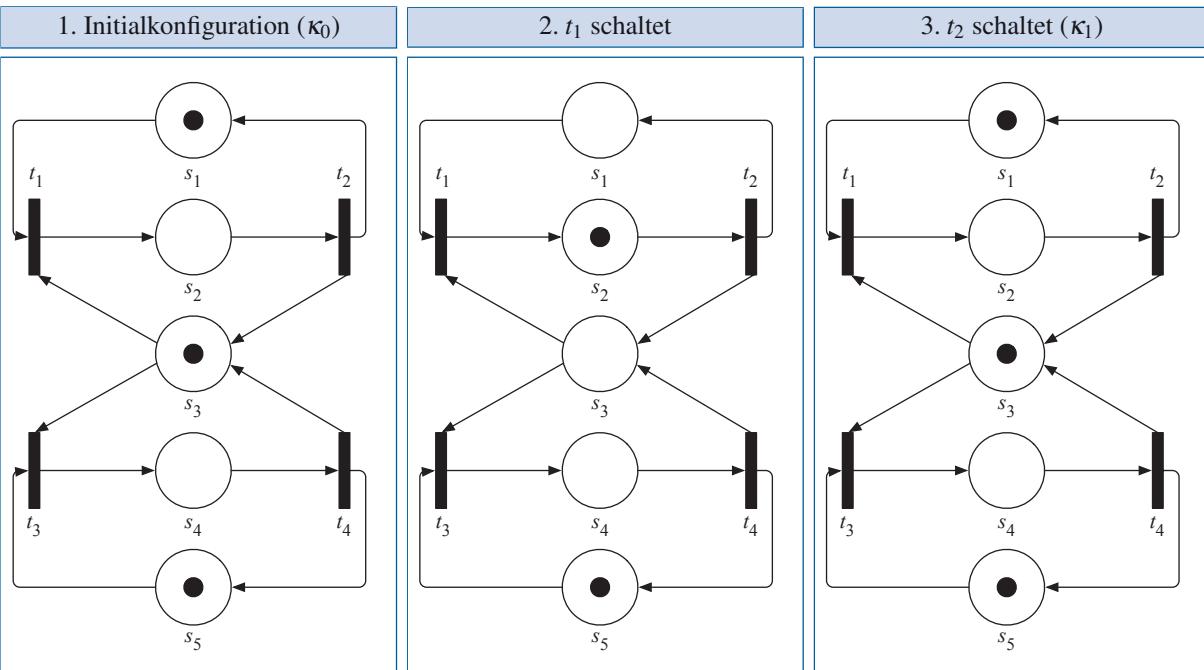


Abbildung 5.43: Petri-Netz-Topologien in der Übersicht



$$\underbrace{\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}}_{\kappa_0} + \underbrace{\begin{pmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix}}_{t_1 \text{ schaltet}} \cdot \underbrace{\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{\kappa_0} + \underbrace{\begin{pmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix}}_{t_2 \text{ schaltet}} \cdot \underbrace{\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}}_{\kappa_1} = \underbrace{\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}}_{\kappa_1}$$

Abbildung 5.44: Vollständiger Umschaltzyklus der ersten Signalgruppe

zellen Transitionen schalten, nicht aus der Vektordarstellung abgeleitet werden kann; sie gibt ausschließlich darüber Auskunft, *wie oft* eine Transition schaltet. Nichtsdestotrotz ist der Parikh-Vektor von großem Nutzen, da wir durch die Multiplikation mit der Inzidenzmatrix die Folgekonfiguration eines Petri-Netzes berechnen können. Bezeichnen wir die aktuelle Konfiguration mit κ und die Folgekonfiguration mit κ' , so gilt die folgende Beziehung:

$$\kappa' = \kappa + I \cdot \Psi \quad (5.19)$$

Abbildung 5.44 demonstriert das Gesagte am Beispiel der weiter oben eingeführten Ampelsteuerung. Dargestellt ist ein kompletter Umschaltzyklus für die erste Signalgruppe. Ausgehend von der Initialkonfigura-

tion schaltet zuerst die Transition t_1 . Hierbei werden die Marken in den Stellen s_1 und s_3 gelöscht und eine neue Marke in s_2 erzeugt. Anschließend schaltet die Transition t_2 und versetzt das Petri-Netz zurück in die Anfangskonfiguration.

Gleichung (5.19) lässt sich in direkter Weise für die *Erreichbarkeitsanalyse* eines Petri-Netzes einsetzen. Wir nennen eine Konfiguration κ_1 erreichbar, wenn eine Sequenz von schaltenden Transitionen existiert, mit der die Anfangskonfiguration κ_0 in κ_1 überführt wird. Falls eine solche Sequenz existiert, besitzt die *Markierungsgleichung*

$$I \cdot \Psi = \kappa_1 - \kappa_0$$

eine Lösung in den natürlichen Zahlen. Beachten Sie, dass die Umkehrung dieser Aussage nicht gilt, d. h., nicht jede natürliche Zahlige Lösung lässt sich auch wirklich in eine entsprechende Transitionssequenz umsetzen. Als Beispiel betrachten wir das Petri-Netz in Abbildung 5.46. Die Lösbarkeit der Markierungsgleichung suggeriert, dass sich eine Konfiguration erreichen lässt, in der ausschließlich die Stelle s_3 mit einer Marke befüllt ist. In Wirklichkeit lässt sich diese Konfiguration nicht herstellen. Die unmarkierte Stelle s_2 sorgt dafür, dass die Transition t_1 niemals schalten kann.

Erreichbarkeitsuntersuchungen gehören zu den wichtigsten Fragestellungen, die mithilfe von Petri-Netzen beantwortet werden können. Neben diesen werden Petri-Netze zur Analyse der folgenden Systemeigenschaften genutzt:

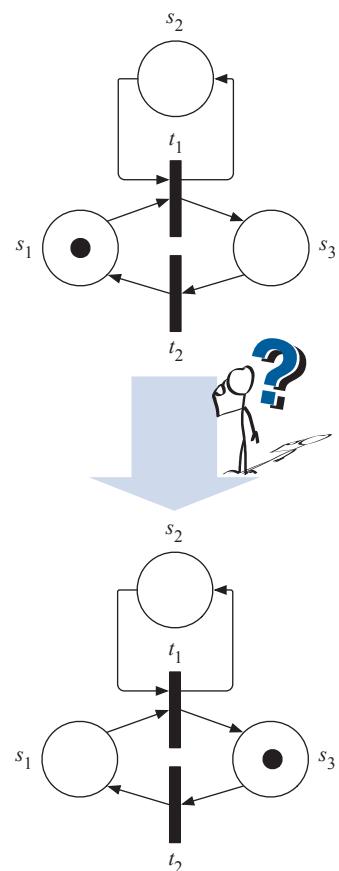
■ Lebendigkeit

Ein Petri-Netz heißt *lebendig*, wenn es für jede Transition t die Eigenschaft erfüllt, dass sich aus jeder erreichbaren Konfiguration eine andere Konfiguration erzeugen lässt, in der t aktiviert ist. Tabelle 5.45 (links) zeigt ein Petri-Netz, das die Lebendigkeitseigenschaft nicht erfüllt. Die Transition t_3 ist bereits aus der Startkonfiguration nicht aktivierbar, da die Stellen s_1 und s_2 nur wechselweise eine Marke enthalten. Damit t_3 schalten kann, müssten jedoch beide Stellen zur selben Zeit eine Marke enthalten.

■ Sicherheit

Ein Petri-Netz heißt *sicher*, wenn die Anzahl der Markierungen in jeder Stelle s einen gewissen Wert $C(s)$ nicht übersteigt. $C(s)$ heißt die *Kapazität* von s . Das mittlere der drei in Tabelle 5.45 dargestellten Petri-Netze ist unsicher. In diesem Beispiel werden die Transitionen t_1 und t_2 abwechselnd aktiviert und immer dann, wenn t_2 schaltet, eine zusätzliche Marke in der Stelle s_3 erzeugt. Die Anzahl

■ Konfigurationsübergang



■ Markierungsgleichung

$$\begin{pmatrix} -1 & 1 \\ 0 & 0 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$$

Abbildung 5.46: Die Lösbarkeit der Markierungsgleichung ist ein notwendiges, aber kein hinreichendes Kriterium für die Erreichbarkeit einer Konfiguration. Obwohl die Markierungsgleichung in diesem Beispiel eine Lösung in den natürlichen Zahlen besitzt, ist der dargestellte Konfigurationsübergang nicht möglich.

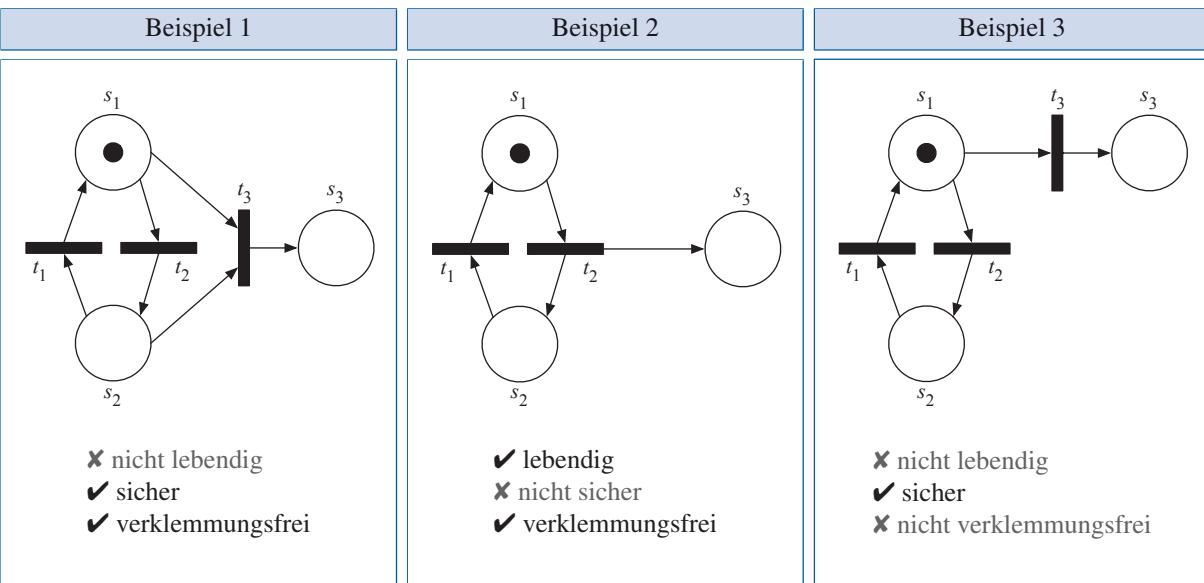


Abbildung 5.45: Petri-Netz-Eigenschaften

der Marken nimmt hierdurch kontinuierlich zu und ist damit nach oben unbeschränkt.

■ Verklemmungsfreiheit

Ein Petri-Netz ist *verklemmungsfrei*, wenn zu jeder Zeit mindestens eine Transition aktiviert ist. Ist ein Petri-Netz nicht verklemmungsfrei, so lässt sich aus der Startmarkierung eine Konfiguration erzeugen, in der keine aktivierte Transitionen existieren. Es entsteht ein Systemstillstand (*Deadlock*). Innerhalb des rechten Petri-Netzes in Tabelle 5.45 lässt sich eine solche Situation leicht herbeiführen. Sobald die Transition t_3 schaltet, lässt sich weder t_1 , t_2 noch t_3 erneut aktivieren.

Die Lebendigkeit und die Verklemmungsfreiheit sind keine vollständig unabhängigen Systemeigenschaften. So ist jedes lebendige Petri-Netz immer auch verklemmungsfrei. Die Umkehrung dieser Schlussfolgerung gilt nicht, wie wir am Beispiel des linken Petri-Netzes in Abbildung 5.45 demonstriert haben. Obwohl das Netz die Lebendigkeiteigenschaft nicht erfüllt, kommt es zu keinem Deadlock, da zu jedem Zeitpunkt eine aktivierte Transition existiert.

5.8 Zelluläre Automaten

In diesem Abschnitt wollen wir einen einführenden Blick auf ein Automatenmodell werfen, das sich grundlegend von den bisher betrachteten unterscheidet. Die Rede ist von *zellulären Automaten*, deren Ursprünge bis in die Vierzigerjahre des letzten Jahrhunderts zurückreichen, in der wissenschaftlichen Literatur aber erst Jahre später aufgearbeitet wurden [78]. Eingesetzt wird das Modell vor allem zur Beschreibung dynamischer, selbstorganisierender Systeme.



Definition 5.16 (Zellulärer Automat)

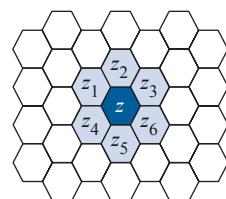
Ein zellulärer Automat (*cellular automaton*), kurz ZA, ist ein 4-Tupel (Z, S, v, δ) . Er besteht aus

- der **Zellmenge** Z ,
- der endlichen **Zustandsmenge** S ,
- der **Nachbarschaftsfunktion** $v : Z \rightarrow S^n$,
- der **Zustandsübergangsfunktion** $\delta : S \times S^n \rightarrow S$

Grob gesprochen setzt sich ein zellulärer Automat aus einer großen Menge Z von Elementarautomaten zusammen, die wir im Folgenden als *Zellen* bezeichnen. Genau wie im Falle des klassischen Automaten befindet sich eine Zelle zu jedem Zeitpunkt in einem von endlich vielen Zuständen. Die Menge der erlaubten Zustände bezeichnen wir mit S . In den folgenden Beispielen werden wir die Zustände allesamt durch verschiedene Farben darstellen, so dass S in diesen Fällen dem verfügbaren Farbvorrat entspricht.

In einem zellulären Automaten agieren die einzelnen Zellen nicht unabhängig voneinander. Ganz im Gegenteil: Sie stehen in ständiger Interaktion. Wie sich eine Zelle verhält, wird zum einen durch ihren eigenen, aktuell eingenommenen Zustand und zum anderen durch den Zustand ihrer Nachbarzellen bestimmt. Die Nachbarschaftsbeziehung eines zellulären Automaten wird durch die Funktion v bestimmt. v bildet eine Zelle z auf einen n -elementigen Vektor ab, der alle Nachbarn von z enthält. Abbildung 5.47 zeigt, dass mit v beliebige Topologien von Nachbarschaftsbeziehungen modelliert werden können; hier demonstriert am Beispiel einer hexagonalen Anordnung, in der jede Zelle von jeweils 6 Nachbarzellen umgeben wird. Am häufigsten werden jedoch die folgenden beiden Nachbarschaftsbeziehungen eingesetzt:

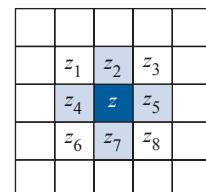
■ Hexagon-Nachbarschaft



$$v(z) = \{z_1, z_2, z_3, z_4, z_5, z_6\}$$

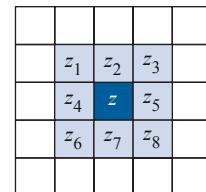
Abbildung 5.47: Mit Hilfe der Nachbarschaftsfunktion lassen sich beliebige Topologien modellieren.

■ Von-Neumann-Nachbarschaft



$$v(z) = \{z_2, z_4, z_5, z_7\}$$

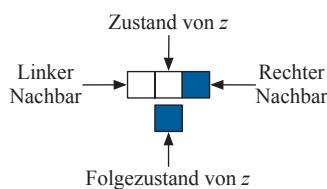
■ Moore-Nachbarschaft



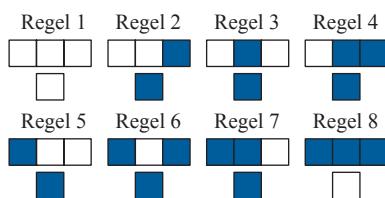
$$v(z) = \{z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8\}$$

Abbildung 5.48: Die Von-Neumann- und Moore-Nachbarschaft im Vergleich

■ Regelschema



■ Vollständiger Regelsatz



■ Initiale Konfiguration



■ Automat in Aktion

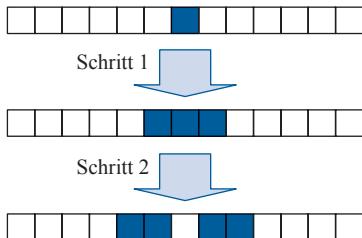


Abbildung 5.49: Lineare zelluläre Automaten basieren auf einer eindimensionalen Zell-Topologie. Der dargestellte Regelsatz definiert einen zweifarbigen Automaten. Welche Farbe eine Zelle im nächsten Rechenschritt annimmt, wird zum einen durch ihre momentane Einfärbung und zum anderen durch die Einfärbungen der beiden Nachbarzellen bestimmt.

■ Von-Neumann-Nachbarschaft

Sind die Zellen in Form von Quadraten auf einer zweidimensionalen Fläche angeordnet, so besteht eine *Von-Neumann-Nachbarschaft* genau dann, wenn sich die betrachteten Zellen eine Kante teilen. Wir sprechen in diesem Zusammenhang auch von einer *4er-Nachbarschaft* (vgl. Abbildung 5.48 oben).

■ Moore-Nachbarschaft

Eine *Moore-Nachbarschaft* wird auch als *8er-Nachbarschaft* bezeichnet. Sie besteht bereits dann, wenn sich die betrachteten Zellen eine Ecke teilen (vgl. Abbildung 5.48 unten). Die Von-Neumann-Nachbarschaft ist vollständig in der Moore-Eigenschaft erhalten.

Nachdem wir den grundsätzlichen Aufbau eines zellulären Automaten erarbeitet haben, ist es an der Zeit, das genaue Schaltverhalten zu untersuchen. Befinden sich die Nachbarzellen z_1, \dots, z_n von z in den Zuständen s_{z_1}, \dots, s_{z_n} , so lässt sich der Nachfolgezustand von z wie folgt berechnen:

$$s'_z = \delta(s_z, s_{z_1}, \dots, s_{z_n})$$

Jede Auswertung von δ entspricht der Änderung des Zustands einer einzelnen Zelle.

Den aktuellen Zustand einer Zelle z bezeichnen wir als *lokale Konfiguration*. Die Gesamtheit aller lokalen Konfigurationen heißt *globale Konfiguration*; sie definiert den Zustand, in dem sich der zelluläre Automat gegenwärtig befindet. Im Gegensatz zu den konventionellen endlichen Automaten werden für die Berechnung des Konfigurationsübergangs keine Eingabezeichen ausgewertet, so dass das zukünftige Verhalten bereits vollständig durch den gegenwärtig eingenommenen Zustand und die Zustände der Nachbarzellen festgelegt ist. In der Einbeziehung der Nachbarzellen liegt die Stärke zellulärer Automaten. Die permanente Rückkopplung macht es möglich, durch die Angabe einiger weniger, einfach aufgebauter Regeln ein komplexes, selbstorganisierendes Verhalten zu erzeugen.

Im Folgenden wollen wir mit den *linearen Automaten* eine Untergruppe der zellulären Automaten genauer untersuchen, die durch die detaillierteren Forschungsarbeiten des britischen Mathematikers Stephen Wolfram einen hohen Bekanntheitsgrad erlangen konnte [108]. Hierbei handelt es sich um zelluläre Automaten mit einer eindimensionalen Topologie, in der wir uns alle Zellen wie auf einer Schnur aufgereiht vorstellen können. Zwei Zellen gelten als benachbart, wenn diese unmittelbar aneinandergrenzen. In linearen Automaten liegt sowohl eine Von-Neumann-

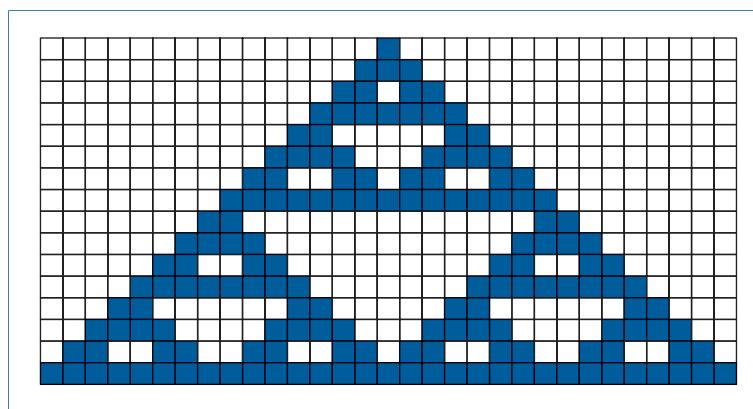


Abbildung 5.50: Das Sierpinski-Dreieck, erzeugt durch einen zellulären Automaten. Im Bereich der fraktalen Geometrie wird es häufig verwendet, um das Prinzip der Selbstähnlichkeit zu demonstrieren. Grob gesprochen ist ein Objekt genau dann selbstähnlich, wenn es als Ganzes die gleiche Struktur aufweist wie seine Teile. Am Beispiel des Sierpinski-Dreiecks lässt sich die Eigenschaft gut erkennen. Trennen wir eines der drei Teil-dreiecke heraus, so erhalten wir erneut ein Sierpinski-Dreieck, das in seiner Struktur dem ursprünglichen ähnlich ist.

als auch eine Moore-Nachbarschaft vor; beide Konzepte sind im eindimensionalen Raum identisch.

Abbildung 5.49 zeigt einen linearen Automaten mit zwei Zuständen. Wie oben angedeutet, codieren wir die Zustände durch das Einfärben der Zellen und bezeichnen einen Automaten dieser Bauart auch als *zweifarbig*. $\delta(s_z, s_{z_l}, s_{z_r})$ ist die Folgefärbung des Zustands z und berechnet sich aus der aktuellen Farbe s_z von z sowie den Farben s_{z_l} und s_{z_r} seiner beiden Nachbarzellen z_l und z_r .

Wie sich der Automat in Aktion verhält, lässt sich in der unteren Hälfte von Abbildung 5.49 beobachten. Zunächst legen wir die *Startkonfiguration* fest. In unserem Beispiel sieht diese so aus, dass alle Zellen bis auf eine weiß eingefärbt sind. Auf der initialen Konfiguration startend, führt der zelluläre Automat jetzt sukzessiv eine Folge von Arbeitsschritten aus. In jedem Schritt gehen alle Zellen synchron in ihren Folgezustand über, so dass an allen Positionen ein potenzieller Farbwechsel stattfindet. Tragen wir die nacheinander berechneten Konfigurationen untereinander auf, so entsteht ein zweidimensionales Bild, in dem die vertikale Achse die Zeitachse bildet und jeder horizontale Schnitt einer einzelnen Konfiguration entspricht.

Abbildung 5.50 klärt auf, welches Bild der weiter oben definierte Beispielautomat erzeugt. Die gewählten Produktionsregeln lassen das sogenannte *Sierpinski-Dreieck* entstehen – eine heute gut untersuchte Struktur aus dem Bereich der fraktalen Geometrie [67, 91].

In Kapitel 6 werden wir den zellulären Automaten erneut begegnen. Dort werden wir zeigen, dass dieser Automatentypus stark genug ist, um mit wenigen Modifikationen eine Turing-Maschine zu simulieren.

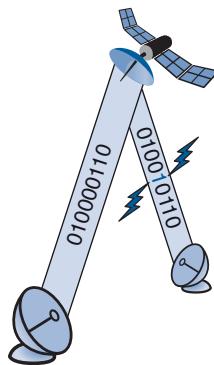
5.9 Übungsaufgaben

Aufgabe 5.1



**Webcode
5237**

Im Bereich der fehlererkennenden Datenübertragung spielt der *Paritätscode* eine wichtige Rolle. Die Codierung verfolgt die Idee, ausschließlich Datenpakete zu versenden, die eine gerade Anzahl Einsen aufweisen. Hierzu werden die Datenpakete vor dem Versenden um ein *Paritätsbit* ergänzt, das die Gesamtzahl der Einsen gerade werden lässt. Die folgende Tabelle listet exemplarisch alle Codewörter des 5-Bit-Paritätscodes auf. x_0, \dots, x_3 entsprechen den Datenbits und p dem künstlich hinzugefügten Paritätsbit.



x_3	x_2	x_1	x_0	p
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1

x_3	x_2	x_1	x_0	p
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

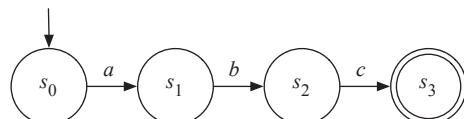
- Erzeugen Sie einen DEA, der die Integrität eines empfangenen Datenpakets überprüft und alle korrekt übertragenen Wörter akzeptiert. Wurde ein einzelnes Bit des Datenpakets während der Übertragung verfälscht, so soll der Automat das Eingabewort zurückweisen.
- Wie verhält sich der von Ihnen konstruierte Automat, falls zwei Bits während der Datenübertragung verfälscht wurden?

Aufgabe 5.2



**Webcode
5866**

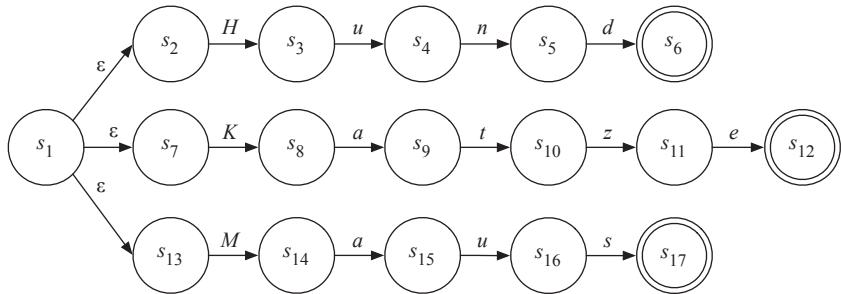
Der endliche Automat $A = (\{s_0, s_1, s_2, s_3\}, \{a, b, c\}, \delta, s_0)$ sei durch das folgende Zustandsübergangsdiagramm gegeben:



A akzeptiert die Sprache $L = \{abc\}$. Handelt es sich hier um einen DEA oder um einen NEA? Begründen Sie Ihre Antwort.

Gegeben sei der folgende nichtdeterministische ε -Akzeptor, der auf die Erkennung bestimmter Worte ausgelegt ist.

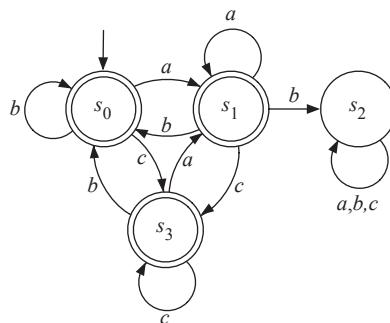
Aufgabe 5.3

Webcode
5892


- Formen Sie den Automaten so um, dass er nur einen einzigen Endzustand besitzt.
- Lässt sich die angewendete Umformung auf einen beliebigen ε -NEA übertragen?
- Lässt sich jeder DEA so umformen, dass er genau einen Endzustand besitzt?

Der endliche Automat $A = (\{s_0, s_1, s_2, s_3\}, \{a, b, c\}, \delta, s_0)$ sei durch das folgende Zustandsübergangsdiagramm gegeben:

Aufgabe 5.4

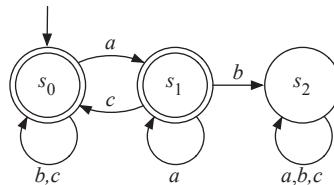
Webcode
5038


- Handelt es sich bei dem abgebildeten Automaten um einen DEA oder um einen NEA?
- Ist der Automat reduziert? Geben Sie gegebenenfalls den zugehörigen Minimalautomaten an.
- Welche Sprache akzeptiert der Automat?

Aufgabe 5.5

**Webcode
5000**

Der endliche Automat $A = (\{s_0, s_1, s_2, s_3\}, \{a, b, c\}, \delta, s_0)$ sei durch das folgende Zustandsübergangsdiagramm gegeben:



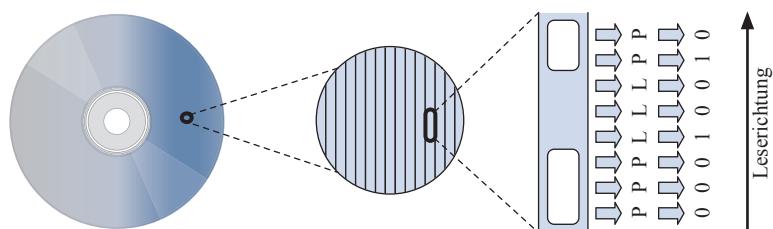
Beschreiben Sie jede der nachfolgend aufgeführten Wortmengen mithilfe eines regulären Ausdrucks.

- Die Menge der Wörter, die den Automaten von s_0 nach s_1 überführen, ohne den Zustand s_0 ein zweites Mal zu besuchen.
- Die Menge der Wörter, die den Automaten von s_0 nach s_0 überführen, ohne den Zustand s_0 zwischendurch ein drittes Mal zu besuchen.
- Die Menge aller Wörter, die der Automat akzeptiert.

Aufgabe 5.6

**Webcode
5295**

Auf der physikalischen Ebene arbeiten CDs, DVDs und Blu-ray Discs alle nach dem gleichen Grundprinzip. Die gespeicherten Daten werden als Folge mikroskopisch kleiner *Pits* (P) und *Lands* (L) auf das Trägermaterial aufgebracht. Pits besitzen die Eigenschaft, den einfallenden Laserstrahl zu streuen, und entsprechen bei nichtbeschreibbaren Medien kleinen Einkerbungen im Trägermaterial. Beim Lesen eines optischen Datenträgers wird der P-L-Datenstrom zunächst in einen binären Datenstrom aus Nullen und Einsen übersetzt. Jeder Übergang von P nach L oder von L nach P entspricht einer Eins, ansonsten wird eine Null codiert.



Konstruieren Sie einen Transduktor, der eine gegebene P-L-Sequenz in den zugehörigen 0-1-Datenstrom zurückübersetzt.

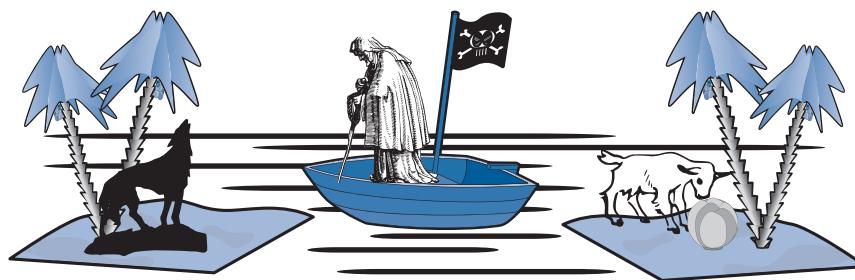
Im **BCD-Code** wird eine Dezimalzahl codiert, indem jede Ziffer in ein 4 Bit breites Datenpaket übersetzt wird:

Aufgabe 5.7**Webcode****5876**

Dezimalziffer	BCD-Codewort	Dezimalziffer	BCD-Codewort
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Konstruieren Sie einen seriellen BCD-Dezimal-Wandler, der einen BCD-codierten Datenstrom einliest und in die ursprüngliche Dezimaldarstellung zurückübersetzt.

Das Wolf-Ziege-Kohlkopf-Problem, kurz WZK-Problem, ist ein bekanntes Denkspiel. Ein Bauer ist mit der Aufgabe betraut, einen Wolf, eine Ziege und einen Kohlkopf auf die andere Uferseite zu bringen. Ihm steht nur ein kleines Boot zur Verfügung, so dass er jeweils nur einen der drei transportieren kann. Lässt er dabei die beiden Tiere alleine, so wird der Wolf die Ziege reißen. Die Ziege ist nicht weniger unschuldig und wird den Kohlkopf fressen, sobald sie der Bauer alleine lässt (vgl. [103]).

Aufgabe 5.8**Webcode****5922**

Modellieren Sie das WZK-Problem mithilfe eines endlichen Automaten. Dieser soll so konstruiert werden, dass jeder Zustand eindeutig eine bestimmte Position der Beteiligten codiert. Beschriften Sie die Transition mit einem Buchstaben aus der Menge $\{BW, BZ, BK\}$, je nachdem ob der Bauer den Wolf, die Ziege oder den Kohlkopf an das andere Ufer bringt.

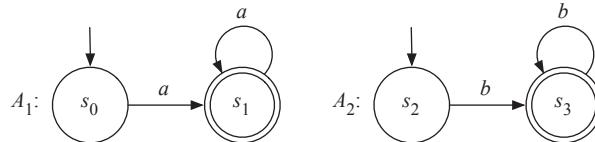
- Wie muss der Bauer die Überfahrt durchführen, damit alle Beteiligten wohlbehalten an das andere Ufer gelangen?
- Lassen sich die Lösungen des WZK-Problems direkt aus der Menge der akzeptierten Eingabewörter des von Ihnen konstruierten Automaten ableiten?

Aufgabe 5.9

Die Akzeptoren A_1 und A_2 seien wie folgt gegeben:



Webcode
5983



- Welche Sprache akzeptieren A_1 und A_2 ?
- Erzeugen Sie aus A_1 und A_2 einen Automaten A mit $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$.
- Erzeugen Sie aus A_1 und A_2 einen Automaten A mit $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

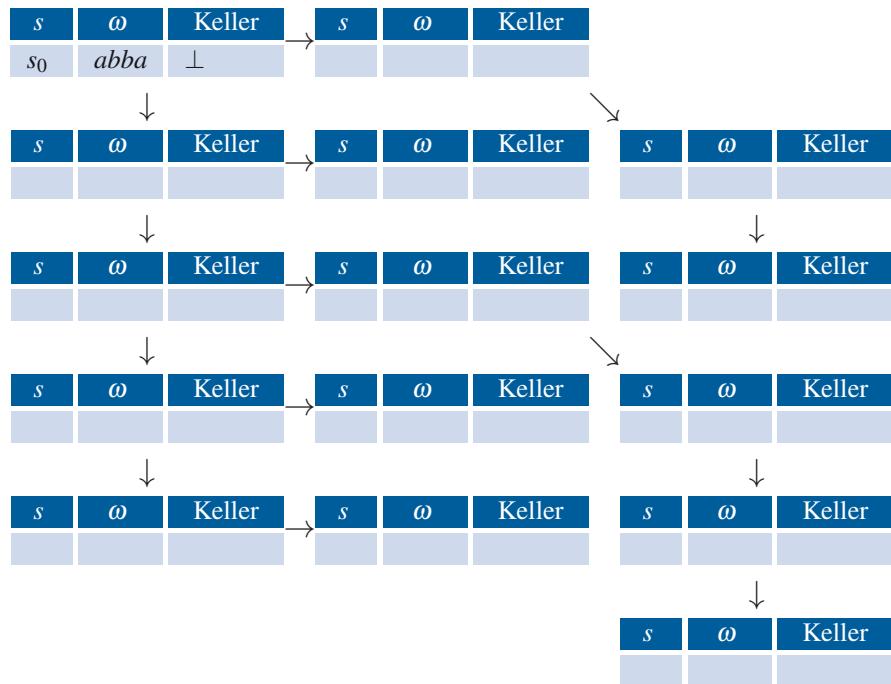
Aufgabe 5.10

In diesem Kapitel haben Sie einen Kellerautomaten kennengelernt, der die Palindromsprache

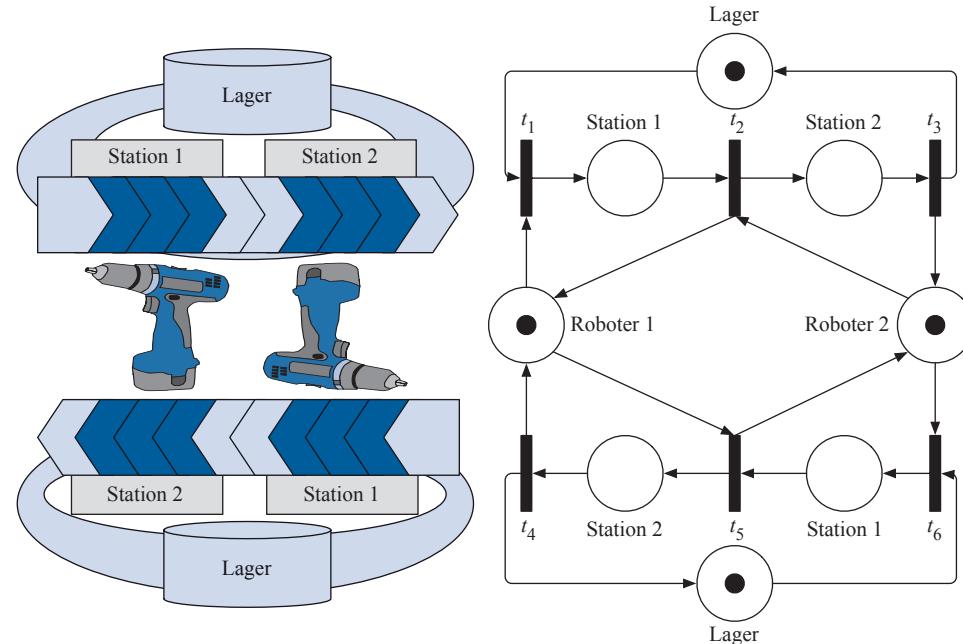
$$L_{\text{Pal}} := \{\varepsilon\} \cup \{\sigma_1 \dots \sigma_n \sigma_n \dots \sigma_1 \mid n \in \mathbb{N}^+, \sigma_i \in \{a, b\}\}$$

Webcode
5929

akzeptiert. Die formale Definition des Automaten ist in Abbildung 5.29 dargestellt. Vervollständigen Sie das folgende Diagramm, das alle Konfigurationsübergänge für das Eingabewort *abba* zusammenfasst (vgl. [52]):



In diesem Kapitel haben Sie gelernt, wie sich dynamische Systemeigenschaften mithilfe von Petri-Netzen beschreiben lassen. In der vorliegenden Übungsaufgabe wird ein Petri-Netz verwendet, um die Abläufe einer industriellen Produktionsanlage zu modellieren. Unsere fiktive Produktionsstraße besteht aus zwei Arbeitsstationen, an denen fest montierte Industrieroboter ihre Arbeit verrichten (vgl. [68]). An den Robotern werden zwei in entgegengesetzte Richtungen laufende Förderbänder vorbeigeführt. Entsprechend den eingestellten Laufrichtungen wird ein Werkstück im oberen Kreislauf zuerst durch den linken und anschließend durch den rechten Roboter bearbeitet. Im unteren Kreislauf ist die Reihenfolge genau andersherum.



Ein Werkstück wird zunächst aus dem Lager entnommen, danach durch beide Arbeitsstationen geführt und anschließend in das Lager zurückbefördert. Erst danach kann die Arbeit mit dem nächsten Werkstück beginnen. Die Roboter können Werkstücke auf beiden Förderbändern bearbeiten, aber nicht gleichzeitig beide Bänder bedienen.

Rechts ist das Petri-Netz-Modell der Produktionsstraße abgebildet. Lager, Roboter und Arbeitsstationen werden durch separate Stellen modelliert. Per Definition ist eine Arbeitsstation genau dann belegt, wenn sich eine Marke in der entsprechenden Stelle befindet.

- Analysieren Sie das Petri-Netz. Zeigen Sie, dass ein Systemstillstand eintreten kann.
- Modifizieren Sie das Modell so, dass ein verklemmungsfreies Petri-Netz entsteht.

Aufgabe 5.11

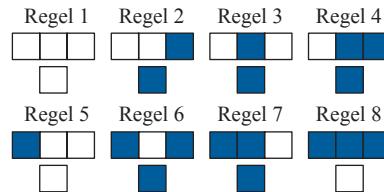


Webcode

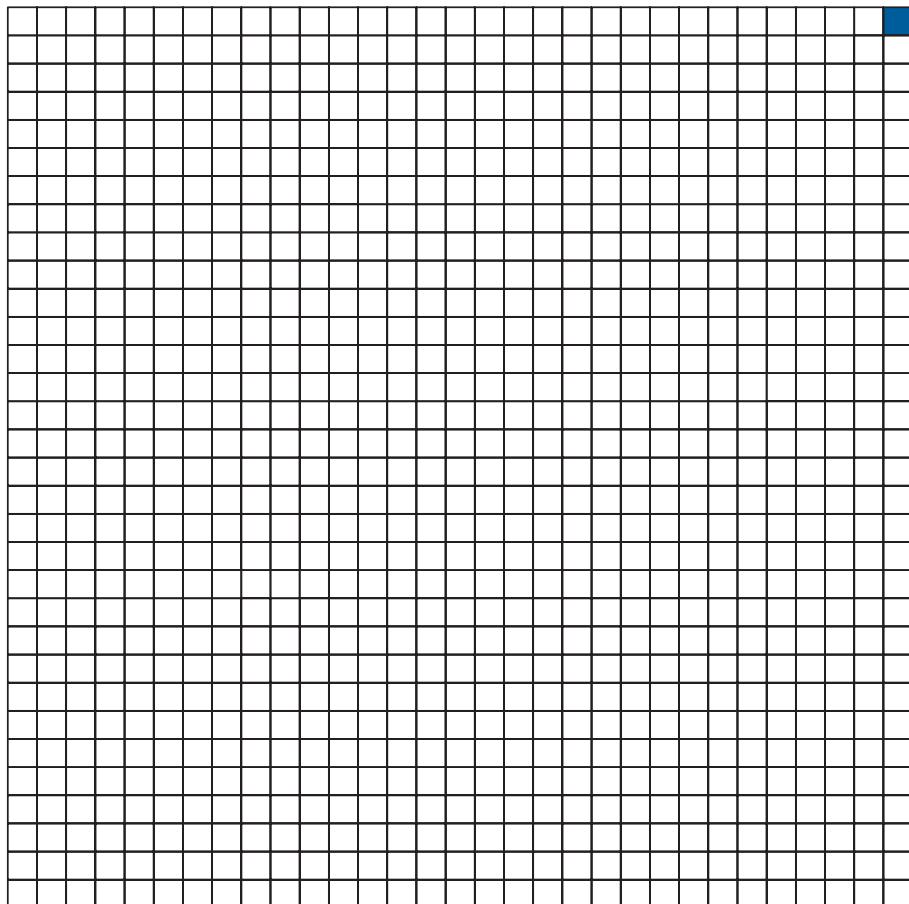
5482

Aufgabe 5.12**Webcode
5599**

In Abschnitt 5.8 haben Sie einen zellulären Automaten zur Erzeugung des Sierpinski-Dreiecks kennen gelernt. In dieser Übungsaufgabe wollen wir die Entstehungsregeln minimal modifizieren (vgl. [108]):



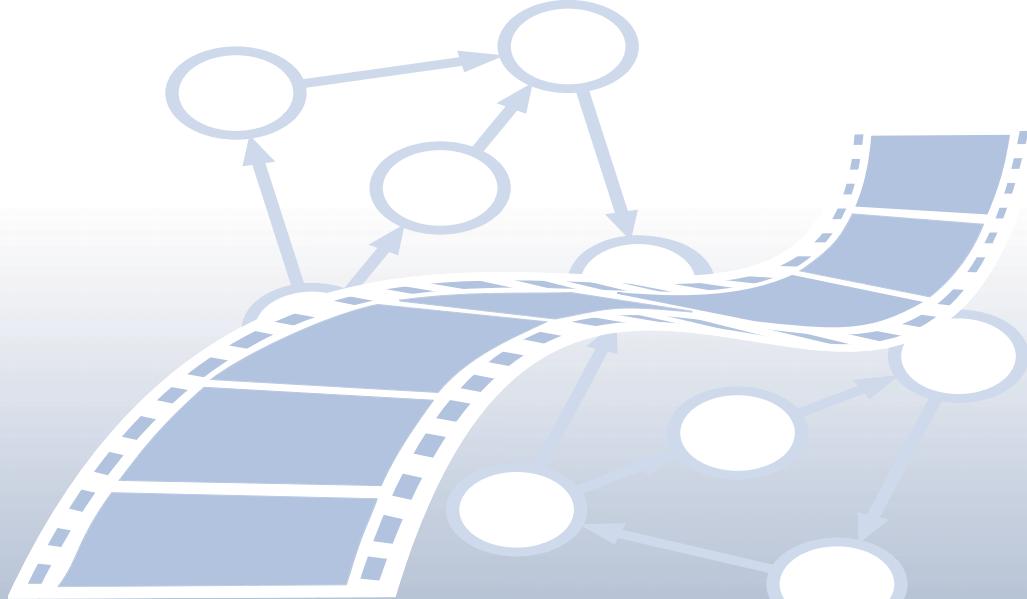
Stellen Sie fest, wie sich das Verhalten des Automaten geändert hat. Vervollständigen Sie hierzu das folgende Diagramm:



6 Berechenbarkeitstheorie

In diesem Kapitel werden Sie ...

- den abstrakten Begriff der Berechenbarkeit durchdringen,
- einfache Programme in der Loop-, While- und Goto-Sprache verfassen,
- den Aufbau primitiv-rekursiver Funktionen verstehen,
- die Funktionsweise und die Ausdrucksfähigkeit von Turing-Maschinen erforschen,
- einen Einblick in Registermaschinen und den Lambda-Kalkül erhalten,
- die Kernaussage der Church'schen These nachvollziehen,
- die theoretischen Grenzen der Berechenbarkeit erfahren,
- eine Reihe von unentscheidbaren Problemen kennen lernen.



■ Beispiel 1

```
add.loop
x0 := x1;
loop x2 do
  x0 := succ(x0)
end
```

1
2
3
4

■ Beispiel 2

```
triple.loop
x0 := x1;
loop x1 do
  x0 := succ(x0)
end;
loop x1 do
  x0 := succ(x0)
end
```

1
2
3
4
5
6
7

Abbildung 6.1: Zwei Programme der Loop-Sprache. Das erste Programm implementiert die Addition zweier natürlicher Zahlen ($x_0 := x_1 + x_2$), das zweite verdreifacht den Eingabewert ($x_0 := 3 \cdot x_1$).

6.1 Berechnungsmodelle

Jeder von uns besitzt eine intuitive Vorstellung davon, was es bedeutet, etwas zu *berechnen*. Bei genauerer Betrachtung entpuppen sich unsere Gedankenmodelle jedoch schnell als zu vage, um daraus handfeste Schlussfolgerungen abzuleiten. Insbesondere reichen unsere informellen Vorstellungen nicht aus, um belastbare Aussagen über die Unberechenbarkeit bestimmter Funktionen zu treffen oder den schon mehrmals recht lax verwendeten Begriff der Entscheidbarkeit auf seine eigenen Füße zu stellen. Kurzum: Wir kommen nicht umhin, den Begriff der *Berechenbarkeit* formal zu präzisieren. Zu diesem Zweck wurde die Berechenbarkeitstheorie geschaffen, die uns die benötigte Grundlage in Form von formalen Berechnungsmodellen zur Verfügung stellt.

In den folgenden Abschnitten werden wir die aus heutiger Sicht wichtigsten Berechnungsmodelle genauer untersuchen. Lassen Sie sich nicht von der großen Anzahl an Modellen abschrecken, auch wenn die Stoffmenge erdrückend erscheint! In Abschnitt 6.2 werden wir im Zusammenhang mit der *Church'schen These* zeigen, dass sich die meisten Modelle nur äußerlich unterscheiden und denselben Berechenbarkeitsbegriff begründen. In diesem Sinne wird sich die Berechenbarkeit als eine tiefgründige Eigenschaft erweisen, die unabhängig von ihrer Form existiert und sich durch diese in keiner Weise beeinflussen lässt.

6.1.1 Loop-Programme

Wir beginnen mit der Diskussion der *Loop-Sprache*, dem einfachsten Berechnungsmodell dieses Kapitels.



Definition 6.1 (Loop-Programme)

Ein Loop-Programm besteht aus den folgenden Komponenten:

- der *Konstanten* 0,
- den *Variablen* x_0, x_1, x_2, \dots ,
- den *Operatoren* succ und pred,
- dem *Zuweisungsoperator* ':= ',
- dem *Kompositionssoperator* ';' ,
- dem *Schleifenkonstrukt* loop do end

Abbildung 6.1 zeigt, wie sich die einzelnen Sprachkonstrukte zu komplexeren Programmen zusammensetzen lassen. Die Syntax folgt den Konstruktionsprinzipien moderner Programmiersprachen, so dass wir auf eine ausschweifende Definition an dieser Stelle verzichten wollen. Über die Semantik der Loop-Sprache wollen wir dagegen nicht so schnell hinweggehen.

Im Zentrum der Loop-Semantik steht der *Speichervektor*

$$\mathbf{v} = (x_0, x_1, \dots, x_n, x_{n+1}, \dots, x_m),$$

der den Zustand aller Programmvariablen zu einem bestimmten Zeitpunkt beschreibt. Per Definition werden die Eingabewerte in den Variablen x_1, \dots, x_n bereitgestellt und das berechnete Ergebnis in x_0 abgelegt. Die restlichen Variablen dienen der internen Speicherung von Zwischenergebnissen. Im Folgenden bezeichnen wir mit $\mathbf{v}[x_i \leftarrow y]$ den Vektor \mathbf{v} , in dem die i -te Komponente durch den Wert y ersetzt wurde.

Die *Übergangsfunktion* $\delta(\mathbf{v}, P)$ nimmt einen Speichervektor \mathbf{v} sowie ein Loop-Programm P entgegen und berechnet daraus den Speichervektor, der aus \mathbf{v} nach der Ausführung von P entsteht. Wir definieren δ induktiv über die Programmstruktur:

■ Wertzuweisung

$$\delta(\mathbf{v}, x_i := 0) := \mathbf{v}[x_i \leftarrow 0] \quad (6.1)$$

$$\delta(\mathbf{v}, x_i := \text{succ}(x_j)) := \mathbf{v}[x_i \leftarrow x_j + 1] \quad (6.2)$$

$$\delta(\mathbf{v}, x_i := \text{pred}(x_j)) := \begin{cases} \mathbf{v}[x_i \leftarrow x_j - 1] & \text{für } x_j > 0 \\ \mathbf{v} & \text{für } x_j = 0 \end{cases} \quad (6.3)$$

■ Komposition

$$\delta(\mathbf{v}, P_1; P_2) := \delta(\delta(\mathbf{v}, P_1), P_2) \quad (6.4)$$

■ Loop-Schleife

$$\delta(\mathbf{v}, \text{loop } x_i \text{ do } P \text{ end}) := \delta(\mathbf{v}, \underbrace{P; P; \dots; P}_{x_i \text{ Kopien}}) \quad (6.5)$$

Abbildung 6.2 zeigt, wie sich das oben eingeführte Loop-Programm add.loop mithilfe der Übergangsfunktion schrittweise simulieren lässt. Sind die Register x_1 und x_2 , wie in unserem Beispiel, mit den Werten 3 und 4 vorinitialisiert, so finden wir in Register x_0 am Ende den erwarteten Ergebniswert 7 vor.

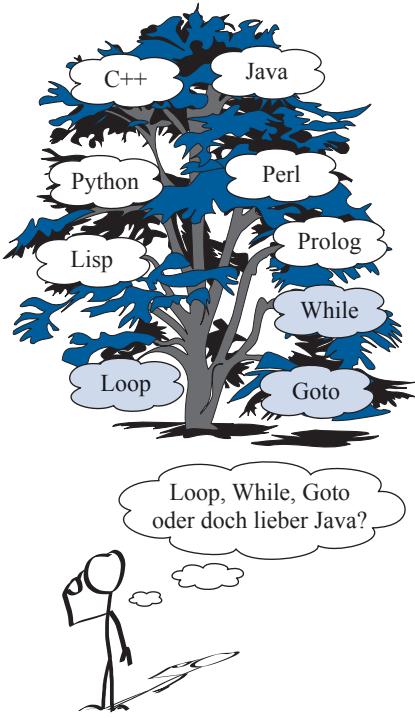
Die Übergangsfunktion δ setzen wir nun ein, um den Begriff der Loop-Berechenbarkeit formal zu definieren:

$$\begin{aligned} & \delta((0, 3, 4), \\ & \quad x_0 := x_2; \\ & \quad \text{loop } x_1 \text{ do } x_0 := \text{succ}(x_0) \text{ end }) \\ = & \delta(\delta((0, 3, 4), x_0 := x_2), \\ & \quad \text{loop } x_1 \text{ do } x_0 := \text{succ}(x_0) \text{ end }) \\ = & \delta((4, 3, 4), \\ & \quad \text{loop } x_1 \text{ do } x_0 := \text{succ}(x_0) \text{ end }) \\ = & \delta((4, 3, 4), \\ & \quad x_0 := \text{succ}(x_0); \\ & \quad x_0 := \text{succ}(x_0); \\ & \quad x_0 := \text{succ}(x_0)) \\ = & \delta(\delta((4, 3, 4), x_0 := \text{succ}(x_0)), \\ & \quad x_0 := \text{succ}(x_0); \\ & \quad x_0 := \text{succ}(x_0)) \\ = & \delta((5, 3, 4), \\ & \quad x_0 := \text{succ}(x_0); \\ & \quad x_0 := \text{succ}(x_0)) \\ = & \delta(\delta((5, 3, 4), x_0 := \text{succ}(x_0)), \\ & \quad x_0 := \text{succ}(x_0)) \\ = & \delta((6, 3, 4), \\ & \quad x_0 := \text{succ}(x_0)) \\ = & (7, 3, 4) \end{aligned}$$

Abbildung 6.2: Wie hier am Beispiel des Programms add.loop demonstriert, wird die Ausführung eines Loop-Programms durch die schrittweise Auswertung der Übergangsfunktion simuliert. Zu Beginn befinden sich in den Variablen x_1 und x_2 die Eingabewerte 3 und 4. Am Ende enthält die Ergebnisvariable x_0 erwartungsgemäß die Summe $3 + 4 = 7$.

Berechnungsmodelle haben nichts mit gutem Programmierstil zu tun! Im Folgenden werden Sie viele Programme kennenlernen, die aus der Sicht der modernen Software-Technik weit von einer empfehlenswerten Lösung entfernt sind.

Behalten Sie stets im Auge, dass es in diesem Kapitel um den abstrakten Begriff der Berechenbarkeit und nicht um die Erstellung effizienter, wartbarer und verständlicher Programme geht. Niemand wird freiwillig einen Algorithmus in der hier entwickelten Loop-, While- oder Goto-Sprache verfassen wollen und soll es auch nicht! Das Einzige, was an dieser Stelle zählt, ist der formale Beweis, dass eine Implementierung aus theoretischer Sicht möglich wäre.



Definition 6.2 (Loop-Berechenbarkeit)

Sei $f : \mathbb{N}^n \rightarrow \mathbb{N}$ eine beliebige Funktion über den natürlichen Zahlen. f heißt *Loop-berechenbar*, falls ein Loop-Programm P mit der folgenden Eigenschaft existiert:

$$\delta((0, x_1, \dots, x_n, 0, \dots), P) = (f(x_1, \dots, x_n), \dots)$$

Mit dieser Definition haben wir implizit festgelegt, dass x_0 sowie alle für die interne Verwendung vorgesehenen Variablen zu Beginn mit dem Wert 0 belegt sind. Das ist der Grund, weshalb wir diese Variablen in unseren Loop-Programmen ohne manuelle Initialisierung bedenkenlos verwenden dürfen.

Auf den ersten Blick erscheinen Loop-Programme schon fast als trivial und wir werden weiter unten in der Tat herausarbeiten, dass nicht jede berechenbare Funktion auch tatsächlich Loop-berechenbar ist. Nichtsdestotrotz ist das Schleifenkonstrukt stark genug, um wichtige Kontrollflussoperatoren zu simulieren. Beispielsweise können wir die Loop-Sprache durch ein If-Konstrukt ergänzen, ohne sie im Kern erweitern zu müssen. Hierzu fassen wir den Befehl

`if ($x_i = 0$) then P end`

ganz einfach als Makro für das folgende Schleifenkonstrukt auf:

```
 $x_j := 1;$ 
loop  $x_i$  do  $x_j := 0$  end;
loop  $x_j$  do  $P$  end
```

Auch im Hinblick auf den Konstantenvorrat sowie die zur Verfügung gestellten Arithmetikoperationen erweist sich die Loop-Sprache als äußerst spartanisch. Zum Glück sind wir hier ebenfalls in der Lage, durch die Definition einiger nützlicher Makros Linderung zu schaffen. So können wir z.B. alle natürlichen Zahlen bedenkenlos als Konstanten verwenden, indem wir diese als Abkürzungen für die folgenden Ausdrücke betrachten:

```
1 := succ(0),
2 := succ(succ(0)),
3 := succ(succ(succ(0))), ...
```

Darüber hinaus ist die Loop-Sprache aussagekräftig genug, um alle elementaren Arithmetikoperatoren zu erzeugen. Abbildung 6.3 zeigt, wie

sich z. B. die Addition ($x_1 + x_2$), die Multiplikation ($x_1 \cdot x_2$), die Potenz ($x_1^{x_2}$) sowie die Hyperpotenz $x_1 \uparrow^2 x_2$ durch Loop-Programme berechnen lassen. Um unnötige Schreibarbeit zu ersparen, benutzen die Implementierungen die bereits definierten Funktionen in Form von *Makros*. Lösen wir alle Makros entsprechend ihrer Definition auf, so erhalten wir ein „echtes“ Loop-Programm, das ausschließlich die in Definition 6.1 eingeführten Operatoren verwendet. Beachten Sie, dass es hierzu nicht ausreicht, die Makro-Definition nur textuell zu ersetzen, da die Verwendung der gleichen Variablennamen unweigerlich zu Namenskonflikten führt. Probleme dieser Art lassen sich beheben, indem die Makro-Variablen umbenannt und die Aufrufparameter vor und hinter dem expandierten Makro-Code umkopiert werden.

Die gewählten Beispiele bringen zwei zentrale Eigenschaften von Loop-Programmen zum Vorschein. Zum einen zeigen sie, dass sich durch die verschachtelte Verwendung des Schleifenoperators auch arithmetische Operationen höherer Grade implementieren lassen. Selbst der ungewöhnliche Up-Arrow-Operator, den wir ausführlich in Abschnitt 2.3.2 besprochen haben, lässt sich problemlos berechnen. Zum anderen wird deutlich, dass ein enger Zusammenhang zwischen der Anzahl der benötigten Schleifenkonstrukte und dem Grad der realisierten Arithmetikoperation besteht. Kommt die Addition mit einer einzigen Schleife aus, so benötigen wir für die Multiplikation zwei und für die Potenzierung bereits drei ineinander geschachtelte Schleifen. Verallgemeinert gilt der folgende Satz:

Satz 6.1

Jedes Loop-Programm zur Berechnung der Funktion $x \uparrow^n y$ benötigt mindestens $n + 2$ Schleifen.

Wir wollen die aus der Beobachtung gewonnene Eigenschaft an dieser Stelle ohne Beweis akzeptieren. Mit dem Mittel der strukturellen Induktion lässt sie sich über dem rekursiven Aufbau von Loop-Programmen formal belegen (siehe z.B. [89]).

Satz 6.1 hat weitreichende Konsequenzen für die Ausdrucksstärke der Loop-Sprache. So folgt aus ihm unmittelbar, dass die Ackermann-Funktion $\text{ack}(n, m)$ nicht Loop-berechenbar ist. Warum dies so ist, geht aus der folgenden Beziehung hervor, die wir in Abschnitt 2.3.2 im Detail herausgearbeitet haben:

$$\text{ack}(n, m) = 2 \uparrow^{n-2} (m + 3) - 3$$

■ $x_0 := \text{add}(x_1, x_2)$

add.loop

```
// Berechnet x0 := x1 + x2
x0 := x1;
loop x2 do
  x0 := succ(x0)
end
```

1
2
3
4
5
6

■ $x_0 := \text{mult}(x_1, x_2)$

mult.loop

```
// Berechnet x0 := x1 · x2
x0 := 0;
loop x2 do
  x0 := add(x1, x0)
end
```

1
2
3
4
5
6

■ $x_0 := \text{power}(x_1, x_2)$

power.loop

```
// Berechnet x0 := x1^{x2}
x0 := 1;
loop x2 do
  x0 := mult(x1, x0)
end
```

1
2
3
4
5
6

■ $x_0 := \text{hyper}(x_1, x_2)$

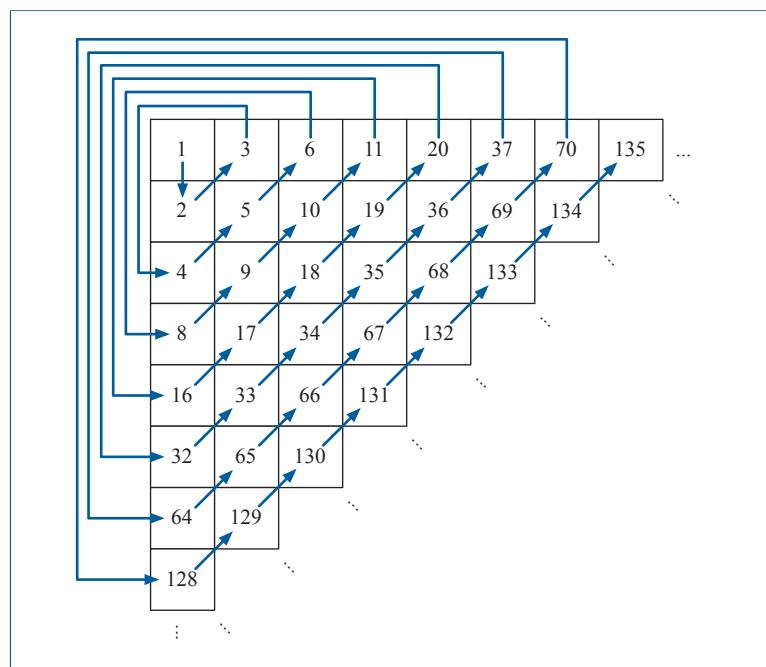
hyper.loop

```
// Berechnet x0 := x1 \uparrow^2 x2
x0 := 1;
loop x2 do
  x0 := power(x1, x0)
end
```

1
2
3
4
5
6

Abbildung 6.3: Alle gängigen Arithmetikoperationen lassen sich mithilfe von Loop-Programmen simulieren.

Abbildung 6.4: Die abgebildete Paarungsfunktion ordnet jedem Tupel $(p, q) \in \mathbb{N}^2$ eine Zahl $\pi(p, q) \in \mathbb{N}$ zu. Ein vergleichender Blick auf Abbildung 2.21 zeigt, dass es sich hierbei um eine Abwandlung der Cantor'schen Paarungsfunktion aus Kapitel 2 handelt. Die gewählte Funktion ist im Gegensatz zum Original nicht mehr bijektiv, erfüllt aber dennoch ihren Zweck: Sie gestattet uns, ein beliebiges Zahlenpaar aus der Menge \mathbb{N}^2 in eine einzige Zahl aus \mathbb{N} hineinzucodieren. Die gewählte Variante besitzt den Vorteil, eine vergleichsweise einfache Rechnungsformel zu besitzen. Hierdurch erfordert die Codierung weniger Rechenschritte als das Original.



Gäbe es ein Loop-Programm, das $\text{ack}(n, m)$ tatsächlich für beliebige Werte von n berechnet, so müsste dieses aufgrund von Satz 6.1 – im Widerspruch zum endlichen Aufbau eines Loop-Programms – unendlich viele Schleifenkonstrukte enthalten.



Korollar 6.1

Die Ackermann-Funktion $\text{ack}(n, m)$ ist nicht Loop-berechenbar.

Neben der Ackermann-Funktion existiert eine wichtige Klasse von Funktionen, die ebenfalls nicht Loop-berechenbar sind. Die Rede ist von *partiellen Funktionen*, die in Abschnitt 2.2 eingeführt wurden und in der theoretischen Informatik eine wichtige Rolle spielen. Partielle Funktionen sind für uns die formale Grundlage, um Programme zu beschreiben, die nicht für alle Eingabekombinationen terminieren.

Ein Blick auf die Gleichungen (6.1) bis (6.5) klärt unmittelbar auf, warum diese Limitierung besteht. Loop-Programme terminieren für jede beliebige Eingabe und können damit im Gegensatz zu Programmiersprachen, die in der realen Software-Entwicklung verwendet werden, keine Endlosschleifen hervorbringen.


Satz 6.2

Alle Loop-berechenbaren Funktionen $f : \mathbb{N}^n \rightarrow \mathbb{N}$ sind total.

Zum Schluss wollen wir uns ein wenig genauer mit den Möglichkeiten beschäftigen, die uns innerhalb eines Loop-Programms für die Speicherung interner Zwischenergebnisse zur Verfügung stehen.

Zunächst sieht die Sprachdefinition vor, dass wir Zugriff auf beliebig viele Variablen haben, die nach Belieben beschrieben oder ausgelesen werden dürfen. Viele Programme ließen sich jedoch einfacher formulieren, wenn wir Zwischenergebnisse auf einem Stapel (*stack*) ablegen könnten. Wie ein Stapel funktioniert, haben wir detailliert in der Diskussion über Kellerautomaten besprochen (vgl. Abschnitt 5.5). Wir werden nun zeigen, dass die Loop-Sprache, so primitiv sie auf den ersten Blick auch wirkt, stark genug ist, um einen Kellerspeicher zu simulieren.

Hierbei verfolgen wir den Ansatz, alle Stapelemente in eine einzige Variable hineinzucodieren. Den Schlüssel hierzu bildet die Cantor'sche Paarungsfunktion aus Abschnitt 2.3.3, die eine eindeutige Abbildung zwischen der Menge \mathbb{N}^2 und \mathbb{N} herstellt. Für unseren Zweck verwenden wir eine abgewandelte Variante, die wie folgt definiert ist:

$$\pi(x, y) = 2^{x+y} + x$$

Abbildung 6.4 zeigt einen Auszug aus der zweidimensionalen Wertetabelle. Die Funktion π sowie deren Umkehrfunktion π^{-1} sind Loop-berechenbar, so dass wir Zahlentupel und einzelne Zahlen beliebig hin- und hercodieren können. Die Implementierungen sind in Abbildung 6.5 dargestellt. Beide machen ausgiebig von den weiter oben eingeführten Makro-Konstrukten Gebrauch.

Nachdem wir nun in der Lage sind, ein beliebiges Zahlenpaar in eine einzige Zahl hineinzucodieren, ist unser Ziel schon so gut wie erreicht. Wir können einen Stapel mit k Elementen n_1, \dots, n_k repräsentieren, indem wir die Paarungsfunktion π rekursiv anwenden:

$$n = \pi(n_k, \pi(n_{k-1}, \dots, \pi(n_1, 0) \dots))$$

Mit n erhalten wir eine einzige natürliche Zahl, die stellvertretend für den gesamten Kellerspeicher steht. Das Element 0 nimmt eine Sonderstellung ein. Es repräsentiert einen leeren Stapel und kann dazu verwendet werden, die Anzahl der Stapelemente zu ermitteln.

Die geleistete Arbeit hat sich gelohnt. Abbildung 6.6 zeigt, wie wir die elementaren Stapeloperationen in der Loop-Sprache implementieren können. new erzeugt einen leeren Stapel, push fügt dem Stapel ein

■ $x_0 := \text{cantor}(x_1, x_2)$

cantor.loop

```

1  x0 := x1 + x2 ;
2  x0 := 2x0 ;
3  x0 := x0 + x1
4
5
```

■ $(x_1, x_2) := \text{cantor}^{-1}(x_0)$

cantor_invers.loop

```

1  loop x0
2    if (2 · 2x3 ≤ x0)
3      x3 := succ(x3)
4    end ;
5
6    // An dieser Stelle gilt:
7    // x3 = max{n | 2n ≤ x0}
8
9    x1 := x0 - 2x3 ;
10   x2 := x3 - x1
11
12
```

Abbildung 6.5: Die Cantor'sche Paarungsfunktion ist Loop-berechenbar.

■ $x_j := \text{new}()$

new.loop

1 $x_j := 0;$

Element hinzu und pop entfernt das oberste Element. Ab sofort dürfen wir alle vier Makrobefehle in unseren Loop-Programmen frei verwenden.

■ $\text{push}(x_i, x_j)$

push.loop

1 $x_j := \text{cantor}(x_i, x_j);$

Abbildung 6.6 zeigt ferner, wie sich die Size-Operation in Form eines Loop-Programms berechnen lässt. $\text{size}(x)$ liefert die Anzahl der Stack-Elemente zurück und ist für den praktischen Umgang mit dieser Datenstruktur unumgänglich. Die Implementierung basiert auf der Tatsache, dass die Anzahl der Elemente eines mit x codierten Stapels stets kleiner ist als der Wert x selbst. Entsprechend lässt sich die Anzahl der Elemente bestimmen, indem wir die Stack-Inhalte der Reihe nach decodieren und einen dedizierten Zähler so lange erhöhen, bis das Stack-Ende erreicht ist.

■ $x_i := \text{pop}(x_j)$

pop.loop

1 $(x_i, x_j) := \text{cantor}^{-1}(x_j)$

6.1.2 While-Programme

■ $x_i := \text{size}(x_j)$

size.loop

1 $x_i := 0;$
 2 **loop** x_j **do**
 3 **if** $(x_j \neq 0)$ **then**
 4 $x_i := \text{succ}(x_i);$
 5 $\text{pop}(x_j)$
 6 **end**
 7 **end**

Die Diskussion über die Loop-Sprache hat gezeigt, dass das eingeführte Schleifenkonstrukt nicht ausdrucksstark genug ist, um alle Funktionen zu berechnen, die auch im intuitiven Sinne berechenbar sind. Die Ackermann-Funktion hat uns die Limitierungen deutlich vor Augen geführt. In diesem Abschnitt wollen wir die Loop-Schleife um ein komplexeres Konstrukt ergänzen, das zu einer echten Erweiterung der Ausdrucksstärke führen wird. Die Rede ist von der *While-Schleife*.



Definition 6.3 (While-Programme)

Ein While-Programm besteht aus den folgenden Komponenten:

- der *Konstanten* 0,
- den *Variablen* x_0, x_1, x_2, \dots ,
- den *Operatoren* succ und pred,
- dem *Zuweisungsoperator* ':= ',
- dem *Kompositionssoperator* ',' ,
- dem *Schleifenkonstrukt* loop do end,
- dem *Schleifenkonstrukt* while do end

Abbildung 6.6: Die Loop-Sprache ist ausdrucksstark genug, um einen Stapel zu simulieren. Die Programme zeigen, wie sich die vier elementaren Stapeloperationen implementieren lassen.

Alle Konstrukte der Loop-Sprache finden sich auch in der While-Sprache wieder und wir übernehmen deren Semantik eins zu eins.

Die While-Sprache wird hierdurch zu einer echten Erweiterung der Loop-Sprache, so dass wir nicht nur jedes Loop-Programm als While-Programm betrachten können, sondern darüber hinaus auf sämtliche der weiter oben definierten Makros zurückgreifen dürfen.

Um die Semantik der While-Sprache vollständig zu definieren, müssen wir nur noch die Bedeutung der While-Schleife festlegen. Zu diesem Zweck definieren wir zuerst die *Terminierungsmenge* T_i :

$$T_i := \{k \in \mathbb{N} \mid \delta(\nu, P^k) = (\dots, x_{i-1}, 0, x_{i+1}, \dots)\}$$

Ausgehend von einem gegebenen Speichervektor ν enthält die Terminierungsmenge alle Zahlen $k \in \mathbb{N}$ mit der Eigenschaft, dass x_i nach der k -ten Wiederholung von P den Wert 0 annimmt. Besonders wichtig wird für uns das Minimum der Menge T_i sein. Dieses beschreibt die minimale Anzahl von Iterationen, die wir P ausführen müssen, um die Bedingung $x_i > 0$ falsch werden zu lassen. Ein Sonderfall liegt vor, falls T_i zur leeren Menge degeneriert. In diesem Fall ist die Bedingung $x_i > 0$ immer erfüllt und wir können in T_i kein Minimum bestimmen.

Basierend auf dem Begriff der Terminierungsmenge legen wir die Semantik des While-Konstruktks wie folgt fest:

$$\delta(\nu, \text{while } x_i \text{ do } P) = \begin{cases} \perp & \text{falls } T_i = \emptyset \\ \delta(\nu, P^{\min T_i}) & \text{falls } T_i \neq \emptyset \end{cases}$$

Für den programmübergreifenden Umgang mit undefinierten Funktionswerten vereinbaren wir die folgende Beziehung:

$$\delta(\perp, P) = \perp$$

Hierdurch wird nachgebildet, dass eine einzige nichtterminierende While-Schleife dazu führt, dass das gesamte Programm niemals anhält.

Wir sind nun in der Lage, den Begriff der While-Berechenbarkeit formal zu definieren.



Definition 6.4 (While-Berechenbarkeit)

Mit $f : \mathbb{N}^n \rightarrow \mathbb{N}$ sei eine partielle Funktion über den natürlichen Zahlen gegeben. f heißt *While-berechenbar*, falls ein While-Programm P mit den folgenden Eigenschaften existiert:

$$\delta((0, x_1, \dots, x_n, 0, \dots), P) = \begin{cases} \perp & \text{falls } f(x_1, \dots, x_n) = \perp \\ (f(x_1, \dots, x_n), \dots) & \text{falls } f(x_1, \dots, x_n) \neq \perp \end{cases}$$

Vielleicht ist Ihnen in der Definition der While-Sprache aufgefallen, dass wir die Loop-Schleife nicht aus dem Sprachschatz entfernt haben. In der Tat hätten wir auf die Aufnahme des Loop-Konstruktts problemlos verzichten können. Die While-Schleife ist so ausdrucksstark, dass wir jede Loop-Schleife, wie in Abbildung 6.7 gezeigt, simulieren können. Dass wir an der Loop-Schleife dennoch festhalten, hat triftige Gründe, die am Ende dieses Abschnitts ersichtlich werden. Der interessierte Leser möge bereits jetzt einen Blick auf Satz 6.3 werfen. Dieser würde falsch werden, wenn wir das Loop-Konstrukt aus der While-Sprache verbannen.

loop.while

```
1 loop x_i do
2   P
3 end
```



loop_simulate.while

```
1 x_j := x_i;
2 while x_j do
3   P;
4   x_j := pred(x_j)
5 end
```

Abbildung 6.7: Der While-Befehl ist ausdrucksstark genug, um den Loop-Befehl zu simulieren.

less.while	greater.while	equal.while
<pre> while x < y do P end </pre>	<pre> while x > y do P end </pre>	<pre> while x = y do P end </pre>
		
less_simulate.while	greater_simulate.while	equal_simulate.while
<pre> z := y - x; while z do P; z := y - x end </pre>	<pre> z := x - y; while z do P; z := x - y end </pre>	<pre> z := (x - y) + (y - x); while z do P; z := (x - y) + (y - x) end </pre>
1 2 3 4 5	1 2 3 4 5	1 2 3 4 5

Abbildung 6.8: Alle gängigen Arithmetikoperationen lassen sich mithilfe von While-Programmen simulieren.

Abbildung 6.8 zeigt, dass sich auch die Form der erlaubten Schleifenbedingungen weiter verallgemeinern lässt. Neben $<$, $>$ und $=$ lassen sich in analoger Weise die Relationen \leq , \geq sowie Vergleiche mit numerischen Konstanten modellieren.

Im Folgenden wollen wir herausarbeiten, wie flexibel die While-Schleife wirklich ist. Kombinieren wir das Schleifenkonstrukt mit den in Abschnitt 6.1.1 eingeführten Push- und Pop-Operationen, so sind wir in der Lage, rekursive Funktionsaufrufe in gewissen Grenzen zu simulieren. Insbesondere dann, wenn der rekursive Aufruf ganz am Ende des Funktionsrumpfes steht, gelingt die Simulation ohne Probleme. In diesem Fall reicht es aus, die Funktionsparameter auf den Stack zu legen und den Funktionsrumpf innerhalb einer While-Schleife auszuführen. Ein rekursiver Aufruf wird simuliert, indem die Aufrufparameter auf den Stack geschoben und im Zuge des Rücksprungs wieder entfernt werden. Die While-Schleife wird verlassen, sobald der Stack keine Elemente mehr enthält.

Beachten Sie, dass rekursive Aufrufe, die in der Mitte des Funktionsrumpfs stehen, nicht auf diese Weise simuliert werden können. Im Gegensatz zu realen Programmiersprachen sieht die While-Sprache keine Möglichkeit vor, eine Rücksprungadresse auf dem Stapel abzulegen.

Abbildung 6.9 demonstriert das Transformationsprinzip am Beispiel der Ackermann-Funktion. Die linke Seite enthält eine konventionelle

ackermann.c <pre> int ack(int n, int m) { if (n == 0) { return m+1; } if (m == 0) { return ack(n-1, 1); } return ack(n-1, ack(n,m-1)); } </pre>	ackermann.while <pre> 1 push(n, stack); 2 push(m, stack); 3 4 while size(stack) > 1 do 5 6 y := pop(stack); 7 x := pop(stack); 8 9 if (x == 0) then 10 push(y+1, stack) 11 end 12 else if (y == 0) then 13 push(x-1, stack); 14 push(1, stack); 15 end 16 else 17 push(x-1, stack); 18 push(x, stack); 19 push(y-1, stack) 20 end 21 end 22 23 x0 := pop(stack); </pre>
--	--

Abbildung 6.9: Die Ackermann-Funktion ist While-berechenbar. Das linke Programm zeigt eine C-Implementierung, die den Funktionswert $\text{ack}(n,m)$ rekursiv berechnet. Mithilfe der Push- und Pop-Makros lässt sich die Rekursion simulieren und das C-Programm in ein äquivalentes While-Programm überführen.

C-Implementierung, die den Funktionswert $\text{ack}(n,m)$ rekursiv berechnet. Die rechte Seite zeigt, wie sich die Rekursion unter Verwendung einer While-Schleife sowie mehrerer Push- und Pop-Operationen simulieren lässt. Mit dem konstruierten Programm haben wir ein wichtiges Ergebnis der Berechenbarkeitstheorie gezeigt. Zusammen mit Korollar 6.1 beweist dessen Existenz, dass die While-Sprache eine größere Ausdrucksstärke als die Loop-Sprache besitzt.

Zum Schluss wollen wir die Frage klären, wie viele While-Schleifen für die Berechnung einer Funktion mindestens benötigt werden. Für Loop-Programme haben wir beobachtet, dass die Anzahl der Schleifenkonstrukte, die für die Berechnung der Funktion $x \uparrow^n y$ benötigt werden, linear mit dem Parameter n wächst. Diese Erkenntnis wirft unweigerlich die Frage auf, ob ein ähnliches Ergebnis auch für While-Programme gilt. Der folgende Satz beantwortet diese Frage negativ:

■ add(3,4)

```

add.goto

M1 : x0 := x2 ;
M2 : if x1 goto M4 ;
M3 : halt ;
M4 : x0 := succ(x0) ;
M5 : x1 := pred(x1) ;
M6 : if x0 goto M2

```

1
2
3
4
5
6

 **Satz 6.3 (Satz von Kleene)**

Jede While-berechenbare Funktion lässt sich durch ein While-Programm mit nur einer While-Schleife berechnen.

Abbildung 6.10: Das dargestellte Goto-Programm berechnet die Summe zweier natürlicher Zahlen ($x_0 = x_1 + x_2$).

■ add(3,4)

```

((0,3,4,0,...), 1)
→ ((4,3,4,0,...), 2)
→ ((4,3,4,0,...), 4)
→ ((5,3,4,0,...), 5)
→ ((5,2,4,0,...), 6)
→ ((5,2,4,0,...), 2)
→ ((6,2,4,0,...), 4)
→ ((6,1,4,0,...), 6)
→ ((6,1,4,0,...), 2)
→ ((6,1,4,0,...), 4)
→ ((7,1,4,0,...), 5)
→ ((7,0,4,0,...), 6)
→ ((7,0,4,0,...), 2)
→ ((7,0,4,0,...), 3)
→ ((7,0,4,0,...), 0)

```

Abbildung 6.11: Schrittweise Ausführung des Goto-Programms aus Abbildung 6.10. Das Programm wird mit den Initialwerten $x_1 = 3$ und $x_2 = 4$ gestartet. Nach Ausführungsende enthält die Ergebnisvariable x_0 erwartungsgemäß den Wert 7.

6.1.3 Goto-Programme

Goto-Programme sind ähnlich einfach aufgebaut wie die weiter oben eingeführten Loop- und While-Programme und kommen gänzlich ohne die konventionellen Schleifenkonstrukte aus. An die Stelle der Loop- und der While-Schleife tritt der bedingte Sprung (If-Goto-Konstrukt).



Definition 6.5 (Goto-Programme)

Ein *Goto-Programm* ist eine Sequenz markierter Anweisungen:

$$M_1 : A_1 ; M_2 : A_2 ; \dots ; M_n : A_n$$

Die Anweisungen A_i bestehen aus den folgenden Komponenten:

- der *Konstanten* 0,
- den *Variablen* x_0, x_1, x_2, \dots ,
- den *Operatoren* succ und pred,
- dem *Zuweisungsoperator* ':= ',
- dem *bedingten Sprungbefehl* if x_i goto M_j ,
- dem *Stoppbefehl* halt

Die Semantik von Loop- und While-Programmen konnten wir induktiv über den Programmaufbau definieren, da die Schleifenkonstrukte keine Möglichkeit vorsehen, um an beliebige Stellen im Quellcode zu springen. Goto-Programme hingegen bieten diese Möglichkeit. Aus diesem Grund wählen wir einen zustandsbasierten Ansatz, wie er uns bereits im Rahmen der Diskussion über endliche Automaten begegnet ist.

Im Kern steht der Begriff der *Konfiguration*, der uns erlaubt, den augenblicklichen Zustand der Programmausführung im Sinne einer Momentaufnahme zu beschreiben.

Konkret wird die Konfiguration eines Goto-Programms durch einen Speichervektor ν und einen *Markenindex* k gebildet. Die Übergangsrelation → beschreibt, wie sich eine gegebene Konfiguration im Zuge der Programmausführung verändert. Die Folgekonfiguration wird durch den Befehl A_k bestimmt, wobei k dem Markierungsindex der aktuellen Konfiguration (ν, k) entspricht. In Abhängigkeit von A_k definieren wir → wie folgt:

- Zuweisung ($M_k : x_i := 0$, $M_k : x_i := \text{succ}(x_j)$, $M_k : x_i := \text{pred}(x_j)$)

$$(\nu, k) \rightarrow (\delta(\nu, A_k), k + 1)$$

Die Definition von δ übernehmen wir unverändert aus der Loop-Sprache.

- Bedingter Sprung ($M_k : \text{if } x_i \text{ goto } M_l$)

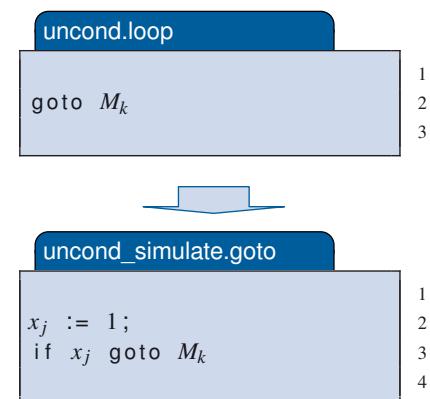
$$(\nu, k) \rightarrow \begin{cases} (\nu, k + 1) & \text{falls } x_i = 0 \\ (\nu, l) & \text{falls } x_i \neq 0 \end{cases}$$

- Stoppbefehl ($M_k : \text{halt}$)

$$(\nu, k) \rightarrow (\nu, 0)$$

Die Abbildungen 6.10 und 6.11 zeigen, wie die Addition zweier Zahlen x_1 und x_2 mithilfe eines Goto-Programms implementiert werden kann. Ersetzen wir den Operator succ durch den Operator pred , so entsteht ein Programm, das die (gesättigte) Subtraktion zweier Zahlen realisiert.

Im Folgenden wollen wir eine Reihe weiterer Vereinbarungen treffen, die den Umgang mit Goto-Programmen deutlich vereinfachen werden. Zum einen lassen wir zu, dass neben dem bedingten Sprung (If-Goto-Konstrukt) ein unbedingter Sprung (Goto-Konstrukt) eingesetzt werden darf. Die Verwendung ist legitim, da wir das Konstrukt, wie in Abbildung 6.12 gezeigt, auf den bedingten Sprung reduzieren können. Zum anderen erlauben wir, dass die If-Bedingungen auch Vergleiche der Form $x_i < x_j$, $x_i > x_j$ oder $x_i = x_j$ enthalten dürfen, wobei x_j entweder für eine andere Variable oder eine Konstante steht. Abbildung 6.13 demonstriert, wie sich die entsprechenden Anweisungen durch native Goto-Programme simulieren lassen.



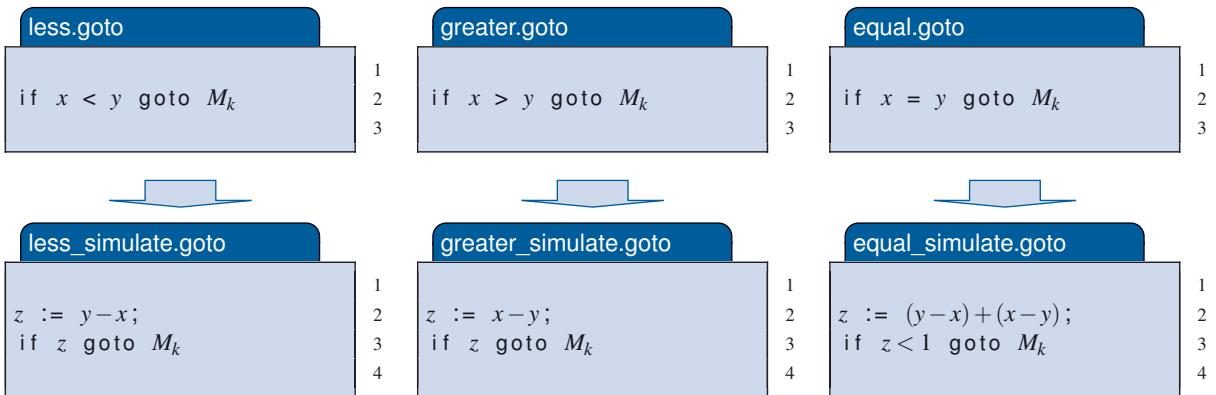


Abbildung 6.13: Die Goto-Sprache ist ausdrucksstark genug, um komplexe If-Bedingungen zu simulieren.

Mit den getroffenen Vereinbarungen können wir viele Programme übersichtlicher formulieren. Eine entsprechende Reimplementierung des Beispielprogramms add.loop ist in Abbildung 6.14 dargestellt.

Mithilfe der Übergangsrelation → sind wir in der Lage, den Begriff der Goto-Berechenbarkeit exakt festzulegen:



Definition 6.6 (Goto-Berechenbarkeit)

Mit $f : \mathbb{N}^n \rightarrow \mathbb{N}$ sei eine partielle Funktion über den natürlichen Zahlen gegeben. f heißt *Goto-berechenbar*, falls ein Goto-Programm P mit den folgenden Eigenschaften existiert:

$$\begin{aligned} ((0, x_1, \dots, x_n, 0, \dots), 1) \not\rightarrow^* ((\dots), 0) &\text{ falls } f(x_1, \dots, x_n) = \perp \\ ((0, x_1, \dots, x_n, 0, \dots), 1) \rightarrow^* ((f(x_1, \dots, x_n), \dots), 0) &\text{ sonst} \end{aligned}$$

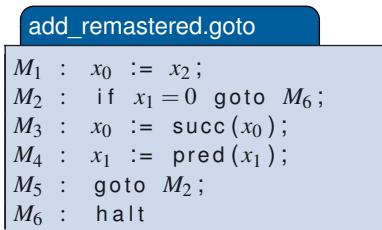


Abbildung 6.14: Das dargestellte Goto-Programm berechnet die Summe zweier natürlicher Zahlen ($x_0 = x_1 + x_2$). Es handelt es sich um eine Umformulierung des Originalprogramms aus Abbildung 6.10.

Berechnet ein Goto-Programm P die Funktion $f(x_1, \dots, x_n)$, so können wir einen konkreten Funktionswert bestimmen, indem wir die EingabevARIABLEN x_1, \dots, x_n auf die gewünschten Werte setzen und das Programm, wie oben gezeigt, ausführen. Für alle x_1, \dots, x_n mit $f(x_1, \dots, x_n) \neq \perp$ terminiert die Programmausführung irgendwann in einer Konfiguration, die den Markierungsindex 0 enthält, und wir können den gesuchten Funktionswert aus der Variablen x_0 auslesen. Gilt $f(x_1, \dots, x_n) = \perp$, so wird der Markierungsindex 0 nicht erreicht.

Auch wenn sich Goto-Programme auf den ersten Blick erheblich von den While-Programmen aus Abschnitt 6.1.2 unterscheiden, sind die

Differenzen rein äußerlicher Natur. Um den Zusammenhang zwischen beiden Sprachen offenzulegen, zeigen wir zunächst, wie sich ein While-Programm durch ein äquivalentes Goto-Programm ersetzen lässt, und anschließend, wie aus einem Goto-Programm ein äquivalentes While-Programm erzeugt werden kann.

Für die folgenden Betrachtungen sei ein beliebiges While-Programm gegeben, das sich aus einer Kette von n Anweisungen A_1 bis A_n zusammensetzt. In einem ersten Schritt übersetzen wir die Sequenz in eine markierte Anweisungskette und schließen diese am Ende mit einem halt-Befehl ab:

$$M_1 : A_1; M_2 : A_2; \dots; M_n : A_n; M_{n+1} : \text{halt}$$

Im zweiten Schritt ersetzen wir jeden While-Befehl der Form

$$M_i : \text{while } x_i \text{ do } P \text{ end}$$

durch das folgende Goto-Konstrukt:

$$\begin{aligned} M_i &: \text{if } x_i = 0 \text{ goto } M_{i+1}; \\ &\quad P; \\ &\quad \text{goto } M_i; \\ M_{i+1} &: \dots \end{aligned}$$

Die Konstruktion stellt sicher, dass das erzeugte Goto-Programm die gleiche (partielle) Funktion berechnet wie das ursprüngliche While-Programm.

Die umgekehrte Schlussrichtung können wir ebenfalls zeigen, müssen hierzu aber ein wenig trickreicher agieren. Um ein Goto-Programm der Form

$$M_1 : A_1; M_2 : A_2; \dots; M_n : A_n$$

in ein äquivalentes While-Programm zu übersetzen, kommen wir nicht umhin, Sprünge zu beliebigen Programmstellen zu simulieren. Hierbei verfolgend wir die Grundidee, den Markenindex eines Goto-Programms in eine dedizierte Variable y hineinzucodieren, die im ursprünglichen Programm nicht verwendet wird. Betten wir die Variable, wie in Abbildung 6.15 gezeigt, in eine Reihe von If-Anweisungen ein, so können wir eine vollständige Fallunterscheidung über alle Markenindizes durchführen. Jeder If-Zweig entspricht der Ausführung eines Befehls des Goto-Programms. Um eine kontinuierliche Programmausführung zu simulieren, umhüllen wir das Programmgerüst mit einer While-Schleife, die genau dann verlassen wird, wenn der Markenindex den Wert 0 annimmt.

Programmgerüst.while

```

y := 1;
while y ≠ 0 do
    if y = 1 then
        ... // A1
    end;
    if y = 2 then
        ... // A2
    end;
    ...
    if y = n then
        ... // An
    end
end

```

Abbildung 6.15: Das abgebildete Programmgerüst erlaubt, Goto-Programme mithilfe der While-Sprache zu simulieren. Die Variable y übernimmt die Rolle des Markenindexes.

Goto-Anweisung	While-Anweisung
$M_k : x_i := 0$	$x_i := 0; y := k + 1$
$M_k : x_i := \text{succ}(x_j)$	$x_i := \text{succ}(x_j); y := k + 1$
$M_k : x_i := \text{pred}(x_j)$	$x_i := \text{pred}(x_j); y := k + 1$
$M_k : \text{goto } M_l$	$y := l$
$M_k : \text{if } x_i \text{ goto } M_l$	$y := k+1; \text{if } x_i \text{ then } y := l$
$M_k : \text{halt}$	$y := 0$

Tabelle 6.1: Transformation von Goto-Anweisungen in die While-Sprache. Sprünge werden durch die Modifikation der Indexvariablen y simuliert.

Innerhalb des Goto-Programms wird dieser Markenindex genau dann erreicht, wenn der halt-Befehl ausgeführt wurde und das Programm terminiert.

Damit sind wir fast am Ziel. Wir müssen nur noch die Einzelanweisungen A_1 bis A_n übersetzen und in das konstruierte Programmgerüst einfügen. Die entsprechenden Transformationen sind in Tabelle 6.1 zusammengefasst. Eine besondere Rolle spielt auch hier wieder die Indexvariable y . Diese wird durch die vorhandenen Arithmetikoperationen so manipuliert, dass im nächsten Iterationsschritt immer der korrekte Folgebefehl ausgeführt wird. Insgesamt liefern die Transformationen einen konstruktiven Beweis für den folgenden Satz:

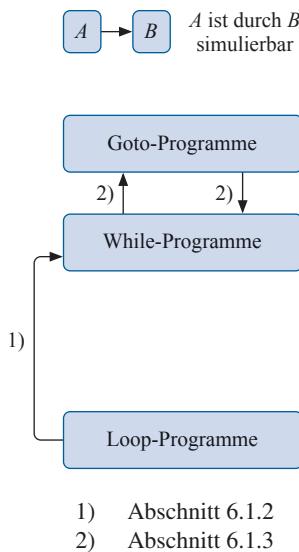


Abbildung 6.16: While-Programme besitzen eine größere Ausdrucksstärke als Loop-Programme. Zwischen While- und Goto-Programmen besteht hingegen kein Unterschied. Die Programme beider Sprachen lassen sich ineinander überführen, ohne die berechnete Funktion zu verändern.

Satz 6.4

Die Klasse der While-berechenbaren Funktionen stimmt mit der Klasse der Goto-berechenbaren Funktionen überein.

Abbildung 6.16 fasst die bisher erarbeiteten Ergebnisse in einer Gesamtübersicht zusammen.

Der oben konstruierte Beweis hält eine weitere Überraschung für uns bereit. Führen wir beide Transformationen für ein beliebiges While-Programm nacheinander durch, so erhalten wir aufgrund des Konstruktionsschemas aus Abbildung 6.15 ein Programm, das aus einer einzigen While-Schleife besteht. Wir haben damit einen Beweis für den Satz von Kleene gefunden, den wir in Abschnitt 6.1.2 ohne Begründung niedergeschrieben haben. Jetzt wird auch der Name klar, mit dem Programme in der speziellen Form aus Abbildung 6.15 häufig bezeichnet werden: Sie liegen in *Kleene'scher Normalform* vor.

6.1.4 Primitiv-rekursive Funktionen

Alle bisher betrachteten Ansätze verfolgen die Idee, den Berechenbarkeitsbegriff auf der Ebene von Programmiersprachen zu erfassen. Aus der Sicht des Informatikers macht diese Vorgehensweise durchaus Sinn, da sich hinter der Loop-, While- und Goto-Sprache die gleichen Konzepte verborgen, die wir aus realen Programmiersprachen kennen. Die Theorie der rekursiven Funktionen verfolgt einen anderen Ansatz und macht den Berechenbarkeitsbegriff auf rein mathematischem Weg zugänglich.

Auf den folgenden Seiten werden wir uns der Thematik schrittweise nähern, da die auftretenden Begriffe erfahrungsgemäß schwieriger zu verinnerlichen sind als jene der anderen Berechnungsmodelle. Wir beginnen mit dem Prinzip der *primitiven Rekursion* und führen erst im Anschluss daran die Menge der *primitiv-rekursiven Funktionen* ein.



Definition 6.7 (Primitive Rekursion)

Mit $g : \mathbb{N} \rightarrow \mathbb{N}$ und $h : \mathbb{N}^3 \rightarrow \mathbb{N}$ seien zwei Funktionen über \mathbb{N} gegeben. Eine Funktion $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ ist nach dem Schema der *primitiven Rekursion* aufgebaut, wenn sie die folgende Form besitzt:

$$f(m, n) = \begin{cases} g(n) & \text{falls } m = 0 \\ h(f(m-1, n), m-1, n) & \text{falls } m > 0 \end{cases}$$

Ein genauer Blick auf das Rekursionsschema zeigt, dass der Funktionswert f in einer Schleife berechnet wird, in der m die Rolle der Schleifenvariablen übernimmt. Ist $m = 0$, so wird der Funktionswert über die Funktion $g(n)$ bestimmt. Ist $m > 0$, so wird der Funktionswert ermittelt, indem die Funktion h auf den berechneten Funktionswert $f(m-1, n)$ sowie die Parameter $m-1$ und n angewendet wird. Auf den ersten Blick mag die Definition willkürlich erscheinen, auf den zweiten Blick wird jedoch schnell deutlich, dass die Formel den Aufbau einer rekursiv implementierten Schleife widerspiegelt. Der Zusammenhang wird sichtbar, wenn die Formelfragmente, wie in Abbildung 6.17 getan, in eine programmähnliche Form gebracht werden.

Das Schema der primitiven Rekursion ist stark genug, um alle üblichen Arithmetikoperationen auszudrücken. Am einfachsten gelingt die Umsetzung in eine primitiv-rekursive Darstellung, wenn wir in zwei Schritten vorgehen. Zunächst implementieren wir die Operationen mit einem

primrek.c

```

int f(m,n)
{
    if (m == 0) {
        // stuff
        return g(n)
    } else {
        // recursive call
        tmp = f(m-1,n);

        // stuff
        return h(tmp,m-1,n);
    }
}

```

Abbildung 6.17: Viele rekursiv programmierte Funktionen sind nach dem Prinzip der primitiven Rekursion aufgebaut.

```

add.c
int add(m,n)
{
    if (m == 0) {
        return n;
    } else {
        return
            succ(add(m-1,n));
    }
}

mult.c
int mult(m,n)
{
    if (m == 0) {
        return 0;
    } else {
        return
            add(mult(m-1,n),n);
    }
}

pow.c
int pow(m,n)
{
    if (m == 0) {
        return 1;
    } else {
        return
            mult(pow(m-1,n),n);
    }
}

```

Programm, das die Form aus Abbildung 6.17 besitzt. Anschließend extrahieren wir die Formelanteile g und h und konstruieren daraus die gesuchte Darstellung.

Basierend auf den Implementierungen aus Abbildung 6.18 erhalten wir für die Addition, die Multiplikation und die Potenzierung die folgenden Ergebnisse:

$$\text{add}(m,n) = \begin{cases} n & \text{falls } m = 0 \\ \text{succ}(\text{add}(m-1,n)) & \text{falls } m > 0 \end{cases} \quad (6.6)$$

$$\text{mult}(m,n) = \begin{cases} 0 & \text{falls } m = 0 \\ \text{add}(\text{mult}(m-1,n),n) & \text{falls } m > 0 \end{cases} \quad (6.7)$$

$$\text{pow}(m,n) = \begin{cases} 1 & \text{falls } m = 0 \\ \text{mult}(\text{pow}(m-1,n),n) & \text{falls } m > 0 \end{cases} \quad (6.8)$$

Nachdem das Grundschema der primitiven Rekursion eingeführt ist, können wir die *primitiv-rekursiven Funktionen* formal erklären:



Definition 6.8 (Primitiv-rekursive Funktionen)

- Die Nullfunktion $f(n) := 0$ ist primitiv-rekursiv.
- Die Nachfolgerfunktion $\text{succ}(n) := n + 1$ ist primitiv-rekursiv.
- Die Projektion $p_i^n(x_1, \dots, x_n) := x_i$ ist primitiv-rekursiv.
- Sind $h : \mathbb{N}^k \rightarrow \mathbb{N}$ und $g_1, \dots, g_k : \mathbb{N}^n \rightarrow \mathbb{N}$ primitiv-rekursiv, dann ist es auch $h(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$.
- Sind $g : \mathbb{N}^n \rightarrow \mathbb{N}$ und $h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ primitiv-rekursiv, dann ist es auch $f(m, x_1, \dots, x_n)$ mit

$$f(0, x_1, \dots, x_n) = g(x_1, \dots, x_n), \\ f(m+1, x_1, \dots, x_n) = h(f(m, x_1, \dots, x_n), m, x_1, \dots, x_n).$$

Abbildung 6.18: Aus der gewählten Programmstruktur lässt sich die primitiv-rekursive Definition der implementierten arithmetischen Operationen sofort ableiten.

Die Definition folgt dem induktiven Schema aus Abschnitt 2.4. Die ersten drei Regeln legen die elementaren primitiven Funktionen fest; natürlich sind dies die Nullfunktion, die Nachfolgerfunktion und die Projektion. Die letzten beiden Regeln geben an, wie sich aus bereits bekannten primitiv-rekursiven Funktionen weitere konstruieren lassen. Die erste Regel erlaubt uns, primitiv-rekursive Funktionen als Parameter einzusetzen (*Komposition*), die zweite besagt, dass die Menge bez. primitiver Rekursion abgeschlossen ist. Das zum Einsatz kommende

■ Lineare Rekursion

factorial.java

```
public static factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

■ Berechnungsbaum für $n = 4$

```

graph TD
    A[factorial(4)] --> B[factorial(3)]
    B --> C[factorial(2)]
    C --> D[factorial(1)]
    D --> E[factorial(0)]
  
```

■ Verzweigende Rekursion

fibonacci.java

```
public static fibo(int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fibo(n-1) + fibo(n-2);
    }
}
```

■ Berechnungsbaum für $n = 6$

```

graph TD
    A[fibo(6)] --> B[fibo(5)]
    A --> C[fibo(4)]
    B --> D[fibo(4)]
    B --> E[fibo(3)]
    C --> F[fibo(3)]
    C --> G[fibo(2)]
    D --> H[fibo(2)]
    D --> I[fibo(1)]
    E --> J[fibo(2)]
    E --> K[fibo(1)]
  
```

Abbildung 6.19: Lineare Rekursion und verzweigende Rekursion im Vergleich

Bildungsschema ist eine verallgemeinerte Variante der primitiven Rekursion aus Definition 6.7.

Legen wir die eingeführten Bildungsregeln streng aus, so entsprechen die Funktionen (6.6) bis (6.8) nicht der in Definition 6.8 festgelegten Struktur. Durch den geschickten Einsatz der Projektionsfunktion können wir diese jedoch leicht in eine entsprechende Form bringen:

$$\begin{aligned}
 \text{add}(0, n) &= p_1^1(n), \\
 \text{add}(m+1, n) &= \text{succ}(p_1^3(\text{add}(m, n), m, n)) \\
 \text{mult}(0, n) &= 0, \\
 \text{mult}(m+1, n) &= \text{add}(p_1^3(\text{mult}(m, n), m, n), p_3^3(\text{mult}(m, n), m, n)) \\
 \text{pow}(0, n) &= \text{succ}(0), \\
 \text{pow}(m+1, n) &= \text{mult}(p_1^3(\text{pow}(m, n), m, n), p_3^3(\text{pow}(m, n), m, n))
 \end{aligned}$$

Die primitive Rekursion ist nur ein Rekursionsschema von vielen. Sie gehört in die größere Gruppe der *linearen Rekursionen*, in der sich z. B. auch die *Kopfrekursion (head recursion)* und die *Endrekursion (tail recursion)* befinden. Von einer linearen Rekursion sprechen wir immer dann, wenn in jeder Rekursionsebene höchstens ein weiterer rekursiver Aufruf initiiert wird. Eine Kopf- bzw. eine Endrekursion liegt vor, wenn der Selbstaufruf die erste bzw. die letzte Aktion nach dem Basisfalltest ist. Hinter dem primitiven Rekursionsschema verbirgt sich somit eine spezielle Kopfrekursion, die den Wert des ersten Parameters in jedem neuen Aufruf um eins verringert.

Die lineare Rekursion verdankt ihren Namen der Eigenschaft, dass jeder Aufruf zu einem Berechnungsbaum in Form einer linearen Kette führt (vgl. Abbildung 6.19 links). Lassen wir dagegen mehrere, nacheinander ausgeführte Aufrufe zu, so entsteht eine Baumstruktur und wir reden von einer *verzweigenden Rekursion* (vgl. Abbildung 6.19 rechts). Sind die rekursiven Aufrufe derart miteinander verknüpft, dass das Ergebnis eines rekursiven Aufrufs als Parameter für den nächsten Aufruf dient, so sprechen wir von einer *verschachtelten Rekursion*. Eine *wechselseitige Rekursion* liegt vor, wenn sich mehrere Funktionen gegenseitig aufrufen.

David Hilbert äußerte im Jahr 1926 die Vermutung, dass alle berechenbaren Funktionen primitiv-rekursiv sind [47]. In diesem Sinne müssten sich alle Rekursionstypen auf die primitive Rekursion reduzieren lassen. Hilberts Annahme wurde durch die Arbeit von Wilhelm Ackermann widerlegt [1]. Mit der Ackermann-Funktion präsentierte er eine Funktion, die nicht primitiv-rekursiv ist, aber mithilfe verschachtelter Rekursionsaufrufe berechnet werden kann. Die Ackermann-Funktion ist uns in diesem Buch bereits mehrfach begegnet, wenn auch nicht in der im Jahr 1928 publizierten Form.

Die Projektionsfunktion p_i^j ist ein wertvolles Hilfsmittel, um primitiv-rekursive Funktionen flexibel umzuformen; weiter oben haben wir sie z. B. dafür eingesetzt, um aus der Parameterliste $\text{pow}(m, n), m, n$ den ersten und den dritten Parameter auszuwählen und an die zweistellige Funktion mult weiterzugeben:

$$\text{mult}(p_1^3(\text{pow}(m, n), m, n), p_3^3(\text{pow}(m, n), m, n))$$

Primitiv-rekursive Funktionen besitzen eine Eigenschaft, die an die Diskussion der Loop-Sprache erinnert: Sie sind allesamt total. Mit einem einfachen induktiven Argument ist die Gültigkeit dieser Aussage leicht einzusehen. Zunächst sind die elementaren primitiv-rekursiven Funktionen offensichtlich total. Des Weiteren lassen die beiden Bildungsregeln (Komposition und primitive Rekursion) erneut totale Funktionen entstehen.

Wie Sie vielleicht schon vermuten, gehen die Gemeinsamkeiten zwischen primitiv-rekursiven Funktionen und Loop-berechenbaren Funktionen noch deutlich weiter. In der Tat lässt sich durch eine geeignete Transformation jede Loop-berechenbare Funktion primitiv-rekursiv formulieren und umgekehrt jede primitiv-rekursive Funktion mithilfe eines Loop-Programms berechnen. Kurzum: Beide Berechnungsmodelle sind äquivalent.

Wir wollen nun genauer betrachten, wie die entsprechenden Transformationen aussehen müssen. Im Folgenden sei P ein Loop-Programm, in dem die Variablen x_1, \dots, x_n vorkommen. Mit a_1, \dots, a_n bezeichnen wir eine beliebige Anfangsbelegung von x_1, \dots, x_n und mit b_1, \dots, b_n die Endbelegung, die sich nach der Ausführung von P einstellt. Wir werden nun herausarbeiten, wie sich P in eine primitiv-rekursive Funktion

$$f_P : \mathbb{N} \rightarrow \mathbb{N}$$

mit der folgenden Eigenschaft transformieren lässt:

$$f_P(\pi(a_0, \dots, a_n)) = \pi(b_0, \dots, b_n)$$

π ist eine beliebige Funktion, die \mathbb{N}^n bijektiv auf \mathbb{N} abbildet, und dient dazu, den Inhalt des kompletten Variablenatzes x_1, \dots, x_n eineindeutig in eine einzige natürliche Zahl hineinzucodieren. Eines ähnlichen Kunstgriffs haben wir uns bereits in Abschnitt 6.1.1 im Zusammenhang mit der Loop-Implementierung eines Stapelspeichers bedient.

Da π eine bijektive Abbildung ist, lässt sich die Zuordnung umkehren. Ausgehend von π^{-1} definieren wir n Umkehrfunktionen

$$\pi_1^{-1} : \mathbb{N} \rightarrow \mathbb{N}, \dots, \pi_n^{-1} : \mathbb{N} \rightarrow \mathbb{N}$$

mit der Eigenschaft

$$\pi_i^{-1}(\pi(x_1, \dots, x_n)) = x_i$$

Über die Funktion π_i^{-1} lässt sich der Wert von x_i aus der Codierung $\pi(x_1, \dots, x_n)$ zurückberechnen (vgl. Abbildungen 6.20 und 6.21). Dass die Funktionen π, π_i^{-1} existieren, ist nach den bisher in diesem Buch errungenen Erkenntnissen selbstverständlich. Dass sie sich primitiv-rekursiv berechnen lassen, dagegen nicht. Im Übungsteil auf Seite 334 werden wir herausarbeiten, dass tatsächlich primitiv-rekursive Funktionen π und π_i^{-1} mit der gesuchten Eigenschaft existieren.

Die gesuchte Funktion f_P lässt sich induktiv aus dem Aufbau eines Loop-Programms P ableiten. Wir unterscheiden 5 Fälle:

- Fall 1: P ist von der Form $x_i := 0$

Wir setzen

$$f_P(\pi(a_0, \dots, a_n)) := \pi(a_0, \dots, a_{i-1}, 0, a_{i+1}, \dots, a_n)$$

- Fall 2: P ist von der Form $x_i := x_j$

Wir setzen

$$f_P(\pi(a_0, \dots, a_n)) := \pi(a_0, \dots, a_{i-1}, a_j, a_{i+1}, \dots, a_n)$$

- Fall 3: P ist von der Form $x_i := \text{succ}(x_j)$ oder $x_i := \text{pred}(x_j)$

Wir setzen

$$f_P(\pi(a_0, \dots, a_n)) := \pi(a_0, \dots, a_{i-1}, a_j \pm 1, a_{i+1}, \dots, a_n)$$

- Fall 4: P ist von der Form $P' ; P''$

Für die Teilausdrücke P' und P'' existieren primitiv-rekursive Funktionen $f_{P'}$ und $f_{P''}$. Wir setzen

$$f_P(\pi(a_0, \dots, a_n)) := f_{P''}(f_{P'}(\pi(a_0, \dots, a_n)))$$

- Fall 5: P ist von der Form $\text{loop } x_i \text{ do } P' \text{ end}$

Für den Teilausdruck P' ist eine primitiv-rekursive Funktion $f_{P'}$ bereits bekannt. Wir setzen

$$f_P(\pi(a_0, \dots, a_n)) := f_{\text{loop}}(a_i, \pi(a_0, \dots, a_n))$$

mit

$$f_{\text{loop}}(0, x) := x$$

$$f_{\text{loop}}(n+1, x) := f_{P'}(f_{\text{loop}}(n, x))$$

Die Hilfsfunktion f_{loop} ist offenbar primitiv-rekursiv und damit auch die konstruierte Funktion f_P .

■ Codierungsschema

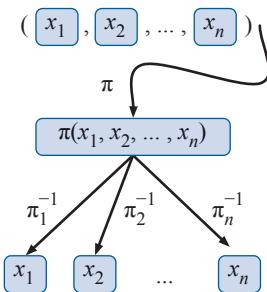


Abbildung 6.20: Mithilfe der Funktion π kann der komplette Variablenzustand x_1, \dots, x_n eines Loop-Programms in eine einzige natürliche Zahl hineincodiert werden.

■ Wertetabelle von π

	$x = 2$	$x = 3$	$x = 4$
$y = 1$	8	13	19
$y = 2$	12	18	25
$y = 3$	17	24	32
	⋮	⋮	⋮

■ Codierung

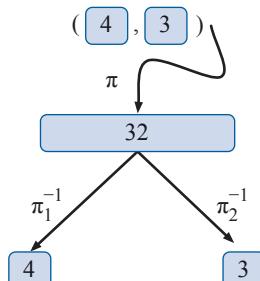


Abbildung 6.21: Demonstration des Codierungsschemas am Beispiel des Zahlentupels (4, 3)

■ Komposition

composition.loop

```

1  y1 := g1(x1, ..., xn);
2  y2 := g2(x1, ..., xn);
3  ...
4  yk := gk(x1, ..., xn);
5  x0 := h(y1, y2, ..., yk)

```

■ Primitive Rekursion

primrek.loop

```

1  y := 0;
2  x0 := g(x1, ..., xn);
3
4  loop m do
5    y := succ(y);
6    x0 := h(x0, y, x1, ..., xn);
7  end

```

Abbildung 6.22: Der rekursive Aufbau primitiv-rekursiver Funktionen wird durch zwei Bildungsschemata bestimmt: Komposition und primitive Rekursion. Beide lassen sich innerhalb der Loop-Sprache formulieren.

Auch die umgekehrte Transformation ist möglich, d.h., wir können jede primitiv-rekursive Funktion f mithilfe eines Loop-Programms berechnen. Zunächst halten wir fest, dass die drei primitiv-rekursiven Elementarfunktionen (Nullfunktion, Nachfolgerfunktion, Projektion) offensichtlich Loop-berechenbar sind. Damit verbleibt die Aufgabe, die Loop-Berechenbarkeit für zwei weitere Fälle zu zeigen:

■ Fall 1: f ist von der Form

$$f(x_1, \dots, x_n) = h(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$$

Abbildung 6.22 (oben) zeigt, wie sich f berechnen lässt.

■ Fall 2: f ist von der Form

$$\begin{aligned} f(0, x_1, \dots, x_n) &= g(x_1, \dots, x_n), \\ f(m+1, x_1, \dots, x_n) &= h(f(m, x_1, \dots, x_n), m, x_1, \dots, x_n) \end{aligned}$$

Abbildung 6.22 (unten) zeigt, wie sich f berechnen lässt.

Der folgende Satz trägt die Früchte unserer Arbeit zusammen:

 **Satz 6.5**

Die Klasse der primitiv-rekursiven Funktionen stimmt mit der Klasse der Loop-berechenbaren Funktionen überein.

μ -Rekursion

Die in Satz 6.5 manifestierte Äquivalenz hat zur Folge, dass sich sämtliche Limitierungen der Loop-Sprache mit einem Schlag auf die Klasse der primitiv-rekursiven Funktionen übertragen. Zum einen ist es unmöglich, partielle primitiv-rekursive Funktionen zu definieren, zum anderen existieren totale Funktionen wie die Ackermann-Funktion, die sich nicht primitiv-rekursiv formulieren lassen.

Abhilfe schafft eine Erweiterung des Rekursionsschemas, die als μ -Rekursion bezeichnet wird. Ausgangspunkt ist eine $n+1$ -stellige Funktion $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$, die durch die Anwendung des μ -Operators wie folgt auf eine n -stellige Funktion reduziert wird:

$$(\mu f)(x_1, \dots, x_n) := \min \left\{ m \mid \begin{array}{l} f(m, x_1, \dots, x_n) = 0 \\ \text{und für alle } k < m \text{ ist} \\ f(k, x_1, \dots, x_n) \neq \perp \end{array} \right\} \quad (6.9)$$

Beachten Sie, dass die Menge auf der rechte Seite von Gleichung (6.9) zur leeren Menge degradieren kann. In diesem Fall ist kein minimales Element vorhanden und der Funktionswert $(\mu f)(x_1, \dots, x_n)$ per Definition gleich \perp . Wir verwenden den μ -Operator als Grundlage, um die Klasse der μ -rekursiven Funktionen zu definieren:



Definition 6.9 (μ -rekursive Funktionen)

Die Klasse der μ -rekursiven Funktionen ist die kleinste Klasse von Funktionen, die alle primitiv-rekursiven Funktionen enthält und außerdem unter der Anwendung des μ -Operators abgeschlossen ist.

Für die Klasse der μ -rekursiven Funktionen existiert eine Analogie zu Satz 6.5. Wir werden zeigen, dass jede While-berechenbare Funktion μ -rekursiv ist und umgekehrt.

Hierzu sei mit P ein beliebiges While-Programm gegeben. Genau wie oben konstruieren wir hieraus eine Funktion $f_P : \mathbb{N} \rightarrow \mathbb{N}$ mit der Eigenschaft

$$f_P(\pi(a_0, \dots, a_n)) = \pi(b_0, \dots, b_n)$$

Die Variablenmengen a_1, \dots, a_n und b_1, \dots, b_n bezeichnen erneut die Anfangs- und die Endbelegung der Programmvariablen. Die Funktion f_P lässt sich wieder induktiv aus der Programmstruktur von P ableiten und wird μ -rekursiv sein.

Für alle Elemente der While-Sprache, die auch in der Loop-Sprache enthalten sind, können wir die weiter oben durchgeföhrten Beweisschritte eins zu eins übernehmen. Es bleibt, den Beweis für Programme P der Form `while x_i do P' end` zu föhren. In diesem Fall wird das Teilloop P' so lange ausgeführt, bis x_i den Wert 0 annimmt und P' in allen vorangegangenen Iterationen terminiert. Definieren wir die (μ -rekursive) Hilfsfunktion $h(m, x)$ als

$$h(m, x) := \underbrace{f_{P'}(f_{P'}(\dots(f_{P'}(x)\dots))),}_{m-\text{mal}}$$

so lässt sich die Anzahl der Iterationen einer (terminierenden) While-Schleife über die Formel

$$\min\{m \mid \pi_i^{-1}(h(m, x)) = 0\} \quad (6.10)$$

berechnen. Mithilfe des μ -Operators können wir die Formel (6.10) in

$$(\mu(\pi_i^{-1} \circ h))(x)$$

```

μ-recursion.while

y := f(0,x1,...,xn);
while y ≠ 0 do
  x0 := succ(x0);
  y := f(x0,x1,...,xn);
end

```

Abbildung 6.23: Die While-Schleife ist ausdrucksstark genug, um den μ -Operator zu simulieren.

umformulieren und erhalten mit

$$f_P(x) = h((\mu(\pi_i^{-1} \circ h))(x), x)$$

die von uns gesuchte Funktion.

Damit entpuppt sich die μ -Rekursion als mächtig genug, um das While-Konstrukt in allen seinen Facetten zu simulieren.

Umgekehrt lässt sich jede μ -rekursive Funktion mithilfe eines While-Programms berechnen. Auch hier können wir sämtliche Beweisschritte übernehmen, die sich auf primitiv-rekursive Formeln beziehen, und müssen nur noch zeigen, dass sich der μ -Operator auf ein äquivalentes While-Konstrukt abbilden lässt.

Hierzu sei mit $f(m, x_1, \dots, x_n)$ eine beliebige μ -rekursive Funktion gegeben. Unter der Induktionsannahme, dass $f(m, x_1, \dots, x_n)$ While-berechenbar ist, können wir die Funktion

$$(\mu f)(x_1, \dots, x_n) := \min \left\{ m \mid \begin{array}{l} f(m, x_1, \dots, x_n) = 0 \\ \text{und für alle } k < m \text{ ist} \\ f(k, x_1, \dots, x_n) \neq \perp \end{array} \right\}$$

wie in Abbildung 6.23 gezeigt, berechnen. Damit sind wir am Ziel und haben den folgenden Satz bewiesen:

Satz 6.6

Die Klasse der μ -rekursiven Funktionen stimmt mit der Klasse der While-berechenbaren Funktionen überein.

Erneut können wir mit einem Schlag auf die Erkenntnisse zurückgreifen, die wir in der Diskussion über die While-Sprache gewonnen haben. Erinnern Sie sich noch an Satz 6.3 – den Satz von Kleene? Dieser besagte, dass sich jede While-berechenbare Funktion durch ein Programm mit nur einer While-Schleife berechnen lässt. Drücken wir die gewonne Erkenntnis in der Nomenklatur der μ -rekursiven Funktionen aus, so erhalten wir auf direktem Weg das folgende Ergebnis:

Korollar 6.2

Jede n -stellige μ -rekursive Funktion $f : \mathbb{N}^n \rightarrow \mathbb{N}$ lässt sich in der Form

$$f(x_1, \dots, x_n) = g(x_1, \dots, x_n, (\mu h)(x_1, \dots, x_n))$$

berechnen, wobei h und g primitiv-rekursiv sind.

6.1.5 Turing-Maschinen

Die Turing-Maschine ist das älteste und gleichzeitig am häufigsten bemühte Modell, um den Berechenbarkeitsbegriff formal zu erfassen. Mit seinem im Jahr 1936 vorgestellten Begriffsgerüst gelingt Turing eine bemerkenswerte Gratwanderung. Zum einen erfüllt die Turing-Maschine in jeder Hinsicht die Anforderungen eines formalen Modells, so dass sie mathematisch präzise Aussagen über den Berechenbarkeitsbegriff erlaubt. Zum anderen ist sie von einer inneren Einfachheit und Klarheit geprägt, die einen überraschend intuitiven Zugang zu dieser komplexen Materie eröffnet. Im Gegensatz zu rein mathematischen Ansätzen, zu denen z. B. der zeitgleich von Alonzo Church entwickelte *Lambda-Kalkül* [18, 19] und die in Abschnitt 6.1.4 vorgestellte Theorie der primitiv-rekursiven Funktionen gehören, erscheint die Turing-Maschine zum Anfassen nah.

6.1.5.1 Einband-Turing-Maschinen

In seiner Originalarbeit motivierte Turing die Konzeption seines Maschinenmodells wie folgt:

„Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child’s arithmetic book.“

Das Zitat zeigt die Unbefangenheit, die sich durch Turings gesamte Arbeit zieht. Er startete seine Überlegungen über die Berechenbarkeit mit dem, was er seit seiner Kindheit zum Rechnen verwendete: einem leeren Stück karierten Papier. Unmittelbar danach nahm Turing dann doch eine erste Abstraktion vor. Er sah, dass die zweidimensionale Gestalt des Rechenpapiers im Grunde genommen keine Rolle spielt. Alle Berechnungen, die wir per Hand auf Papier durchführen können, sind auch auf einem eindimensionalen Band möglich – wenngleich nicht immer mit der gleichen Eleganz (vgl. Abbildung 6.24).

„[...] I think that it is agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on tape divided into squares.“

Turing lässt weitere Annahmen folgen. Zunächst geht er davon aus, dass es nur endlich viele Symbole gibt, mit denen die Felder seines Bandes gefüllt werden können. Er ging außerdem davon aus, dass sich das

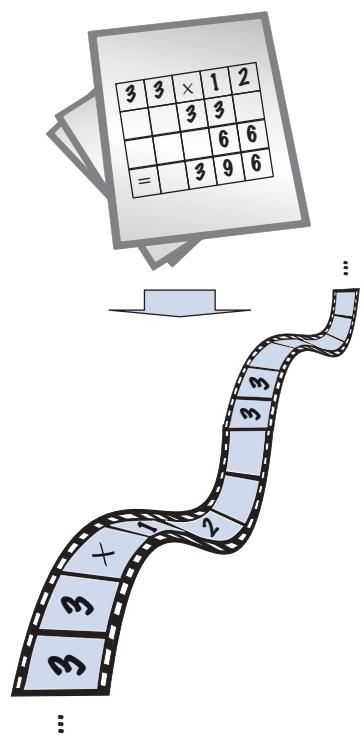


Abbildung 6.24: Der britische Mathematiker Alan Turing machte ein unendlich langes Band zur Grundlage seines Rechenmodells. Aus mathematischer Sicht ist ein solches ausreichend, da sich alle Rechenoperationen, die auf einem zweidimensionalen Blatt durchgeführt werden können, auf einem eindimensionalen Band nachvollziehen lassen.

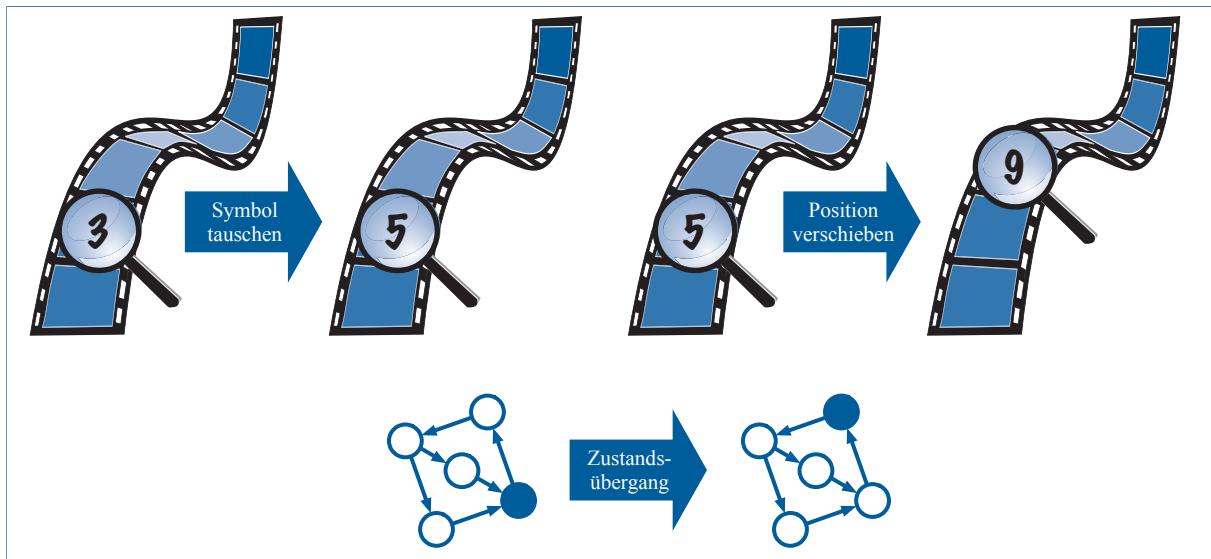


Abbildung 6.25: Turing definierte wenige primitive Elementaroperationen, aus denen komplexe Berechnungen erwachsen. In jedem Bearbeitungsschritt kann eine Turing-Maschine das aktuell betrachtete Symbol durch ein anderes ersetzen und das Betrachtungsfenster (*observed square*) verschieben. Die ausgeführten Aktionen gehen mit einem potenziellen Wechsel des inneren Zustands (*state of mind*) einher.

menschliche Gehirn im Zuge einer Berechnung zu jedem Zeitpunkt in einem von endlich vielen Zuständen befindet.

„We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. [...] We will also suppose that the number of states of mind which will be taken into account is finite.“

Anschließend definiert Turing eine Menge von Elementaroperationen, aus denen sich komplexe Berechnungen zusammensetzen. Diese erlauben, das Symbol des aktuell betrachteten Felds auszutauschen und die Aufmerksamkeit auf eines der Nachbarfelder zu lenken:

„The simple operations must therefore include: (a) Changes of the symbol on one of the observed squares. (b) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.“

Beide Aktionen werden durch einen möglichen Wechsel des internen Zustands begleitet:

„It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following: (A) A possible change (a) of symbol together with a possible change of state of mind. (B) A possible change (b) of observed squares, together with a possible change of state of mind.“

Abbildung 6.25 fasst die von Turing eingeführten Elementaroperationen bildlich zusammen.

Damit ist es an der Zeit, den Begriff der Turing-Maschine formal zu präzisieren. Wir orientieren uns dabei an der Nomenklatur, die wir für die Beschreibung endlicher Automaten in Kapitel 5 erfolgreich eingesetzt haben.



Alan Mathison Turing (1912 – 1954)

Definition 6.10 (Turing-Maschine)

Eine (deterministische) Turing-Maschine, kurz TM, ist ein 7-Tupel $(S, \Sigma, \Pi, \delta, s_0, \square, E)$. Sie besteht aus

- der endlichen Zustandsmenge S ,
- dem endlichen Eingabealphabet Σ ,
- dem Bandalphabet Π mit $\Pi \supset \Sigma$,
- der Zustandsübergangsfunktion $\delta : S \times \Pi \rightarrow S \times \Pi \times \{\leftarrow, \rightarrow\}$,
- dem Startzustand s_0 ,
- dem Blank-Symbol $\square \in \Pi \setminus \Sigma$,
- der Menge der Endzustände (Finalzustände) $E \subseteq S$.

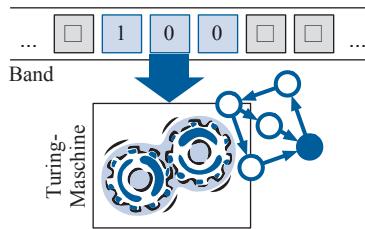
Zu Beginn befindet sich jede Turing-Maschine in ihrem Startzustand s_0 . Das zu verarbeitende Eingabewort $\omega \in \Sigma^*$ steht bereits auf dem Band und der virtuelle Schreib-Lese-Kopf ist über dem ersten Eingabezeichen positioniert. Beachten Sie, dass die Maschine über ein Band verfügt, dass sich in beide Richtungen unendlich weit ausbreitet. Um die noch freien Felder von dem Eingabewort unterscheiden zu können, vereinbaren wir, dass alle Felder links und rechts der Eingabesequenz mit dem Blank-Symbol \square beschrieben sind.

Ausgehend von dieser Initialkonfiguration führt eine Turing-Maschine die folgenden Aktionen durch (vgl. Abbildung 6.26):

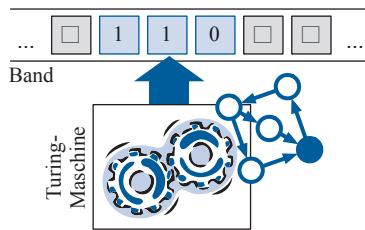
Alan Turing wurde am 23. Juni 1912 in London-Paddington geboren. Bereits in seiner frühen Jugend wurde seine außerordentliche mathematische Begabung sichtbar, genauso wie sein Unvermögen, sich gesellschaftlichen Normen und staatlichen Autoritäten zu beugen. Turings wissenschaftliches Vermächtnis ist beachtlich. Neben seinen grundlegenden Beiträgen zur Theorie der Berechenbarkeit lieferte er während des zweiten Weltkriegs wertvolle Erkenntnisse im Bereich der Kryptoanalyse. Im Jahr 1950 schlug er mit dem *Turing-Test* ein Verfahren vor, mit dem sich der Intelligenzbegriff auf Maschinen übertragen lässt [99].

Im Jahr 1952 sollte Turings Karriere ein abruptes Ende finden. Als die Polizei sein Haus nach einem Einbruch untersuchte, gestand er eine homosexuelle Beziehung ein. Das prüde England der Fünfzigerjahre reagierte erbarmungslos und sprach Turing in einem Strafverfahren der sexuellen Perversion schuldig. Die angeordnete Zwangstherapien machten aus ihm einen gebrochenen Mann. Zwei Jahre später, am 7. Juni 1954, wurde Alan Turing im Alter von 42 Jahren neben den Resten eines vergifteten Apfels tot aufgefunden.

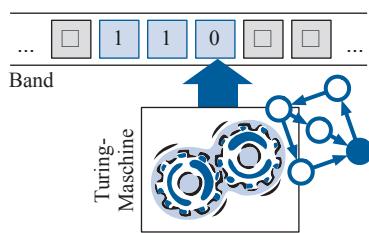
- Zeichen lesen



- Zeichen schreiben



- Kopf bewegen



- Zustand wechseln

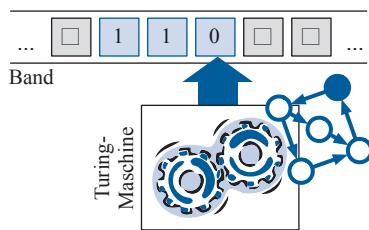


Abbildung 6.26: Die Turing-Maschine in Aktion

- Der Schreib-Lese-Kopf liest das aktuelle Bandzeichen σ ein

- Der Funktionswert $(s', \sigma', r) = \delta(s, \sigma)$ wird berechnet

- Das Bandzeichen wird durch σ' ersetzt

- Der Kopf wird nach links ($r = ' \leftarrow '$) oder rechts ($r = ' \rightarrow '$) bewegt

- Der Folgezustand s' wird eingenommen

Hierin bezeichnet s den Zustand, in dem sich die Turing-Maschine aktuell befindet. Die Übergangsfunktion δ ist eine partielle Funktion und muss deshalb nicht für alle Eingabekombinationen aus der Menge $S \times \Pi$ definiert sein. In der Konsequenz kann die Turing-Maschine (gewolltermaßen) in einen Zustand geraten, der keine weitere Aktion mehr zulässt. In diesem Fall bleibt die Maschine stehen und die Berechnung ist beendet.

Nachdem wir die Funktionsweise einer Turing-Maschine informell festgelegt haben, wollen wir das Gesagte in eine mathematisch präzise Form überführen. Wie schon im Falle des endlichen Automaten und seiner diversen Varianten spielen der Begriff der Konfiguration und die darauf definierte Übergangsrelation → die zentrale Rolle.



Definition 6.11 (Konfiguration (TM))

Sei $T = (S, \Sigma, \Pi, \delta, s_0, \square, E)$ eine beliebige Turing-Maschine. Jedes Tripel (v, s, ω) mit $v, \omega \in \Pi^+$ und $s \in S$ heißt eine *Konfiguration* von T . Die Übergangsrelation \rightarrow_T definieren wir wie folgt:

- Rechtsbewegung: $\delta(s, \sigma) = (s', \sigma', \rightarrow), \rho, \sigma \in \Pi$

$$(v\rho, s, \sigma\omega) \rightarrow_T \begin{cases} (v\rho\sigma', s', \omega) & \text{falls } \omega \neq \varepsilon \\ (v\rho\sigma', s', \square) & \text{falls } \omega = \varepsilon \end{cases}$$

- Linksbewegung: $\delta(s, \sigma) = (s', \sigma', \leftarrow), \rho, \sigma \in \Pi$

$$(v\rho, s, \sigma\omega) \rightarrow_T \begin{cases} (v, s', \rho\sigma'\omega) & \text{falls } v \neq \varepsilon \\ (\square, s', \rho\sigma'\omega) & \text{falls } v = \varepsilon \end{cases}$$

Informell gesprochen besitzen die Variablen v , ρ , s , σ und ω in dem Ausdruck $(v\rho, s, \sigma\omega)$ die folgende Bedeutung: $v\rho$ und $\sigma\omega$ bilden zusammen den Bandinhalt, wobei sich der Schreib-Lese-Kopf aktuell über dem Zeichen σ befindet. s ist der aktuelle Zustand der Turing-Maschine. In der Festlegung der Übergangsrelation müssen wir berücksichtigen,

dass sich der Schreib-Lese-Kopf beliebig nach links und rechts und damit insbesondere über die Grenzen des Eingabewortes hinweg bewegen darf. Unsere Definition trägt diesem Verhalten Rechnung, indem der Bandinhalt beim Überschreiten der Wortgrenze um ein Blank-Symbol verlängert wird.

Abbildung 6.27 demonstriert die Konfigurationsübergänge anhand eines konkreten Beispiels. Die Zustandsübergangsfunktion δ ist aus Gründen der Übersichtlichkeit in Form einer Tabelle definiert. Im unteren Teil der Abbildung sind die Konfigurationsübergänge dargestellt, die während der Bearbeitung des Eingabeworts $\omega = 111111$ entstehen. Die Übergangsfunktion δ ist so definiert, dass sich die Turing-Maschine im Startzustand s_0 zunächst nach rechts über alle vorgefundenen Einsen hinwegbewegt und das erste vorgefundene Blank-Symbol durch eine 1 ersetzt. Zeitgleich wechselt sie in den Zustand s_1 und beginnt, den Schreib-Lese-Kopf nach links zurückzubewegen. Sobald sich dieser links über das erste Eingabezeichen hinausschiebt, kehrt die Maschine mit einer Rechtsbewegung auf das erste Eingabezeichen zurück und stoppt im Terminalzustand s_2 . Insgesamt haben wir mit diesem ersten Beispiel eine Maschine kennen gelernt, die das Eingabewort $\omega = 1^n$ in das Ausgabewort $f(\omega) = 1^{n+1}$ überführt.

Analog zu den Begriffen der Loop-, While- und Goto-Berechenbarkeit sind wir jetzt in der Lage, den Begriff der *Turing-Berechenbarkeit* formal zu definieren:



Definition 6.12 (Turing-Berechenbarkeit)

Mit $f : \Sigma^* \rightarrow \Sigma^*$ sei eine beliebige partielle Funktion über Wörtern des Eingabealphabets Σ gegeben. f heißt *Turing-berechenbar*, falls eine Turing-Maschine $T = (S, \Sigma, \Pi, \delta, s_0, \square, E)$ mit den folgenden Eigenschaften existiert:

- T terminiert unter Eingabe von ω genau dann in einem Endzustand $s_e \in E$, wenn $f(\omega) \neq \perp$
- In diesem Fall gilt: $(\square, s_0, \omega) \xrightarrow{*} (\square^*, s_e, f(\omega)\square^*)$

Informell stellt sich der Begriff der Turing-Berechenbarkeit wie folgt dar: Um den Funktionswert $f(\omega)$ zu berechnen, wird zunächst die Zeichenfolge ω an einer beliebigen Stelle auf das Band geschrieben und der Schreib-Lese-Kopf auf das erste Zeichen positioniert. Anschließend wird die Maschine gestartet und die weiter oben im Detail beschriebenen Berechnungsschritte durchgeführt. Terminiert die Maschine in einem Endzustand, so können wir den Funktionswert $f(\omega)$ vom Band

Turing-Maschine

$$S = \{s_0, s_1, s_2\}$$

$$\Sigma = \{1\}$$

$$\Pi = \{1, \square\}$$

$$E = \{s_2\}$$

Übergangstabelle

	1	\square
s_0	$(s_0, 1, \rightarrow)$	$(s_1, 1, \leftarrow)$
s_1	$(s_1, 1, \leftarrow)$	$(s_2, \square, \rightarrow)$
s_2	—	—

Konfigurationsübergänge

\rightarrow	$(\square, s_0, 111111)$
\rightarrow	$(\square 1, s_0, 11111)$
\rightarrow	$(\square 11, s_0, 1111)$
\rightarrow	$(\square 111, s_0, 111)$
\rightarrow	$(\square 1111, s_0, 11)$
\rightarrow	$(\square 11111, s_0, 1)$
\rightarrow	$(\square 111111, s_0, \square)$
\rightarrow	$(\square 11111, s_1, 11)$
\rightarrow	$(\square 1111, s_1, 111)$
\rightarrow	$(\square 111, s_1, 1111)$
\rightarrow	$(\square 11, s_1, 11111)$
\rightarrow	$(\square 1, s_1, 111111)$
\rightarrow	$(\square, s_1, 1111111)$
\rightarrow	$(\square, s_1, \square 1111111)$
\rightarrow	$(\square\square, s_2, 1111111)$

Abbildung 6.27: Konfigurationsübergänge für das Eingabewort $\omega = 111111$. Die Bandposition, auf der sich der Schreib-Lese-Kopf aktuell befindet, ist farblich hervorgehoben.

■ Turing-Maschine

$$S = \{s_0, s_1\}$$

$$\Sigma = \{1\}$$

$$\Pi = \{1, \square\}$$

$$E = \{s_0, s_1\}$$

■ Übergangstabelle

1	\square
$(s_1, 1, \rightarrow)$	$(s_1, \square, \rightarrow)$
$(s_0, 1, \leftarrow)$	$(s_0, \square, \leftarrow)$

■ Konfigurationsübergänge

$$\begin{aligned} & (\square, s_0, 111111) \\ \rightarrow & (\square 1, s_1, 111111) \\ \rightarrow & (\square, s_0, 111111) \\ \rightarrow & (\square 1, s_1, 111111) \\ \rightarrow & (\square, s_0, 111111) \\ \rightarrow & (\square 1, s_1, 111111) \\ \rightarrow & (\square, s_0, 111111) \\ \rightarrow & (\square 1, s_1, 111111) \\ \rightarrow & (\square, s_0, 111111) \\ \rightarrow & (\square 1, s_1, 111111) \\ \rightarrow & (\square, s_0, 111111) \\ \rightarrow & (\square 1, s_1, 111111) \\ \rightarrow & (\square, s_0, 111111) \\ \dots & \end{aligned}$$

Abbildung 6.28: Die konstruierte Turing-Maschine terminiert für keine Eingabe. Sie berechnet mit $f(\omega) = \perp$ die an allen Stellen undefinierte Funktion. Der untere Teil der Abbildung zeigt die Konfigurationsübergänge für das Beispielwort $\omega = 111111$.

lesen. Der Schreib-Lese-Kopf steht in diesem Fall über dem ersten Zeichen von $f(\omega)$. Da in einem Konfigurationsübergang neue Blank-Symbole entstehen können, wird das Ergebnis in der Finalkonfiguration links und rechts von beliebig vielen Blank-Symbolen (\square^*) eingerahmt.

Zwei Fällen müssen wir unsere besondere Beachtung schenken. Zum einen ist es möglich, dass die Turing-Maschine zwar terminiert, aber keinen Endzustand erreicht. Zum anderen besteht die Möglichkeit, dass die Maschine in eine Endlosschleife gerät und niemals anhält (vgl. Abbildung 6.28). In beiden Fällen betrachten wir die Berechnung als gescheitert und weisen ihr den Funktionswert \perp zu. Turing-Maschinen sind damit auf natürliche Weise in der Lage, partielle Funktionen zu berechnen.

Vergleichen wir den Berechenbarkeitsbegriff mit jenen der Abschnitte 6.1.1 bis 6.1.2, so unterscheidet er sich in einem wesentlichen Punkt: Er ist für Funktionen über der Wortmenge Σ^* definiert und nicht, wie bisher, über der Menge der natürlichen Zahlen. Um den Berechenbarkeitsbegriff für unsere Zwecke nutzbar zu machen, müssen wir eine geeignete Codierung finden, die Zahlenwerte auf Wörter der Menge Σ^* abbildet. Zwei Codierungen sind in diesem Zusammenhang von besonderer Bedeutung:

■ Binäre Codierung

Die Ein- und Ausgabewerte werden im Binärformat auf das Band geschrieben (vgl. Abbildung 6.29 oben). Die Codierung entspricht jener, die in aktuellen Computersystemen zum Einsatz kommt. Interessant ist sie vor allem im Zusammenhang mit Komplexitätsuntersuchungen, da sich viele Ergebnisse direkt auf reale Rechnerarchitekturen übertragen lassen.

■ Unäre Codierung

Die Ein- und Ausgabewerte werden durch Einserfolgen entsprechender Länge repräsentiert (vgl. Abbildung 6.29 unten). Die unäre Codierung besitzt den Vorteil, dass sich viele Algorithmen besonders einfach in eine entsprechende Turing-Maschine übersetzen lassen. Für Komplexitätsbetrachtungen ist sie nicht geeignet, da bereits das Schreiben einer Zahl n einen linear steigenden Aufwand verursacht.

Um die folgenden Betrachtungen nicht unnötig zu verkomplizieren, werden wir auf die einfachere und für unsere Zwecke völlig ausreichende unäre Codierung zurückgreifen.

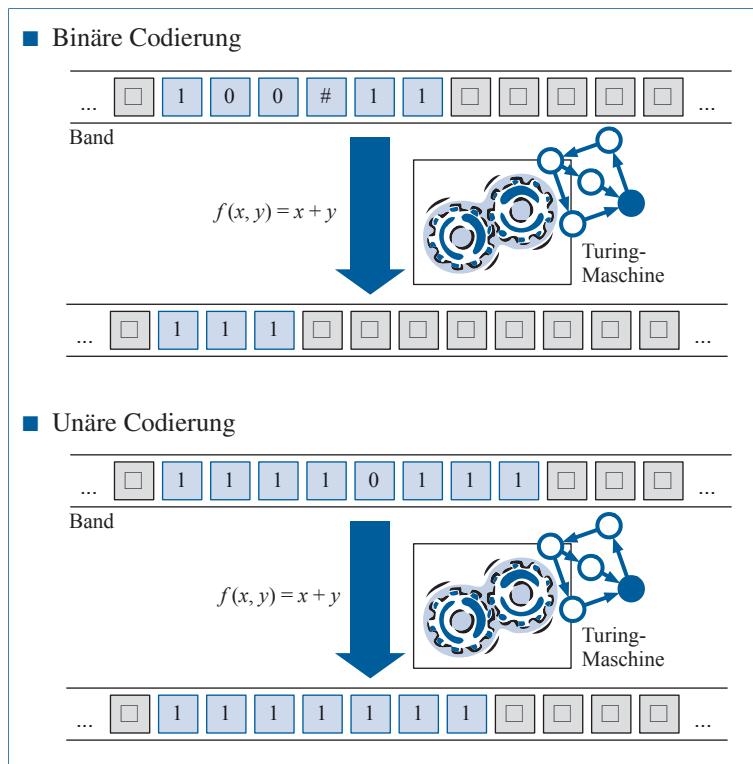


Abbildung 6.29: Die binäre und die unäre Codierung im Vergleich. Bei der binären Codierung werden die Eingabewerte als eine Folge von Nullen und Einsen auf das Eingabeband geschrieben und liegen damit in dem gleichen Zahlenformat vor, das auch in realen Rechnersystemen zum Einsatz kommt. Da sich mit m Bits 2^m verschiedene Binärmuster unterscheiden lassen, werden im Umkehrschluss $\lceil \log_2 n \rceil$ Bandstellen benötigt, um die Zahl n binär codiert auf das Eingabeband zu schreiben. Die unäre Codierung ist die weitaus primitivere Variante; sie stellt eine Zahl n dar, indem eine Folge von n Einsen hintereinander auf das Eingabeband geschrieben wird. Die Null kann dazu verwendet werden, um zwei Codierungsmuster voneinander zu trennen. Die unäre Codierung bietet den Vorteil, dass sich viele Algorithmen besonders einfach in entsprechende Turing-Maschinen übersetzen lassen. Aus diesem Grund werden wir im Folgenden häufig auf sie zurückgreifen, auch wenn sie unbestritten die geringere Praxisbedeutung besitzt.

Auf dieser Grundlage können wir die Funktionsweise der in Abbildung 6.27 eingeführten Turing-Maschine auch numerisch interpretieren: Sie berechnet die Nachfolgerfunktion $f(x) = \text{succ}(x)$. Weiter oben haben wir bereits angedeutet, wie die Maschine arbeitet. Über der ersten Eins startend bewegt die Maschine den Schreib-Lese-Kopf so weit nach rechts, bis das erste Blank-Symbol erscheint. Dieses wird durch eine Eins ersetzt und der unär codierte Eingabewert hierdurch um eins erhöht. Anschließend spult die Maschine das Band zurück, indem sie den Schreib-Lese-Kopf in die Ausgangsposition zurückbewegt.

Abbildung 6.30 zeigt die Konstruktion einer Turing-Maschine zur Berechnung der Vorgängerfunktion $f(x) = \text{pred}(x)$. Genau wie die erste Maschine bewegt sie den Schreib-Lese-Kopf an das Ende der Eingabesequenz. Sobald ein Blank-Symbol erscheint, erfolgt eine Linksbewegung und die vorgefundene Eins wird mit einem Blank-Symbol überschrieben. Jetzt steht der gesuchte Wert $x - 1$ auf dem Band und die Maschine bewegt den Schreib-Lese-Kopf auf seine ursprüngliche Position zurück.

■ Turing-Maschine

$$S = \{s_0, s_1, s_2, s_3\}$$

$$\Sigma = \{1\}$$

$$\Pi = \{1, \square\}$$

$$E = \{s_3\}$$

■ Übergangstabelle

	1	\square
s_0	$(s_0, 1, \rightarrow)$	$(s_1, \square, \leftarrow)$
s_1	$(s_2, \square, \leftarrow)$	$(s_3, \square, \rightarrow)$
s_2	$(s_2, 1, \leftarrow)$	$(s_3, \square, \rightarrow)$
s_3	—	—

■ Konfigurationsübergänge

	$(\square, s_0, 111111)$
\rightarrow	$(\square 1, s_0, 11111)$
\rightarrow	$(\square 11, s_0, 1111)$
\rightarrow	$(\square 111, s_0, 111)$
\rightarrow	$(\square 1111, s_0, 11)$
\rightarrow	$(\square 11111, s_0, 1)$
\rightarrow	$(\square 111111, s_0, \square)$
\rightarrow	$(\square 11111, s_1, 1\square)$
\rightarrow	$(\square 1111, s_2, 1\square\square)$
\rightarrow	$(\square 111, s_2, 1\square\square\square)$
\rightarrow	$(\square 11, s_2, 1\square\square\square\square)$
\rightarrow	$(\square 1, s_2, 1\square\square\square\square\square)$
\rightarrow	$(\square, s_2, 1\square\square\square\square\square\square)$
\rightarrow	$(\square\square, s_3, 1\square\square\square\square\square\square)$

Abbildung 6.30: Die dargestellte Turing-Maschine berechnet die Funktion $f(x) = \text{pred}(x)$.

Komplexere Operationen lassen sich nach demselben Schema realisieren. Als Beispiel zeigt Abbildung 6.31 eine Turing-Maschine zur Addition zweier Zahlen x und y . Zu Beginn werden beide Operanden unär codiert und zusammen mit einer separierenden Null auf das Band geschrieben. Um die Zahlen zu addieren, geht die Maschine in drei Schritten vor: Zunächst wird temporär der Wert $x + y + 1$ erzeugt, indem ausgehend von der Startposition die trennende Null gesucht und durch eine Eins ersetzt wird. Anschließend wird der Schreib-Lese-Kopf, genau wie im Falle der Berechnung von $\text{pred}(x)$, an das Ende bewegt und die letzte Eins gelöscht. Jetzt steht das gesuchte Ergebnis $x + y$ auf dem Band und die Maschine muss nur noch zurückgespult werden.

In Abbildung 6.32 wird eine weitere Turing-Maschine definiert, die in den nachfolgenden Betrachtungen noch eine wichtige Rolle spielen wird. Die Rede ist von der Move-Maschine, die das komplette Eingabewort um eine Stelle nach rechts verschiebt. Hierzu wird in jedem Schritt das aktuelle Bandzeichen eingelesen, mit dem Vorgängersymbol überschrieben und der Schreib-Lese-Kopf anschließend eine Stelle nach rechts bewegt. Der Vorgang wird so lange wiederholt, bis das Blank-Symbol eingelesen wird. In diesem Fall ist das gesamte Wort verschoben und die Maschine terminiert nach dem Rückspulen des Bands im Endzustand s_e . Beachten Sie, dass die Zustandsmenge der Move-Maschine maßgeblich durch die Anzahl der Elemente des Bandalphabets Π bestimmt wird. Damit sich die Maschine an das gelesene Vorgängersymbol „erinnern“ kann, müssen wir für jedes Element des Bandalphabets einen dedizierten Zustand definieren.

Die Move-Maschine macht sich eine häufig verwendete Erweiterung des Basismodells zu Nutze, die wir nicht unbeachtet übergehen wollen. Der Übergang $\delta(s_0, \square) = (s_2, \square, \circlearrowright)$ drückt aus, dass der Schreib-Lese-Kopf an seiner aktuellen Position verharren soll. Ein solches Verhalten ist im Basismodell nicht vorgesehen; hier wird der Kopf in jedem Schritt entweder nach links, oder nach rechts bewegt. Trotzdem handelt es sich um keine Erweiterung im eigentlichen Sinne, da wir das Verhalten sehr einfach im Basismodell simulieren können. Anstatt direkt in den Zustand s_2 überzugehen, lassen wir die Maschine zunächst in einen Zwischenzustand s' wechseln. Hierbei bewegen wir den Kopf nach links, ohne den Bandinhalt zu verändern. Anschließend erfolgt der Übergang nach s_2 , gekoppelt mit einer Rechtsbewegung. Über einen solchen Zwischenschritt lässt sich simulieren, dass sich der Schreib-Lese-Kopf während eines Zustandsübergangs scheinbar nicht bewegt. Damit können wir das neu hinzugekommene Symbol \circlearrowright bedenkenlos verwenden; es handelt sich lediglich um eine Schreiberleichterung und erfordert keine Anpassung des zugrunde liegenden Berechnungsmodells.

Im Folgenden wollen wir mehrere Erweiterungen der ursprünglichen Turing-Maschine diskutieren, mit denen sich viele Berechnungen deutlich bequemer durchführen lassen. Dabei werden wir die erstaunliche Beobachtung machen, dass keine zu einer echten Steigerung der Ausdrucksfähigkeit führen wird. Sämtliche Erweiterungen lassen sich im Basismodell simulieren.

6.1.5.2 Einseitig und linear beschränkte Turing-Maschinen

Einseitig beschränkte Turing-Maschinen verwenden ein Band, das sich nur in eine Richtung unendlich weit ausbreitet. Ohne Beschränkung der Allgemeinheit wollen wir von einem nach links beschränkten Band ausgehen und die Felder mit den natürlichen Zahlen nummerieren. Der Bandanfang besitzt den Index 0 und speichert das erste Zeichen der Eingabesequenz. Der Schreib-Lese-Kopf einer einseitig beschränkten Turing-Maschine kann sich nicht über das Bandende hinausbewegen. Eine angeforderte Linksbewegung wird ignoriert und der Schreib-Lese-Kopf verharrt in seiner Position.

Einseitig beschränkte Turing-Maschinen lassen sich durch das Basismodell simulieren, indem das Bandende durch ein dediziertes Symbol \diamond markiert und die Übergangsfunktion für jeden Zustand $s \in S$ um die Regel

$$\delta(s, \diamond) = (s, \diamond, \rightarrow)$$

ergänzt wird. Hierdurch wird der Schreib-Lese-Kopf auf die Startposition zurückbewegt, sobald der Bandanfang verlassen wird.

Die Umkehrung gilt ebenfalls, d. h., wir können jede Turing-Maschine durch eine einseitig beschränkte Turing-Maschine simulieren. Wir beginnen, indem wir den Bandanfang erneut mit dem dedizierten Symbol \diamond markieren. Anschließend bewegen wir den Schreib-Lese-Kopf nach rechts auf das erste Zeichen der Eingabe und starten die Maschine. Solange sich der Kopf rechts des ersten Eingabezeichens befindet, verläuft die Berechnung wie gehabt. Bewegt die Maschine den Schreib-Lese-Kopf jedoch über die linke Grenze hinaus, treffen wir also auf das vorher eingefügte Symbol \diamond , so müssen wir ein wenig Sonderarbeit leisten. Wir schaffen zunächst Platz für ein neues Zeichen, indem wir den gesamten Bandinhalt, analog zur Arbeitsweise der Move-Maschine aus Abbildung 6.32, um eine Stelle nach rechts verschieben. Anschließend führen wir die Berechnung in gewohnter Weise fort. Damit erweist sich die einseitige Beschränkung des Bandes als harmloser, als es der erste Blick vermuten lässt. Sie führt zu einem Automatenmodell, dessen Berechnungsstärke mit jener des Basismodells übereinstimmt.

Turing-Maschine

$$S = \{s_0, s_1, s_2, s_3, s_4\}$$

$$\Sigma = \{1, 0\}$$

$$\Pi = \{1, 0, \square\}$$

$$E = \{s_4\}$$

Übergangstabelle

	1	0	\square
s_0	$(s_0, 1, \rightarrow)$	$(s_1, 1, \rightarrow)$	—
s_1	$(s_1, 1, \rightarrow)$	—	$(s_2, \square, \leftarrow)$
s_2	$(s_3, \square, \leftarrow)$	—	$(s_4, \square, \rightarrow)$
s_3	$(s_3, 1, \leftarrow)$	—	$(s_4, \square, \rightarrow)$
s_4	—	—	—

Abbildung 6.31: Die Add-Maschine berechnet die Summe zweier unär codierter Operanden x und y .

Turing-Maschine

$$S = \{s_\sigma \mid \sigma \in \Sigma\} \cup \{s_0, s_1, s_2\}$$

$$\Pi = \Sigma \cup \{\square\}$$

$$E = \{s_2\}$$

Übergangstabelle

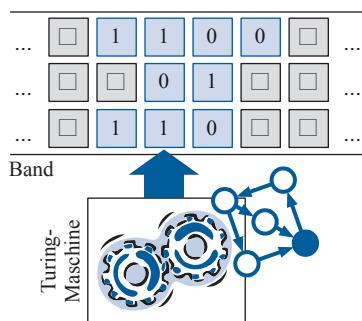
	σ'	\square
s_0	$(s_{\sigma'}, \square, \rightarrow)$	$(s_2, \square, \circlearrowleft)$
s_σ	$(s_{\sigma'}, \sigma, \rightarrow)$	$(s_1, \sigma, \leftarrow)$
s_1	$(s_1, \sigma', \leftarrow)$	$(s_2, \square, \rightarrow)$
s_2	—	—

Abbildung 6.32: Die Move-Maschine verschiebt den Bandinhalt um eine Stelle nach rechts.

■ Mehrspur-Turing-Maschine

$$\Sigma = \{1, 0\}$$

$$\Pi = \{1, 0, \square\}$$



■ Simulation im Basismodell

$$\Sigma = \{1, 0\}^3$$

$$\Pi = \{1, 0, \square\}^3$$

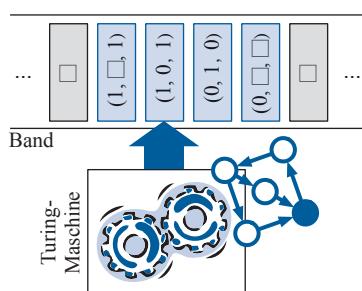


Abbildung 6.33: Eine Mehrspur-Turing-Maschine lässt sich durch die Erweiterung des Eingabe- und des Bandalphabets im Basismodell simulieren.

Anders stellt sich die Situation für *linear beschränkte Turing-Maschinen* dar. Einer solchen Maschine stehen für die Berechnung nur diejenigen Felder zur Verfügung, die zur Codierung der Eingabesequenz benötigt wurden. Mit anderen Worten: Es ist nicht erlaubt, den Schreib-Lese-Kopf nach links oder rechts über die Eingabesequenz hinauszubewegen. Dass diese Einschränkung die Berechnungsstärke massiv beeinflusst, liegt auf der Hand. So ist es unmöglich, eine Ausgabe zu produzieren, die länger ist als die Eingabe selbst. In Abschnitt 6.4 werden wir dieses Automatenmodell erneut aufgreifen und zeigen, dass es trotz seiner offensichtlichen Limitierungen dennoch eine große Rolle in der theoretischen Informatik spielt.

6.1.5.3 Mehrspur-Turing-Maschinen

Eine *k-Spur-Turing-Maschine* besteht aus einem Band, das in k separate Spuren unterteilt ist. Die einzelnen Spuren werden von fest aneinander gekoppelten Schreib-Lese-Köpfen angesprochen (vgl. Abbildung 6.33 oben). Ähnlich dem Prinzip, das konventionellen Festplattenlaufwerken zugrunde liegt, können sich die Köpfe alle gleichzeitig nach links oder rechts, jedoch nicht unabhängig voneinander bewegen.

Bei genauerer Betrachtung entpuppt sich das Konzept der Mehrspur-Turing-Maschine als eine eher marginale Erweiterung des Basismodells. Eine *k*-Spur-Turing-Maschine mit dem Eingabealphabet Σ und dem Bandalphabet Π lässt sich mit einer konventionellen Turing-Maschine simulieren, indem die Zeichen der k Spuren in ein Einzelzeichen hineincodiert werden. Um dies zu erreichen, ersetzen wir Σ ganz einfach durch Σ^k und Π durch Π^k . Abbildung 6.33 (unten) demonstriert die Transformation auf grafische Weise.

6.1.5.4 Mehrband-Turing-Maschinen

Eine in vielerlei Hinsicht wertvolle Weiterentwicklung ist die *Mehrband-Turing-Maschine*. Im Gegensatz zur Mehrspur-Turing-Maschine gestattet sie, dass alle Schreib-Lese-Köpfe unabhängig voneinander bewegt werden dürfen. Um das Verhalten einer solchen Maschine vollständig zu erfassen, müssen wir die Übergangsfunktion δ geringfügig anpassen. Für eine k -Band-Maschine besitzt sie die folgende Form:

$$\delta(s, \sigma_1, \dots, \sigma_k) = (s', \sigma'_1, \dots, \sigma'_k, r_1, \dots, r_k)$$

Die Funktionsdefinition ist wie folgt zu lesen: Befindet sich die Turing-Maschine im Zustand s und steht der i -te Schreib-Lese-Kopf über dem Symbol σ_i , so geht die Maschine in den Zustand s' über und ersetzt das Zeichen σ_i auf dem i -ten Band durch das Zeichen σ'_i . Zusätzlich werden die Schreib-Lese-Köpfe entsprechend den Richtungsangaben r_i bewegt. Ist $r_i = \leftarrow$, so fährt der i -te Kopf eine Stelle nach links. Im Fall $r_i = \rightarrow$ findet eine Rechtsbewegung statt.

Obwohl die Mehrband-Turing-Maschine weit mächtiger erscheint als die ursprüngliche Einband-Maschine, lässt sie sich ebenfalls auf das Basismodell zurückführen (vgl. Abbildung 6.34). Die Grundidee besteht darin, die k -Band-Turing-Maschine auf eine Mehrspur-Turing-Maschine zu reduzieren und die unterschiedlichen Positionen der Schreib-Lese-Köpfe in symbolischer Form auf dem Band zu speichern. Hierzu verwenden wir eine Mehrspur-Turing-Maschine mit $2k$ Spuren. Jedem Band der zu simulierenden k -Band-Turing-Maschine ordnen wir 2 separate Spuren zu. Während die *Datenspur* eine symbolweise Bandkopie enthält, verwendet die *Positionsspur* ausschließlich die Zeichen \square und \uparrow . Das Symbol \uparrow markiert die Position, an der sich der simulierte Schreib-Lese-Kopf aktuell befindet.

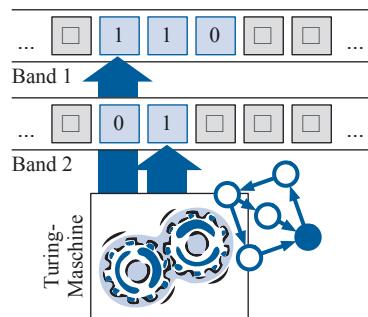
Um den Zustandsübergang einer k -Band-Maschine nachzuempfinden, muss die Mehrspurmaschine zunächst die Positionen der simulierten Schreib-Lese-Köpfe bestimmen. Hierzu wird das Band zurückgespult, bis auf allen Spuren ein Blank-Symbol erscheint. Danach wird der Kopf sukzessive nach rechts bewegt. Sobald eine Pfeilmarkierung auf einer der Positionsspuren erscheint, merkt sich die Maschine das auf der Datenspur vorgefundene Zeichen durch den Übergang in einen speziell hierfür vorgesehenen Zustand. Da wir die Symbolinformation hierdurch vollständig in die Zustandsmenge hineincodieren, wächst diese selbst für kleine Mehrband-Turing-Maschinen enorm an. Nichtsdestotrotz bleibt die Anzahl der Zustände in jedem Fall endlich.

Nachdem die letzte Pfeilmarkierung gefunden wurde, steht eindeutig fest, in welche Folgekonfiguration die Maschine übergeht. Die für jedes Band auszuführenden Aktionen werden jetzt nacheinander simuliert. Hierzu wird auf der aktuell bearbeiteten Positionsspur erneut die Pfeilmarkierung gesucht und das zugehörige Zeichen auf der Datenspur ersetzt. Danach wird eine Kopfbewegung nach links oder rechts simuliert, indem die Pfeilmarkierung auf der Positionsspur in die jeweilige Richtung verschoben wird.

Mehrband-Turing-Maschine

$$\Sigma = \{1, 0\}$$

$$\Pi = \{1, 0, \square\}$$



Simulation im Mehrspurmodell

$$\Sigma = \{1, 0, \uparrow\}$$

$$\Pi = \{1, 0, \uparrow, \square\}$$

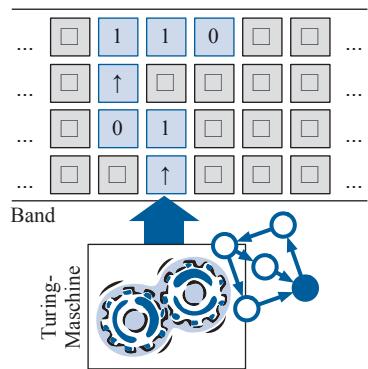


Abbildung 6.34: Eine Mehrband-Turing-Maschine mit k unabhängigen Bändern lässt sich auf eine Mehrspur-Turing-Maschine mit $2k$ Bändern abbilden. Jedes Band wird durch zwei Spuren repräsentiert. Die Datenspur ist eine Kopie des Bandinhalts. Auf der Positionsspur ist die Lage des simulierten Schreib-Lese-Kopfes verzeichnet.

■ Replace-Maschine

$$S = \{s_0, s_1\}$$

$$\Sigma = \{1, 0\}$$

$$\Pi = \{1, 0, \square\}$$

$$E = \{s_1\}$$

	1	0	\square
s_0	$(s_0, 1, \rightarrow)$	$(s_1, 1, \rightarrow)$	—
s_1	—	—	—

■ Erase-Maschine

$$S = \{s_2, s_3, s_4\}$$

$$\Sigma = \{1, 0\}$$

$$\Pi = \{1, 0, \square\}$$

$$E = \{s_4\}$$

	1	0	\square
s_2	$(s_2, 1, \rightarrow)$	—	$(s_3, \square, \leftarrow)$
s_3	$(s_4, \square, \leftarrow)$	—	—
s_4	—	—	—

■ Rewind-Maschine

$$S = \{s_5, s_6\}$$

$$\Sigma = \{1, 0\}$$

$$\Pi = \{1, 0, \square\}$$

$$E = \{s_6\}$$

	1	0	\square
s_5	$(s_5, 1, \leftarrow)$	—	$(s_6, \square, \rightarrow)$
s_6	—	—	—

Abbildung 6.35: Die Funktionsweise der Add-Maschine aus Abbildung 6.31 setzt sich aus drei Arbeitsschritten zusammen. Jeder Einzelschritt lässt sich mithilfe einer separaten Turing-Maschine beschreiben.

6.1.5.5 Maschinenkomposition

Betrachten wir die bisher konstruierten Turing-Maschinen genauer, so zeigen diese auf der obersten Ebene fast alle eine sequenzielle Struktur. Am Beispiel der in Abbildung 6.31 eingeführten Maschine zur Addition zweier Zahlen x und y wird die Struktur besonders deutlich; die Berechnung der Summe erfolgt hier in drei nacheinander ausgeführten Schritten. Im ersten Schritt wird die trennende Null durch eine Eins ersetzt und damit das Zwischenergebnis $x + y + 1$ hergestellt. Im zweiten Schritt wird die letzte Eins gelöscht und die Maschine im dritten Schritt zurückgespult. Insgesamt entpuppt sich die Additions-Maschine als die Komposition

- der Maschine Replace zum Ersetzen der Null,
- der Maschine Erase zum Löschen der letzten Eins und
- der Maschine Rewind zum Zurückspulen des Bands.

Alle drei Einzelmaschinen sind in Abbildung 6.35 zusammengefasst.

Im Umkehrschluss gibt uns diese Beobachtung eine einfache Möglichkeit an die Hand, um komplexe Funktionen durch die Komposition mehrerer einfach aufgebauter Maschinen zu realisieren. Zu diesem Zweck seien n beliebige Turing-Maschinen gegeben:

$$\begin{aligned} T_1 &= (S_1, \Sigma_1, \Pi_1, \delta_1, s_{10}, \square, \{e_{10}, \dots, e_{1m_1}\}) \\ T_2 &= (S_2, \Sigma_2, \Pi_2, \delta_2, s_{20}, \square, \{e_{20}, \dots, e_{2m_2}\}) \\ &\dots \\ T_n &= (S_n, \Sigma_n, \Pi_n, \delta_n, s_{n0}, \square, \{e_{n0}, \dots, e_{nm_n}\}) \end{aligned}$$

Um die Betrachtungen einfach zu halten, treffen wir die folgenden Vereinbarungen:

- Ohne Beschränkung der Allgemeinheit wollen wir annehmen, dass die Zustandsmengen S_1, \dots, S_n paarweise disjunkt sind. Zustandsmengen, die keine gemeinsamen Elemente enthalten, lassen sich herstellen, indem die Maschinenzustände vor der Komposition entsprechend umbenannt werden.
- Nur solche Zustände sind als Endzustände zugelassen, die keine weiteren Übergänge mehr erlauben. Auch diese Forderung lässt sich durch eine geringfügige Modifikation der Zustandsmenge und der Übergangsfunktion erfüllen.

Unter den beschriebenen Voraussetzungen lassen sich T_1, \dots, T_n zu einer Kompositionsmaschine vereinigen, indem wir sie an k Nahtstellen

$$(e_1, s_1), (e_2, s_2), \dots, (e_k, s_k)$$

miteinander verbinden. Hierin bezeichnet e_i einen Endzustand einer Turing-Maschine und s_i den Startzustand einer anderen. Den Übergang von e_i nach s_i gestalten wir so, dass der Schreib-Lese-Kopf an der aktuellen Position verharrt und der Bandinhalt nicht verändert wird. Hierzu fügen wir für jede Nahtstelle die folgende Regel hinzu:

$$\delta(e_i, \sigma) = (s_i, \sigma, \circlearrowleft)$$

Insgesamt hat die Kompositionsmaschine damit die Form

$$T' = (S', \Sigma', \Pi', \delta', s', \square, E')$$

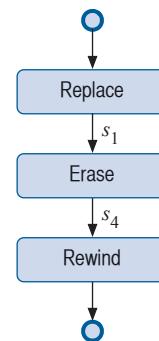
mit

$$\begin{aligned} S' &:= S_1 \cup \dots \cup S_n \\ \Sigma' &:= \Sigma_1 \cup \dots \cup \Sigma_n \\ \Pi' &:= \Pi_1 \cup \dots \cup \Pi_n \\ \delta' &:= \delta_1 \cup \dots \cup \delta_n \cup \{(e_i, \sigma) \mapsto (s_i, \sigma, \circlearrowleft) \mid 1 \leq i \leq k\} \\ s' &:= s_{10} \\ E' &:= E_1 \cup \dots \cup E_n \end{aligned}$$

Die Festlegung der Endzustände ist nicht die einzige mögliche. Alternative Definitionen nehmen diejenigen Zustände aus der Menge E' heraus, die eine Nahtstelle zu einer anderen Maschine besitzen. In diesem Fall würde das Ergebnis der Berechnung \perp lauten, falls die Turing-Maschine an einer Nahtstelle (e_i, s_i) stehen bleibt.

Mithilfe von Graphen lässt sich die Komposition von Turing-Maschinen übersichtlich darstellen. In dieser Darstellung wird jede Nahtstelle, die eine Turing-Maschine T_1 mit einer Turing-Maschine T_2 verbindet, durch einen Pfeil symbolisiert, dessen Kante mit dem Endzustand von T_1 beschriftet ist. Auf die Angabe des Startzustands von T_2 kann verzichtet werden, da dieser eindeutig bestimmt ist. Abbildung 6.36 demonstriert das Gesagte am Beispiel der Additionsmaschine.

Kompositionsmaschine



Übergangstabelle

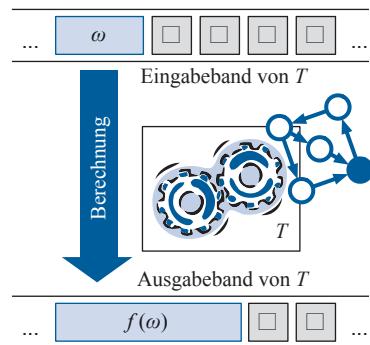
	1	0	\square
s_0	$(s_0, 1, \rightarrow)$	$(s_1, 1, \rightarrow)$	—
s_1	$(s_2, 1, \circlearrowleft)$	$(s_2, 0, \circlearrowleft)$	$(s_2, \square, \circlearrowleft)$
s_2	$(s_2, 1, \rightarrow)$	—	$(s_3, \square, \leftarrow)$
s_3	$(s_4, \square, \leftarrow)$	—	—
s_4	$(s_5, 1, \circlearrowleft)$	$(s_5, 0, \circlearrowleft)$	$(s_5, \square, \circlearrowleft)$
s_5	$(s_5, 1, \leftarrow)$	—	$(s_6, \square, \rightarrow)$
s_6	—	—	—

Abbildung 6.36: Mit dem Mittel der Komposition lassen sich Turing-Maschinen zu komplexeren Maschinen zusammenschalten.

6.1.5.6 Universelle Turing-Maschinen

In den vorangegangenen Abschnitten haben wir untersucht, wie sich grundlegende arithmetische Operationen mithilfe von Turing-Maschinen berechnen lassen. In diesem Abschnitt werden wir herausarbeiten, dass Turing-Maschinen sogar in der Lage sind, sich selbst zu

■ Turing-Maschine T



■ Universelle Turing-Maschine U

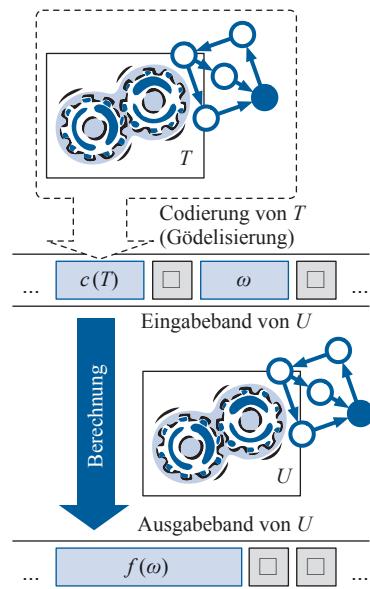


Abbildung 6.37: Prinzip der universellen Turing-Maschine. Neben dem Eingabewort ω wird die Gödelnummer einer anderen Turing-Maschine T auf das Eingabeband geschrieben. Die universelle Turing-Maschine U simuliert das Verhalten von T und produziert dieselbe Ausgabe, die T für das Eingabewort ω produzieren würde.

simulieren. Eine Turing-Maschine U , die eine beliebige andere Maschine T simulieren kann, heißt *universell*. Abbildung 6.37 veranschaulicht die grundsätzliche Arbeitsweise einer solchen Maschine:

- Als Eingabe nimmt U die Beschreibung einer anderen Turing-Maschine T in codierter Form sowie ein Eingabewort ω entgegen.
- Als Ausgabe schreibt U dieselbe Sequenz auf das Band, die auch die Originalmaschine T für die Eingabe ω produzieren würde.

Im Gegensatz zu allen anderen vorgestellten Maschinen ist die Funktion der universellen Turing-Maschine nicht auf eine Spezialaufgabe beschränkt. In ihrer Funktionsweise ist sie dem modernen Computer sehr ähnlich; sie agiert als Interpreter, der das Verhalten von T Schritt für Schritt auf dem Eingabewort simuliert. Die codierte Form von T entspricht dem Programm, das die universelle Maschine in die Lage versetzt, beliebige Berechnungen durchzuführen.

So weit die Theorie. Aber wie können wir eine Turing-Maschine dazu bewegen, eine andere Maschine zu simulieren? Wir wollen uns der Antwort schrittweise nähern und zunächst ein Verfahren ersinnen, mit dessen Hilfe eine Turing-Maschine codiert werden kann. Die Codierung ist notwendig, um die Turing-Maschine T zusammen mit dem Eingabewort ω auf dem Band der universellen Maschine zu speichern.

Für unsere Betrachtungen gehen wir von einer Maschine der Form

$$T = (\{s_1, s_2, \dots, s_n\}, \{0, 1, \square\}, \delta, s_1, \square, \{s_2\}) \quad (6.11)$$

aus. Neben den Symbolen 0, 1 und \square sind keine weiteren Zeichen auf dem Band zugelassen. s_1 bezeichnet den Startzustand und s_2 den (einzigsten) Endzustand. Die Beschränkung besitzt den Vorteil, dass das Eingabe- und das Bandalphabet sowie der Start- und Endzustand eindeutig festgelegt sind und sich zwei Turing-Maschinen nur noch in der Übergangsfunktion δ unterscheiden können. Hierdurch ist es ausreichend, eine Codierung der Übergangsfunktion δ auf das Band zu schreiben, um eine Turing-Maschine eindeutig zu charakterisieren.

Abbildung 6.38 zeigt eine mögliche Codierung der weiter oben eingeführten Erase-Maschine. Die Umsetzung in eine Binärzahl erfolgt in zwei Schritten. Zunächst wird für jeden Eintrag der Übergangstabelle eine Binärzahl erzeugt, indem die fünf Komponenten (Startzustand, Eingabesymbol, Folgesymbol, Ausgabesymbol und Richtung) in Form von Einserketten unär codiert und mit der Null als Trennzeichen zusammengefügt werden. Anschließend werden die erstellten Bitsequenzen

nach dem gleichen Schema zu einer großen Binärzahl verschmolzen. Die erzeugte Zahl heißt die *Gödelnummer* der Turing-Maschine T und die Codierung wird als *Gödelisierung* bezeichnet.

Beachten Sie, dass es eine Vielzahl von Möglichkeiten gibt, um Turing-Maschinen zu codieren, und die hier gewählte nur eine unter vielen ist. Wir wollen deshalb klären, welche Minimalanforderungen eine Codierung erfüllen muss, damit sie für die Gödelisierung einer Turing-Maschine eingesetzt werden kann.

- Offensichtlich muss die Gödelisierung gewährleisten, dass die Übergangstabelle durch die erzeugte Gödelnummer eindeutig festgelegt ist. Diese Eigenschaft ist genau dann erfüllt, wenn die Menge der Turing-Maschinen *injektiv* in die Menge der natürlichen Zahlen abgebildet wird. In der hier gewählten Codierung ist die Injektivität gewährleistet, da wir die unär codierten Symbole eindeutig durch die Null als Trennzeichen voneinander unterscheiden können.
- Die Gödelisierung stellt im Allgemeinen keine Eins-zu-eins-Beziehung zwischen der Menge der Turing-Maschinen und der Menge der Binärzahlen her. Es können also immer Binärzahlen existieren, die keiner Turing-Maschine entsprechen. Wir wollen nur solche Codierungen zulassen, für die wir entscheiden können, ob eine gültige oder eine ungültige Gödelnummer vorliegt.
- Die Injektivität einer Gödelisierung gewährleistet, dass die Übergangstabelle einer Turing-Maschine eindeutig durch ihre Gödelnummer bestimmt ist. Damit wir die Codierung für unsere Zwecke einsetzen können, müssen wir zusätzlich fordern, dass sie *berechenbar* ist. Das heißt, dass eine Turing-Maschine existieren muss, die aus einer beliebigen Übergangstabelle eine Gödelnummer berechnet und aus einer beliebigen Gödelnummer die Übergangstabelle extrahiert.

Verallgemeinert lesen sich die aufgestellten Eigenschaften wie folgt:



Definition 6.13 (Gödelisierung)

Die Funktion $c : M \rightarrow \mathbb{N}$ heißt *Gödelisierung*, wenn sie die folgenden Eigenschaften erfüllt:

- c ist injektiv
- Die Bildmenge $c(M)$ ist entscheidbar
- $c : M \rightarrow \mathbb{N}$ und $c^{-1} : c(M) \rightarrow M$ sind berechenbar

■ Erase-Maschine

	1	0	\square
s_1	$(s_3, 1, \rightarrow)$	—	$(s_2, \square, \leftarrow)$
s_3	$(s_2, \square, \leftarrow)$	—	—
s_2	—	—	—



$$\begin{aligned}(s_1, 1) &\mapsto (s_3, 1, \rightarrow) \\ (s_1, \square) &\mapsto (s_2, \square, \leftarrow) \\ (s_3, 1) &\mapsto (s_2, \square, \leftarrow)\end{aligned}$$

■ Codierung

Symbol	Code	Symbol	Code
s_1	1	\leftarrow	1
s_2	11	\rightarrow	11
s_3	111	(0
0	1	,	0
1	11)	0
\square	111	\leftrightarrow	0

$$(s_1, 1) \mapsto (s_3, 1, \rightarrow)$$



010110001110110110110

$$(s_1, \square) \mapsto (s_2, \square, \leftarrow)$$



010111000110111010

$$(s_3, 1) \mapsto (s_2, \square, \leftarrow)$$



0111011000110111010

■ Gödelnummer

0101100011101101100101110...
...001101110100111011000110111010

Abbildung 6.38: Gödelisierung der Erase-Maschine

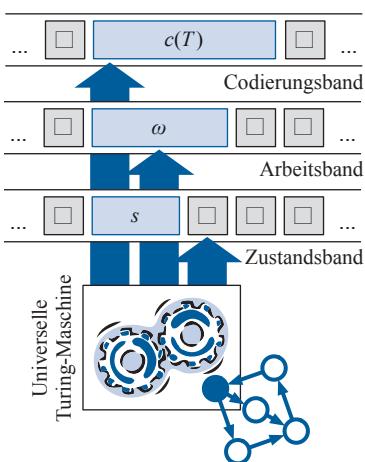


Abbildung 6.39: 3-Band-Implementierung der universellen Turing-Maschine. Das *Codierungsband* speichert die Gödelnummer $c(T)$ der zu simulierenden Maschine. Auf dem *Arbeitsband* wird die Arbeitsweise von T nachvollzogen und auf dem *Zustandsband* der simulierte Zustand von T abgelegt.

Die Implementierung einer universellen Turing-Maschine kann ebenfalls auf unterschiedliche Weise erfolgen. Eine einfache Möglichkeit besteht in der Verwendung einer 3-Band-Maschine (vgl. Abbildung 6.39). Auf dem ersten Band – dem *Codierungsband* – nimmt sie die Gödelnummer der zu simulierenden Turing-Maschine sowie das Eingabewort ω entgegen. Im ersten Schritt wird ω auf das zweite Band – das *Arbeitsband* – verschoben. Die Gödelnummer von T verbleibt auf dem *Codierungsband* und dient während der gesamten Berechnung als Merkhilfe für die auszuführenden Aktionen. Das dritte Band ist das *Zustandsband*. In codierter Form speichert es zu jedem Zeitpunkt den Zustand, in dem sich die simulierte Turing-Maschine T während der Abarbeitung von ω befinden würde. Da die Maschine, wie oben vereinbart, im Zustand s_1 startet, wird der Inhalt des *Zustandsbands* vorab mit der Sequenz $\dots \square 1 \square \dots$ initialisiert.

Ein Berechnungsschritt von T wird durch die folgenden Aktionen simuliert: Zunächst liest die Maschine die Unärcodierung des aktuellen Zustands von Band 3 und das nächste zu verarbeitende Zeichen von Band 2. Anschließend wird auf Band 1 nach dem auszuführenden Übergang gesucht. Befindet sich die simulierte Maschine beispielsweise im Zustand s_3 (111) und wird das Eingabezeichen 1 (11) eingelesen, so wird nach dem Bitmuster 111011000 gesucht. Die Gödelnummer von T ist so konstruiert, dass diese Bitsequenz niemals mehrfach auftreten kann. Wurde das Bitmuster gefunden, so liest die Maschine die nachfolgenden Bits der Form $1^i 0 1^j 0 1^k$ ein. Die Sequenzen 1^i , 1^j und 1^k entsprechen der unären Codierung des Folgezustands, des zu schreibenden Zeichens sowie der auszuführenden Kopfbewegung. Die Information wird durch die universelle Turing-Maschine ausgewertet und in entsprechende Aktionen umgesetzt. Das Folgezeichen wird auf das *Arbeitsband* geschrieben und die Kopfbewegung ausgeführt. Der neu einzunehmende Zustand wird auf das *Zustandsband* geschrieben.

Die universelle Turing-Maschine bricht die skizzierte Bearbeitungssequenz ab, sobald kein passendes Bitmuster auf Band 1 gefunden wurde. In diesem Fall ist für den aktuellen Zustand und das gefundene Eingabezeichen kein Übergang definiert und die simulierte Maschine würde die Bearbeitung stoppen. Jetzt entscheidet der Inhalt auf Band 3 über den Ausgang der Berechnung. Befindet sich die simulierte Maschine in einem Endzustand, so kopiert die universelle Maschine den Inhalt des zweiten auf das erste Band und geht ihrerseits in einen Endzustand über. Befindet sich auf Band 3 eine abweichende Sequenz, so stoppt die universelle Maschine, ohne in einen Endzustand überzugehen.

Die vergleichsweise grobe Beschreibung des Arbeitsprinzips soll uns an dieser Stelle genügen. Würden wir das genaue Verhalten der univer-

sellten Turing-Maschine auf der Zustandsebene vollständig auscodieren, so würde die Übergangstabelle den Umfang dieses Buchs sprengen.

Auf den ersten Blick mag die Universalität als eine Eigenschaft erscheinen, die nur sehr großen Turing-Maschinen vorbehalten ist. Dass dieser Eindruck täuscht, wird der nächste Abschnitt zeigen. Dort werden wir zunächst eine alternative Notation einführen und anschließend die kleinstmögliche universelle Turing-Maschine kennen lernen.

6.1.5.7 Zelluläre Turing-Maschinen

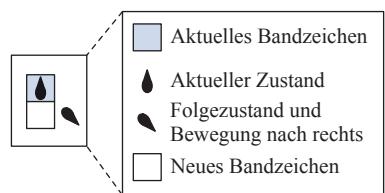
Die bisherige Darstellung der Turing-Maschine orientierte sich im Kern an Turings Originalarbeit; wir stellten uns eine mechanische Einheit vor, die einen Schreib-Lese-Kopf frei über einem unendlich langem Band hin- und herbewegt. In diesem Abschnitt wollen wir eine andere Darstellungsvariante diskutieren, die sich an jener des linearen Automaten aus Abschnitt 5.8 orientiert.

Linearen Automaten liegt eine eindimensionale Zelltopologie zugrunde. Die Zellen sind nebeneinander angeordnet und erstrecken sich in beide Richtungen in das Unendliche. Sie bilden exakt das unendliche Band nach, das wir für die Modellierung der Turing-Maschine benötigen. Der Bandinhalt wird durch die Färbungen der Zellen dargestellt, so dass wir die zur Verfügung stehende Farbmenge in direkter Weise als das Bandalphabet einer Turing-Maschine interpretieren können.

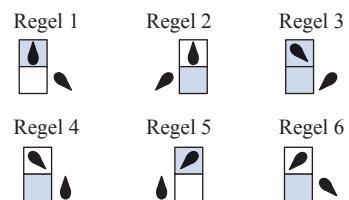
Die geschilderten Gemeinsamkeiten zwischen linearen Automaten und Turing-Maschinen dürfen nicht darüber hinwegtäuschen, dass sich beide Modelle in einem wesentlichen Punkt unterscheiden. Während die Berechnung in einem linearen Automaten verteilt erfolgt und alle Zellen parallel eine Zustandsänderung durchführen, arbeitet eine Turing-Maschine mit einem dedizierten Schreib-Lese-Kopf, der sich zu jeder Zeit an einer wohldefinierten Position befindet.

Um das Verhalten einer Turing-Maschine trotzdem adäquat zu beschreiben, bedarf es einer geringfügigen Modifikation des linearen Automaten. Wir erweitern das Modell, indem wir eine *Kopfzelle* (*head cell*) definieren, die als Schreib-Lese-Kopf fungiert. Das Schaltverhalten des erweiterten linearen Automaten legen wir analog zur Funktionsweise der Turing-Maschine fest. In jedem Berechnungsschritt wird die Kopfzelle umgefärbt und anschließend um eine Position nach links oder rechts geschoben. Außerdem reichern wir die Kopfzelle um einen zusätzlichen Zustand an, der mit dem Zustand der modellierten Turing-Maschine identisch ist.

Regelschema



Vollständiger Regelsatz



Automat in Aktion

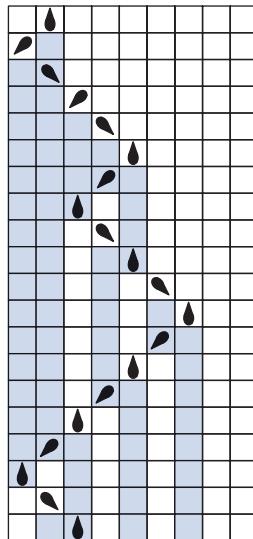
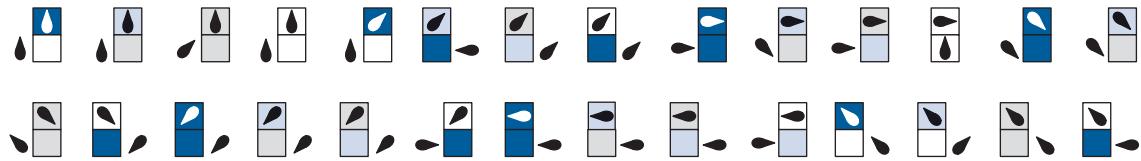
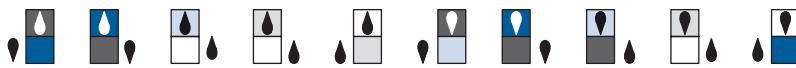


Abbildung 6.40: Durch eine Modifikation des Grundmodells lassen sich lineare zelluläre Automaten für die Simulation von Turing-Maschinen einsetzen [108].

- Marvin Minskys universelle Turing-Maschine aus dem Jahr 1962 [73]



- Stephen Wolframs universelle 2,5-Turing-Maschine aus dem Jahr 2002 [108]



- Die kleinstmögliche universelle Turing-Maschine: Wolframs 2,3-Maschine aus dem Jahr 2002 [108]

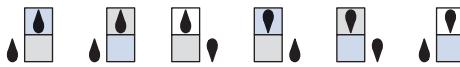


Abbildung 6.41: Die Jagd nach der kleinsten universellen Turing-Maschine fand im Jahr 2007 ihr erfolgreiches Ende. In diesem Jahr bewies der Brite Alex Smith die Universalität von Wolframs 2,3-Maschine.

Abbildung 6.40 zeigt, wie sich eine Turing-Maschine in der Notation des modifizierten linearen Automaten beschreiben lässt. Die dargestellte Maschine besitzt drei Zustände, die durch den Drehwinkel des verwendeten Keilsymbols unterschieden werden. Die Zellen können eine von zwei Farben annehmen, so dass der zelluläre Automat einer Turing-Maschine mit einem zweielementigen Bandalphabet entspricht. Das Verhalten wird durch insgesamt 6 Regeln bestimmt. Jede Regel wird durch zwei Farbfelder und zwei Keilsymbole beschrieben. Das obere Farbfeld beschreibt das aktuelle und das untere das neu zu schreibende Bandzeichen. Die Richtung des oberen Keils gibt an, in welchem Zustand sich die Maschine befinden muss, damit die entsprechende Regel angewendet werden kann. Der untere Keil definiert den Folgezustand und die auszuführende Kopfbewegung. Ist das Keilsymbol links des Folgezustands eingezeichnet, bewegt sich der Schreib-Lese-Kopf nach links, ist es rechts eingezeichnet, bewegt er sich nach rechts. In unserem Beispiel führen die Regeln 1, 3, 4 und 6 eine Kopfbewegung nach rechts und die Regeln 2 und 5 eine Kopfbewegung nach links aus.

Abbildung 6.41 zeigt drei historisch bedeutsame Turing-Maschinen, dargestellt in der Notation der zellulären Automaten. Alle drei Maschinen sind universell, d. h., sie sind in der Lage, jede andere Turing-Maschine zu simulieren. Die erste wurde im Jahr 1962 von dem amerikanischen Computerwissenschaftler Marvin Minsky vorgestellt. Seine universelle Maschine unterscheidet 7 Zustände und 4 Farben. In der von Stephen Wolfram verwendeten Nomenklatur wird die Maschine als 7,4-Maschine bezeichnet. Im Vergleich zur Originalarbeit von Turing war Minsky ein großer Wurf gelungen. Dort erstreckt sich die Beschreibung einer universellen Maschine noch über vier ganze Seiten [97].

Im Jahr 2002 stellte der britische Mathematiker Stephen Wolfram eine weiterentwickelte Maschine vor, die ebenfalls universell ist, aber mit weniger Zuständen auskommt. Während Minskys Modell 7 Zustände benötigt, kommt die neue 2,5-Maschine mit nur 2 Zuständen aus. Die Anzahl der Farben musste Wolfram allerdings von 4 auf 5 erhöhen. Bedeutender sollte jedoch seine zeitgleich veröffentlichte 2,3-Maschine werden (Abbildung 6.41 unten). Wolfram schlug die Maschine als einen potenziellen Kandidaten für die kleinstmögliche Turing-Maschine vor, die die Eigenschaft der Universalität erfüllt [108]. Auch wenn es Wolfram nicht gelang, seine Vermutung selbst zu belegen, konnte er ein wichtiges Zwischenresultat erzielen: Er bewies, dass 2 Farben und 2 Zustände *nicht* ausreichen, um die Eigenschaft der Universalität zu erreichen. Wäre die 2,3-Maschine also tatsächlich universell, so wäre es gleichzeitig die kleinste universelle Maschine, die überhaupt existiert.

Wolframs Vermutung wurde im Jahr 2007 zur Gewissheit. In diesem Jahr gelang dem 20-jährigen Briten Alex Smith der Nachweis, dass die 2,3-Maschine die Eigenschaft der Universalität erfüllt. Das Rätsel um die kleinstmögliche universelle Turing-Maschine hat damit ein erfolgreiches Ende gefunden.

6.1.6 Alternative Berechnungsmodelle

Neben den bisher vorgestellten Berechnungsmodellen wurden in der Vergangenheit weitere entwickelt. Bekannte Vertreter sind

- die Registermaschine,
- der Lambda-Kalkül von Church und Kleene,
- die Post'sche Tag-Maschine und
- die dynamische Logik.

Besitzt Wolframs 2,3-Maschine eine praktische Anwendung? Es scheint gute Gründe zu geben, die Frage mit Nein zu beantworten, schließlich lehrt uns die Erfahrung, dass die Programmierung einer Maschine schwieriger wird, je einfacher sie aufgebaut ist. Die 2,3-Maschine scheint diese empirisch gewonnene Vermutung auf eindringliche Weise zu bestätigen. Deutliche Hinweise liefert der Compiler, den Smith im Rahmen seines Universitätsbeweises konstruierte. Der Compiler hat die Aufgabe, eine beliebige Turing-Maschine T in ein Eingabewort zu übersetzen, das die 2,3-Maschine dazu veranlasst, T zu simulieren. Wolfram äußert sich hierüber wie folgt:

„But his ‘compiler’ doesn’t make terribly compact or efficient code. In fact, for anything but the simplest cases, the code tends to be astronomically large and horrendously inefficient.“ [109]

Lösen wir uns dagegen von den klassischen Denkmustern, die uns die aktuelle Computertechnik auferlegt, so fällt die Antwort weniger eindeutig aus. Für Wolfram ist z. B. eine völlig andere Art des Computers denkbar:

„Perhaps one day there’ll even be practical molecular computers built from this very 2,3 Turing machine. With tapes a bit like RNA strands, and heads moving up and down like ribosomes. When we think of nanoscale computers, we usually imagine carefully engineering them to mimic the architecture of the computers we know today. But one of the lessons [...] is that there’s a completely different way to operate.“ [109]

Erst die Zukunft kann zeigen, ob und wenn ja, in welcher Form die 2,3-Maschine den Sprung in die praktische Anwendung schaffen wird. Unabhängig davon ist die Maschine der Beweis dafür, dass die Universalität keine Eigenschaft ist, die einer komplexen Maschinerie bedarf. Bereits wenige, einfache Regeln reichen aus, um sie zu erreichen.

Die beiden Erstgenannten werden wir in ihren Grundzügen kurz vorstellen, aber nicht in der gleichen Tiefe diskutieren wie die weiter oben eingeführten Modelle. Informationen zu den Post'schen Tag-Maschinen gibt [82]. Dynamische Logiken werden ausführlich in [41] besprochen.

6.1.6.1 Registermaschinen

Die *Registermaschine* verkörpert ein Berechnungsmodell, das der Architektur realer Computersysteme sehr nahe kommt. Die Frage, was wir im Detail unter diesem Begriff zu verstehen haben, wird in der Literatur jedoch unterschiedlich beantwortet. Es existiert eine Vielzahl von Maschinentypen, die sich sowohl in der Anzahl und der Beschaffenheit der Register als auch im Befehlssatz unterscheiden. Das hier vorgestellte Modell wird in der Literatur meist als *verallgemeinerte Registermaschine* oder als *Random Access Machine*, kurz RAM, bezeichnet.

Abbildung 6.42 skizziert den Aufbau einer verallgemeinerten Registermaschine. Auf der obersten Ebene verfügt sie über ein *Eingabeband*, ein *Ausgabeband* und eine *Zentraleinheit*. Die Zentraleinheit untergliedert sich in den *Akkumulator A*, den *Befehlszähler B*, das *Programm P* und den *Speicher*, der durch abzählbar viele Register $R[1], R[2], \dots$ gebildet wird. Jedes Register kann über eine eindeutig vergebene Adresse direkt angesprochen und mit einer natürlichen Zahl beliebiger Größe beschrieben werden. Das Programm besteht aus einer endlichen Anweisungsfolge $P[1], \dots, P[m]$, die über den Befehlszähler B adressiert wird. Die vorhandenen Bänder dienen der Ein- und Ausgabe von Daten und werden über einen Lese- und einen Schreibkopf angesteuert. Der Lesekopf steht über der i -ten Zelle z_i des Eingabebands und der Schreibkopf über der j -ten Zelle z'_j des Ausgabebands. Initial seien i und j gleich 0, B gleich 1 und der Inhalt aller Register gleich 0. Nachdem die Registermaschine gestartet wurde, führt sie so lange den Befehl $P[B]$ aus, bis der Befehlszähler den Wert 0 erreicht.

Der Befehlssatz der Registermaschine ist in Tabelle 6.2 zusammengefasst. Mithilfe der Befehle `INP` oder `OUT` wird ein Zeichen vom Eingabeband eingelesen oder auf das Ausgabeband geschrieben. Der Lese- und der Schreibkopf bewegen sich ausschließlich von links nach rechts, so dass alle Zeichen nur einmal, in unveränderbarer Reihenfolge eingelesen bzw. ausgegeben werden können. Die Befehle `LDA` und `STA` dienen dem Datentransfer zwischen Akkumulator und Speicher. Insgesamt unterstützt die Registermaschine drei Adressierungsarten: die *unmittelbare Adressierung* (`LDA #n`), die *absolute Adressierung* (`LDA n, STA n`) und die *indirekte Adressierung* (`LDA (n), STA (n)`). Des Weiteren verfügt

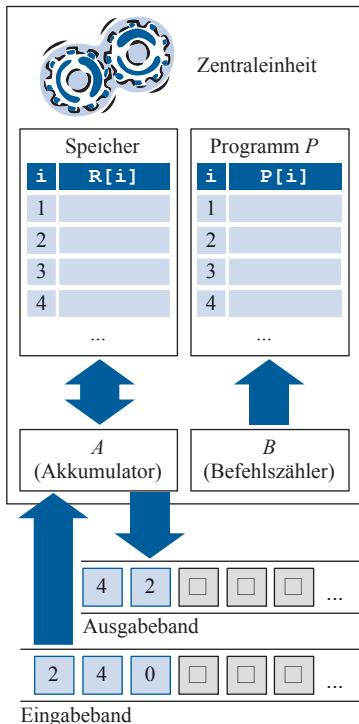


Abbildung 6.42: Allgemeiner Aufbau der verallgemeinerten Registermaschine

Befehl	Beschreibung	Aktion
INP	Überträgt den nächsten Wert vom Eingabeband in den Akkumulator	$A := z_i$ $B := B + 1, i := i + 1$
OUT	Schreibt den Akkumatorinhalt auf das Ausgabeband	$z'_j := A$ $B := B + 1, j := j + 1$
LDA #n	Lädt den Akkumulator mit dem Wert n	$A := n$ $B := B + 1$
LDA n	Lädt den Akkumulator mit dem Inhalt von Register n	$A := R[n]$ $B := B + 1$
LDA (n)	Lädt den Akkumulator über ein indirekt adressiertes Register	$A := R[R[n]]$ $B := B + 1$
STA n	Überträgt den Akkumatorinhalt in das Register n	$R[n] := A$ $B := B + 1$
STA (n)	Überträgt den Akkumatorinhalt in ein indirekt adressiertes Register	$R[R[n]] := A$ $B := B + 1$
ADD #n	Erhöht den Akkumatorinhalt um einen konstanten Wert	$A := A + n$ $B := B + 1$
SUB #n	Verringert den Akkumatorinhalt um einen konstanten Wert	$A := \max\{0, A - n\}$ $B := B + 1$
JMP n	Direkter Sprung zur n -ten Programmanweisung	$B := n$
BEQ i,n	Indirekter Sprung in Abhängigkeit des Akkumatorinhalts	$B := n$ falls $A = i$ $B := B + 1$ falls $A \neq i$

Tabelle 6.2: Vollständiger Befehlssatz der verallgemeinerten Registermaschine

die Registermaschine über primitive Arithmetikfähigkeiten in Form der Befehle ADD und SUB. Die Subtraktion wird auch hier gesättigt durchgeführt, da die Register per Definition keine negativen Zahlen aufnehmen können. Die Befehle JMP und BEQ dienen zur Steuerung des Kontrollflusses. JMP beschreibt B mit einem konstanten Wert und verursacht hierdurch einen Sprung an eine beliebige Programmstelle. BEQ implementiert einen bedingten Sprung, der nur dann ausgelöst wird, wenn der Akkumulator einen bestimmten Wert enthält.

```

mirror.asm

START: LDA #1
       STA 1
READ:  LDA 1
       ADD #1
       STA 1
       INP
       BEQ 0, WRITE
       STA (1)
       JMP READ
WRITE: LDA 1
       SUB #1
       STA 1
       BEQ 1, HALT
       LDA (1)
       OUT
       JMP WRITE
HALT:  JMP 0

```

Abbildung 6.43: Das dargestellte Beispielprogramm veranlasst die Registermaschine, den Inhalt des Eingabebands in umgekehrter Reihenfolge auf dem Ausgabeband wiederzugeben [27]. Um die Programmstruktur zu verdeutlichen, wurden die absoluten Sprungadressen durch symbolische Bezeichner ersetzt.

Abbildung 6.43 demonstriert die Funktionsweise der Registermaschine anhand eines konkreten Beispiels. Das dargestellte Programm liest eine Folge von Zahlen vom Eingabeband und gibt diese in umgekehrter Reihenfolge auf dem Ausgabeband aus. Die Ziffer 0 markiert das Ende der Eingabefolge und wird nicht auf das Ausgabeband geschrieben.

Um die gewünschte Funktionalität zu erreichen, werden die Register $R[2], R[3], \dots$ als Kellerspeicher eingesetzt, der zunächst in aufsteigender Richtung beschrieben und danach in absteigender Richtung wieder ausgelesen wird. Das Register $R[1]$ wird als Indexzeiger verwendet, der zu jedem Zeitpunkt auf das Kopfelement des Kellerspeichers verweist. Tabelle 6.3 zeigt im Detail, wie die Eingabesequenz 2,4,0 verarbeitet wird. Nach 41 Berechnungsschritten terminiert die Maschine und hinterlässt wie erwartet die Ziffernfolge 4,2 auf dem Ausgabeband.

Die verallgemeinerte Registermaschine ist in der Lage, eine beliebige Turing-Maschine zu simulieren. Eine einfache Möglichkeit besteht darin, mit den Registern $R[2], R[3], \dots$ ein einseitig beschränktes Band nachzubilden und die Position des Schreib-Lese-Kopfes in Register $R[1]$ zu speichern. Das Verhalten der Turing-Maschine wird in drei Einzelschritten simuliert. Zuerst wird der Inhalt des Eingabebands auf das virtuelle Arbeitsband kopiert. Anschließend wird das Verhalten der Turing-Maschine schrittweise nachvollzogen und am Ende der finale Inhalt des virtuellen Arbeitsbands auf das Ausgabeband kopiert.

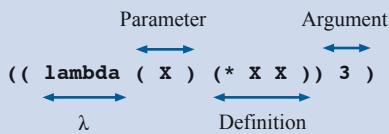
Umgekehrt lässt sich jede Registermaschine mithilfe einer Turing-Maschine simulieren. Um die Transformation zu verstehen, halten wir zunächst fest, dass wir jedes Programm so modifizieren können, dass nur endlich viele Register verwendet werden. Die Transformation ist möglich, da der Wertebereich eines Registers – im Gegensatz zu realen Computerarchitekturen – unbeschränkt ist. Über die Cantor'sche Paarungsfunktion können wir damit den Inhalt des gesamten Registersatzes in ein einziges Register packen.

Eine Registermaschine mit n Registern lässt sich mit einer einseitig beschränkten Turing-Maschine mit $n+3$ Bändern simulieren. Auf den ersten n Bändern wird der Inhalt des Registersatzes gespeichert, auf den verbleibenden 3 Bändern wird das Eingabeband, das Ausgabeband und der Akkumulator nachgebildet. Für das Befehlsregister wird kein zusätzliches Band benötigt. Da jedes Programm nur endlich viele Anweisungen besitzt, können wir den Wert von B in die Zustandsfolge hineincodieren, die von der Turing-Maschine nacheinander durchlaufen wird. Wenngleich die angestellten Überlegungen sehr informeller Natur sind, erklären sie im Kern, warum die Registermaschine die gleiche Berechnungsstärke wie die Turing-Maschine besitzt.

Zyklus	B	Befehl	Eingabeband	Ausgabeband	A	R[1]	R[2]	R[3]	R[4]
01	1	LDA #1	2, 4, 0	–	1	0	0	0	0
02	2	STA 1	2, 4, 0	–	1	1	0	0	0
03	3	LDA 1	2, 4, 0	–	1	1	0	0	0
04	4	ADD #1	2, 4, 0	–	2	1	0	0	0
05	5	STA 1	2, 4, 0	–	2	2	0	0	0
06	6	INP	4, 0	–	2	2	0	0	0
07	7	BEQ 0, WRITE	4, 0	–	2	2	0	0	0
08	8	STA (1)	4, 0	–	2	2	2	0	0
09	9	JMP READ	4, 0	–	2	2	2	0	0
10	3	LDA 1	4, 0	–	2	2	2	0	0
11	4	ADD #1	4, 0	–	3	2	2	0	0
12	5	STA 1	4, 0	–	3	3	2	0	0
13	6	INP	0	–	4	3	2	0	0
14	7	BEQ 0, WRITE	0	–	4	3	2	0	0
15	8	STA (1)	0	–	4	3	2	4	0
16	9	JMP READ	0	–	4	3	2	4	0
17	3	LDA 1	0	–	3	3	2	4	0
18	4	ADD #1	0	–	4	3	2	4	0
19	5	STA 1	0	–	4	4	2	4	0
20	6	INP	–	–	0	4	2	4	0
21	7	BEQ 0, WRITE	–	–	0	4	2	4	0
22	10	LDA 1	–	–	4	4	2	4	0
23	11	SUB #1	–	–	3	4	2	4	0
24	12	STA 1	–	–	3	3	2	4	0
25	13	BEQ 1, HALT	–	–	3	3	2	4	0
26	14	LDA (1)	–	–	4	3	2	4	0
27	15	OUT	–	4	4	3	2	4	0
28	16	JMP WRITE	–	4	4	3	2	4	0
29	10	LDA 1	–	4	3	3	2	4	0
30	11	SUB #1	–	4	2	3	2	4	0
31	12	STA 1	–	4	2	2	2	4	0
32	13	BEQ 1, HALT	–	4	2	2	2	4	0
33	14	LDA (1)	–	4	2	2	2	4	0
34	15	OUT	–	4,2	2	2	2	4	0
35	16	JMP WRITE	–	4,2	2	2	2	4	0
36	10	LDA 1	–	4,2	2	2	2	4	0
37	11	SUB #1	–	4,2	1	2	2	4	0
38	12	STA 1	–	4,2	1	1	2	4	0
39	13	BEQ 1, HALT	–	4,2	1	1	2	4	0
40	17	JMP 0	–	4,2	1	1	2	4	0
41	0	–	–	4,2	1	1	2	4	0

Tabelle 6.3: Ablaufprotokoll für die Eingabesequenz 2,4,0

Der Lambda-Kalkül ist das mathematische Fundament aller funktionalen Programmiersprachen. Der älteste Vertreter dieser Gruppe ist Lisp. Die Sprache wurde im Jahr 1958 von John McCarthy ins Leben gerufen und ist nach Fortran die zweitälteste Computersprache überhaupt. Genau wie Prolog wird Lisp vor allem für Anwendungen aus dem Bereich der künstlichen Intelligenz eingesetzt. Lisp-Programme besitzen einen einfachen Aufbau, da lediglich zwei Grundelemente unterschieden werden: Atome und Listen. Letztere dürfen sich selbst als Element enthalten und ermöglichen so die Konstruktion komplexer Datenstrukturen. Sowohl die interne Arbeitsweise als auch die Notation von Lisp orientieren sich an jenen des Lambda-Kalküls:



Lisp erreicht seine große Flexibilität aufgrund der Eigenschaft, nicht zwischen Daten und Anweisungen zu unterscheiden. Jedes Lisp-Programm ist eine Liste und wie jedes andere Objekt zur Laufzeit manipulierbar. Umgekehrt kann jede Liste als Programm interpretiert und durch den Interpreter ausgeführt werden.

Lisp ist die älteste, aber nicht die einzige Programmiersprache, die sich an der Arbeitsweise des Lambda-Kalküls orientiert. Das funktionale Programmierparadigma wurde in der Vergangenheit in verschiedene Richtungen weiterentwickelt und um zusätzliche Konzepte angereichert. Im Zuge dieser Entwicklungen sind unter anderem die Sprachen Haskell, ML, Miranda und Scheme entstanden. Obwohl sich das Aussehen der Quelltexte teilweise erheblich unterscheidet, basieren sie alle auf dem gleichen operativen Kern: dem Lambda-Kalkül von Alonzo Church und Stephen Kleene.

6.1.6.2 Lambda-Kalkül

Der Lambda-Kalkül (kurz λ -Kalkül) wurde in den Dreißigerjahren von Alonzo Church und Stephen Kleene entwickelt. Er wurde mit dem Ziel entworfen, komplexe mathematische Funktionen durch die Kombination allgemein gehaltener Rechenvorschriften zu definieren. Die grundlegende Operation des Lambda-Kalküls ist die Anwendung einer Funktion f auf ein Argument x , geschrieben als $(f x)$. Ist z. B. add eine Funktion zur Addition zweier Zahlen, so berechnet $((\text{add } x) y)$ die Summe $x + y$. Indem wir eine Variable mithilfe des λ -Operators binden, lassen sich aus bestehenden Funktionen neue erzeugen. So bezeichnet $(\lambda x. ((\text{add } x) x))$ eine von x abhängige Funktion, deren Ergebniswert durch $((\text{add } x) x) = 2 \cdot x$ festgelegt ist. Die Anwendung des λ -Operators wird als *Abstraktion* bezeichnet.

Damit haben wir bereits alle Grundbausteine des Lambda-Kalküls kennen gelernt. Formal definieren wir die Menge der *Lambda-Terme* (kurz λ -Terme) wie folgt:



Definition 6.14 (λ -Terme)

Sei $V = \{x_1, x_2, \dots\}$ eine Menge von Variablen. Dann gilt:

- Jede Variable $x_i \in V$ ist ein λ -Term.
- Sind f und g λ -Terme, dann ist (fg) ein λ -Term.
- Ist f ein λ -Term und $x_i \in V$, dann ist $(\lambda x_i.f)$ ein λ -Term.

λ -Ausdrücke lassen sich nach dieser Definition freizügig kombinieren. Eine Funktion kann beliebige Lambda-Terme als Argumente erhalten und somit auch auf Funktionen angewendet werden. Wie das folgende Beispiel zeigt, kann sich eine Funktion sogar selbst als Argument entgegennehmen:

$$((\lambda x.x)(\lambda x.x))$$

Mithilfe von *Konversionsregeln* lässt sich ein λ -Term in einen äquivalenten Term überführen, der die gleiche Funktion beschreibt. Auf diese Weise werden die λ -Terme in Äquivalenzklassen aufgeteilt und wir schreiben $f = g$, falls f und g der gleichen Klasse angehören. Im Einzelnen handelt es sich um die folgenden drei Regeln:

- α -Konversion

$$(\lambda x.f) = (\lambda y.f[x \leftarrow y])$$

Die Konversion erlaubt die Umbenennung von Variablen, ist jedoch nur unter gewissen Einschränkungen anwendbar. Zum einen muss die Variable y so gewählt werden, dass durch die Substitution keine neuen Bindungen innerhalb von f entstehen. Zum anderen werden nur diejenigen Vorkommen von x ersetzt, die in f ungebunden vorkommen.

■ β -Konversion

$$((\lambda x.f) g) = f[x \leftarrow g]$$

Diese Regel entspricht dem Einsetzen des Parameters g in die Funktion f . Der Funktionswert wird berechnet, indem alle Vorkommen der Variablen x durch g ersetzt werden. Auch hier dürfen nur jene Vorkommen von x ersetzt werden, die in f nicht durch einen weiteren λ -Operator gebunden sind.

■ η -Konversion

$$(\lambda x.fx) = f$$

Die Konversion besagt, dass zwei Funktionen genau dann gleich sind, wenn sie für alle Belegungen der Eingangsgröße den gleichen Funktionswert berechnen. Die η -Konversion ist optional und kann aus dem Kalkül entfernt werden, ohne seine grundlegenden Eigenschaften zu verändern.

Ein λ -Term liegt in *Normalform* vor, wenn keine β -Konversion mehr anwendbar ist. Der Kalkül erfüllt die nach Alonzo Church und J. Barkley Rosser benannte *Church-Rosser-Eigenschaft*, aus der unter anderem folgt, dass die Reihenfolge, in der α - und β -Konversionen angewendet werden, keine Rolle spielt (vgl. Abbildung 6.44).

Beachten Sie, dass sich nicht jeder λ -Ausdruck in endlich vielen Schritten in eine Normalform überführen lässt. Wie das Beispiel in Abbildung 6.45 zeigt, können unendlich lange Ableitungssequenzen entstehen. Der abgebildete λ -Ausdruck Y wird aufgrund seiner speziellen Eigenschaft als *Fixpunktoperator* bezeichnet. Er spielt innerhalb des λ -Kalküls eine prominente Rolle und kann dazu verwendet werden, beliebige rekursive Funktionen zu definieren.

Im direkten Vergleich mit den meisten anderen Berechnungsmodellen wirkt der λ -Kalkül minimalistisch, schließlich beschränkt er sich auf die Definition weniger fundamentaler Konversionsregeln, die eine rein syntaktische Manipulationen von Ausdrücken erlauben. Trotzdem ist der λ -Kalkül genauso ausdrucksstark wie die in Abschnitt 6.1.5 ausführlich beschriebene Turing-Maschine. Entsprechende Resultate wurden von Kleene und Turing in den Dreißigerjahren bewiesen [59, 98].

■ Beispiel

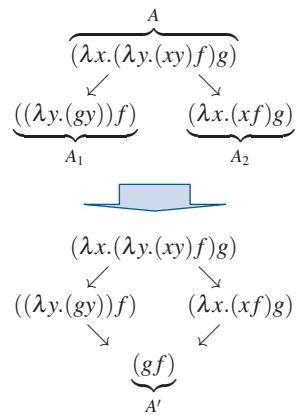


Abbildung 6.44: Der Lambda-Kalkül erfüllt die Church-Rosser-Eigenschaft. Lässt sich ein Ausdruck A in die Ausdrücke A_1 und A_2 umformen, so lassen sich diese durch weitere Umformungen immer wieder zu einem gemeinsamen Ausdruck A' zusammenführen.

$$Y := (\lambda f.((\lambda x.(f(xx)))(\lambda x.(f(xx)))))$$



$$\begin{aligned} & (Yg) \\ &= ((\lambda f.((\lambda x.(f(xx)))(\lambda x.(f(xx))))))g \\ &= ((\lambda x.(g(xx)))(\lambda x.(g(xx)))) \\ &= (g((\lambda x.(g(xx)))(\lambda x.(g(xx)))))) \\ &= (g(Yg)) \\ &= (g(g(Yg)))) \\ &= (g(g(g(Yg)))) \\ &= (g(g(g(g(Yg)))))) \\ &= \dots \end{aligned}$$

Abbildung 6.45: Für beliebige λ -Terme g erfüllt der Fixpunktoperator Y die Eigenschaft $(Yg) = (g(Yg))$. Es entsteht eine unendlich lange Ableitungssequenz.

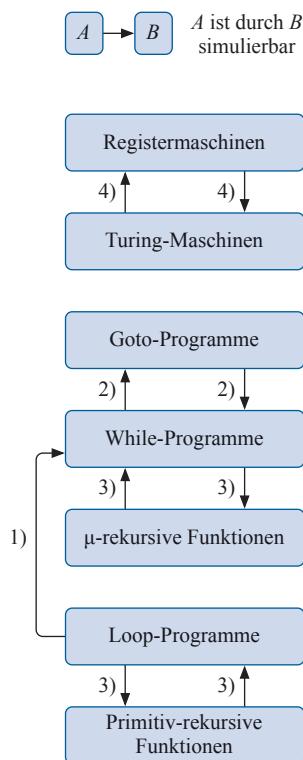


Abbildung 6.46: Ausdrucksstärke der verschiedenen Berechnungsmodelle

6.2 Church'sche These

In den vorangegangenen Abschnitten haben wir verschiedene Berechnungsmodelle eingeführt, die den Begriff der *Berechenbarkeit* mathematisch präzise beschreiben. Im Rahmen unserer Betrachtungen konnten wir bereits einige wichtige Teilergebnisse erringen (vgl. Abbildung 6.46). So haben wir in Abschnitt 6.1.2 am Beispiel der Ackermann-Funktion herausgearbeitet, dass die Loop-Sprache berechnungsschwächer als die While-Sprache ist und in Abschnitt 6.1.3 die Äquivalenz von While- und Goto-Programmen bewiesen. Anschließend haben wir in Abschnitt 6.1.4 gezeigt, dass die Menge der primitiv-rekursiven Funktionen mit der Menge der Loop-berechenbaren Funktionen und die Menge der μ -rekursiven Funktionen mit der Menge der While-berechenbaren Funktionen übereinstimmt. Abschließend haben wir in Abschnitt 6.1.6 die Äquivalenz zwischen Turing-Maschinen und Registermaschinen skizziert.

Ein wichtiger Beweis steht bisher noch aus, bevor wir den Kreis endgültig schließen können. Die Rede ist von der Äquivalenz zwischen While-Programmen und Turing-Maschinen. Dass sich beide tatsächlich als gleich ausdrucksstark erweisen werden, ist ein beeindruckendes Ergebnis der Berechenbarkeitstheorie, schließlich könnten beide Modelle in ihrem Aufbau und ihrer Struktur kaum unterschiedlicher sein. Die While-Sprache folgt dem Ansatz, den wir aus der klassischen Software-Technik kennen. Im Kern ist sie der Prototyp des imperativen Programmierparadigmas und entsprechend einfach lassen sich While-Programme in reale Sprachen übersetzen. Im Gegensatz hierzu erinnern Turing-Maschinen in ihrem Aufbau an die klassische Hardware-Technik. Interpretieren wir das Band als Speicher und den Zustandsautomat als Prozessor, so lässt sich die Funktionsweise einer Turing-Maschine nahezu eins zu eins mit der eines modernen Computers gleichsetzen. Auch dort werden Daten permanent vom Speicher in den Prozessor geladen, verändert und in den Speicher zurückgeschrieben.

Um die Äquivalenz beider Berechnungsmodelle zu beweisen, werden wir in zwei Schritten vorgehen. Zuerst werden wir demonstrieren, dass sich jedes While-Programm mithilfe einer Turing-Maschine simulieren lässt. Anschließend zeigen wir die Umkehrung.

Die Simulation eines While-Programms P gelingt am einfachsten mit einer Turing-Maschine mit k Bändern, wobei wir den Parameter k so wählen, dass jede in P vorkommende Variable x_i auf einem separaten Band Platz findet. Um die einzelnen Operationen eines While-Programms auf der Turing-Maschine zu simulieren, definieren wir die

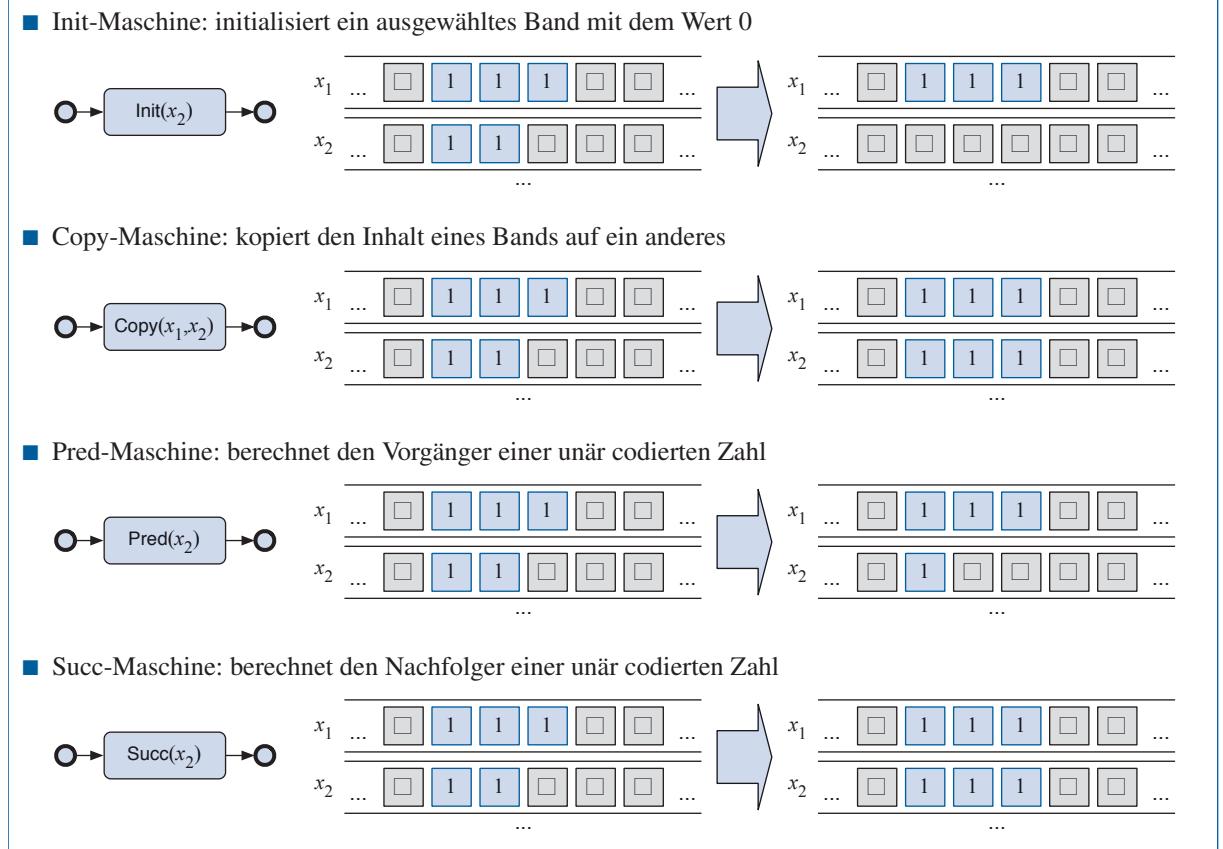
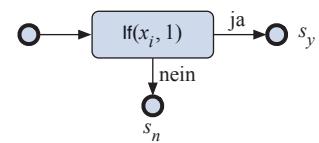


Abbildung 6.47: Elementarmaschinen zur Simulation der While-Sprache

vier in Abbildung 6.47 dargestellten Elementarmaschinen. Mit ihnen sind wir in der Lage, alle Zuweisungs- und Arithmetikoperationen der While-Sprache abzubilden.

Außerdem definieren wir für alle $k \in \mathbb{N}$ eine Maschine $\text{If}(x_i, k)$, die den Inhalt des i -ten Bands mit dem Wert k vergleicht. Abbildung 6.49 zeigt exemplarisch die Definition der Maschine $\text{If}(x_i, 1)$. Anders als die weiter oben vorgestellten Elementarmaschinen besitzt die If-Maschine zwei Endzustände. Ist der Inhalt des i -ten Bands gleich k , so wird der Zustand s_y und in allen anderen Fällen der Zustand s_n eingenommen.

Damit haben wir alle Bausteine in unserem Repertoire, um ein beliebiges While-Programm durch die Komposition der Elementarmaschinen in eine äquivalente Turing-Maschine zu übersetzen. Das Konstruktions-



	0	1	□
s_0	$(s_n, 0, \circlearrowleft)$	$(s_1, 1, \rightarrow)$	$(s_n, \square, \circlearrowleft)$
s_1	$(s_n, 0, \leftarrow)$	$(s_n, 1, \leftarrow)$	$(s_y, \square, \leftarrow)$
s_n	—	—	—
s_y	—	—	—

Abbildung 6.49: Die If-Maschine

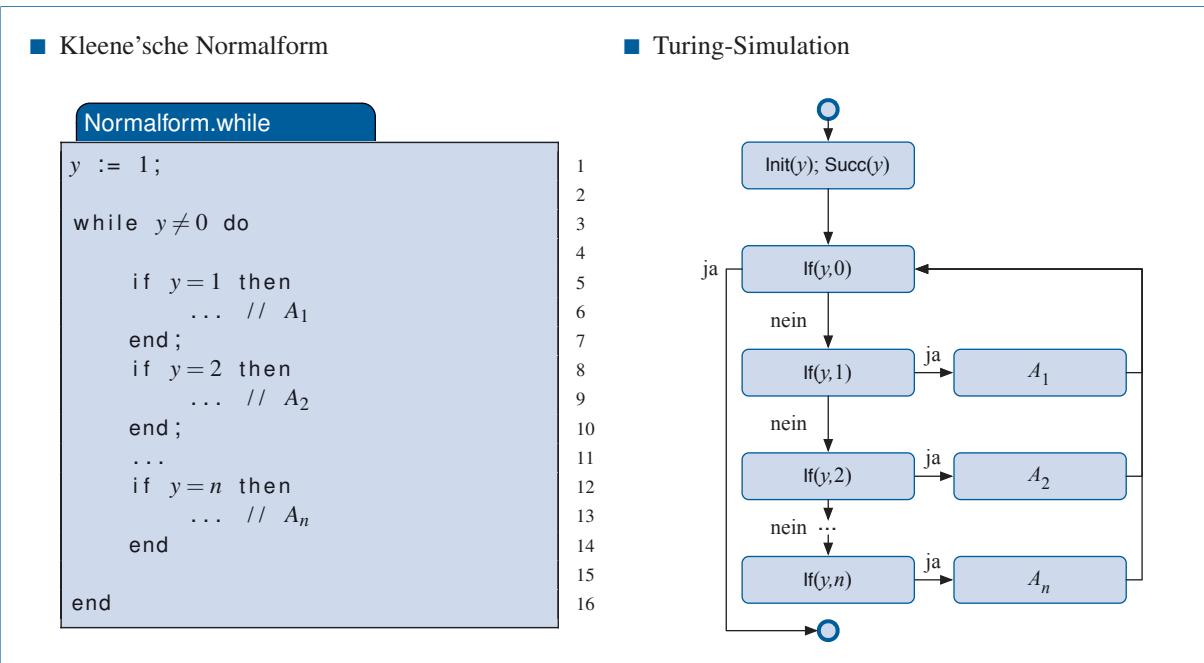


Abbildung 6.48: Simulation von While-Programmen mithilfe von Turing-Maschinen

schema in Abbildung 6.48 zeigt, wie sich ein While-Programm in Kleene'scher Normalform in eine Turing-Maschine überführen lässt. Den Kern der Konstruktion bildet die Simulation der While-Schleife. Diese wird durch eine If-Maschine realisiert, die in jedem Schleifendurchlauf den Wert des Bandes y überprüft. Ist $y = 0$, so geht die Turing-Maschine in den Endzustand über und terminiert. Ist $y \neq 0$, so wird mithilfe weiterer If-Maschinen eine Fallunterscheidung durchgeführt, die eins zu eins der Struktur des While-Programms entspricht. Trifft die i -te If-Bedingung zu, so wird die While-Anweisung A_i simuliert. Diese besteht aus einer Reihe arithmetischer Manipulationen, die mithilfe der Elementarmaschinen aus Abbildung 6.47 nachempfunden werden können.

Da sich jedes While-Programm in ein äquivalentes Programm in Kleene'scher Normalform übersetzen lässt, ist gezeigt, dass jedes While-Programm durch eine Turing-Maschine simuliert werden kann. Damit sind wir unserem Ziel, die Äquivalenz zwischen Turing-Maschinen und While-Programmen zu zeigen, einen großen Schritt näher gekommen:


Satz 6.7

Jede While-berechenbare Funktion ist Turing-berechenbar.

Wir wenden uns nun der umgekehrten Richtung zu: der Simulation von Turing-Maschinen mit While-Programmen. Wir werden den Beweis indirekt führen, indem wir auf eines unserer erzielten Zwischenergebnisse aus Abschnitt 6.1.3 zurückgreifen. Dort haben wir gezeigt, dass jede While-berechenbare Funktion auch Goto-berechenbar ist und umgekehrt. Wir nutzen dieses Resultat aus und zeigen, dass sich jede Turing-Maschine durch ein Goto-Programm simulieren lässt. Aus diesem Ergebnis folgt dann unmittelbar die Äquivalenz zwischen Turing-Maschinen und While-Programmen.

Für die folgende Betrachtung sei eine beliebige Turing-Maschine

$$T = (S, \Sigma, \Pi, \delta, s_0, \square, E)$$

gegeben. Das Bandalphabet Π betrachten wir als geordnete Menge, d. h., jedes Element $\pi \in \Pi$ besitzt eine eindeutige Position, die wir mit $\langle \pi \rangle$ ($1 \leq \langle \pi \rangle \leq |\Pi|$) bezeichnen.

Wir wollen nun versuchen, eine Konfiguration der Turing-Maschine T innerhalb eines Goto-Programms darzustellen. Wie in Abschnitt 6.1.5 eingeführt, ist eine Konfiguration ein Tripel (v, s, ω) mit

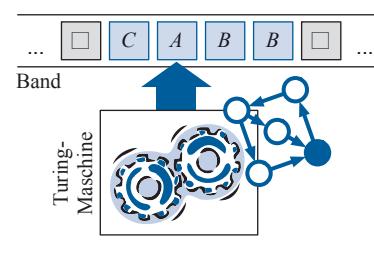
$$v = \{\rho_m, \dots, \rho_1\}, \quad s \in S, \quad \omega = \{\sigma_1, \dots, \sigma_n\}.$$

Die Variable s repräsentiert den aktuellen Zustand der Maschine und die Zeichenfolgen ρ_m, \dots, ρ_1 und $\sigma_1, \dots, \sigma_n$ entsprechen den Bandsymbolen links und rechts des Lesekopfes. Der Kopf selbst befindet sich über dem Zeichen σ_1 . Beachten Sie, dass die Indizes der Zeichen in v absteigend und in ω aufsteigend notiert sind. Die Reihenfolge wurde bewusst so gewählt; sie stellt sicher, dass sich das Zeichen mit dem kleinsten Index in unmittelbarer Kopfnähe befindet. Wie wir gleich sehen werden, ist es hierdurch möglich, eine Konfigurationsänderung ohne große Umwege auf die Multiplikation, die Division und die Modulo-Berechnung abzubilden.

Um eine Konfiguration zu repräsentieren, konstruieren wir ein Goto-Programm mit drei Variablen x_v , x_s und x_ω (vgl. Abbildung 6.50). Der aktuelle Zustand der Turing-Maschine wird in der Variablen x_s gespeichert. x_v und x_ω codieren den Inhalt von v und ω in Form einer b -adischen Zahlendarstellung:

$$x_v := \sum_{i=1}^m \langle \rho_i \rangle \cdot b^i \tag{6.12}$$

$$x_\omega := \sum_{i=1}^n \langle \sigma_i \rangle \cdot b^i \tag{6.13}$$



■ Bandalphabet

$$\Pi = \{A, B, C, \square\}$$

$$\langle A \rangle = 1$$

$$\langle B \rangle = 2$$

$$\langle C \rangle = 3$$

■ Bandcodierung ($b = 5$)

$$v = C$$

$$x_v = \langle C \rangle \cdot 5^1$$

$$= 15$$

$$\omega = ABB$$

$$x_\omega = \langle A \rangle \cdot 5^1 + \langle B \rangle \cdot 5^2 + \langle B \rangle \cdot 5^3$$

$$= 1 \cdot 5 + 2 \cdot 25 + 2 \cdot 125$$

$$= 305$$

Abbildung 6.50: Simulation von Turing-Maschinen mit Goto-Programmen. Die aktuelle Konfiguration der simulierten Maschine wird durch den Inhalt der drei Variablen x_v , x_s und x_ω nachgebildet.

■ Linksbewegung

$$\delta(s_i, \sigma) = (s_j, \sigma', \leftarrow)$$



simleft.goto

```

1    $x_\omega := x_\omega \text{ div } b;$ 
2    $x_\omega := \langle\sigma'\rangle + b \cdot x_\omega;$ 
3    $x_\omega := (x_v \text{ mod } b) + b \cdot x_\omega;$ 
4    $x_v := x_v \text{ div } b;$ 
5    $x_s := j$ 

```

■ Rechtsbewegung

$$\delta(s_i, \sigma) = (s_j, \sigma', \rightarrow)$$



simright.goto

```

1    $x_\omega := x_\omega \text{ div } b;$ 
2    $x_\omega := \langle\sigma'\rangle + b \cdot x_\omega;$ 
3    $x_v := b \cdot x_v + (x_\omega \text{ mod } b);$ 
4    $x_\omega := x_\omega \text{ div } b;$ 
5    $x_s := j$ 

```

Abbildung 6.51: Der Konfigurationsübergang einer Turing-Maschine lässt sich innerhalb eines Goto-Programms durch eine Folge arithmetischer Operationen nachbilden.

Die Zahl b ist die *Basis* der Zahlendarstellung und wird so gewählt, dass sie die Anzahl der Symbole des Bandalphabets Π übersteigt. Hierdurch erreichen wir eine eindeutige Abbildung von Π^* in die Menge der natürlichen Zahlen. Vielleicht ist Ihnen aufgefallen, dass die Gleichungen (6.12) und (6.13) für den Fall $b = 10$ nichts weiter als eine mathematische Beschreibung der uns vertrauten Dezimalzahlen sind. Jeder Wert $\langle\rho_i\rangle$ bzw. $\langle\sigma_i\rangle$ entspricht in diesem Fall einer einzelnen Ziffer. Die Ziffer 0 wird in unserer Darstellung bewusst vermieden, da die Folgen 0, 00 und 000 allesamt einen unterschiedlichen Bandinhalt repräsentieren, numerisch aber nicht voneinander unterschieden werden können.

Abbildung 6.51 zeigt, wie sich ein Konfigurationsübergang mithilfe arithmetischer Operationen simulieren lässt. Zunächst wird das Zeichen σ_1 durch das Folgezeichen σ' ersetzt. Dies geschieht in zwei Schritten. Im ersten Schritt wird das Zeichen σ_1 aus ω entfernt, indem der Wert x_ω durch die Basis b ganzzahlig dividiert wird. Im zweiten Schritt wird x_ω mit b multipliziert und der Wert $\langle\sigma'\rangle$ addiert. Hierdurch wird das Zeichen σ' an das linke Ende von ω angehängt.

Anschließend wird die Bewegung des Schreib-Lese-Kopfes simuliert. Im Falle einer Linksbewegung wird das Zeichen ρ_1 mithilfe der Modulo-Operation bestimmt und der Zeichenkette ω von links hinzugefügt. Die anschließende Division von x_v durch b führt dazu, dass das Zeichen ρ_1 aus v verschwindet. Die Rechtsbewegung erfolgt analog. Jetzt wird das Zeichen σ_1 bestimmt und von rechts an die Zeichenkette v angehängt. Danach wird σ_1 durch die Divisionsanweisung aus der Zeichenkette ω entfernt. Beschreiben wir die Variable x_s am Ende noch mit dem Index des Folgezustands, so ist der Konfigurationsübergang vollständig ausgeführt.

Fügen wir die Puzzle-Stücke in der richtigen Art und Weise zusammen, so können wir jede Turing-Maschine mithilfe eines Goto-Programms simulieren. Wie in Abbildung 6.52 (links) skizziert, besteht das Programm auf der obersten Ebene aus drei sequenziell durchlaufenden Teilen. In M_1 werden die intern verwendeten Variablen x_v , x_ω und x_s initialisiert. In M_3 wird der Inhalt der Variablen analysiert und das berechnete Ergebnis in die AusgabevARIABLE x_0 geschrieben. Beide Programmteile sind offensichtlich Goto-berechenbar, da sie lediglich eine Reihe von Umcodierungen vornehmen.

Die eigentliche Arbeit wird in Programmabschnitt M_2 verrichtet (vgl. Abbildung 6.52 rechts). Zu Beginn wird mithilfe der Modulo-Operation der Index des Zeichens σ_1 ermittelt und in der Variablen x_σ gespeichert. In Abhängigkeit von x_σ und dem Inhalt der Zustandsvariablen x_s springt das Programm eine bestimmte Marke an. Konkret implementiert der

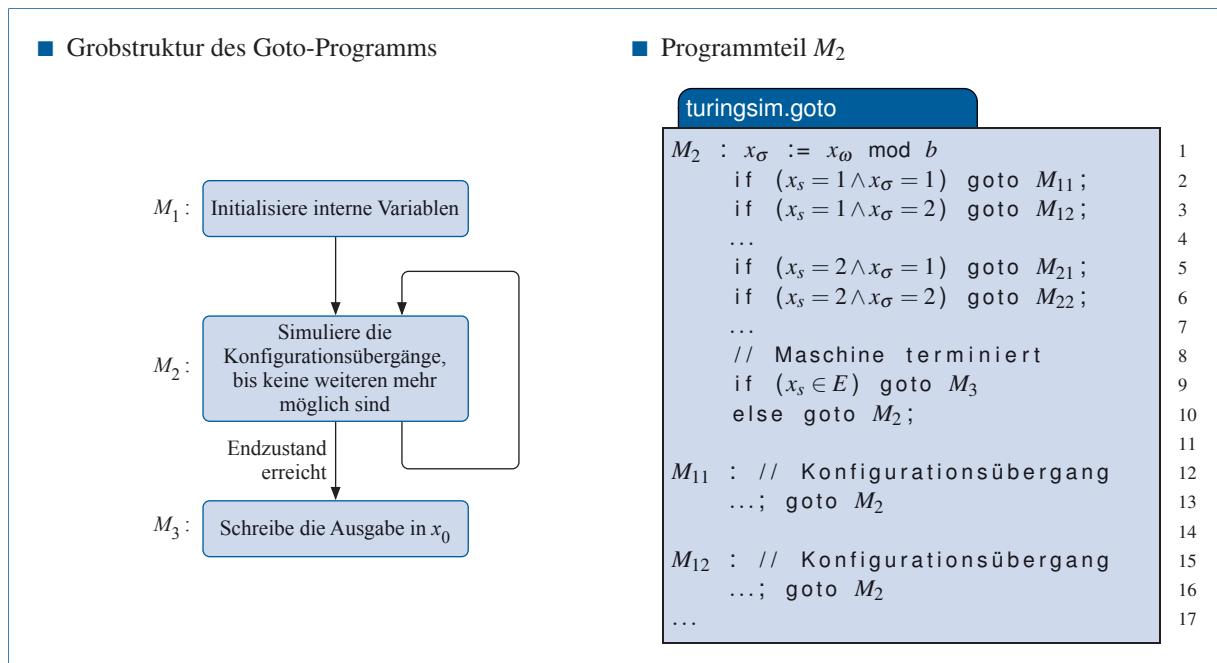


Abbildung 6.52: Jede Turing-Maschine lässt sich mit einem Goto-Programm simulieren.

Programmtext ab Marke $M_{i(\sigma)}$ den Zustandsübergang $\delta(s_i, \sigma)$ nach dem in Abbildung 6.51 entwickelten Schema.

Damit haben wir gezeigt, dass sich jede Turing-Maschine durch ein Goto-Programm simulieren lässt, und Satz 6.4 stellt sicher, dass auch die Übersetzung in ein While-Programm gelingt. Damit sind wir am Ziel unserer Überlegungen angekommen und dürfen den folgenden Satz in unseren Wissensfundus aufnehmen:

Satz 6.8

Jede Turing-berechenbare Funktion ist While-berechenbar.

Zusammen beweisen die Sätze 6.7 und 6.8 die Äquivalenz zwischen While-Programmen und Turing-Maschinen. Beziehen wir die bisher erarbeiteten Äquivalenzen ebenfalls in die Betrachtung mit ein, so entsteht das in Abbildung 6.53 dargestellte Gesamtbild. Zwei Aspekte sind an dieser Stelle von besonderer Bedeutung:

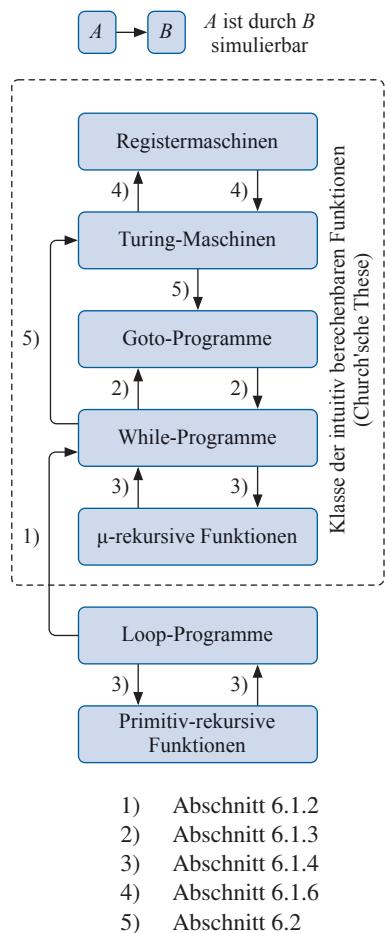


Abbildung 6.53: Fast alle Berechnungsmodelle besitzen ihren äußerlichen Unterschieden zum Trotz exakt die gleiche Ausdrucksstärke. Diese empirische Beobachtung veranlasste Alonzo Church zur Formulierung seiner berühmten These.

■ So verschieden die Ansätze auch sind: nahezu alle Berechnungsmodelle besitzen die gleiche Ausdrucksstärke. Der Berechenbarkeitsbegriff bleibt damit stets derselbe, egal ob wir ihn über die While-Sprache, die Goto-Sprache, das Konzept der Turing-Maschine oder eines der anderen vorgestellten Modelle definieren. Eine Ausnahme bildet die Loop-Sprache. In Abschnitt 6.1.1 haben wir am Beispiel der Ackermann-Funktion gezeigt, dass diese ausdrucksschwächer ist als die While-Sprache.

■ In der Vergangenheit wurde mehrmals versucht, die Menge der berechenbaren Funktionen durch die Angabe eines ausdrucksstärkeren Berechnungsmodells zu vergrößern. Allen Anstrengungen zum Trotz wurde ein ausdrucksstärkeres Berechnungsmodell bis heute nicht gefunden. Selbst so ausgefallene Konzepte wie der *Quantenrechner* [79] oder das *DNA computing* [4] konnten die Grenze des maschinell Berechenbaren nicht verschieben.

Der amerikanische Mathematiker Alonzo Church vermutete bereits im Jahr 1936, dass der intuitive Berechenbarkeitsbegriff mit dem Begriff der Turing-Berechenbarkeit zusammenfällt. Genau dies ist der Inhalt der berühmten *Church'schen These* [19]:

These 6.1 (Church, 1936)

Die Klasse der Turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein.

Der Begriff der *intuitiv berechenbaren Funktion* bedarf an dieser Stelle besonderer Aufmerksamkeit. Er bezeichnet eine Funktion, die von einem Menschen – in welcher Form auch immer – ausgerechnet werden kann. Damit besagt die Church'sche These nichts anderes, als dass jede Funktion, die überhaupt in irgendeiner Weise berechenbar ist, auch durch eine Turing-Maschine berechnet werden kann.

Die Church'sche These ist kein Satz im mathematisch präzisen Sinne, da der Begriff der intuitiv berechenbaren Funktion keine formale Definition besitzt. Gäbe es diese, so hätten wir uns – bewusst oder unbewusst – bereits auf ein konkretes Berechnungsmodell festgelegt und die eigentliche Bedeutung dieses Begriffs ad absurdum geführt. Folgerichtig wird es niemals möglich sein, die Church'sche These zu beweisen. Wir können lediglich Indizien für ihre Gültigkeit sammeln und genau dies ist Forschern in der Vergangenheit vielfach gelungen.

6.3 Entscheidbarkeit

Nachdem wir in den vorangegangenen Abschnitten den Berechenbarkeitsbegriff formal erfasst haben, sind wir nun in der Lage, den bereits mehrfach gefallenen Begriff der *Entscheidbarkeit* ebenfalls auf ein stabiles Fundament zu stellen.



Definition 6.15 (Entscheidbarkeit, Semi-Entscheidbarkeit)

Eine Sprache $L \subseteq \Sigma^*$ heißt *entscheidbar*, falls die *charakteristische Funktion* $\chi_L : \Sigma^* \rightarrow \{0, 1\}$ mit

$$\chi_L(\omega) = \begin{cases} 1 & \text{falls } \omega \in L \\ 0 & \text{falls } \omega \notin L \end{cases}$$

berechenbar ist. L heißt *semi-entscheidbar*, falls die *partielle charakteristische Funktion* $\chi'_L : \Sigma^* \rightarrow \{1\}$ mit

$$\chi'_L(\omega) = \begin{cases} 1 & \text{falls } \omega \in L \\ \perp & \text{falls } \omega \notin L \end{cases}$$

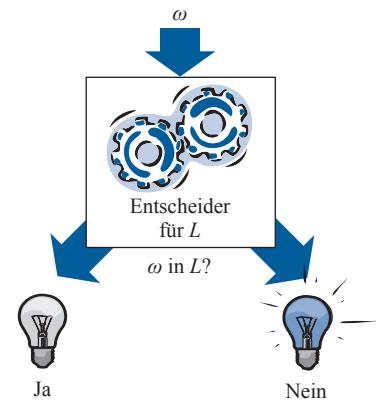
berechenbar ist.

Im Kern dieser Definition steht der Begriff der *charakteristischen Funktion*. Sie ist das formale Bindeglied zwischen dem auf Funktionen ausgelegten Berechenbarkeitsbegriff und dem für Mengen bzw. Sprachen formulierten Entscheidbarkeitskriterium.

Abbildung 6.54 demonstriert, wie sich die beiden Entscheidbarkeitsbegriffe bildlich erfassen lassen. Für eine entscheidbare Sprache L existiert eine algorithmisch arbeitende Maschine, die ein Element $\omega \in L$ entgegennimmt und die Frage beantwortet, ob ω zu L gehört oder nicht. In der bildlichen Darstellung werden die beiden möglichen Antworten durch eine separate Glühlampe symbolisiert, von denen genau eine nach endlicher Zeit zu leuchten beginnt. Wann unsere gestellte Frage „Ist $\omega \in L$ “ durch eine glühende Lampe beantwortet wird, steht in den Sternen. Nichtsdestotrotz ist sichergestellt, dass wir sowohl für den Fall $\omega \in L$ als auch für den Fall $\omega \notin L$ irgendwann eine Antwort erhalten werden. Wir müssen uns also lediglich ein wenig in Geduld üben und lange genug warten.

Im Gegensatz zu einem Entscheider besitzt ein *Semi-Entscheider* nur eine einzige Glühlampe. Wird er mit einem Element $\omega \in L$ gestartet, so beginnt die Lampe nach endlicher Zeit zu leuchten. Für $\omega \notin L$ lässt sich

■ Entscheidbarkeit



■ Semi-Entscheidbarkeit

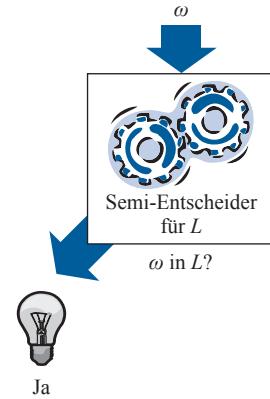


Abbildung 6.54: Bildliche Darstellung der Entscheidungsbegriffe

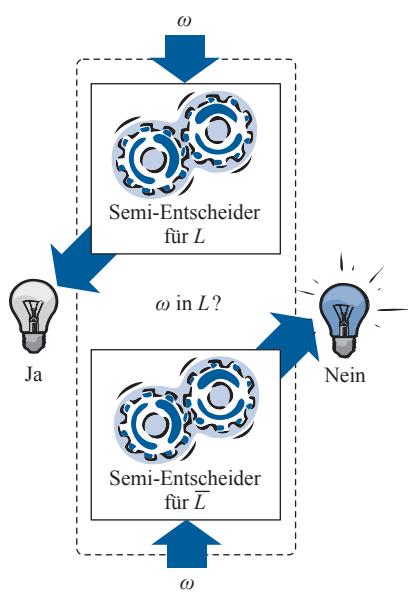


Abbildung 6.55: Sind sowohl L als auch das Komplement \bar{L} semi-entscheidbar, so lässt sich die Sprache L entscheiden.

keine Aussage treffen, da wir nicht wissen können, ob sich die Maschine innerhalb einer Endlosschleife befindet oder zu einem späteren Zeitpunkt vielleicht doch noch eine positive Antwort liefern wird. Damit ist die Semi-Entscheidbarkeit gleichbedeutend mit einer Halbaussage. Die gestellte Frage „Ist $\omega \in L$?“ wird nur im positiven Fall nach endlicher Zeit beantwortet. Fällt die Antwort negativ aus, so zeigt die Maschine keinerlei Reaktion.

In manchen Fällen reicht die Eigenschaft der Semi-Entscheidbarkeit trotzdem aus, um eine Sprache L zu entscheiden. Dies ist immer dann der Fall, wenn neben L auch das Komplement \bar{L} semi-entscheidbar ist. Abbildung 6.55 zeigt auf grafische Weise, wie sich die Semi-Entscheider für L und \bar{L} zu einem Entscheider für L kombinieren lassen. Beide Semi-Entscheider werden mit dem Eingabewort ω versorgt und parallel gestartet. Liegt ω in L , so reagiert der erste Semi-Entscheider nach einer endlichen Zeitspanne. Ist ω nicht in L enthalten, so wird dies durch den zweiten Semi-Entscheider irgendwann angezeigt. Damit haben wir einen konstruktiven Beweis für den folgenden Satz gefunden:

Satz 6.9

Eine Sprache L ist genau dann entscheidbar, wenn sowohl L als auch \bar{L} semi-entscheidbar sind.

Die eingeführten Entscheidbarkeitsbegriffe sind eng mit den Begriffen der *Abzählbarkeit* und der *Aufzählbarkeit* verwandt. Auch diese wollen wir formal einführen:



Definition 6.16 (Abzählbarkeit, Aufzählbarkeit)

Eine Sprache $L \neq \emptyset$ heißt

- *abzählbar*, falls eine bijektive Abbildung $f : \mathbb{N} \rightarrow L$ existiert.
- (*rekursiv*) *aufzählbar*, falls eine surjektive und berechenbare Abbildung $f : \mathbb{N} \rightarrow L$ existiert.

Dem Begriff der Abzählbarkeit sind wir bereits in Kapitel 2 im Rahmen der Mächtigkeitsbetrachtungen verschiedener Mengen begegnet. Schon dort haben wir eine Menge genau dann als abzählbar bezeichnet, wenn es möglich ist, die natürlichen Zahlen und die Elemente von L paarweise einander zuzuordnen. Die bijektive Abbildung f beschreibt, wie

sich die Elemente gruppieren lassen, und liefert uns gleichzeitig eine auflistende Darstellung für L :

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

Beachten Sie, dass die Funktion f lediglich existieren, aber nicht zwangsläufig berechenbar sein muss. Ist Letzteres dennoch der Fall, so sprechen wir von einer *aufzählbaren*, oder ausführlicher von einer *rekursiv aufzählbaren* Menge oder Sprache. Plakativ gesprochen sind solche Sprachen dadurch charakterisiert, dass wir deren Elemente der Reihe nach *aufsagen* können (vgl. Abbildung 6.56). Dies ist möglich, da f berechenbar ist und wir die Funktionswerte $f(i)$ für $i = 0, 1, 2, \dots$ daher nacheinander ausrechnen können.

Beachten Sie ferner, dass es im Fall der Aufzählbarkeit keine Rolle spielt, ob ein Element mehrfach aufgesagt wird; wichtig ist nur, dass jedes Element irgendwann erscheint. Dies ist der Grund, warum wir in Definition 6.15 lediglich gefordert haben, dass die Funktion f surjektiv sein muss und nicht bijektiv, wie im Fall der Abzählbarkeit.

Zwischen der Aufzählbarkeit und der Semi-Entscheidbarkeit einer Sprache besteht eine Verwandtschaft, die sehr viel enger ist, als es der erste Blick vermuten lässt. Zunächst ist jede aufzählbare Sprache L auch semi-entscheidbar, schließlich können wir alle Elemente der Reihe nach aufsagen und genau dann stoppen, wenn wir das gesuchte Element ω gefunden haben. Ist $\omega \in L$, so werden wir das Element nach endlich vielen Schritten antreffen. Ist $\omega \notin L$, so fahren wir für immer fort.

Interessanterweise gilt auch die umgekehrte Schlussrichtung: Jede semi-entscheidbare Sprache L ist aufzählbar. Auf den ersten Blick erscheint unser Vorhaben gewaltig: Wir müssen einen Semi-Entscheider so ansteuern, dass er nacheinander die Elemente von L identifiziert, und gleichzeitig darauf achten, dass er niemals in eine Endlosschleife gerät.

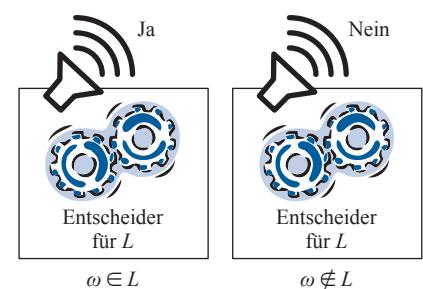
Zum Erfolg verhilft uns erneut die Cantor'sche Paarungsfunktion $\pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ aus Abschnitt 2.3.3. π stellt eine Zuordnung zwischen der Menge aller Tupel $(i, j) \in \mathbb{N}^2$ und der Menge der natürlichen Zahlen her. Um eine semi-entscheidbare Sprache L aufzuzählen, gehen wir wie folgt vor:

- In einer unendlichen Schleife berechnen wir nacheinander die Elemente

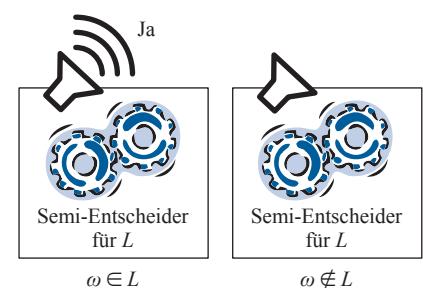
$$\pi^{-1}(0), \pi^{-1}(1), \pi^{-1}(2), \dots$$

Als Ergebnis erhalten wir eine Folge, in der jedes Tupel $(i, j) \in \mathbb{N}^2$ irgendwann auftaucht (vgl. Abbildung 6.57).

■ Entscheidbarkeit



■ Semi-Entscheidbarkeit



■ Aufzählbarkeit

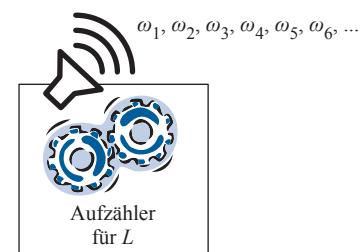


Abbildung 6.56: Entscheidbarkeit, Semi-Entscheidbarkeit und Aufzählbarkeit im Vergleich

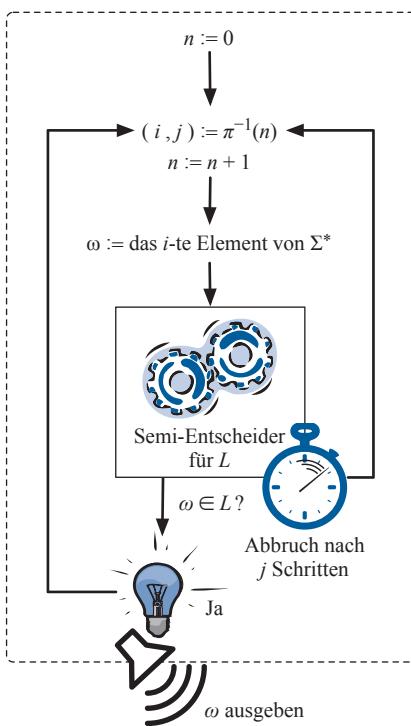


Abbildung 6.57: Mit einigen Tricks und Kniffen ist es möglich, eine *Cantor-Maschine* zu konstruieren, die alle Elemente einer semi-entscheidbaren Sprache nacheinander aufzählt.

■ Für jedes Tupel (i, j) starten wir den Semi-Entscheider mit dem i -ten Element von Σ^* . Stellt dieser die Sprachzugehörigkeit innerhalb von j Schritten fest, so gibt er das Wort aus. Ist der Semi-Entscheider nach j Schritten noch zu keinem Ergebnis gekommen, brechen wir die Berechnung ab und fahren mit dem nächsten Tupel fort. Da für jedes Wort $\omega \in L$ ein $j \in \mathbb{N}$ mit der Eigenschaft existiert, dass der Semi-Entscheider die Sprachzugehörigkeit in j Schritten positiv beantwortet, werden nach und nach die Elemente von L erzeugt.

Damit ist es uns gelungen, den folgenden Satz zu beweisen:

Satz 6.10 (Aufzählbarkeit und Semi-Entscheidbarkeit)

Eine Sprache $L \neq \emptyset$ ist genau dann aufzählbar, wenn sie semi-entscheidbar ist.

Kombinieren wir die Aussagen der Sätze 6.9 und 6.10, so erhalten wir ohne weiteres Zutun das folgende Ergebnis:

Korollar 6.3

Eine Sprache $L \neq \emptyset$ ist genau dann entscheidbar, wenn L und \bar{L} aufzählbar sind.

Beachten Sie, dass wir den Entscheidbarkeitsbegriff für Sprachen und damit für spezielle Mengen formuliert haben, wir aber an vielen Stellen von entscheidbaren oder unentscheidbaren *Problemen* reden. Der Zusammenhang lässt sich einfach herstellen, indem wir für eine beliebige Eigenschaft E zunächst die folgende Sprache definieren:

$$L_E := \{\omega \in \Sigma^* \mid \omega \text{ erfüllt die Eigenschaft } E\}$$

Das Problem „Erfüllt ω die Eigenschaft E ?“ bezeichnen wir als entscheidbar, wenn die zugehörige Sprache L_E entscheidbar ist. Über die gleiche Reduktion lässt sich der Begriff der Semi-Entscheidbarkeit übertragen.

6.4 Akzeptierende Turing-Maschinen

Im Abschnitt 6.3 haben wir gezeigt, wie sich der für Mengen bzw. Sprachen ausgelegte *Entscheidbarkeitsbegriff* durch die Rückführung

auf den Begriff der Berechenbarkeit formal absichern lässt. Von zentraler Bedeutung ist in diesem Zusammenhang die charakteristische Funktion, die gewissermaßen als Bindeglied zwischen den beiden Welten fungiert. Für die Betrachtung von Entscheidungsproblemen ist es manchmal einfacher, diesen Umweg einzusparen und die Definition der Turing-Maschine so anzupassen, dass diese von Natur aus als Entscheider arbeitet. Wie dies funktioniert, werden wir jetzt herausarbeiten.

Wir beginnen mit der Definition der von Turing-Maschinen akzeptierten Sprachen (vgl. Abbildung 6.58):

Satz 6.11 (Turing-Akzeptanz)

Eine Turing-Maschine $T = (S, \Sigma, \Pi, \delta, s_0, \square, E)$ akzeptiert das Wort $\omega \in \Sigma^*$, falls sie unter Eingabe von ω in einem Endzustand terminiert. Die von T akzeptierte Sprache $\mathcal{L}(T)$ ist definiert als

$$\mathcal{L}(T) := \{\omega \in \Sigma^* \mid T \text{ akzeptiert } \omega\}$$

Um zu testen, ob ein Wort ω von einer Turing-Maschine T akzeptiert wird, müssen wir lediglich das Band mit ω befüllen und T starten. Hält die Maschine nach endlich vielen Schritten in einem Endzustand an, so gilt ω als akzeptiert. Terminiert T in einen Zustand $s \notin E$, so wird das Eingabewort ω abgelehnt. Das Gleiche gilt für den Fall, dass T in eine Endlosschleife gerät und überhaupt nicht terminiert (Abbildung 6.59). Beachten Sie, dass die Akzeptanz eines Wortes ω nur davon abhängt, ob der final eingenommene Zustand ein Endzustand ist. Der generierte Bandinhalt ist für die Akzeptanz von ω bedeutungslos.

Offensichtlich gelten die folgenden Zusammenhänge:

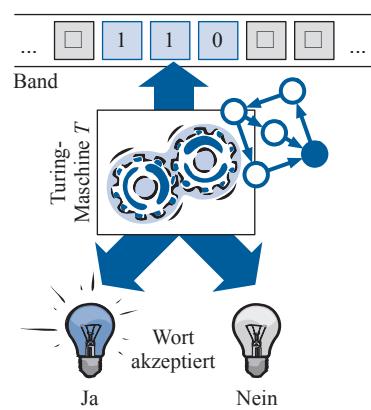
- Eine Sprache L ist genau dann semi-entscheidbar, wenn eine Turing-Maschine T existiert, die L akzeptiert.
- Eine Sprache L ist genau dann entscheidbar, wenn eine Turing-Maschine T existiert, die L akzeptiert und für jede Eingabe terminiert.

Zusammen mit Satz 6.10 erhalten wir

Satz 6.12

Die Menge der von Turing-Maschinen akzeptierten Sprachen entspricht der Menge der rekursiv aufzählbaren Sprachen.

- Fall 1: ω wird akzeptiert ($\omega \in \mathcal{L}(T)$)



- Fall 2: ω wird abgelehnt ($\omega \notin \mathcal{L}(T)$)

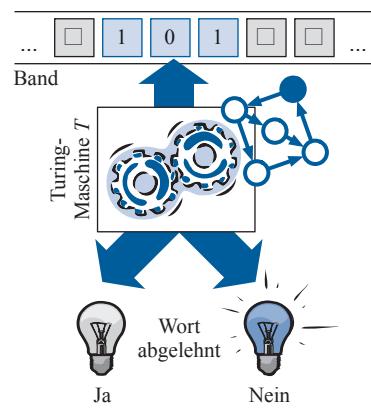
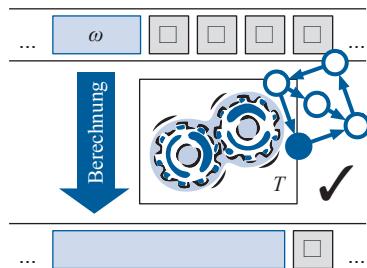


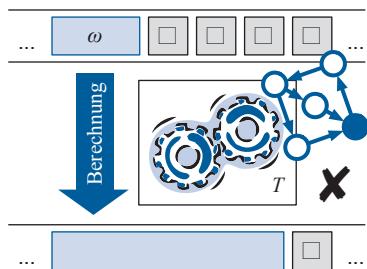
Abbildung 6.58: Die Turing-Maschine als Akzeptor

■ Fall 1



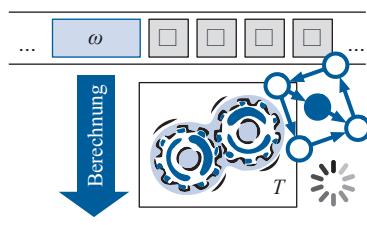
Die Turing-Maschine terminiert
in einem Endzustand $s \in E$
 $\Rightarrow \omega \in \mathcal{L}(T)$

■ Fall 2



Die Turing-Maschine terminiert
in einem Zustand $s \notin E$
 $\Rightarrow \omega \notin \mathcal{L}(T)$

■ Fall 3



Die Turing-Maschine gerät in
eine Endlosschleife
 $\Rightarrow \omega \notin \mathcal{L}(T)$

Abbildung 6.59: Arbeitsweise akzeptierender Turing-Maschinen

Bevor wir mit unseren Betrachtungen über Turing-akzeptierbare Sprachen fortfahren, wollen wir die bisher verwendete Definition der Übergangsfunktion δ geringfügig auflockern. Genau wie im Falle des endlichen Automaten werden wir auch hier nichtdeterministische Zustandsübergänge zulassen und δ zukünftig als *Übergangsrelation* bezeichnen. Die Turing-Maschine wird hierdurch um Entscheidungspunkte angereichert, die in Abhängigkeit der getätigten Auswahl zu unterschiedlichen Berechnungssequenzen führen. Analog zum Konzept des nichtdeterministischen endlichen Automaten wollen wir ein Wort ω als akzeptiert ansehen, wenn mindestens eine Berechnungssequenz in einem Finalzustand endet.

Der Übergang von einer Übergangsfunktion zu einer Übergangsrelation δ wirft die Frage auf, ob der hinzugefügte Nichtdeterminismus das Modell der akzeptierenden Turing-Maschine erweitert oder nur der Schreiberleichterung dient. Im Falle des endlichen Automaten haben wir herausgearbeitet, dass sich jeder nichtdeterministische Automat auf ein deterministisches Pendant reduzieren lässt. Wir werden nun zeigen, dass eine entsprechende Reduktion auch für Turing-Maschinen möglich ist – wenn auch nicht mit derselben Eleganz.

Der Kern der Reduktion basiert auf der Tatsache, dass der eingeschlagene Berechnungspfad an einem Entscheidungspunkt in nur endlich viele unterschiedliche Richtungen gelenkt werden kann. Die Endlichkeit hat zur Folge, dass wir jede terminierende Berechnungssequenz durch eine Zahlenfolge

$$e_1, \dots, e_n \quad \text{mit} \quad 1 \leq e_i \leq w \quad (6.14)$$

repräsentieren können, in der n die Anzahl der getroffenen Entscheidungen und w die maximale Anzahl der Wahlmöglichkeiten bezeichnet. Obwohl unendlich viele Sequenzen der Form (6.14) existieren, lassen sie sich auf einfache Weise nacheinander berechnen. Für $w = 2$ können wir die Sequenzen beispielsweise wie folgt aufzählen:

$$\{(1), (2), (1,1), (1,2), (2,1), (2,2), (1,1,1), (1,1,2), \dots\} \quad (6.15)$$

Wir werden nun eine deterministische Turing-Maschine konstruieren, die alle möglichen Berechnungspfade der Reihe nach simuliert. Hierzu erzeugt die Maschine alle endlichen Berechnungsfolgen der Form (6.14) und arbeitet diese anschließend nacheinander ab. Um die Konstruktion übersichtlich zu gestalten, verwenden wir eine 3-Band-Maschine (vgl. Abbildung 6.60). Band 1 – das Eingabeband – enthält eine Kopie des Eingabeworts ω und bleibt während der gesamten Berechnung unverändert. Die anderen beiden Bänder werden benötigt, um

die möglichen Berechnungsfolgen der nichtdeterministischen Turing-Maschine nacheinander zu simulieren. In jeder Einzelsimulation werden drei Schritte durchlaufen:

- Im ersten Schritt wird die zu simulierende Berechnungsfolge auf Band 2 – dem Auswahlband – erzeugt. Hierzu generiert die Maschine nacheinander die in (6.14) beschriebenen Folgen.
- Im zweiten Schritt wird die Master-Kopie vom Eingabeband auf Band 3 – das Arbeitsband – kopiert.
- Jetzt wird auf dem Arbeitsband das Verhalten der nichtdeterministischen Maschine simuliert. In jedem Schritt entscheidet die Zahlenfolge auf Band 2 über den einzuschlagenden Berechnungspfad.

Terminiert die Maschine in einem Endzustand, so haben wir einen akzeptierenden Berechnungspfad gefunden und das Eingabewort ist ein Element der Sprache $\mathcal{L}(T)$. Enthält das Auswahlband eine Folge der Länge n und konnte nach n Schritten noch kein Endzustand erreicht werden, so wird die Simulation abgebrochen und der Vorgang mit der nächsten Berechnungsfolge wiederholt.

Auf diese Weise wird das Verhalten der nichtdeterministischen Maschine vollständig nachgeahmt. Akzeptiert diese das Wort ω , so existiert ein akzeptierender Berechnungspfad. Dieser wird von der konstruierten deterministischen Maschine irgendwann simuliert und das Wort ω ebenfalls akzeptiert. Wird ω durch die nichtdeterministische Maschine nicht akzeptiert, so wird das deterministische Pendant eine Sequenz nach der anderen simulieren und stets scheitern. Die Maschine terminiert nicht und ω ist kein Element der akzeptierten Sprachen. Damit haben wir den folgenden Satz bewiesen:

Satz 6.13

Jede nichtdeterministische Turing-Maschine kann durch eine deterministische Turing-Maschine simuliert werden.

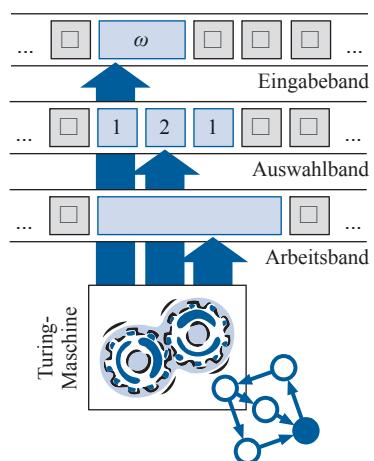


Abbildung 6.60: Simulation einer nichtdeterministischen Turing-Maschine mithilfe einer deterministischen 3-Band-Maschine. Das Eingabeband enthält eine Master-Kopie des Eingabeworts ω und wird nicht verändert. Auf dem Auswahlband werden nacheinander die Berechnungspfade erzeugt, die von der nichtdeterministischen Turing-Maschine durchlaufen werden können. Für jeden Pfad simuliert die Maschine das jeweilige Verhalten auf dem Arbeitsband und geht im Erfolgsfall in einen Endzustand über.

Wir werden die nichtdeterministische Turing-Maschine nun zum Erkennen von Sprachen einsetzen. Dabei wird sich das Maschinenmodell als leistungsfähiger erweisen, als es der erste Blick vermuten lässt. In der Tat werden wir zeigen, dass jede Typ-0-Sprache, d. h. jede Sprache, die sich generativ mithilfe einer Grammatik erzeugen lässt, durch eine Turing-Maschine akzeptiert werden kann.

■ Grammatik G

$$\begin{aligned} S &\rightarrow \varepsilon \\ S &\rightarrow SS \\ S &\rightarrow [S] \\ S &\rightarrow (S) \end{aligned}$$

■ Wortproblem

Gilt $[()] \in \mathcal{L}(G)$?

■ Simulation

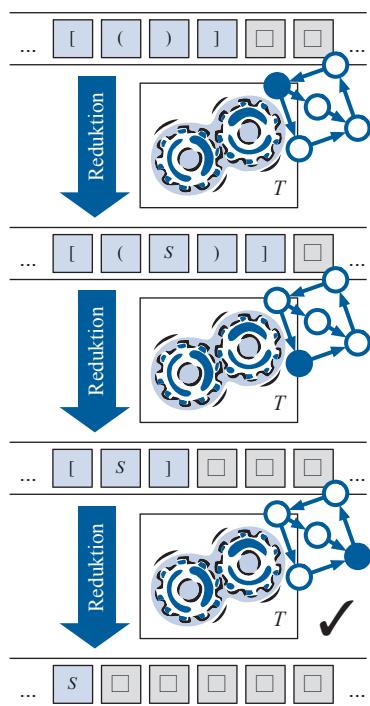


Abbildung 6.61: Für jede Typ-0-Sprache L existiert eine Turing-Maschine T , die L akzeptiert.

Für die nachstehenden Betrachtungen sei mit $G = (V, \Sigma, P, S)$ eine beliebige Typ-0-Grammatik gegeben. Wie in Abschnitt 4.1 dargelegt, besteht die Sprache $\mathcal{L}(G)$ aus allen Wörtern $\omega \in \Sigma^*$, die aus dem Startsymbol S ableitbar sind. Mit anderen Worten: Für jedes Wort $\omega \in \mathcal{L}(G)$ existiert eine Ableitungssequenz der Form

$$S \Rightarrow \omega_1 \Rightarrow \omega_2 \Rightarrow \dots \Rightarrow \omega \quad (6.16)$$

mit $\omega_i \in (\Sigma \cup V)^*$. Um eine akzeptierende Turing-Maschine für $\mathcal{L}(G)$ zu konstruieren, müssen wir die in (6.16) dargestellte Ableitungsrichtung umkehren: Genau dann, wenn es der Maschine gelingt, das Eingabewort ω auf dem Band in das Startsymbol S zurückzuführen, wird es akzeptiert.

Die Rückführung wird durch die Tatsache erschwert, dass im Allgemeinen mehrere anwendbare Regeln zur Auswahl stehen. Durch die geleistete Vorarbeit können wir das Problem jedoch einfach lösen, indem wir die Turing-Maschine nichtdeterministisch konstruieren. Wir nutzen an dieser Stelle unser Wissen aus, dass zwischen nichtdeterministischen und deterministischen Maschinen kein prinzipieller Unterschied besteht.

Das Arbeitsprinzip der konstruierten Maschine folgt dem in Abbildung 6.61 skizzierten Schema. Initial wird das Eingabewort ω auf das Band geschrieben und die Turing-Maschine gestartet. Diese wählt zunächst eine Produktion der Form $v_1 \rightarrow v_2$ aus und sucht anschließend die Zeichenfolge v_2 auf dem Eingabeband. Sowohl die Auswahl als auch die Suche geschehen nichtdeterministisch. Ist die Zeichenkette v_2 nicht vorhanden, so terminiert die Turing-Maschine in einem Zustand außerhalb der Finalmenge, andernfalls wird v_2 durch v_1 ersetzt. Der Austausch muss mit Bedacht erfolgen, da v_2 und v_1 im Allgemeinen aus unterschiedlich vielen Symbolen bestehen. Folgerichtig muss ein Teil des Bandinhalts vor der Ersetzung um eine entsprechende Anzahl von Symbolen nach links oder rechts verschoben werden. Die Maschine geht in einen Endzustand über, sobald das Startsymbol S auf dem Eingabeband steht. Ist der Bandinhalt von S verschieden, so wird der gesamte Vorgang wiederholt. Durch die nichtdeterministische Konstruktion ist sichergestellt, dass S genau dann erzeugt werden kann, wenn eine Ableitungssequenz der Form (6.16) existiert. Damit ist die Gültigkeit des folgenden Satzes bewiesen:

Satz 6.14

Zu jeder Typ-0-Sprache L existiert eine Turing-Maschine, die L akzeptiert.

Handelt es sich bei G um eine kontextsensitive Grammatik, so gilt für alle Produktionen $l \rightarrow r$ die Beziehung $|l| \leq |r|$. Da die simulierende Turing-Maschine T die Ersetzung in umgekehrter Richtung nachvollzieht, bewegt sich der Schreib-Lese-Kopf niemals über die Grenzen des Eingabeworts hinaus. Mit anderen Worten: T ist linear beschränkt. Damit haben wir ein weiteres wichtiges Resultat der Berechenbarkeitstheorie erzielt:



Satz 6.15

Zu jeder Typ-1-Sprache L existiert eine nichtdeterministische, linear beschränkte Turing-Maschine, die L akzeptiert.

Anders als in Satz 6.14 dürfen wir auf den Zusatz „nichtdeterministisch“ nicht verzichten. Zwar haben wir gezeigt, dass sich jede nichtdeterministische Turing-Maschine durch eine deterministische simulieren lässt, allerdings ändert die von uns gezeigte Reduktion den benötigten Bandplatz. Die konstruierte Maschine ist damit nicht mehr länger linear beschränkt.

Die Umkehrung der Sätze 6.14 und 6.15 gilt ebenfalls, d. h., die von allgemeinen bzw. linear beschränkten Turing-Maschinen akzeptierten Sprachen gehen nicht über die Typ-0- bzw. die Typ-1-Sprachen hinaus.

Wir wollen nun grob skizzieren, wie sich ein Turing-Akzeptor T in eine äquivalente Grammatik G übersetzen lässt. Die Konstruktion basiert auf der Grundidee, die Konfigurationen von T vollständig in die Wortmenge von G hineinzucodieren. Ist $T = (S, \Sigma, \Pi, \delta, s_0, \square, E)$ die zu simulierende Turing-Maschine und $G = (V, \Sigma_G, P, S)$ die zu konstruierende Grammatik, dann wählen wir die Mengen V und Σ_G wie folgt:

$$V := (S \times \Pi), \quad \Sigma_G := \Pi.$$

Die Wahl ermöglicht uns, jede Konfiguration

$$((\rho_m \dots \rho_1), s, (\sigma_1 \dots \sigma_n))$$

von T in das Wort

$$\rho_m \dots \rho_1(s, \sigma_1) \sigma_2 \dots \sigma_n \in (V \cup \Sigma_G)^*$$

zu übersetzen. Ferner zeigt Abbildung 6.62, wie die Übergangsrelation δ in direkter Weise auf die Produktionenmenge P abgebildet werden kann. Die erzeugten Regeln sind so gestaltet, dass alle Berechnungsschritte von T eins zu eins nachgeahmt werden können. Fügen wir die Einzelteile in der richtigen Art und Weise zusammen, so entsteht eine Grammatik, die $\mathcal{L}(T)$ nach dem folgenden Arbeitsprinzip erkennt:

■ Reduktionsziel

Turing-Maschine
 $T = (S, \Sigma, \Pi, \delta, s_0, \square, E)$



Grammatik
 $G = (V, \Sigma_G, P, S)$

■ Simulation der Linksbewegung

$$\delta(s, \sigma) = (s', \sigma', \leftarrow)$$



$$\rho(s, \sigma) \rightarrow (s', \rho) \sigma'$$

■ Simulation der Rechtsbewegung

$$\delta(s, \sigma) = (s', \sigma', \rightarrow)$$



$$(s, \sigma) \rho \rightarrow \sigma' (s', \rho)$$

Abbildung 6.62: Um eine Turing-Maschine T in eine Grammatik G mit $\mathcal{L}(T) = \mathcal{L}(G)$ zu übersetzen, wird die Übergangsrelation δ eins zu eins auf die Produktionenmenge von G abgebildet.

In diesem Abschnitt haben wir herausarbeitet, dass zwischen Typ-0- und Typ-1-Sprachen auf der einen Seite und den akzeptierenden Turing-Maschinen auf der anderen Seite ein enger Zusammenhang besteht. Insbesondere attestierte uns Satz 6.17, dass die Typ-1-Sprachen genau diejenigen sind, die sich von nichtdeterministischen, linear beschränkten Turing-Maschinen akzeptieren lassen. Auf das Wort „nichtdeterministisch“ können wir nicht ohne Weiteres verzichten, da die Methode, mit der wir einen nichtdeterministischen in einen deterministischen Turing-Akzeptor übersetzen haben, die Eigenschaft der linearen Beschränktheit zerstört. Natürlich wäre es trotzdem denkbar, dass eine Reduktion existiert, die einen nichtdeterministischen linear beschränkten Akzeptor (LBA) in einen deterministischen Akzeptor übersetzt und die Eigenschaft der linearen Beschränktheit bewahrt. Diese Fragestellung wird als *erstes LBA-Problem* bezeichnet.

Bis heute wartet das erste LBA-Problem auf seine Lösung. Würde es eine positive Antwort erfahren, so könnten wir das Wort „nichtdeterministisch“ in Satz 6.17 streichen. In diesem Fall wären die Klasse der Typ-1-Sprachen und die Klasse der linear beschränkten (deterministischen) Turing-Maschinen identisch.

Das erste LBA-Problem ist nicht zu verwechseln mit dem *zweiten LBA-Problem*. Hinter diesem verbirgt sich die Frage nach der Komplement-Abgeschlossenheit linear beschränkter Turing-Maschinen. Diese Eigenschaft bedeutet, dass mit jeder Sprache $L \subseteq \Sigma^*$, die von einer linear beschränkten Turing-Maschine akzeptiert wird, auch das Komplement \bar{L} von einem LBA akzeptiert wird. Seit seiner Formulierung in den Sechzigerjahren rechnete man fast 30 Jahre mit einer negativen Antwort. Erst im Jahr 1987 wurde das Rätsel um das zweite LBA-Problem gelüftet – zur Überraschung vieler mit einer positiven Antwort [53, 95].

- Im ersten Schritt wird die Startkonfiguration für ein beliebiges Wort $\omega \in \Sigma_G^*$ abgeleitet. Hierzu wird die Produktionenmenge von P mit entsprechenden Regeln versehen.
- Ausgehend von den erzeugten Startkonfigurationen wird das Verhalten der Turing-Maschine simuliert. Die Grundlage hierfür bilden die in Abbildung 6.62 dargestellten Produktionen.
- Terminierte die simulierte Turing-Maschine in einem Finalzustand $s \in E$, so wird das Eingabewort ω wiederhergestellt. Das entstehende Wort ist am Ende frei von Nonterminalzeichen und damit Bestandteil von $\mathcal{L}(G)$. Terminiert die Maschine in einem Zustand $s \notin E$, so unterbleibt die Rekonstruktion. Damit ist ω nicht aus dem Startsymbol ableitbar und folglich kein Bestandteil von $\mathcal{L}(G)$. Gerät die Turing-Maschine in eine Endlosschleife, so produziert die Grammatik eine unendliche Ableitungssequenz. Dies ist ebenfalls gleichbedeutend mit $\omega \notin \mathcal{L}(G)$.

Jede von einer Turing-Maschine T akzeptierte Sprache L lässt sich somit von einer Typ-0-Grammatik erzeugen. In Kombination mit Satz 6.14 erhalten wir das folgende Ergebnis:



Satz 6.16 (Turing-Maschinen und Typ-0-Sprachen)

Die Klasse der Typ-0-Sprachen ist mit der Klasse der von Turing-Maschinen akzeptierten Sprachen identisch.

Der Zusammenhang zwischen der simulierten Turing-Maschine T und der erzeugten Grammatik G geht noch weiter. Ist T linear beschränkt, d. h., bewegt sich der Schreib-Lese-Kopf von T niemals über die Grenzen des Eingabeworts hinaus, so lässt sich die Grammatik so formulieren, dass alle Produktionen $l \rightarrow r$ die Beziehung $|l| \leq |r|$ erfüllen. Mit anderen Worten: G ist eine Typ-1-Grammatik. Beziehen wir die Aussage von Satz 6.15 mit ein, so ergibt sich der folgende Zusammenhang:



Satz 6.17 (Turing-Maschinen und Typ-1-Sprachen)

Die Klasse der Typ-1-Sprachen ist mit der Klasse der Sprachen identisch, die von nichtdeterministischen, linear beschränkten Turing-Maschinen akzeptiert werden.

Damit entpuppen sich die verschiedenen Varianten von Turing-Akzeptoren und Grammatiken als alternative Beschreibungsformen für exakt dieselben Sprachklassen.

6.5 Unentscheidbare Probleme

In diesem Abschnitt wollen wir uns einem der wichtigsten Erkenntnisse der Berechenbarkeitstheorie zuwenden. Die Rede ist von der Existenz unentscheidbarer Mengen oder gleichbedeutend: der Existenz unentscheidbarer Probleme. Für viele Software- und Hardware-Entwickler mag die Beschäftigung mit dieser Materie als wenig praxisrelevant erscheinen, schließlich können sich die wenigsten wissenschaftlich daran erinnern, mit einem unentscheidbaren Problem jemals konfrontiert gewesen zu sein.

Dass es unentscheidbare Probleme geben muss, ist nach den geleisteten Vorarbeiten aber leicht einzusehen. Ist eine Menge $M \subseteq \Sigma^*$ entscheidbar, so existiert eine Turing-Maschine T_M zur Berechnung der charakteristischen Funktion χ_M . Da nur abzählbar viele Turing-Maschinen existieren, können demzufolge auch nur abzählbar viele entscheidbare Mengen existieren. In Abschnitt 2.1.2 haben wir jedoch herausgearbeitet, dass die Menge 2^{Σ^*} überabzählbar viele Elemente enthält. Plakativ gesprochen besagt dieses Ergebnis, dass nicht genug Turing-Maschinen zur Verfügung stehen, um alle Mengen $M \subseteq \Sigma^*$ zu entscheiden (vgl. Abbildung 6.63). Damit ist die Existenz unentscheidbarer Mengen unausweichlich.

Die getätigte Überlegung wirft die Frage auf, warum wir in der Praxis dennoch selten auf unentscheidbare Probleme stoßen. Hauptverantwortlich ist die Tatsache, dass wir die Probleme, mit denen wir uns hauptsächlich beschäftigen, nicht zufällig wählen. Die meisten aus unserer Sicht *interessanten* Fragestellungen weisen eine vergleichsweise reguläre Struktur auf und sind aus diesem Grund sehr häufig entscheidbar. Nichtsdestotrotz werden die Beispiele in diesem Abschnitt die weitverbreitete Ansicht widerlegen, dass unentscheidbare Probleme ausschließlich pathologischer Natur sind. In der Tat reihen sich in diese Problemklasse viele Fragestellungen ein, die in der Praxis von handfestem Interesse sind. Die Konsequenzen der Berechenbarkeitstheorie sind damit weit mehr als mathematische Spielereien; sie zeigen uns unverrückbare Grenzen auf, die tief in die Praxis der modernen Soft- und Hardware-Entwicklung hineinreichen.

6.5.1 Halteproblem

Wir beginnen unsere Untersuchungen mit verschiedenen Varianten des *Halteproblems* für Turing-Maschinen. Anschließend werden wir mit den gewonnenen Ergebnissen den Satz von Rice beweisen. Dieser Satz

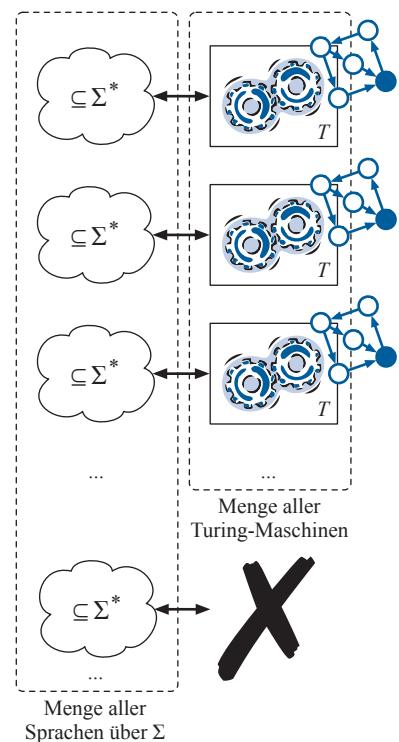


Abbildung 6.63: Es existieren überabzählbar viele Teilmengen von Σ^* , aber nur abzählbar viele Turing-Maschinen. Damit ist die Existenz unentscheidbarer Mengen unausweichlich.

Tabelle 6.4: Ein einfaches Diagonalisierungsargument zeigt die Unentscheidbarkeit des Halteproblems. In der dargestellten Tabelle sind horizontal alle Wörter $\omega \in \Sigma^*$ und vertikal alle Turing-Maschinen aufgelistet. Der Tabelleneintrag (i, j) gibt Antwort auf die Frage, ob die Turing-Maschine T_i unter Eingabe von ω_j hält. Wäre das Halteproblem entscheidbar, so ließe sich eine Turing-Maschine H' konstruieren, die den Diagonaleintrag (i, i) bestimmt und genau dann terminiert, wenn die gefundene Antwort „nein“ lautet. H' taucht aber nicht in der Liste auf, im Widerspruch zur Tabellenkonstruktion.

	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7
T_1	ja	ja	nein	ja	nein	ja	nein
T_2	nein	ja	nein	ja	ja	nein	ja
T_3	ja	ja	nein	nein	nein	ja	ja
T_4	ja	nein	nein	ja	ja	ja	nein
T_5	ja	ja	nein	ja	nein	nein	nein
T_6	nein	nein	ja	ja	ja	nein	ja
T_7	nein	nein	ja	ja	ja	nein	nein

wird von so allgemeiner Natur sein, dass sich hieraus viele Aussagen der Berechenbarkeitstheorie fast von selbst ergeben.



Definition 6.17 (Allgemeines Halteproblem)

Das *allgemeine Haltproblem* lautet wie folgt:

- Gegeben: Turing-Maschine T und Eingabewort ω
- Gefragt: Terminiert T unter Eingabe von ω ?

Mit seiner wegbereitenden Arbeit aus dem Jahr 1936 zerschlug Turing alle Hoffnungen, das Halteproblem zu lösen.



Satz 6.18 (Turing, 1936)

Das allgemeine Halteproblem ist unentscheidbar.

Die Aussage dieses Satzes ist weitreichend. Er besagt, dass kein systematisches Verfahren existieren kann, das für alle Turing-Maschinen T und alle Wörter ω immer korrekt entscheidet, ob T , angewendet auf ω , nach endlich vielen Schritten terminiert.

Für den Beweis der Unentscheidbarkeit nehmen wir an, dass ein Entscheidungsverfahren für das Halteproblem existiert, und führen die Annahme zu einem Widerspruch. Diesen werden wir durch ein Diagonalisierungsargument herbeiführen, das jenem aus Abschnitt 2.3.3 sehr

ähnlich ist. Dort haben wir das Prinzip der Diagonalisierung verwendet, um die Überabzählbarkeit der reellen Zahlen zu zeigen.

Damit wir das Diagonalisierungsargument anwenden können, konstruieren wir zunächst eine Matrix, wie sie in Tabelle 6.4 ausschnittsweise dargestellt ist. Auf der vertikalen Achse listen wir alle Turing-Maschinen auf. Wählen wir die Reihenfolge so, dass alle Maschinen anhand ihrer Gödelnummer in aufsteigender Reihenfolge angeordnet sind, so ist gewährleistet, dass sich jede Turing-Maschine eindeutig einer bestimmten Zeile zuordnen lässt und damit mit Sicherheit in der vorgenommenen Auflistung vorkommt. Auf der horizontalen Achse ordnen wir die Wortmenge Σ^* so an, dass jedes Element $\omega \in \Sigma^*$ in einer bestimmten Spalte erscheint. Abschließend verzeichnen wir in der i -ten Zeile und der j -ten Spalte, ob die Turing-Maschine T_i unter Eingabe von ω_j nach endlich vielen Schritten terminiert.

Wäre das Halteproblem entscheidbar, so würde eine Turing-Maschine H existieren, die neben einem Eingabewort ω eine Turing-Maschine T in codierter Form entgegennimmt und stets korrekt bestimmt, ob T bei Eingabe von ω terminiert. Die fiktive Turing-Maschine H ist nichts anderes als eine Maschine zur Berechnung der soeben konstruierten Matrix. Wie in Abbildung 6.64 skizziert, konstruieren wir aus H eine zweite Maschine H' . Diese berechnet für das Eingabewort ω_i zunächst das Matrixelement (i, i) und verhält sich reziprok zu der erhaltenen Antwort. Hält die Maschine T_i bei Eingabe von ω an $((i, i) = „ja“)$, so geht H' in eine Endlosschleife über. Rechnet T_i dagegen für immer weiter $((i, i) = „nein“)$, so terminiert H' in einem Finalzustand.

Da H' selbst eine Turing-Maschine ist, müssen wir sie in einer bestimmten Zeile unserer konstruierten Matrix vorfinden; der Aufbau der Matrix garantiert ja gerade, dass alle Maschinen der Reihe nach aufgezählt werden. Doch egal, in welcher Zeile wir auch nachschauen: Die Diagonalkonstruktion führt immer einen Widerspruch herbei. Für alle $i \in \mathbb{N}$ gilt $H' \neq T_i$, da T_i die Eingabe ω_i genau dann akzeptiert, wenn sie von H' abgelehnt wird. Der Widerspruch macht deutlich, dass wir die Annahme über die Existenz von H fallen lassen müssen und es keine Maschine geben kann, die das Halteproblem entscheidet.

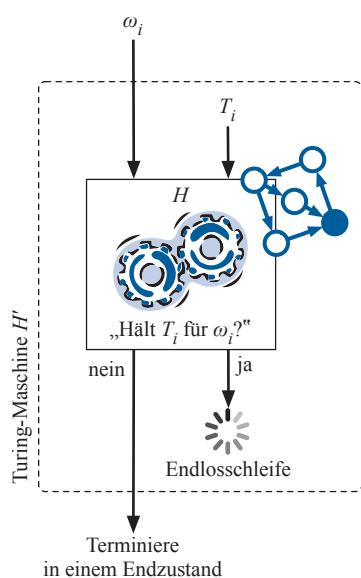


Abbildung 6.64: Gäbe es eine Turing-Maschine H , die das Halteproblem entscheidet, so könnten wir diese zu einer Maschine H' umbauen, die genau dann für das Eingabewort ω_i terminiert, wenn die Turing-Maschine T_i bei Eingabe von ω_i unendlich lange rechnet. Mithilfe des Diagonalisierungsarguments können wir die Konstruktion von H' als widersprüchlich entlarven und daraus schließen, dass die Maschine H nicht existieren kann.

Halteproblem auf leerem Band

Neben dem allgemeinen Halteproblem existiert eine abgeschwächte Variante, die wie folgt definiert ist:

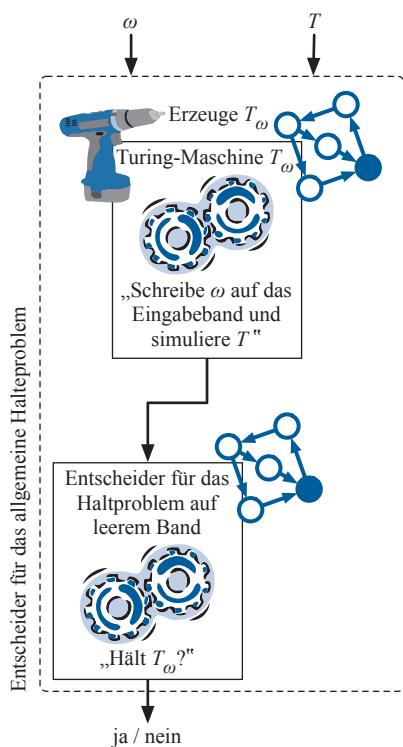


Abbildung 6.65: Reduktion des allgemeinen Halteproblems auf das Halteproblem auf leerem Band. Wären wir in der Lage, das Halteproblem auf leerem Band zu lösen, so könnten wir einen Entscheider für das allgemeine Halteproblem konstruieren. Aus der Unentscheidbarkeit des allgemeinen Halteproblems folgt unmittelbar, dass auch das Halteproblem auf leerem Band nicht entschieden werden kann.

Definition 6.18 (Halteproblem auf leerem Band)

Das *Halteproblem auf leerem Band* lautet wie folgt:

- Gegeben: Turing-Maschine T
- Gefragt: Terminiert T mit der Eingabe ε ?

Während das allgemeine Halteproblem fordert, dass wir die Terminierungseigenschaft für beliebige Turing-Maschinen und beliebige Eingaben ω entscheiden können, betrachtet das spezielle Halteproblem nur den Fall $\omega = \varepsilon$, d. h., ein entsprechender Algorithmus müsste das Verhalten einer Turing-Maschine nur für den Fall analysieren, dass sie mit einem leeren Band gestartet wird. Das spezielle Halteproblem ist damit augenscheinlich einfacher zu lösen als das allgemeine.

Nichtsdestotrotz reiht sich auch die abgeschwächte Variante in die Riege der unentscheidbaren Probleme ein. Dieses Ergebnis ist leicht einzusehen, da sich das allgemeine Halteproblem auf das Halteproblem auf leerem Band zurückführen lässt. Abbildung 6.65 zeigt, wie eine entsprechende Reduktion durchgeführt werden kann. Um zu entscheiden, ob eine Turing-Maschine für ein Eingabewort $\omega \in \Sigma^*$ hält, konstruieren wir zunächst eine Turing-Maschine T_ω , die alle Zeichen von ω auf das Band schreibt und anschließend T simuliert. T_ω wird mit einem leeren Band gestartet und terminiert genau dann, wenn die Originalmaschine T mit der Eingabe ω terminiert. Gäbe es eine Turing-Maschine, die das Halteproblem auf leerem Band entscheidet, so wären wir demnach auch in der Lage, das allgemeine Halteproblem zu entscheiden. Aus Satz 6.18 folgt daher sofort, dass auch das spezielle Halteproblem unentscheidbar sein muss.

Satz 6.19

Das Halteproblem auf leerem Band ist unentscheidbar.

6.5.2 Satz von Rice

Die Unentscheidbarkeit des Halteproblems hat uns gezeigt, dass Aussagen über Turing-Maschinen existieren, die sich einer maschinellen Beweisbarkeit entziehen. Kein algorithmisches Verfahren ist in der Lage, die Terminierungseigenschaft für alle Turing-Maschinen T_i und alle Eingabewörter ω_j stets korrekt vorherzusagen. Durch eine geeignete

Reduktion waren wir darüber hinaus in der Lage, auch das Halteproblem auf leerem Band als unentscheidbar zu identifizieren. In diesem Abschnitt wollen wir uns mit der Frage beschäftigen, ob noch weitere Aussagen über Turing-Maschinen existieren, die nicht algorithmisch entschieden werden können. So viel vorweg: Wir werden eine verblüffende Antwort erhalten.

Im Folgenden bezeichnet T eine beliebige Turing-Maschine, f_T die von T berechnete Funktion und E eine funktionale Eigenschaft von T (also eine Eigenschaft von f_T). E soll *nichttrivial* sein, d.h., es gibt mindestens eine Maschine, die die untersuchte Eigenschaft besitzt, und mindestens eine Maschine, die sie nicht besitzt. Die folgende Aufzählung enthält eine exemplarische Auswahl möglicher Eigenschaften.

- T berechnet eine konstante Funktion
- Alle Ausgaben von T sind mindestens n Zeichen lang
- T berechnet eine totale Funktion
- T generiert zweimal das gleiche Zeichen in Folge

Der Phantasie sind an dieser Stelle keine Grenzen gesetzt. Wir wollen nun ausloten, welche Konsequenzen sich aus der Existenz eines Entscheidungsverfahrens für E ergeben. Hierzu führen wir zunächst die Turing-Maschine T_{\perp} ein, die für keine Eingabe terminiert. Zugegebenermaßen ist T_{\perp} eine vergleichsweise langweilige Maschine, da sie die überall undefinierte Funktion berechnet. Für den Moment wollen wir annehmen, dass T_{\perp} die gewählte Eigenschaft E erfüllt. Da E nichttrivial ist, existiert mindestens eine weitere Maschine $T_{\overline{E}}$, die E nicht erfüllt. Wir fassen zusammen:

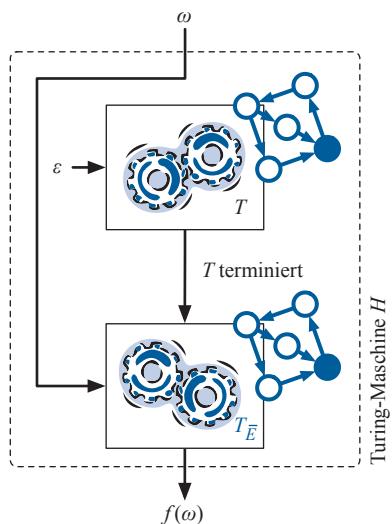
- T_{\perp} erfüllt die Eigenschaft E
 $T_{\overline{E}}$ erfüllt die Eigenschaft E nicht

Die Maschinen T und $T_{\overline{E}}$ vereinen wir nun zu einer gemeinsamen Maschine H . Wie der obere Teil von Abbildung 6.66 zeigt, wird innerhalb von H zunächst die Maschine T mit dem leeren Eingabewort ε gestartet. Hält diese nach endlich vielen Schritten an, so wendet H die Maschine $T_{\overline{E}}$ auf das Eingabewort ω an.

Um das Verhalten von H zu verstehen, unterscheiden wir zwei Fälle:

- T terminiert nicht
 In diesem Fall ist H funktional identisch mit T_{\perp} und erfüllt die Eigenschaft E .

■ Erster Fall: T_{\perp} erfüllt E



■ Zweiter Fall: T_{\perp} erfüllt E nicht

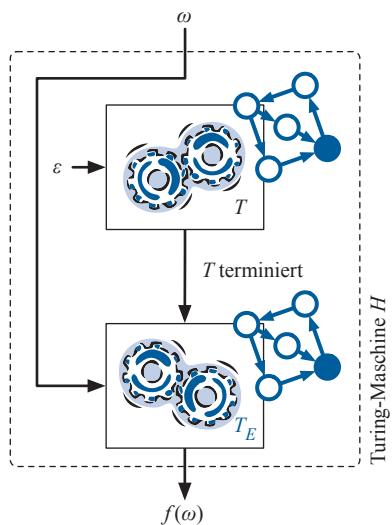


Abbildung 6.66: Kernstück des Beweises für den Satz von Rice. Über die dargestellte Zusammenschaltung wird ein direkter Zusammenhang zwischen der untersuchten Maschineneigenschaft E und dem Halteproblem hergestellt.

Der Satz von Rice macht alle Hoffnungen zunichte, nichttriviale Aussagen über Turing-Maschinen mit einem maschinell arbeitenden Verfahren beweisen zu können. Aufgrund der bewiesenen Äquivalenzen ist die Aussage nicht auf Turing-Maschinen beschränkt und lässt sich ohne Änderung auf die anderen Berechnungsmodelle übertragen. Die Grenzen, die uns der Satz von Rice auferlegt, reichen damit tief in die Praxis der realen Software-Entwicklung hinein. So folgt daraus unmittelbar, dass es keinen Algorithmus geben kann, der für ein beliebiges Programm maschinell verifiziert, ob es sich entsprechend seiner Spezifikation verhält. Selbst so einfache Probleme wie die Frage nach der Existenz von Endlosschleifen entziehen sich einer algorithmischen Lösung.

Im Bereich der Software-Verifikation wurden in der Vergangenheit zahlreiche Methoden und Verfahren entwickelt, die zum Ziel haben, gewisse Eigenschaften über Programme formal zu beweisen. Verliert die Software-Verifikation durch den Satz von Rice nicht auf einen Schlag ihre Berechtigung? Die Antwort ist Nein. Unbestritten folgt aus dem Satz von Rice, dass kein Verifikationswerkzeug existieren kann, das für *jedes* Programm die korrekte Antwort liefert. Er schließt jedoch nicht aus, dass Verfahren existieren, die für die *meisten* Programme oder vielleicht sogar für *alle praxisrelevanten* Programme korrekt arbeiten. In der Tat wurde in der Vergangenheit eine Vielzahl von Algorithmen entwickelt, die theoretisch betrachtet unvollständig sind, in der Praxis aber gut funktionieren.

Trotzdem lehrt uns die Berechenbarkeitstheorie, dass Programme existieren, für die jedes Verifikationswerkzeug versagen muss. Die Unvollständigkeit ist dabei kein Fehler in der Verifikations-Software; sie ist eine genauso fundamentale wie unvermeidliche Eigenschaft, die wir in der gleichen Weise akzeptieren müssen wie die Naturgesetze in der Physik.

■ T terminiert

In diesem Fall ist H funktional identisch mit $T_{\overline{E}}$ und erfüllt die Eigenschaft E nicht.

Voilà. Mit der vorgenommenen Konstruktion ist es uns gelungen, einen direkten Zusammenhang zwischen der Eigenschaft E und der Terminierung von T herzustellen. Würde ein Verfahren existieren, das E entscheidet, so könnten wir das Halteproblem für jede beliebige Maschine T lösen, indem wir H nach dem gezeigten Schema konstruieren. Kurzum: Wir hätten ein Entscheidungsverfahren für das Halteproblem gefunden.

Beachten Sie, dass die obige Überlegung stets unter der Annahme stand, dass die gewählte Eigenschaft E auf T_{\perp} zutrifft. Sollte dies nicht der Fall sein, so modifizieren wir die Maschine H wie in der unteren Hälfte von Abbildung 6.66 gezeigt. Anstelle von $T_{\overline{E}}$ starten wir eine beliebige Maschine T_E , die E erfüllt. Die Fallunterscheidung liest sich jetzt wie folgt:

■ T terminiert nicht

In diesem Fall ist H funktional identisch mit T_{\perp} und erfüllt die Eigenschaft E nicht.

■ T terminiert

In diesem Fall ist H funktional identisch mit T_E und erfüllt die Eigenschaft E .

Wiederum ist es uns gelungen, einen Eins-zu-eins-Zusammenhang zwischen E und der Terminierung von T herzustellen. Gäbe es ein Entscheidungsverfahren für die Eigenschaft E , so könnten wir das Halteproblem ebenfalls lösen. Die bereits bewiesene Unentscheidbarkeit des Halteproblems führt damit unweigerlich zu der Erkenntnis, dass ein Entscheidungsverfahren für E nicht existieren kann. Genau dies ist die Aussage des berühmten Satzes von Henry Gordon Rice aus dem Jahr 1953.

Satz 6.20 (Satz von Rice)

Mit E sei eine nichttriviale funktionale Eigenschaft von Turing-Maschinen gegeben. Dann ist das folgende Problem unentscheidbar:

■ Gegeben: Turing-Maschine T

■ Gefragt: Besitzt T die Eigenschaft E ?

Die Tragweite des Satzes von Rice ist enorm. In einem Rundumschlag macht er die Hoffnung zunichte, irgendeine nichttriviale Eigenschaft über Turing-Maschinen algorithmisch entscheiden zu können. Seine Allgemeinheit macht diesen Satz zu einer der wertvollsten Aussagen der theoretischen Informatik – wenngleich wir uns alle sicher eine positivere Lösung der Berechenbarkeitsfrage gewünscht hätten.

6.5.3 Reduktionsbeweise

Erinnern Sie sich noch an den Unentscheidbarkeitsbeweis des Halteproblems auf leerem Band? Das Ergebnis hatten wir nicht direkt bewiesen. Stattdessen zeigten wir, dass das Halteproblem auf leerem Band stark genug ist, um das allgemeine Halteproblem zu lösen. Mit anderen Worten: Wir hatten das allgemeine Halteproblem auf das Halteproblem auf leerem Band *reduziert*. Aus unserem Wissen über die Unentscheidbarkeit des allgemeinen Halteproblems konnten wir dann schließen, dass auch das Halteproblem auf leerem Band unentscheidbar sein muss.

Die Reduktionstechnik ist in der Berechenbarkeitstheorie von so großer Bedeutung, dass wir sie in diesem Abschnitt in eine allgemein verwendbare Form bringen wollen. Wir beginnen mit der formalen Definition des bisher nur informell verwendeten Reduktionsbegriffs.



Definition 6.19 (Reduzierbarkeit)

Mit $L \subseteq \Sigma^*$ und $L' \subseteq \Gamma^*$ seien zwei Sprachen gegeben. L ist genau dann auf L' reduzierbar, geschrieben als $L \leq L'$, falls eine totale Funktion $f : \Sigma^* \rightarrow \Gamma^*$ mit den folgenden Eigenschaften existiert:

- f ist berechenbar
- $\omega \in L \Leftrightarrow f(\omega) \in L'$

Ist eine Sprache L auf eine andere Sprache L' reduzierbar, so sind wir in der Lage, die Frage $\omega \in L$ durch eine äquivalente Frage über L' zu beantworten. Wie in Abbildung 6.67 gezeigt, berechnen wir hierzu zunächst das Element $f(\omega)$. Mithilfe dieses Elements können wir die Mengenzugehörigkeit durch ein Entscheidungsverfahren für L' beantworten. Vor allem aber erlaubt die Reduktionstechnik, die bewiesene Unentscheidbarkeit einer Sprache bzw. eines Problems L auf eine andere Sprache bzw. ein anderes Problem L' zu übertragen. Unsere Überlegungen offenbaren den folgenden Zusammenhang:

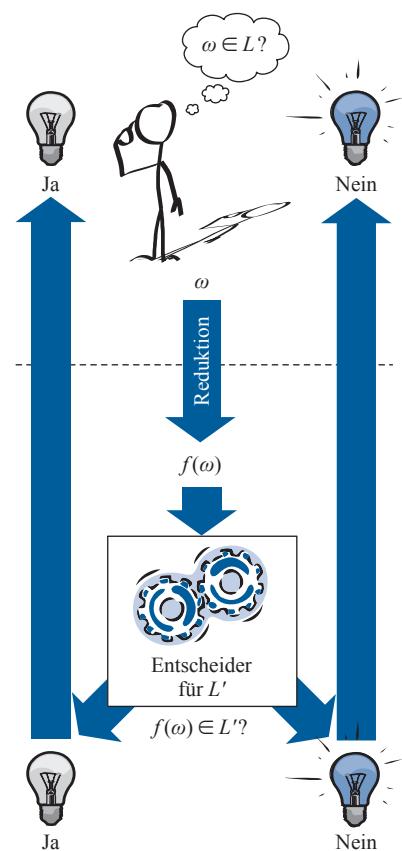


Abbildung 6.67: Die Reduktionstechnik in Aktion. Um die Mengenzugehörigkeit $\omega \in L$ zu entscheiden, wird zunächst das Element $f(\omega)$ berechnet und anschließend die Frage $f(\omega) \in L'$ beantwortet.

■ Alphabet

$$\Sigma := \{0, 1\}$$

■ Wortpaare

$$(01, 1)$$

$$(0, 000)$$

$$(01000, 01)$$

■ Lösung

$$\boxed{01000} \boxed{0} \boxed{0} \boxed{01}$$

$$\boxed{01} \boxed{000} \boxed{000} \boxed{1}$$

Abbildung 6.68: Das Post'sche Korrespondenzproblem besitzt für die dargestellte Instanz eine Lösung.



Satz 6.21

Ist L eine unentscheidbare Sprache, so ist jede Sprache L' mit $L \leq L'$ ebenfalls unentscheidbar.

Haben wir also erst einmal eine Reihe von unentscheidbaren Sprachen oder Problemen gefunden, so können wir diese mithilfe der Reduktions-technik um weitere ergänzen. Achten Sie stets darauf, die Richtung der Reduktion nicht zu verwechseln. Die Unentscheidbarkeit einer Sprache L' wird bewiesen, indem eine unentscheidbare Sprache L auf L' reduziert wird und nicht umgekehrt. In unserem obigen Beweis haben wir exakt diese Reihenfolge eingehalten: Wir haben das allgemeine Halteproblem auf das Halteproblem auf leerem Band reduziert.

■ Alphabet

$$\Sigma := \{0, 1\}$$

■ Wortpaare

$$(011, 100)$$

$$(11, 110)$$

$$(010, 011)$$

■ Lösung müsste wie folgt beginnen

$$\boxed{11} \boxed{?} \boxed{?} \dots$$

$$\boxed{110} \boxed{?} \boxed{?} \dots$$

Abbildung 6.69: Das Post'sche Korrespondenzproblem besitzt für die dargestellte Instanz keine Lösung.

6.5.4 Das Post'sche Korrespondenzproblem

In diesem Abschnitt wenden wir uns dem *Post'schen Korrespondenzproblem* zu. Es wurde im Jahr 1946 von dem polnisch-US-amerikanischen Mathematiker Emil Leon Post formuliert und gehört heute zu den wichtigsten unentscheidbaren Problemen. Seine Bedeutung beruht weniger auf seiner inhaltlichen Aussage, sondern in erster Linie auf der Eigenschaft, dass es sich vergleichsweise einfach auf andere Probleme reduzieren lässt. In dieser Funktion gehört das Post'sche Korrespondenzproblem zu den wichtigsten Instrumenten für das Führen von Unentscheidbarkeitsbeweisen.



Definition 6.20 (Post'sches Korrespondenzproblem)

Das *Post'sche Korrespondenzproblem* (*Post's Correspondence Problem*, kurz PCP) lautet wie folgt:

- Gegeben: Wortpaare $(x_1, y_1), \dots, (x_n, y_n)$ mit $x_i, y_i \in \Sigma^+$
- Gefragt: Gibt es eine Folge i_1, \dots, i_k mit der Eigenschaft

$$x_{i_1} x_{i_2} \dots x_{i_k} = y_{i_1} y_{i_2} \dots y_{i_k}?$$

Abbildung 6.68 veranschaulicht das Post'sche Korrespondenzproblem anhand einer Tupelfolge mit drei Wortpaaren. Unsere Aufgabe ist es herauszufinden, ob wir die linken und rechten Komponenten im Sinne

- Wortpaare
 $(001, 0), (01, 011), (01, 101), (10, 001)$
- Erste Sequenz

01	10	01	10	10	01	001	01	10	01	10	01	10	01	10	01	10	01	001	10	01	10	10
01	001	01	10	001	001	01	10	10	10	01	001	01	001	001	001	001	001	01	10	01	10	10
001	01	001	10	10	01	001	10	001	001	01	10	001	001	001	01	001	001	01	001	01	001	001
01	001	10	001	001	01																	

- Zweite Sequenz

011	001	101	001	001	011	0	011	001	101	001	101	001	001	101	001	001	101	001	001	101	001	
011	0	001	001	011	0	101	001	0	0	101	001	001	001	001	001	011	0	011	0	0	0	0
101	001	101	001	0	011	0	001	001	011	0	001	0	001	0	0	101	001	0	0	101	0	0
101	0	011	0	001	0	0	101															

Abbildung 6.70: Auch diese Instanz des Post'schen Korrespondenzproblems besitzt eine Lösung.

von Definition 6.20 zu zwei gleichen Wortketten zusammenfügen können. Wie in der Abbildung gezeigt, besitzt die abgebildete Instanz des Korrespondenzproblems in der Tat eine Lösung.

In Abbildung 6.69 ist eine zweite Probleminstanz dargestellt, die keine Lösung besitzt. Verglichen mit dem ersten Beispiel müssen wir allerdings ein wenig tiefgründiger argumentieren, schließlich gilt es zu zeigen, dass sich alle bildbaren Wortketten voneinander unterscheiden. Auf den ersten Blick ist dies kein leichtes Unterfangen, da die Längen der erzeugbaren Symbolketten nicht nach oben beschränkt sind. Auf den zweiten Blick wird trotzdem deutlich, warum die abgebildete Probleminstanz unlösbar ist. Gäbe es eine Lösung, so müsste die erste Kette mit 11 und die zweite Kette mit 110 beginnen; alle anderen Kombinationen führen unmittelbar zu verschiedenen Symbolsequenzen. Da jetzt die erste Kette die Kürzere ist, muss die zweite Kette das fehlende Zeichen irgendwann wieder aufholen. Dies ist jedoch unmöglich, da die rechte Seite in allen Wortpaaren genauso viele oder mehr Zeichen enthält als die linke.

Dass sich das Post'sche Korrespondenzproblem selbst für harmlos anmutende Wortpaare oft nur schwer lösen lässt, verdeutlicht das Beispiel

■ MPCP-Instanz

Alphabet:

$$\Sigma := \{0, 1\}$$

Tupelfolge:

$$\begin{array}{ll} (011 & , \quad 100) \\ (11 & , \quad 110) \\ (010 & , \quad 011) \end{array}$$



■ PCP-Instanz

Alphabet:

$$\Sigma := \{0, 1\} \cup \{\square, \Box\}$$

Tupelfolge:

$$\begin{array}{ll} (\square 0 \square 1 \square 1 \square & , \quad \square 1 \square 0 \square 0) \\ (0 \square 1 \square 1 \square & , \quad \square 1 \square 0 \square 0) \\ (1 \square 1 \square & , \quad \square 1 \square 1 \square 0) \\ (0 \square 1 \square 0 \square & , \quad \square 0 \square 1 \square 1) \\ (\Box & , \quad \Box \Box) \end{array}$$

Abbildung 6.71: Transformation einer MPCP-Instanz in eine äquivalente PCP-Instanz. Die neu hinzugefügten Symbole \square und \Box sorgen dafür, dass die konstruierten Ketten immer mit dem ersten Wortpaar beginnen müssen.

in Abbildung 6.70. Es stammt aus [89] und ist so konstruiert, dass erst nach 66 Zusammenfügungen zwei identische Zeichenketten entstehen.

Bevor wir die Unentscheidbarkeit des Post'schen Korrespondenzproblems beweisen, führen wir noch eine leicht modifizierte Variante ein.



Definition 6.21 (Modifiziertes Korrespondenzproblem)

Das *modifizierte Korrespondenzproblem* (*Modified Post's Correspondence Problem*, kurz MPCP) lautet wie folgt:

- Gegeben: Wortpaare $(x_1, y_1), \dots, (x_n, y_n)$ mit $x_i, y_i \in \Sigma^+$
- Gefragt: Gibt es eine Folge i_2, \dots, i_k mit der Eigenschaft

$$x_1 x_{i_2} \dots x_{i_k} = y_1 y_{i_2} \dots y_{i_k} ?$$

Das modifizierte Korrespondenzproblem unterscheidet sich in einem nahezu unscheinbaren Punkt. Während die zu bildenden Ketten in der allgemeinen Variante mit einem beliebigen Wortpaar (x_{i_1}, y_{i_1}) gestartet werden können, müssen sie in der modifizierten Variante immer mit dem Wortpaar (x_1, y_1) beginnen.

Wie Sie vielleicht schon vermuten, gilt $\text{MPCP} \leq \text{PCP}$, d. h., wir sind in der Lage, das modifizierte Korrespondenzproblem mithilfe der allgemeinen Variante zu lösen. Bevor wir eine entsprechende Reduktion konstruieren, wollen wir die Ausgangssituation nochmals rekapitulieren. Wir unterscheiden zwei Fälle:

- PCP besitzt für $(x_1, y_1), \dots, (x_n, y_n)$ keine Lösung. In diesem Fall können die x - und y -Komponenten der gegebenen Worttupel nicht zu einer gleichlautenden Zeichensequenz kombiniert werden. Damit existiert erst recht keine Kombination, die x_1 und y_1 als Erstes verwendet. Folgerichtig besitzt auch MPCP für die gegebene Tupelfolge keine Lösung.
- PCP besitzt für $(x_1, y_1), \dots, (x_n, y_n)$ eine Lösung. In diesem Fall existiert mindestens eine Indexfolge i_1, \dots, i_k mit $x_{i_1}, \dots, x_{i_k} = y_{i_1}, \dots, y_{i_k}$. Gilt für alle Indexfolgen die Beziehung $i_1 \neq 1$, so besitzt zwar das PCP-Problem eine Lösung, nicht jedoch das MPCP-Problem.

Die Reduktion von MPCP auf PCP muss so erfolgen, dass jede PCP-Lösung zwangsläufig mit dem Paar (x_1, y_1) beginnen muss. In der Tat

sind wir mit einem kleinen Trick in der Lage, dies zu erreichen. Hierzu ergänzen wir das Alphabet Σ um zwei neue Symbole und modifizieren die Wortpaare (x_i, y_i) so, dass sich x_i und y_i für $i \neq 1$ im ersten Symbol unterscheiden (vgl. Abbildung 6.71). Beachten Sie, dass die Lösbarkeit des Korrespondenzproblems durch die Modifikation nicht beeinflusst wird: Das MPCP-Problem ist für eine Wortfolge W genau dann lösbar, wenn es für die modifizierte Folge W' lösbar ist. Da jetzt jede PCP-Lösung von W' zwangsläufig mit dem Paar (x_1, y_1) beginnen muss, erhalten wir den folgenden Zusammenhang:

- Das MPCP-Problem besitzt für W eine Lösung
- \Leftrightarrow Das MPCP-Problem besitzt für W' eine Lösung
- \Leftrightarrow Das PCP-Problem besitzt für W' eine Lösung

Damit sind wir unserem Ziel, die Unentscheidbarkeit von PCP zu beweisen, einen großen Schritt näher gekommen. Nach Satz 6.21 reicht es aus, die Unentscheidbarkeit des einfacheren MPCP-Problems zu zeigen. Die konstruierte Reduktion sorgt dafür, dass sich das Ergebnis in direkter Weise auf das allgemeine Korrespondenzproblem überträgt.

Die Unentscheidbarkeit von MPCP lässt sich durch die Reduktion des Halteproblems auf leerem Band beweisen. Hierzu sei mit $T = (S, \Sigma, \Pi, \delta, s_0, \square, E)$ eine beliebige Turing-Maschine gegeben. Wir werden zeigen, dass wir das Halteproblem lösen könnten, wenn es einen Entscheider für MPCP gäbe. Die Grundidee ist bestechend einfach: Wir werden für die Maschine T eine MPCP-Instanz konstruieren, die das Verhalten von T nachahmt. Die Folge $(x_1, y_1), \dots, (x_k, y_k)$ setzt sich aus den folgenden Wortpaaren zusammen:

- Startpaar
 $((\varepsilon), (\varepsilon)\langle \kappa_0 \rangle)$
- Übergangspaare
 $((\kappa_i), (\kappa_j))$ für alle Konfigurationsübergänge $\kappa_i \rightarrow \kappa_j$
- Abschlusspaare
 $((\kappa_e)\langle \varepsilon \rangle, \langle \varepsilon \rangle)$ für alle Endkonfigurationen κ_e

Um die Idee dieser Konstruktion zu verstehen, nehmen wir an, die Turing-Maschine T durchläuft nacheinander die Konfigurationen $\kappa_0, \kappa_1, \kappa_2, \kappa_3$. Abbildung 6.72 zeigt, wie die Konfigurationsübergänge innerhalb der MPCP-Instanz nachgebildet werden und sich die gebildeten Wortketten in jedem Schritt verlängern. Von entscheidender Bedeu-

- Initialkonfiguration: κ_0

$$x : \langle \varepsilon \rangle$$

$$y : \langle \varepsilon \rangle \langle \kappa_0 \rangle$$

- Übergang von κ_0 auf κ_1

$$x : \langle \varepsilon \rangle \langle \kappa_0 \rangle$$

$$y : \langle \varepsilon \rangle \langle \kappa_0 \rangle \langle \kappa_1 \rangle$$

- Übergang von κ_1 auf κ_2

$$x : \langle \varepsilon \rangle \langle \kappa_0 \rangle \langle \kappa_1 \rangle$$

$$y : \langle \varepsilon \rangle \langle \kappa_0 \rangle \langle \kappa_1 \rangle \langle \kappa_2 \rangle$$

- Übergang von κ_2 auf κ_3

$$x : \langle \varepsilon \rangle \langle \kappa_0 \rangle \langle \kappa_1 \rangle \langle \kappa_2 \rangle$$

$$y : \langle \varepsilon \rangle \langle \kappa_0 \rangle \langle \kappa_1 \rangle \langle \kappa_2 \rangle \langle \kappa_3 \rangle$$

- Abschluss, falls κ_3 Endzustand ist

$$x : \langle \varepsilon \rangle \langle \kappa_0 \rangle \langle \kappa_1 \rangle \langle \kappa_2 \rangle \langle \kappa_3 \rangle \langle \varepsilon \rangle$$

$$y : \langle \varepsilon \rangle \langle \kappa_0 \rangle \langle \kappa_1 \rangle \langle \kappa_2 \rangle \langle \kappa_3 \rangle \langle \varepsilon \rangle$$

Abbildung 6.72: Das Verhalten einer Turing-Maschine lässt sich mithilfe von Wortsequenzen nachbilden. Die Abschlussregeln werden dabei so gewählt, dass die untere Sequenz die obere genau dann einholen kann, wenn die simulierte Turing-Maschine terminiert.

■ PCP-Instanz

$$(x_1, y_1), \dots, (x_n, y_n), \quad x_i, y_i \in \Sigma^+$$



■ Grammatik G_1

$$G_1 := (S_{G_1}, \Sigma_{G_1}, P_{G_1}, s_{G_1})$$

$$S_{G_1} := \{S_1\}$$

$$s_{G_1} := S_1$$

$$\Sigma_{G_1} := \Sigma \cup \{i_1, \dots, i_n\}$$

■ Produktionenmenge P_{G_1}

$$S_1 \rightarrow i_1 x_1 | \dots | i_n x_n$$

$$S_1 \rightarrow i_1 S_1 x_1 | \dots | i_n S_1 x_n$$

■ Grammatik G_2

$$G_2 := (S_{G_2}, \Sigma_{G_2}, P_{G_2}, s_{G_2})$$

$$S_{G_2} := \{S_2\}$$

$$s_{G_2} := S_2$$

$$\Sigma_{G_2} := \Sigma \cup \{i_1, \dots, i_n\}$$

■ Produktionenmenge P_{G_2}

$$S_2 \rightarrow i_1 y_1 | \dots | i_n y_n$$

$$S_2 \rightarrow i_1 S_2 y_1 | \dots | i_n S_2 y_n$$

Abbildung 6.73: Reduktion von PCP auf das Schnittpproblem kontextfreier Sprachen

tung ist die Eigenschaft, dass die obere Kette der unteren um eine Konfiguration hinterherhinkt. Terminiert T nicht, so wird die erste Kette ständig eine Konfiguration voraus sein, und die MPCP-Instanz besitzt keine Lösung. Terminiert T , so kommen die Abschlussregeln ins Spiel. Diese sind so konstruiert, dass die obere Kette den fehlenden Konfigurationsübergang aufholt. Nach der Anwendung einer Abschlussregel sind die gebildeten Ketten dann identisch und die MPCP-Instanz gelöst. Insgesamt haben wir damit einen direkten Zusammenhang zwischen der Lösbarkeit von MPCP und der Terminierung von Turing-Maschinen hergestellt. Wäre MPCP entscheidbar, so könnten wir auch das Halteproblem entscheiden – im Widerspruch zu unseren bisherigen Erkenntnissen.

Die Beweisskizze bringt die Kernidee der Reduktion gut zum Vorschein, wenngleich in sehr informeller Weise. Um den Beweis in mathematischer Präzision zu führen, müssen wir zusätzlich zeigen, wie sich die Konfigurationen κ_i textuell beschreiben lassen. Oben sind wir einfach stillschweigend davon ausgegangen, dass eine entsprechende Codierung existiert. Die Abstraktion wurde an dieser Stelle bewusst vorgenommen, um die Kernidee des Beweises klar herauszuschälen. Eine mathematisch präzise Ausführung findet sich z. B. in [52].

6.5.5 Weitere unentscheidbare Probleme

In diesem Abschnitt werden wir unser Wissen über das Post'sche Korrespondenzproblem nutzen, um weitere Probleme als unentscheidbar zu identifizieren. Konkret handelt es sich um Probleme aus dem Bereich der formalen Sprachen, die wir in Kapitel 4 ausführlich diskutiert haben. Im Einzelnen werden wir die Unentscheidbarkeit der folgenden Probleme beweisen:

■ Schnittproblem für kontextfreie Grammatiken

Gegeben: Kontextfreie Grammatiken G_1 und G_2

Gefragt: $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \emptyset$?

■ Mehrdeutigkeitsproblem für kontextfreie Grammatiken

Gegeben: Kontextfreie Grammatik G

Gefragt: Besitzt jedes Wort $\omega \in \mathcal{L}(G)$ eine eindeutige Ableitung?

■ Leerheitsproblem für kontextsensitive Grammatiken

Gegeben: Kontextsensitive Grammatik G

Gefragt: $\mathcal{L}(G) \neq \emptyset$?

Als Erstes werden wir die Unentscheidbarkeit des Schnittproblems kontextfreier Sprachen durch die Reduktion des Post'schen Korrespondenzproblems beweisen. Hierzu bilden wir die Tupelfolge $(x_1, y_1), \dots, (x_n, y_n)$ einer gegebenen Instanz des Korrespondenzproblems so auf zwei Grammatiken G_1 und G_2 ab, dass die folgende Beziehung gewährleistet ist:

$$\text{PCP-Instanz hat eine Lösung} \Leftrightarrow \mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \emptyset$$

Abbildung 6.73 legt die Konstruktion von G_1 und G_2 offen. Die Grammatik G_1 ist so konstruiert, dass diejenigen Sequenzen abgeleitet werden können, deren Suffixe aus den bildbaren Wortsequenzen der PCP-Instanz bestehen. Die Definition von G_2 erfolgt analog für die Komponenten y_1, \dots, y_n . Die Präfixe der abgeleiteten Wörter werden aus den Symbolen i_1, \dots, i_n gebildet. Sie sind eine textuelle Repräsentation der Auswahlindizes des PCP und dürfen auf keinen Fall fehlen. Deren Existenz stellt sicher, dass wir ein Wort ω nur dann sowohl mit G_1 als auch mit G_2 ableiten können, wenn beide Ableitungen die gleiche Regellienfolge einhalten. Diese Eigenschaft ist der Schlüssel zum Erfolg. Ein Wort liegt jetzt genau dann in der Schnittmenge $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$, wenn die betrachtete PCP-Instanz eine Lösung besitzt. Könnten wir das Schnittproblem für kontextfreie Sprachen entscheiden, so könnten wir durch die konstruierte Reduktion auch das Post'sche Korrespondenzproblem entscheiden. Aus dem Widerspruch ergibt sich unmittelbar das folgende Ergebnis:



Satz 6.22

Das Schnittproblem ist für kontextfreie Grammatiken unentscheidbar.

Als Nächstes wenden wir uns dem Mehrdeutigkeitsproblem kontextfreier Sprachen zu. Hinter diesem verbirgt sich die Frage, ob für eine gegebene Typ-2-Grammatik G ein Wort ω existiert, das auf unterschiedliche Weise aus dem Startsymbol abgeleitet werden kann. Auch hier können wir die Unentscheidbarkeit durch eine Reduktion des Post'schen Korrespondenzproblems erhalten. Wie in Abbildung 6.74 gezeigt, folgt die Grammatik G weitgehend dem Konstruktionsprinzip, das uns weiter oben die Unentscheidbarkeit des Schnittproblems zeigten ließ. Die Produktionenmenge ist so aufgebaut, dass das Startsymbol S zunächst in eines der beiden Nonterminale S_1 oder S_2 überführt wird. Die weiteren Ableitungen sind mit jenen von G_1 und G_2 aus Abbildung 6.73 identisch. Existieren in G zwei verschiedene Ableitungssequenzen, die das

■ PCP-Instanz

$$(x_1, y_1), \dots, (x_n, y_n)$$



■ Grammatik G

$$G_1 := (S_G, \Sigma_G, P_G, s_G)$$

$$S_G := \{S, S_1, S_2\}$$

$$s_G := S$$

$$\Sigma_G := \Sigma \cup \{i_1, \dots, i_n\}$$

■ Produktionenmenge P_G

$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow i_1 x_1 \mid \dots \mid i_n x_n$$

$$S_1 \rightarrow i_1 S_1 x_1 \mid \dots \mid i_n S_1 x_n$$

$$S_2 \rightarrow i_1 y_1 \mid \dots \mid i_n y_n$$

$$S_2 \rightarrow i_1 S_2 y_1 \mid \dots \mid i_n S_2 y_n$$

Abbildung 6.74: Reduktion von PCP auf das Mehrdeutigkeitsproblem

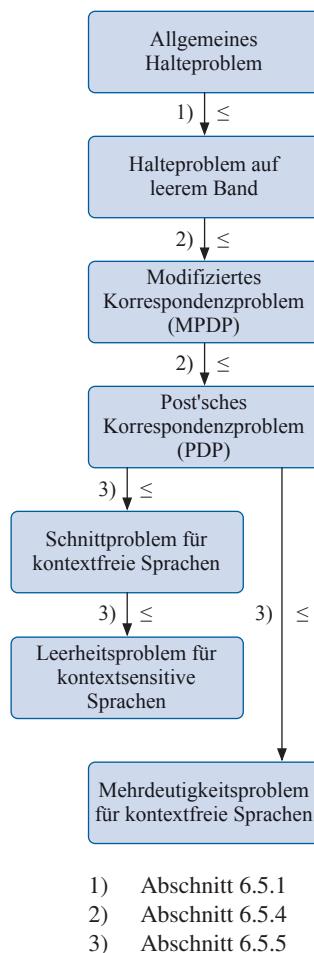


Abbildung 6.75: Durch die Reduktion des Post'schen Korrespondenzproblems lässt sich eine Reihe weiterer Probleme als unentscheidbar entlarven.

gleiche Wort ω entstehen lassen, so muss eine Sequenz das Nonterminal S_1 und die andere das Nonterminal S_2 enthalten. In diesem Fall ist dann auch die PCP-Instanz lösbar. Die Umkehrung gilt ebenfalls. Ist die PCP-Instanz unlösbar, so kann es in G keine zwei Ableitungssequenzen geben, die das gleiche Wort ω erzeugen. Damit haben wir das Post'sche Korrespondenzproblem erfolgreich auf das Mehrdeutigkeitsproblem reduziert und dessen Unentscheidbarkeit bewiesen.

Satz 6.23

Das Mehrdeutigkeitsproblem ist für kontextfreie Grammatiken unentscheidbar.

Als Drittes wenden wir uns dem Leerheitsproblem für kontextsensitiven Sprachen zu. Anders als in den ersten beiden Beispielen werden wir das Post'sche Korrespondenzproblem nicht direkt reduzieren. Stattdessen machen wir von unserem Wissen über die Unentscheidbarkeit des Schnittproblems kontextfreier Sprachen Gebrauch. Wir zeigen, dass das Schnittproblem kontextfreier Sprachen entscheidbar wäre, wenn wir das Leerheitsproblem kontextsensitiver Sprachen entscheiden könnten.

L_1 und L_2 bezeichnen zwei beliebige kontextfreie Sprachen, für die wir das Schnittproblem entscheiden wollen. Für die folgende Betrachtung nutzen wir zwei elementare Eigenschaften kontextsensitiver Sprachen aus. Zum einen ist die Menge der Typ-1-Sprachen eine Obermenge der Typ-2-Sprachen. Damit sind sowohl L_1 als auch L_2 gleichzeitig Typ-1-Sprachen. Zum anderen ist die Menge der Typ-1-Sprachen bez. der Schnittoperation abgeschlossen, d. h., die Schnittmenge $L_1 \cap L_2$ ist eine kontextsensitive Sprache. Könnten wir das Leerheitsproblem für kontextsensitive Sprachen entscheiden, so hätten wir einen Weg gefunden, das Schnittproblem kontextfreier Sprachen ebenfalls zu lösen. Damit ist es uns abermals gelungen, mit der Reduktionstechnik ein wichtiges Unberechenbarkeitsresultat zu erzielen.

Satz 6.24

Das Leerheitsproblem ist für kontextsensitive Sprachen unentscheidbar.

Insgesamt entsteht der in Abbildung 6.75 dargestellte Zusammenhang. Allein die Unentscheidbarkeit des allgemeinen Halteproblems reichte aus, um alle anderen Probleme mithilfe der Reduktionstechnik ebenfalls als unentscheidbar zu identifizieren.

6.6 Übungsaufgaben

In diesem Kapitel haben Sie gelernt, wie sich die While-Sprache durch die Definition von Makros erweitern lässt, ohne ihre Ausdrucksstärke zu verändern. Unter anderem wurde gezeigt, wie komplexe Schleifenbedingungen der Form $x < y$, $x > y$ und $x = y$ auf die Standardkonstrukte reduziert werden können. Zeigen Sie, dass eine ähnliche Reduktion auch für die booleschen Konnektive \wedge und \vee möglich ist. Tragen Sie Ihre Lösung in das jeweils rechtsstehende Listing ein:

Aufgabe 6.1

Webcode
6237

and.while <pre>while x&y do P end</pre>	and_reduced.while <pre>1 2 3 4 5</pre>
or.while <pre>while x∨y do P end</pre>	or_reduced.while <pre>1 2 3 4 5</pre>

Betrachten Sie die folgende Implementierung der Fakultätsfunktion:

Aufgabe 6.2

Webcode
6104

factorial.c <pre>int factorial(m,n) { if (m == 0) { return 1; } return mult(factorial(m-1, n), m); }</pre>	<pre>1 2 3 4 5 6 7 8</pre>
--	----------------------------

Extrahieren Sie aus dem Programm die primitiv-rekursive Form der berechneten Funktion. Welche Bedeutung hat der Parameter n ?

Aufgabe 6.3**Webcode
6006**

Gegeben sei die Funktion $\pi : \mathbb{N}^2 \rightarrow \mathbb{N}$ mit

$$\pi(x, y) = \binom{x+y+1}{2} + x.$$

Analysieren Sie die Funktionswerte, indem Sie die folgende Tabelle vervollständigen:

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$	$x = 5$
$y = 0$						
$y = 1$						
$y = 2$						
$y = 3$						
$y = 4$						
$y = 5$						

Offensichtlich ist π eine spezielle Variante der Cantor'schen Paarungsfunktion, die \mathbb{N}^2 bijektiv auf \mathbb{N} abbildet.

- Zeigen Sie, dass π eine primitiv-rekursive Funktion ist.
- Zeigen Sie, dass auch die beiden Umkehrfunktionen $\pi_1^{-1} : \mathbb{N} \rightarrow \mathbb{N}$ und $\pi_2^{-1} : \mathbb{N} \rightarrow \mathbb{N}$ mit $x = \pi_1^{-1}(\pi(x, y))$ und $y = \pi_2^{-1}(\pi(x, y))$ primitiv-rekursiv sind.
- Zeigen Sie, dass sich die Mengen \mathbb{N}^k und \mathbb{N} mithilfe primitiv-rekursiver Funktionen bijektiv aufeinander abbilden lassen.

Aufgabe 6.4**Webcode
6150**

In diesem Kapitel haben Sie mit der verallgemeinerten Registermaschine ein bekanntes Berechnungsmodell kennen gelernt. Wir wollen unser Augenmerk auf den Registersatz lenken, der in der gewählten Darstellung aus unendlich vielen Registern besteht, die jedes für sich eine beliebige natürliche Zahl speichern können.

Ändert sich die Berechnungsstärke der Registermaschine, wenn wir...

- die Anzahl der Register auf endlich viele reduzieren?
- in einem Register nur noch Werte aus einem endlichen Intervall speichern dürfen?
- die Restriktionen aus a) und b) miteinander kombinieren?

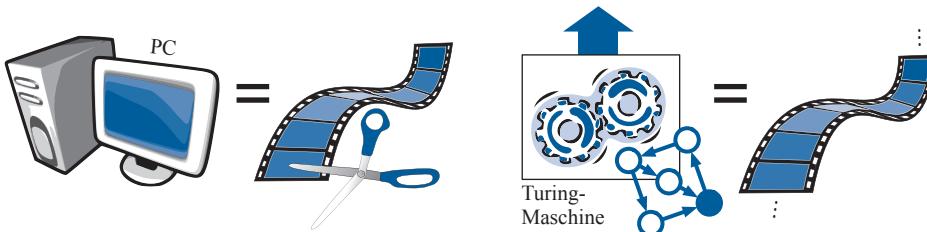
Diese Aufgabe soll Ihnen ein Gefühl für die strukturelle Komplexität von Turing-Maschinen vermitteln. Um die Betrachtung nicht komplizierter als nötig zu gestalten, wollen wir ausschließlich Maschinen mit einem zweielementigen Bandalphabet Π und einem einzigen Endzustand betrachten (vgl. [74]). Wir nehmen weiter an, dass der Schreib-Lese-Kopf drei mögliche Bewegungen vollziehen kann (Linksschritt, Rechtsschritt, Position halten). $T(n)$ bezeichne die Anzahl der Möglichkeiten, eine solche Turing-Maschine mit n Zuständen zu konstruieren.

Aufgabe 6.5**Webcode****6968**

Stellen Sie eine Formel für $T(n)$ auf und vervollständigen Sie die nachstehende Tabelle:

n	$T(n)$	n	$T(n)$
1		6	
2		7	
3		8	
4		9	
5		10	

Viele Experten sehen in der Turing-Maschine dasjenige Berechnungsmodell, das dem modernen Computer, wie wir ihn heute kennen, am nächsten kommt. Von einem formalen Standpunkt aus ist diese Sichtweise angreifbar, da alle gegenwärtig verfügbaren und zukünftig gebauten Computer nur über einen endlich großen Speicher verfügen. Turing-Maschinen besitzen hingegen ein unendliches langes Band.

Aufgabe 6.6**Webcode****6784**

Die Endlichkeit des Speichers hat zur Konsequenz, dass reale Computer nur endlich viele Zustände einnehmen können. In Wirklichkeit sind sie damit nichts anderes als endliche Automaten und folglich berechnungsschwächer als Turing-Maschinen. Warum werden Turing-Maschinen von den meisten Experten dennoch als das adäquatere Modell zur Beschreibung realer Computer angesehen?

Aufgabe 6.7

Webcode
6355

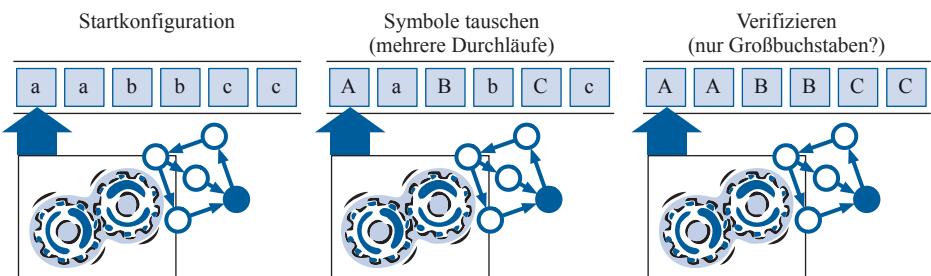
In dieser Aufgabe betrachten wir eine Turing-Maschine T mit

$$\mathcal{L}(T) := \{a^n b^n c^n \mid n \in \mathbb{N}^+\}$$

Die Implementierung von T ist durch die folgende Übergangstabelle festgelegt:

	a	A	b	B	c	C	\square
s_a	(s_b, A, \rightarrow)	(s_a, A, \rightarrow)	—	—	—	—	—
s_b	(s_b, a, \rightarrow)	—	(s_c, B, \rightarrow)	(s_b, B, \rightarrow)	—	—	—
s_c	—	—	(s_c, b, \rightarrow)	—	(s_t, C, \rightarrow)	(s_c, C, \rightarrow)	—
s_t	—	—	—	—	(s_r, c, \leftarrow)	—	$(s_v, \square, \leftarrow)$
s_r	(s_r, a, \leftarrow)	(s_a, A, \rightarrow)	(s_r, b, \leftarrow)	(s_r, B, \leftarrow)	(s_r, c, \leftarrow)	(s_r, C, \leftarrow)	$(s_a, \square, \rightarrow)$
s_v	—	(s_v, A, \leftarrow)	—	(s_v, B, \leftarrow)	—	(s_v, C, \leftarrow)	$(s_e, \square, \rightarrow)$
s_e	—	—	—	—	—	—	—

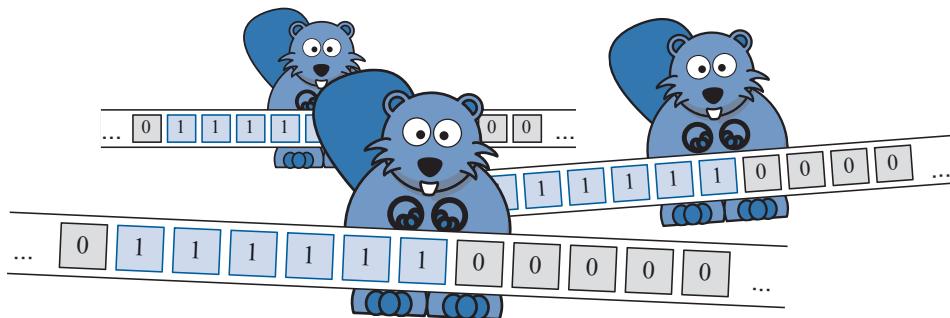
Die Maschine arbeitet in mehreren Schritten. Zunächst bewegt sie den Schreib-Lese-Kopf von links nach rechts über das Eingabewort ω und tauscht je ein a , ein b und ein c durch die Hilfssymbole A , B und C aus. Dieser Vorgang wird durch die Zustände s_a , s_b und s_c gesteuert. Anschließend wird in Zustand s_t geprüft, ob noch weitere c 's auf dem Band vorhanden sind. Falls ja, fährt der Schreib-Lese-Kopf an den Anfang zurück (Zustand s_r) und wiederholt den geschilderten Ersetzungsprozess. Wurde das letzte c ersetzt, so geht die Maschine in die Verifikationsphase über (Zustand s_v). In dieser bewegt die Maschine den Schreib-Lese-Kopf auf die erste Bandposition zurück und überprüft, ob sämtliche der ursprünglich vorhandenen Symbole ersetzt wurden.



T terminiert im Endzustand s_e , wenn der Überprüfungsprozess erfolgreich war. Dies ist genau dann der Fall, wenn das Eingabewort ω die gleiche Anzahl a 's, b 's und c 's enthält.

- Simulieren Sie die Konfigurationsübergänge für das Eingabewort $\omega = aabbcc$.
- Ist T eine deterministische Turing-Maschine?

Im Jahr 1962 rief der ungarische Mathematiker Tibor Radó den Wettbewerb des *fleißigen Bibers* ins Leben. Fleißige Biber (*busy beaver*) der Größe n sind Turing-Maschinen mit n Zuständen, die möglichst viele Einsen auf ein mit Nullen vorinitialisiertes Band schreiben [11, 84]. Endlosschleifen sind dabei explizit verboten, d. h., die Turing-Maschine muss nach endlich vielen Schritten terminieren. Der Endzustand zählt nicht als Zustand.

Aufgabe 6.8**Webcode****6834**

Unser Interesse gilt der Frage, wie viele Einsen ein fleißiger Biber der Größe n höchstens erzeugen kann. Wir bezeichnen diesen Maximalwert mit $B(n)$. Da für jedes n nur endlich viele Turing-Maschinen existieren, ist der Wert der *Biberfunktion* B für alle n wohldefiniert. Trotzdem sind die Funktionswerte nur bis $n = 4$ exakt bekannt. Es gilt:

$$B(1) = 1$$

$$B(2) = 4$$

$$B(3) = 6$$

$$B(4) = 13$$

Die Bestimmung der Funktionswerte ist keinesfalls trivial. Für $n = 4$ gibt es bereits mehr als eine halbe Billion Turing-Maschinen, die als fleißige Biber in Frage kommen. Damit verwundert es kaum, dass für $B(5)$ nur noch Abschätzungen existieren. Im Jahr 1983 wurde ein fleißiger Biber gefunden, der 240 Einsen erzeugt. Ein Jahr später konnte der Wert durch einen anderen Biber zunächst auf 501 und dann auf 1915 erhöht werden. Im Jahr 1989 wurde schließlich ein Biber gefunden, der 4098 Einsen auf das Band schreibt. Ob diese Zahl dem Funktionswert $B(5)$ entspricht, ist gegenwärtig ungewiss. Jeder ins Rennen geschickte Biber erlaubt uns lediglich, den Funktionswert $B(n)$ nach unten abzuschätzen.

Es wächst der Wunsch nach einem Algorithmus, der die *Biberfunktion* $B(n)$ für beliebige n ausrechnen kann. Machen Sie die Hoffnung über die Existenz eines solchen Algorithmus zu nichts, indem Sie die *Biberfunktion* B als unberechenbar entlarven. Gehen Sie dabei ähnlich vor wie im Beweis der Unentscheidbarkeit des Halteproblems, indem Sie zunächst annehmen, dass $B(n)$ mithilfe eines Programms P berechnet werden kann. Konstruieren Sie anschließend ein Programm P' , das P verwendet und die Annahme zu einem Widerspruch führt.

Aufgabe 6.9
Webcode
6926

In dieser Aufgabe wollen wir uns mit den Eigenschaften der sogenannten *Diagonalsprache* L_D beschäftigen. Ähnlich wie in der Diskussion des Halteproblems listen wir zunächst alle Turing-Maschinen entlang der vertikalen Achse einer Matrix auf und versehen die horizontale Achse mit einer Aufzählung der Wörter aus Σ^* . Nun wird in der i -ten Zeile und der j -ten Spalte eingetragen, ob das Eingabewort ω_j von der Maschine T_i akzeptiert wird oder nicht. Auf diese Weise erhalten wir mit der i -ten Zeile eine tabellarische Beschreibung der Sprache $\mathcal{L}(T_i)$.

	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6
T_1	ja	ja	nein	ja	nein	ja
T_2	nein	ja	nein	ja	ja	nein
T_3	ja	ja	nein	nein	nein	ja
T_4	ja	nein	nein	ja	ja	ja
T_5	ja	ja	nein	ja	nein	nein
T_6	nein	nein	ja	ja	ja	nein

Aus der abgebildeten Matrix können wir die *Diagonalsprache* L_D wie folgt ableiten:

$$L_D := \{\omega_i \mid \omega_i \text{ wird von } T_i \text{ nicht akzeptiert}\}$$

Bildlich gesprochen erhalten wir L_D , indem wir die Hauptdiagonale der konstruierten Matrix herabsteigen und den Zelleninhalt für jedes Element invertieren.

Welche der folgenden Aussagen sind wahr? Begründen Sie Ihre Antwort.

- a) L_D ist entscheidbar
- b) L_D ist aufzählbar
- c) $\overline{L_D}$ ist aufzählbar

Aufgabe 6.10
Webcode
6475

In dieser Aufgabe beschäftigen wir uns ausschließlich mit denjenigen Turing-Maschinen, die für alle Eingabewörter ω terminieren. Genau wie in der vorherigen Aufgabe listen wir die Maschinen entlang der vertikalen Achse einer Matrix auf und versehen die horizontale Achse mit einer Aufzählung der Wörter aus Σ^* . Auch hier tragen wir in der i -ten Zeile und der j -ten Spalte ein, ob das Eingabewort ω_j der Sprache $\mathcal{L}(T_i)$ angehört oder nicht.

Die Diagonalsprache L_D definieren wir wie gehabt:

$$L_D := \{\omega_i \mid \omega_i \text{ wird von } T_i \text{ nicht akzeptiert}\}$$

Aus den Maschinen T_i konstruieren wir jetzt eine Cantor-Maschine C , die nach dem folgenden Prinzip arbeitet: Steht das Wort ω_i auf dem Eingabeband, so startet C die Maschine T_i und antwortet mit „ja“, falls T_i mit „nein“ antwortet und umgekehrt. Offensichtlich akzeptiert C die Diagonalsprache L_D . Da die aufgerufenen Turing-Maschinen T_i für alle Eingaben terminieren, hält auch C für alle Eingaben nach endlich vielen Schritten an. Damit müsste die Cantor-Maschine ebenfalls in der oben konstruierten Matrix auftauchen, im Widerspruch zu der durchgeföhrten Diagonalkonstruktion. Lösen Sie den Widerspruch in diesem Gedanken-gang auf.

In dieser Aufgabe beschäftigen wir uns mit einem prominenten Spezialfall des allgemeinen Halteproblems.

Aufgabe 6.11

Webcode
6132

Definition 6.22 (Spezielles Halteproblem)

Das *spezielle Halteproblem* lautet wie folgt:

- Gegeben: Turing-Maschine T
- Gefragt: Terminiert T mit der Eingabe $\lceil T \rceil$?

$\lceil T \rceil$ bezeichnet die Gödelnummer von T .

- a) Zeigen Sie, dass auch das spezielle Halteproblem unentscheidbar ist.
- b) Lässt sich die Unentscheidbarkeit aus dem Satz von Rice ableiten?

Welche der folgenden Aussagen sind äquivalent?

Aufgabe 6.12

Webcode
6833

- a) Die Sprache L ist aufzählbar
- b) Die Sprache L ist semi-entscheidbar
- c) L ist eine Typ-0-Sprache
- d) Es existiert eine Turing-Maschine T mit $\mathcal{L}(T) = L$
- e) Die charakteristische Funktion χ_L ist berechenbar
- f) Die partielle charakteristische Funktion χ'_L ist berechenbar

Aufgabe 6.13**Webcode
6839**

Gegeben sei die nachstehende Folge von Wortpaaren aus der Menge $\{\diamond, \square\}^+ \times \{\diamond, \square\}^+$:

$$(\diamond\square, \square\square), (\diamond, \diamond\square\diamond), (\square\diamond\diamond, \diamond\diamond)$$

- Lösen Sie das Post'sche Korrespondenzproblem für die angegebene Instanz.
- Warum konnten Sie eine Lösung finden, obwohl das Problem unentscheidbar ist?

Aufgabe 6.14**Webcode
6673**

In diesem Kapitel haben Sie mit dem modifizierten Post'schen Korrespondenzproblem, kurz MPCP, eine spezielle Variante des PCP kennen gelernt.

- Besitzt MPCP für die Tupelfolge $(01, 011), (110, 010), (001, 10)$ eine Lösung?
- Zeigen Sie $\text{PCP} \leq \text{MPCP}$, indem Sie eine Reduktion von PCP auf MPCP konstruieren.

Aufgabe 6.15**Webcode
6754**

Das Post'sche Korrespondenzproblem haben wir für Paare (x_i, y_i) von Wörtern über einem beliebigen Alphabet Σ definiert. Gilt $\Sigma = \{0, 1\}$, so sprechen wir von einem *binären Korrespondenzproblem*, kurz BPCP. Zeigen Sie $\text{PCP} \leq \text{BPCP}$, indem Sie eine Reduktion von PCP auf BPCP konstruieren.

7

Komplexitätstheorie

In diesem Kapitel werden Sie ...

- den abstrakten Begriff der algorithmischen Komplexität begreifen,
- das O-Kalkül zur Algorithmenbewertung einsetzen,
- eine Übersicht über die wichtigsten Komplexitätsklassen erhalten,
- an die Grenze der praktischen Berechenbarkeit vordringen,
- NP-harte und NP-vollständige Probleme zu unterscheiden lernen,
- das P-NP-Problem verstehen.



7.1 Algorithmische Komplexität

Während sich die Berechenbarkeitstheorie mit der prinzipiellen Lösbarkeit von Problemen beschäftigt, untersucht die Komplexitätstheorie die Effizienz der eingesetzten Algorithmen. Für die praktische Informatik ist die Komplexitätstheorie von unschätzbarem Wert, schließlich lässt sich ein Entscheidungsverfahren nur dann auf reale Probleme anwenden, wenn sich sein Ressourcenverbrauch in realistischen Grenzen bewegt. Die praktische Anwendbarkeit eines Verfahrens wird vor allem durch seine Laufzeit und seinen Speicherplatzverbrauch bestimmt; beide Ressourcen stehen uns heute wie auch in der Zukunft nur in begrenztem Maße zur Verfügung.

In Kapitel 1 haben Sie am Beispiel des Problems PRIME bereits einen Eindruck erhalten, wie schnell die theoretische Berechenbarkeit in der Praxis an ihre Grenze stößt. Lösen wir PRIME auf naive Weise, so nimmt die Anzahl der Divisionen exponentiell mit der Bitbreite der zu testenden Zahl zu. Geringe Bitbreiten reichen aus, um selbst modernste Computeranlagen Jahrzehnte zu beschäftigen.

Die in der theoretischen Informatik durchgeführten Komplexitätsbe- trachtungen verfolgen das Ziel, die Laufzeit- und den Speicherplatzbedarf eines Algorithmus auf einer abstrakten Ebene zu bestimmen. Ins- besondere sollen die gewonnenen Ergebnisse unabhängig von

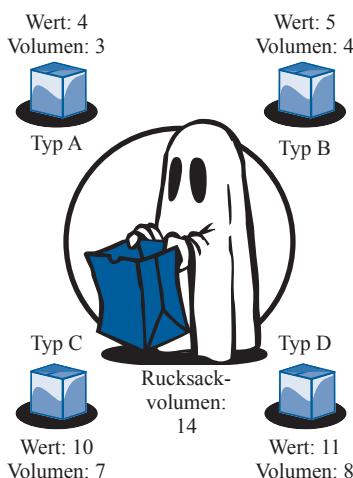


Abbildung 7.1: Hinter dem Rucksackproblem verbirgt sich die Aufgabe, das Diebesgut so auszuwählen, dass der erzielte Gewinn maximal ist.

- der Programmiersprache,
- dem installierten Betriebssystem,
- der Prozessorleistung,
- der Speicherausstattung und
- der zugrunde liegenden Computerarchitektur

sein. Dies ist der Grund, warum wir im Folgenden stets mit einheitenlosen Zahlen agieren werden.

Um einen tieferen Einblick in die Laufzeit- und Platzkomplexität realer Algorithmen zu erhalten, werden wir in diesem Abschnitt zwei Lösungsmöglichkeiten des *Rucksackproblems* diskutieren. Mit Begriffen des Alltags lässt sich das Problem wie folgt veranschaulichen: Ein Ganove findet in einem Tresor verschiedene Gegenstände vor, die in Abhängigkeit ihres Typs ein bestimmtes Volumen besitzen und auf dem Schwarzmarkt einen fixen Geldbetrag einbringen. Der Ganove sieht

sich mit der Aufgabe konfrontiert, seinen Rucksack so mit Diebesgut zu füllen, dass der erbeutete Wert maximal wird. Dabei nehmen wir an, dass die Gegenstände jedes Typs in unbegrenzter Anzahl vorhanden sind und wir den Wert und das Volumen als ganzzahlige Größen beschreiben können. Abbildung 7.1 demonstriert das Dilemma des Ganoven anhand eines konkreten Beispiels.

Mathematisch gesehen verbirgt sich hinter dem Rucksackproblem die folgende Aufgabe:



Definition 7.1 (Rucksackproblem)

Gegeben sei eine Zahl $v \in \mathbb{N}^+$ sowie eine Menge von Wertepaaren

$$(c_1, v_1), \dots, (c_n, v_n) \text{ mit } c_i, v_i \in \mathbb{N}^+, 1 \leq i \leq n$$

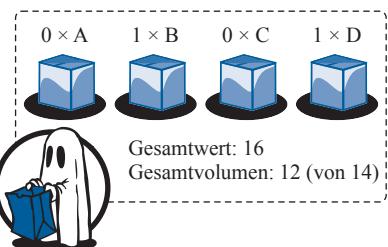
Gesucht ist eine Folge ganzzahliger Werte $a_1, \dots, a_n \in \mathbb{N}$, so dass

$$\sum_{i=1}^n a_i v_i \leq v \text{ und } \sum_{i=1}^n a_i c_i \text{ maximal ist.}$$

Obwohl das Problem auf den ersten Blick simpel erscheint, existiert selbst für kleine Rucksäcke eine Vielzahl an Auswahlmöglichkeiten. Für unser Eingangsbeispiel sind in Abbildung 7.2 zwei konkrete Wahlmöglichkeiten exemplarisch aufgeführt. Der erste Rucksack ist mit einem Typ-B-Gegenstand und einem Typ-D-Gegenstand bestückt und kommt auf einen Gesamtwert von $5 + 11 = 16$. Obwohl das Rucksackvolumen mit dieser Befüllung noch nicht komplett ausgeschöpft ist, lässt sich kein anderes Element mehr dazupacken, schließlich ist der kleinste Gegenstand (Typ A) größer als das Restvolumen. Wählt der Ganove stattdessen zwei Typ-C-Elemente, so wird das Rucksackvolumen komplett ausgeschöpft und ist mit einem Gesamtwert von 20 die bessere Wahl. Später werden wir zeigen, dass wir mit dieser Befüllung eine optimale Lösung vor uns haben, d.h., keine andere Wahl führt zu einem höheren Gesamtwert der entwendeten Ware.

Wir wollen uns der algorithmischen Lösung des Rucksackproblems nähern, indem wir zunächst eine abgeschwächte Variante untersuchen. Anstatt eine konkrete Auswahl an Gegenständen zu berechnen, versuchen wir zunächst den maximal erreichbaren Gewinn zu ermitteln. Anschließend machen wir uns Gedanken darüber, wie wir die erstellten Algorithmen so erweitern können, dass aus dem Ergebnis eine Liste der zu entwendenden Gegenstände extrahiert werden kann.

■ Erste Zusammenstellung



■ Zweite Zusammenstellung

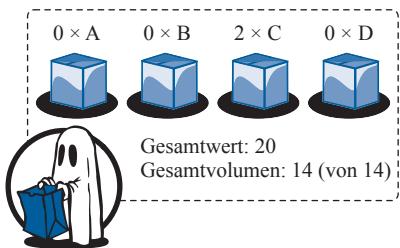


Abbildung 7.2: Zwei Auswahlmöglichkeiten für das Rucksackproblem aus Abbildung 7.1.

```

knapsack_recursive.c
1 int knapsack( int n )
2 {
3     int i, best, tmp;
4
5     best = 0;
6     for ( i = 0; i < m; i++ ) {
7         if ( n - size[i] >= 0 ) {
8             tmp = knapsack(n-size[i]);
9             tmp += gain[i];
10            if ( tmp > best ) {
11                best = tmp;
12            }
13        }
14    }
15    return best;
16 }

knapsack.c
1 int knapsack( int n )
2 {
3     int i, j, tmp;
4
5     for ( i = 0; i < m; i++ ) {
6         for ( j = 0; j <= n; j++ ) {
7             if ( j - size[i] >= 0 ) {
8                 tmp = best[j-size[i]];
9                 tmp += gain[i];
10                if ( tmp > best[j] )
11                    best[j] = tmp;
12            }
13        }
14    }
15    return best[n];
16 }

17
18

```

Abbildung 7.3: Rekursive und iterative Implementierung des Rucksackproblems

Das linke Listing in Abbildung 7.3 zeigt, wie das abgeschwächte Rucksackproblem rekursiv gelöst werden kann. Die Funktion knapsack nimmt die Größe eines Rucksacks als Parameter entgegen und berechnet für diesen den maximal erzielbaren Gewinn. Während der Ausführung greift die Funktion auf die globalen Arrays size und gain zurück. Für $0 \leq i < m$ enthalten die Elemente size[i] und gain[i] das Volumen bzw. den Wert des i -ten Gegenstands und bleiben während der gesamten Berechnung unverändert. Wir nehmen für die Berechnung an, dass die Variable m mit der Anzahl der zur Verfügung stehenden Gegenstände initialisiert ist.

Intern arbeitet die Funktion nach einem einfachen Prinzip. In einer Schleife wird über alle zur Auswahl stehenden Elementen iteriert und in der i -ten Iteration hypothetisch das i -te Element ausgewählt. Anschließend wird über einen rekursiven Aufruf die optimale Befüllung für den dann verbleibenden Rucksackinhalt berechnet und die beste der bisher gefundenen Lösungen in der Variablen best gespeichert.

Angewendet auf unser Beispielszenario aus Abbildung 7.1 liefert der Funktionsaufruf knapsack(14) den Wert 20 zurück. Wie dieser Wert zu stande kommt, zeigt Abbildung 7.4 anhand des Rekursionsbaums, den der Algorithmus intern durchläuft.

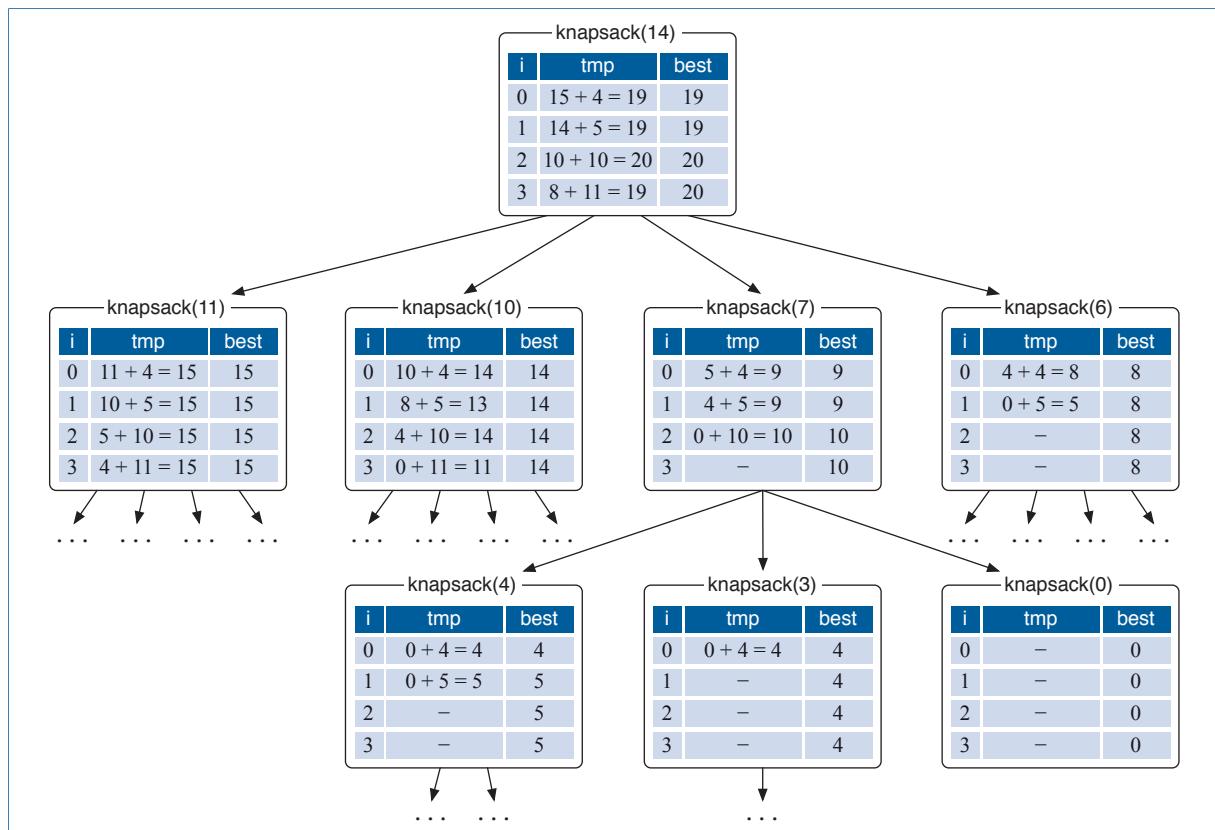


Abbildung 7.4: Auszug aus der internen Berechnungsfolge der rekursiven Implementierung

Der Rekursionsbaum gibt bereits wertvolle Hinweise auf die Laufzeitkomplexität des Algorithmus. In jedem Funktionsaufruf wird der Schleifenkörper m -mal durchlaufen und in jeder Iteration ein rekursiver Aufruf getätigt. Da der Aufrufparameter n hierbei um mindestens 1 verringert wird, besitzt der Rekursionsbaum eine maximale Tiefe von n . Damit enthält der vollständig expandierte Rekursionsbaum bis zu m^n Knoten, so dass sich die Laufzeit des Algorithmus nach oben durch die Funktion m^n abschätzen lässt. In der Nomenklatur der Komplexitätstheorie wird die Proportionalitätsbeziehung in der Form $O(m^n)$ notiert. Was sich hinter dieser Notation formal verbirgt, werden wir im nächsten Abschnitt klären.

Für die Bestimmung des Speicherplatzbedarfs müssen wir beachten, dass ein realer Rechner in jedem rekursiven Aufruf einen neuen Stack-

Rahmen erzeugen muss, in dem alle lokalen Variablen der Funktion zwischengespeichert werden. Die maximale Anzahl der gleichzeitig gespeicherten Stack-Rahmen entspricht der maximalen Tiefe des Rekursionsbaums und ist damit gleich n . Mit anderen Worten: Der Speicherplatzbedarf steigt linear mit der Größe des Rucksacks und bleibt durch den zweiten Parameter m unbeeinflusst.

Die rekursive Implementierung von $\text{knapsack}(n)$ besitzt die folgende Komplexität:



	Laufzeit	Speicherplatz
	$O(m^n)$	$O(n)$

Die Analyse offenbart, dass die rekursive Lösung des abgeschwächten Rucksackproblems in der Praxis kaum brauchbar ist. Verantwortlich ist die Laufzeit, die mit steigender Rucksackgröße exponentiell zunimmt.

j	$i = 0$	$i = 1$	$i = 2$	$i = 3$
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	4	4	4	4
4	4	5	5	5
5	4	5	5	5
6	8	8	8	8
7	8	9	10	10
8	8	10	10	11
9	12	12	12	12
10	12	13	14	14
11	12	14	15	15
12	16	16	16	16
13	16	17	18	18
14	16	18	20	20

Tabelle 7.1: Mit dem Mittel der dynamischen Programmierung lässt sich das Rucksackproblem bestechend einfach lösen.

Dynamische Programmierung

Dennoch ist das Rucksackproblem effizienter lösbar. Den Schlüssel hierzu liefert das Prinzip der dynamischen Programmierung, dem wir bereits in Abschnitt 4.4.4 im Rahmen der Diskussion über den CYK-Algorithmus begegnet sind. Folgt unser Ganove diesem Prinzip, so ersetzt er die Variable best durch eine Tabelle der Länge n und speichert in $\text{best}[j]$ den maximal zu erbeutenden Wert für einen Rucksack mit dem Fassungsvermögen j (vgl. Abbildung 7.3 rechts).

Das Array dient als Arbeitsbereich und wird in einem iterativen Prozess permanent aktualisiert. Im ersten Iterationsschritt wird best so ausgefüllt, als hätten wir ausschließlich Typ-A-Gegenstände zur Verfügung. Im nächsten Iterationsschritt wird auch die Auswahl von B erwogen, im übernächsten die Auswahl von C und so fort. Der Ganove geht nun so vor, dass er in jeder Iteration die optimale Lösung für $\text{best}[i]$ aus den bereits berechneten Werten ermittelt.

Angenommen, der Wert $\text{best}[14]$ enthält am Ende der zweiten Iteration den Wert 18. Der Wert besagt, dass unser Ganove einen maximalen Gewinn von 18 erzielen kann, wenn er nur Typ-A- und Typ-B-Gegenstände stiebt. Basierend auf unserem Beispielszenario stellt er folgende Überlegung an: Stiebt er einen Typ-C-Gegenstand, so besitzt dieser den Wert

knapsack_final.c <pre> void knapsack(int n) { int i, j, tmp; for (i = 0; i < m; i++) { for (j = 0; j <= n; j++) { if (j - size[i] >= 0) { tmp = best[j - size[i]]; tmp += gain[i]; if (tmp > best[j]) { best[j] = tmp; item[j] = i; } } } } } </pre>	main.c <pre> 1 int size[] = { 3, 4, 7, 8 }; 2 int gain[] = { 4, 5, 10, 11 }; 3 int best[15], item[15]; 4 int m = 4; 5 6 int main() 7 { 8 int j; 9 10 knapsack(14); 11 12 for (j=14; j>=3; j-=size[item[j]]) 13 printf("%d ", item[j]); 14 15 return 0; 16 } 17 18 19 </pre>
--	--

Abbildung 7.5: Durch eine minimale Programmerweiterung lässt sich nicht nur der maximal zu erbeutende Gewinn, sondern auch die optimale Bepackung des Rucksacks berechnen. Die zu stehlenden Gegenstände lassen sich auf einfache Weise durch die Rückverfolgung der gespeicherten Tabelleneinträge bestimmen.

10 und es bleibt im Rucksack ein Restvolumen von 7 übrig. Die optimale Lösung, um das Restvolumen zu füllen, kennen wir aber bereits; sie entspricht dem Wert von `best[7]`. Ist die Summe aus `best[7]` und 10 größer als der bisher berechnete Wert 18, so lohnt es sich, einen Typ-C-Gegenstand zu entwenden. Andernfalls bleibt der Ganove bei Typ-A- und Typ-B-Gegenständen. Angewendet auf unser ursprüngliches Beispielszenario liefert der Algorithmus das in Tabelle 7.1 dargestellte Ergebnis. Den maximal zu erbeutenden Wert kann unser Ganove am Ende der Berechnung im Feld `best[14]` ablesen.

Um die Komplexität dieser Implementierungsvariante zu bestimmen, werfen wir erneut einen Blick auf das rechte Listing in Abbildung 7.3. Insgesamt durchläuft die Funktion `knapsack` zwei verschachtelte Schleifen. Während die äußere über alle Gegenstände iteriert und damit m -mal ausgeführt wird, erstreckt sich die innere Schleife über alle Rucksackvolumina von 0 bis n . Die Laufzeit des Algorithmus nimmt somit proportional mit dem Produkt $m \cdot n$ zu.

Auch der Speicherplatzverbrauch fällt moderat aus. Neben dem neu hinzugefügten Array `best` benötigen wir keine weiteren Hilfsvariablen. Das

Array besitzt n Elemente, so dass der Speicherplatz linear mit dem Volumen der betrachteten Rucksäcke wächst. Dass sich der Speicherhunger der betrachteten Implementierung in Grenzen hält, haben wir einer besonderen Eigenschaft des Rucksackproblems zu verdanken. Da wir zur Bestimmung des Feldelements $\text{best}[j]$ ausschließlich die Ergebnisse der vorangegangenen Iteration benötigen, können wir ältere Zwischenergebnisse verwerfen. Damit ist es vollkommen ausreichend, zu jedem Zeitpunkt nur eine einzige der in Tabelle 7.1 dargestellten Spalten im Speicher vorzuhalten. Anders als es z. B. im CYK-Algorithmus aus Abschnitt 4.4.4 der Fall war, können wir das Rucksackproblem hierdurch mit einem eindimensionalen Array lösen.

j	$i = 0$	$i = 1$	$i = 2$	$i = 3$
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	4,A	4,A	4,A	4,A
4	4,A	5,B	5,B	5,B
5	4,A	5,B	5,B	5,B
6	8,A	8,A	8,A	8,A
7	8,A	9,B	10,C	10,C
8	8,A	10,B	10,B	11,D
9	12,A	12,A	12,A	12,A
10	12,A	13,B	14,C	14,C
11	12,A	14,B	15,C	15,C
12	16,A	16,A	16,A	16,A
13	16,A	17,B	18,C	18,C
14	16,A	18,B	20,C	20,C

Tabelle 7.2: Interne Berechnungsabfolge des finalen Algorithmus. Durch die Rückverfolgung der Einträge lässt sich die optimale Rucksackbepackung extrahieren.

Mit dem Mittel der dynamischen Programmierung lässt sich das Rucksackproblem mit der folgenden Komplexität lösen:

Laufzeit	Speicherplatz
$O(m \cdot n)$	$O(n)$

Die bisher vorgestellten Implementierungsvarianten geben dem Ganoven zwar über den maximal zu erbeutenden Gewinn Auskunft, liefern ihm jedoch keinerlei Hinweis, welche Gegenstände er dafür auswählen muss. Genau dies ist aber die Aufgabe des ursprünglich in Definition 7.1 formulierten Rucksackproblems. Legen wir die zweite Implementierungsvariante zugrunde, so können wir dieses Problem mit einer minimalen Modifikation lösen. Hierzu ergänzen wir das Array best um ein zweites Array item , in dem wir den zuletzt hinzugefügten Gegenstandstyp speichern. Jedes Mal, wenn das Array-Element $\text{best}[j]$ beschrieben wird, aktualisieren wir ebenfalls das Element $\text{item}[j]$. Die entsprechend modifizierte Implementierung ist in Abbildung 7.5 zusammengefasst.

Durch die Rückverfolgung der gespeicherten Einträge kann unser Ganove eine optimale Rucksackbepackung für unser Beispieldaten ablesen. Über den Typ des ersten Gegenstandes gibt der Inhalt von $\text{item}[14] (= C)$ Auskunft (vgl. Tabelle 7.2). Um den zweiten Gegenstand zu bestimmen, berechnen wir das Volumen von C und subtrahieren das Ergebnis von 14. Es gilt $\text{item}[14-7] (= C)$, so dass wir einen zweiten Typ-C-Gegenstand entwinden. Mit diesem Gegenstand ist der Rucksack restlos vollgepackt und die optimale Lösung gefunden. Gut, dass die meisten Ganoven noch nie etwas vom Prinzip der dynamischen Programmierung gehört haben...

7.1.1 O-Kalkül

Weiter oben haben wir zur Angabe der algorithmischen Komplexität bereits auf die *O-Notation* zurückgegriffen, ohne uns um eine genaue Begriffsbestimmung zu kümmern. Dies wollen wir jetzt nachholen und die durchgeföhrten Laufzeit- und Speicherplatzbetrachtungen gleichzeitig auf eine abstraktere Ebene heben. Hierzu werden wir zunächst eine formale Definition der verschiedenen Notationsvarianten vornehmen und anschließend die Gültigkeit mehrerer Rechenregeln erarbeiten. Als Ergebnis werden wir mit dem *O-Kalkül* eine Beschreibungsform erhalten, die den Umgang mit Wachstumsfunktionen deutlich erleichtert.

Im Kern verfolgt die O-Notation die Idee, Wachstumsfunktionen anhand ihrer Wertentwicklungen in Äquivalenzklassen einzuteilen, die von den realen Funktionswerten abstrahieren. Formal definieren wir die Notation wie folgt:



Definition 7.2 (O-Notation)

Die Funktion $f : \mathbb{N} \rightarrow \mathbb{R}^+$ fällt in die Komplexitätsklasse

$$O(g(n)),$$

falls eine Konstante $c \in \mathbb{R}^+$ und ein $n_0 \in \mathbb{N}$ existieren, so dass

$$f(n) \leq c \cdot g(n)$$

für alle $n \geq n_0$ gilt.

Das Notationszeichen O ist eines von mehreren *Landau-Symbolen*, benannt nach dem deutschen Mathematiker Edmund Georg Hermann Landau (vgl. Abbildung 7.6).

Zwei Punkten wollen wir an dieser Stelle unsere besondere Aufmerksamkeit schenken.

- In Definition 7.2 wird lediglich gefordert, dass die Beziehung $f(n) \leq c \cdot g(n)$ für alle n erfüllt sein muss, die *größer oder gleich* einer gewissen Konstanten n_0 sind. Die getätigte Einschränkung legt einen Kerngedanken der Komplexitätsanalyse offen. Ziel ist es, das Laufzeitverhalten und den Platzbedarf eines Programms für sehr große Eingaben zu analysieren. Wie sich die ermittelten Wachstumsfunktionen auf einem endlichen Anfangsstück verhalten, spielt keine

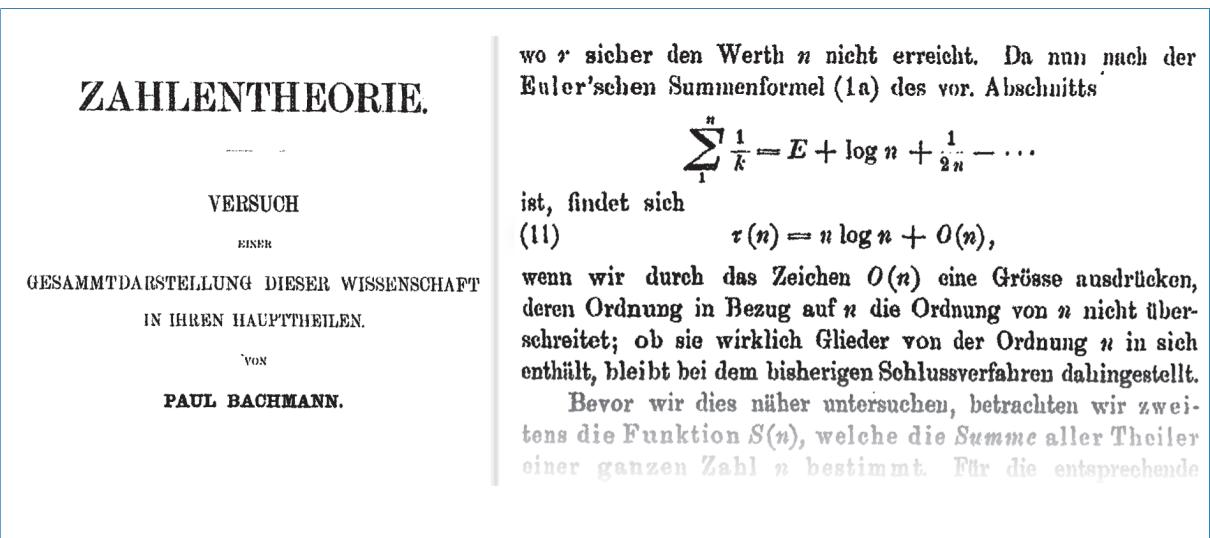


Abbildung 7.6: Die O-Notation hat sich heute als Standard für die Beschreibung von Komplexitätsklassen etabliert. Die verwendeten Symbole wurden vor allem durch die Arbeiten des deutschen Mathematikers Edmund Landau bekannt, ihr wahrer Ursprung reicht jedoch bis in das Jahr 1894 zurück. Dort taucht die O-Notation im zweiten Band von Paul Bachmanns Werk „Zahlentheorie: Versuch einer Gesamtdarstellung dieser Wissenschaft“ auf Seite 401 auf [5]. In [64] äußert sich Landau wie folgt über die Notation: „Das Zeichen $O(g(x))$ habe ich zuerst bei Herrn Bachmann vorgefunden. Das hier hinzugefügte Zeichen $o(g(x))$ entspricht dem, was ich in meinen Abhandlungen früher mit $\{g(x)\}$ zu bezeichnen pflegte.“

Rolle. Da wir den Größenparameter n gegen unendlich streben lassen, sprechen wir auch von der *asymptotischen Komplexität* eines Programms oder Algorithmus.

- Die in Definition 7.2 vorkommende Konstante c dürfen wir beliebig wählen. Konkret hat dies zu Folge, dass die O-Notation von sämtlichen konstanten Faktoren abstrahiert. Fällt eine Funktion $f(n)$ in die Komplexitätsklasse $O(g(n))$, so gehört auch die Funktion $k \cdot f(n)$ zu dieser Klasse – unabhängig davon, wie groß wir die Konstante k auch wählen. Diese Eigenschaft ist gewollt, da wir sämtliche Komplexitätsbetrachtungen auf abstrakten Größen durchführen, die bewusst von einer konkreten Einheit abstrahieren. Kurzum: Die wichtige Information ist für uns die asymptotische Zunahme der Kenngröße und nicht deren realer Wert.

Behalten Sie stets im Gedächtnis, dass der Ausdruck $O(g(n))$ eine *Menge von Funktionen* beschreibt und die Schreibweise $f(n) \in O(g(n))$ ausdrückt, dass die Funktion f in die durch g definierte Komplexitätsklasse fällt. Obwohl dies die formal korrekte Schreibweise ist, hat sich in der

Praxis die Notation $f(n) = O(g(n))$ durchgesetzt. Aus mathematischer Sicht ist diese Schreibweise schlicht falsch. Trotzdem ist sie heute so weit verbreitet, dass wir uns nicht dagegen sträuben wollen.

Mithilfe der O -Notation sind wir in der Lage, das asymptotische Wachstum einer Funktion $f(n)$ nach oben abzuschätzen. In analoger Weise drücken die Symbole Ω und Θ eine Abschätzung nach unten bzw. in beide Richtungen aus.



Definition 7.3 (Ω -, Θ -Notation)

Die Funktion $f : \mathbb{N} \rightarrow \mathbb{R}^+$ fällt in die Komplexitätsklasse

$$\Omega(g(n)),$$

falls eine Konstante $c \in \mathbb{R}^+$ und ein $n_0 \in \mathbb{N}$ existieren, so dass

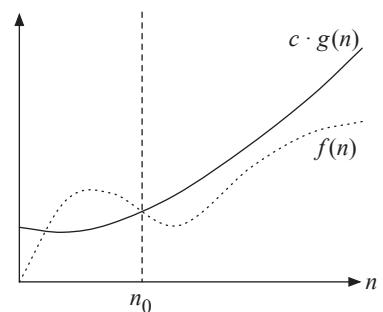
$$f(n) \geq c \cdot g(n)$$

für alle $n \geq n_0$ gilt. f fällt in die Komplexitätsklasse

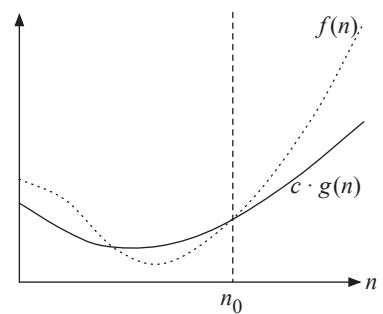
$$\Theta(g(n)),$$

falls f sowohl in $O(g(n))$ als auch in $\Omega(g(n))$ liegt.

■ $O(n)$



■ $\Omega(n)$



■ $\Theta(n)$

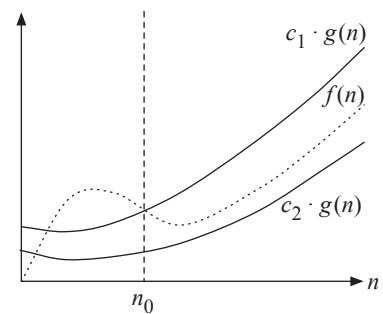


Abbildung 7.7 stellt die Bedeutung der drei Landau-Symbole O , Ω und Θ bildlich gegenüber.

Neben O , Ω und Θ existieren mit o und ω zwei weitere, wenn auch weniger gebräuchliche Symbole. Die Beziehung $f(n) = o(g(n))$ bzw. $f(n) = \omega(g(n))$ drückt aus, dass die Funktion f echt langsamer bzw. echt schneller als g wächst. Formal definieren wir die Symbole so:



Definition 7.4 (o -, ω -Notation)

Die Funktion $f : \mathbb{N} \rightarrow \mathbb{R}^+$ fällt in die Komplexitätsklasse

$$o(g(n)) \quad \text{bzw.} \quad \omega(g(n)),$$

falls für alle Konstanten $c \in \mathbb{R}^+$ ein $n_0 \in \mathbb{N}$ existiert, so dass

$$c \cdot f(n) \leq g(n) \quad \text{bzw.} \quad c \cdot f(n) \geq g(n)$$

für alle $n \geq n_0$ gilt.

Abbildung 7.7: Die Landau-Symbole O , Ω und Θ im Vergleich

In den meisten Informatikbüchern wird die Komplexität von Algorithmen in der O-Notation angegeben. Tatsächlich scheint aber die Θ -Komplexität das adäquatere Messinstrument zu sein, da die Laufzeit oder der Speicherplatzbedarf durch die O-Notation lediglich nach oben abgeschätzt wird. Des Weiteren werden Ausdrücke der Form $O(g(n))$ in der Praxis häufig missinterpretiert. So suggeriert der Ausdruck $O(2^n)$, dass sich Probleme aus dieser Klasse ausschließlich durch Algorithmen lösen lassen, deren Laufzeit oder Speicherbedarf exponentiell mit der Eingangsgröße n zunimmt. Da die O-Notation aber nur eine Abschätzung nach oben vornimmt, liegen z. B. auch alle Algorithmen mit linearem Ressourcenverbrauch in der Komplexitätsklasse $O(2^n)$. Die Angabe der Θ -Komplexität würde Missverständnisse dieser Art vorab vermeiden.

In der Praxis wird schlicht und einfach deshalb auf die O-Notation zurückgegriffen, weil sich die Θ -Komplexität für viele Probleme nur schwer berechnen lässt. Um zu zeigen, dass ein Problem in $\Theta(g(n))$ liegt, müssen wir nachweisen, dass $g(n)$ sowohl eine asymptotische obere Schranke ($O(g(n))$) als auch eine asymptotische untere Schranke ($\Omega(g(n))$) ist. Ersteres kann wie gewohnt dadurch geschehen, dass ein konkreter Algorithmus mit der entsprechenden Komplexität angegeben wird. Der Nachweis einer unteren Schranke gestaltet sich in den meisten Fällen ungleich schwieriger. Hier müssen wir nachweisen, dass *alle* Algorithmen, die das untersuchte Problem lösen, mindestens die durch $g(n)$ definierte Komplexität besitzen. In der Tat existieren nur wenige Probleme, für die der formale Nachweis einer exakten unteren Komplexitätsschranke bisher gelungen ist. Dies ist der Grund, warum wir in der Praxis fast immer auf die O-Notation angewiesen sind und auf die aussagekräftigere Θ -Notation verzichten müssen.

Die Mengen $o(g(n))$ bzw. $\omega(g(n))$ lassen sich aus den verwandten Mengen $O(g(n))$ und $\Omega(g(n))$ konstruieren, indem diejenigen Funktionen herausgenommen werden, die in der gleichen Komplexitätsklasse wie $g(n)$ liegen. Es gelten die folgenden Zusammenhänge:

$$\begin{aligned} o(g(n)) &= O(g(n)) - \Theta(g(n)) \\ \omega(g(n)) &= \Omega(g(n)) - \Theta(g(n)) \end{aligned}$$

So ähnlich die Notationen auf den ersten Blick erscheinen mögen, so unterschiedlich ist ihr Einsatzzweck. O, Ω und Θ dienen in erster Linie zur Bestimmung der Komplexität eines Algorithmus. In diesem Fall wird das Ziel verfolgt, die wahre Komplexitätsklasse so genau wie möglich zu treffen. o und ω dienen stattdessen der Abgrenzung von Komplexitätsklassen. Können wir z. B. die Beziehung $\log n = o(\sqrt{n})$ zeigen, so ist der Beweis erbracht, dass beide Funktionen unterschiedlichen Komplexitätsklassen angehören, die Logarithmus-Funktion also asymptotisch langsamer wächst als Wurzel- n . Dagegen besitzt die Beziehung $\log n = O(\sqrt{n})$ eine deutlich geringere Aussagekraft. Sie wäre auch dann erfüllt, wenn beide Funktionen asymptotisch gleich schnell wachsen würden.

Bisher haben wir die O-Notation dazu verwendet, um die asymptotische Komplexität einzelner Algorithmen zu beschreiben. Im Folgenden werden wir den Begriff weiter fassen und auch von der asymptotischen Komplexität von *Problemen* reden.



Definition 7.5 (Problemkomplexität)

P sei ein beliebiges Ja-Nein-Problem. A sei die Menge aller Algorithmen, die P entscheiden. Wir schreiben

- $P = O(g(n))$, falls für ein $a \in A$ gilt: $f_a = O(g(n))$
- $P = \Omega(g(n))$, falls für alle $a \in A$ gilt: $f_a = \Omega(g(n))$
- $P = \Theta(g(n))$, falls $P \in O(g(n))$ und $P \in \Omega(g(n))$

Die Wachstumsfunktion $f_a : \mathbb{N} \rightarrow \mathbb{R}^+$ beschreibt den Ressourcenverbrauch von a für eine Eingabe der Länge n .

7.1.2 Rechnen im O-Kalkül

In diesem Abschnitt werden wir zunächst eine alternative Definition der Laundau-Symbole ins Spiel bringen und anschließend mehrere Rechen-

regeln einführen, die uns einen algebraischen Umgang mit den Komplexitätsklassen ermöglichen.

Sicher erinnern Sie sich, dass wir die Bedeutung der Landau-Symbole O und Ω mithilfe von Ungleichungen der Form $f(n) \leq c \cdot g(n)$ bzw. $f(n) \geq c \cdot g(n)$ definiert haben. Diese Art der Darstellung entspricht zwar weitgehend der intuitiven Vorstellung eines asymptotischen Wachstums, ist für den Beweis von Komplexitätsaussagen aber nur bedingt geeignet.

Abhilfe schafft eine alternative Charakterisierung des asymptotischen Wachstums, die auf der Grenzwertbetrachtung des Quotienten $\frac{f(n)}{g(n)}$ beruht. Divergiert der Quotient gegen ∞ bzw. konvergiert er gegen 0, so können wir daraus schließen, dass die Funktion f asymptotisch schneller bzw. asymptotisch langsamer wächst als g . Konvergiert der Quotient gegen einen konstanten Wert größer null, so zeigen f und g das gleiche asymptotische Wachstum. Insgesamt erhalten wir die folgende Grenzwertregel:



Satz 7.1 (Grenzwertregel)

Für die Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ gilt der folgende Zusammenhang:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = O(g(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) = \Omega(g(n))$$

Neben O und Ω lassen sich auch die anderen Landau-Symbole Θ , \circ und ω über eine entsprechende Grenzwertbetrachtung charakterisieren. Tabelle 7.3 fasst die Ergebnisse in einer Übersicht zusammen.

Den praktischen Nutzen der Grenzwertregel wollen wir am Beispiel des asymptotischen Wachstums von Polynomen der Form

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \quad (7.1)$$

erproben. Fordern wir die Eigenschaft $a_k > 0$, so entspricht k dem *Grad* des Polynoms p und es gilt die folgende Beziehung:

$$\lim_{n \rightarrow \infty} \frac{p(n)}{n^k} = \lim_{n \rightarrow \infty} \left(a_k + \frac{a_{k-1}}{n} + \dots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k} \right) = a_k$$

Wegen $a_k > 0$ folgt aus der Grenzwertregel unmittelbar der folgende Satz:

Obere asymptotische Schranke (f wächst höchstens so schnell wie g)		
$f(n) = O(g(n))$	$\exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 f(n) \leq c \cdot g(n)$	$0 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
Untere asymptotische Schranke (f wächst mindestens so schnell wie g)		
$f(n) = \Omega(g(n))$	$\exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 f(n) \geq c \cdot g(n)$	$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$
Exakte asymptotische Schranke (f wächst genauso schnell wie g)		
$f(n) = \Theta(g(n))$	$\exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
Starke obere asymptotische Schranke (f wächst langsamer als g)		
$f(n) = o(g(n))$	$\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c \cdot f(n) \leq g(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
Starke untere asymptotische Schranke (f wächst schneller als g)		
$f(n) = \omega(g(n))$	$\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c \cdot f(n) \geq g(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$

Tabelle 7.3: Landau-Symbole in der Übersicht**Satz 7.2 (Asymptotisches Wachstum von Polynomen)**

Sei $p(n)$ ein Polynom vom Grad k . Dann gilt $p(n) = \Theta(n^k)$

Betrachten wir den Aufbau des Polynoms (7.1) von einem abstrakten Standpunkt aus, so können wir $p(n)$ als eine Summe von Teiltermen unterschiedlicher Komplexitätsklassen ansehen. Zwischen den Summanden des Polynoms besteht die folgende Hierarchiebeziehung:

$$a_k n^k = o(a_{k+1} n^{k+1})$$

Ferner hat uns Satz 7.2 gezeigt, dass das asymptotische Wachstum eines Polynoms ausschließlich durch den Summanden $a_k n^k$ und damit durch den Teilausdruck mit dem stärksten asymptotischen Wachstum bestimmt wird. Wir wollen dieses Ergebnis verallgemeinern und betrachten jetzt die Summe zweier Funktionen f und g mit $f(n) = o(g(n))$. Unter Verwendung der oben eingeführten Grenzwerte können wir diese Eigenschaft auch so formulieren:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

Da sich der Quotient der Nulllinie immer weiter annähert, existiert ein $n_0 \in \mathbb{N}$ mit

$$0 \leq \frac{f(n)}{g(n)} \leq 1 \quad \text{für alle } n \geq n_0.$$

Jetzt können wir die Summe $f(n) + g(n)$ für $n \geq n_0$ wie folgt abschätzen:

$$f(n) + g(n) \leq g(n) + g(n) \leq 2 \cdot g(n) \quad (7.2)$$

Voilà. Auch hier wird die Komplexitätsklasse ausschließlich durch den dominierenden Summanden g bestimmt und wir haben einen Beweis für den folgenden Satz gefunden:

Satz 7.3 (Summation im O-Kalkül)

Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ zwei Funktionen mit $f = o(g(n))$. Dann gilt:

$$f(n) + g(n) = O(g(n))$$

Der Satz hat weitreichende Konsequenzen für das Rechnen im O-Kalkül, schließlich wird die asymptotische Komplexität einer Summe ausschließlich durch denjenigen Summanden mit der höchsten Wachstumsordnung bestimmt. Haben wir diesen identifiziert, so können wir die Komplexitätsklasse durch das Weglassen aller anderen Terme sofort bestimmen.

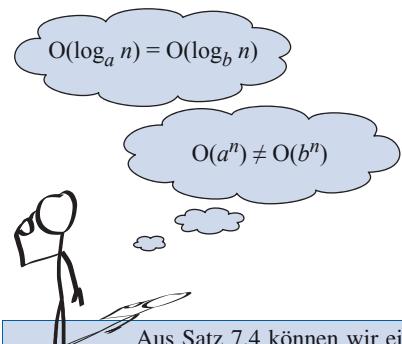
Satz 7.3 ist zugleich die Begründung, warum wir niemals Ausdrücke wie z. B. $O(3 \cdot n^3 + 7 \cdot n^2)$ angeben werden. Da die Komplexitätsklasse einzig und alleine durch den kubischen Term $3 \cdot n^3$ bestimmt wird, verwenden wir die vereinfachte Schreibweise $O(n^3)$.

Dass wir den konstanten Faktor 3 ebenfalls streichen können, ist eine weitere zentrale Eigenschaft des O-Kalküls. Gilt nämlich $f(n) = O(k \cdot g(n))$ für eine Konstante $k \in \mathbb{R}^+$, so existieren per Definition Konstanten $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$ mit $f(n) \leq c \cdot (k \cdot g(n))$ für alle $n \geq n_0$. Aus $c \cdot (k \cdot g(n)) = (c \cdot k) \cdot g(n)$ folgt sofort die Beziehung $f(n) = O(g(n))$ und wir erhalten auf einen Schlag das nachstehende Ergebnis:

Satz 7.4 (Multiplikation im O-Kalkül)

Für beliebige Funktionen $g : \mathbb{N} \rightarrow \mathbb{R}^+$ und Konstanten $k \in \mathbb{R}^+$ gilt:

$$O(k \cdot g(n)) = O(g(n))$$



Aus Satz 7.4 können wir eine bemerkenswerte Eigenschaft über das asymptotische Wachstum der Logarithmus-Funktion ableiten. Zunächst halten wir fest, dass der Logarithmus für beliebige Basen a und b die folgende Beziehung erfüllt:

$$\begin{aligned} \log_a x &= \log_a b^{\log_b x} \\ &= (\log_a b) \cdot \log_b x \end{aligned}$$

Die Umformung zeigt, dass sich die verschiedenen Logarithmus-Funktionen nur durch einen konstanten Faktor voneinander unterscheiden. Jetzt greift Satz 7.4 und garantiert uns für beliebige Basen a und b die Beziehung

$$O(\log_a n) = O(\log_b n)$$

Die asymptotische Komplexität der Logarithmusfunktion bleibt durch die konkrete Wahl der Basis gänzlich unbeeinflusst. Aus diesem Grund dürfen wir die Komplexitätsklasse ruhigen Gewissens in der basenunabhängigen Form

$$O(\log n)$$

angeben, ohne an Präzision zu verlieren. Beachten Sie, dass sich diese Eigenschaft nicht auf die Umkehrfunktion des Logarithmus – die Exponentialfunktion – überträgt. Hier gilt für $a \neq b$ die Beziehung

$$O(a^n) \neq O(b^n),$$

d. h., jede Basis definiert hier in der Tat eine eigene Komplexitätsklasse.

Wachstumsfunktion	
$f_0(n)$	$= O(1)$
$f_1(n)$	$= O(\log \log \log n)$
$f_2(n)$	$= O(\log \log n)$
$f_3(n)$	$= O(\sqrt{\log n})$
$f_4(n)$	$= O(\log n)$
$f_5(n)$	$= O((\log n)^2)$
$f_6(n)$	$= O(\sqrt[3]{n})$
$f_7(n)$	$= O(n)$
$f_8(n)$	$= O(n \log n)$
$f_9(n)$	$= O(\sqrt{n^3})$
$f_{10}(n)$	$= O(n^2)$
$f_{11}(n)$	$= O(n^3)$
$f_{12}(n)$	$= O(n^{\log n})$
$f_{13}(n)$	$= O(2^{\sqrt{n}})$
$f_{14}(n)$	$= O(2^n)$
$f_{15}(n)$	$= O(e^n)$
$f_{16}(n)$	$= O(n!)$
$f_{17}(n)$	$= O(n^n)$
$f_{18}(n)$	$= O(2^{n^2})$
$f_{19}(n)$	$= O(2^{2^n})$

Tabelle 7.4: Hierarchische Anordnung verschiedener Wachstumsfunktionen

7.2 Komplexitätsklassen

Mithilfe der verschiedenen Laudau-Notationen sind wir in der Lage, Wachstumsfunktionen in verschiedene Äquivalenzklassen einzuteilen. Diese bilden untereinander eine Hierarchie, die in Tabelle 7.4 auszugsweise zusammengefasst ist. Alle gelisteten Funktionen erfüllen die Beziehung

$$\lim_{n \rightarrow \infty} \frac{f_i}{f_{i+1}} = 0, \quad 0 \leq i \leq 18 \quad (7.3)$$

Gleichung (7.3) ist gleichbedeutend mit $f_i(n) = o(f_{i+1})$, so dass jede der abgebildeten Funktionen bewiesenermaßen eine eigenständige Komplexitätsklasse definiert.

In der Realität sind die vorgestellten Wachstumsfunktionen unterschiedlich oft anzutreffen. Von besonderer Bedeutung sind dort die folgenden Komplexitätsklassen:

■ $O(1)$ (Konstanter Ressourcenverbrauch)

Diese Komplexitätsklasse spielt nahezu ausschließlich in der Speicherplatzanalyse eine Rolle. Der Speicherplatzverbrauch eines solchen Algorithmus ist konstant und wird durch die Eingabegröße n nicht beeinflusst. Läge die Laufzeit in der Klasse $O(1)$, so könnte der Algorithmus immer nur eine konstante Anzahl an Eingabezeichen auswerten – unabhängig von der wahren Eingabelänge. Es erfordert eine gehörige Portion destruktiven Scharfsinns, um einen sinnvollen Algorithmus mit dieser Eigenschaft hervorzubringen.

■ $O(\log n)$ (Logarithmisches Wachstum)

Viele Algorithmen, die nach dem *Teile-und-herrsche-Prinzip (divide and conquer)* arbeiten, fallen in diese Kategorie. Ein logarithmischer Zeitaufwand entsteht z. B. dann, wenn ein Problem der Länge n in jedem Verarbeitungsschritt auf ein Problem der Länge $\frac{n}{2}$ reduziert wird. Beispiele sind die Suche innerhalb eines Binärbaums oder das ordnungserhaltende Einfügen in eine sortierte Liste [90].

■ $O(n)$ (Lineares Wachstum)

Die Laufzeit bzw. der Speicherplatzverbrauch eines solchen Algorithmus nimmt proportional mit der Eingabegröße n zu. Algorithmen mit dieser Eigenschaft gibt es zuhauf. Bekannte Vertreter linearer Laufzeitalgorithmen sind z. B. die Suche eines Elements in einem unsortierten Array oder die Multiplikation zweier n -stelliger Vektoren.

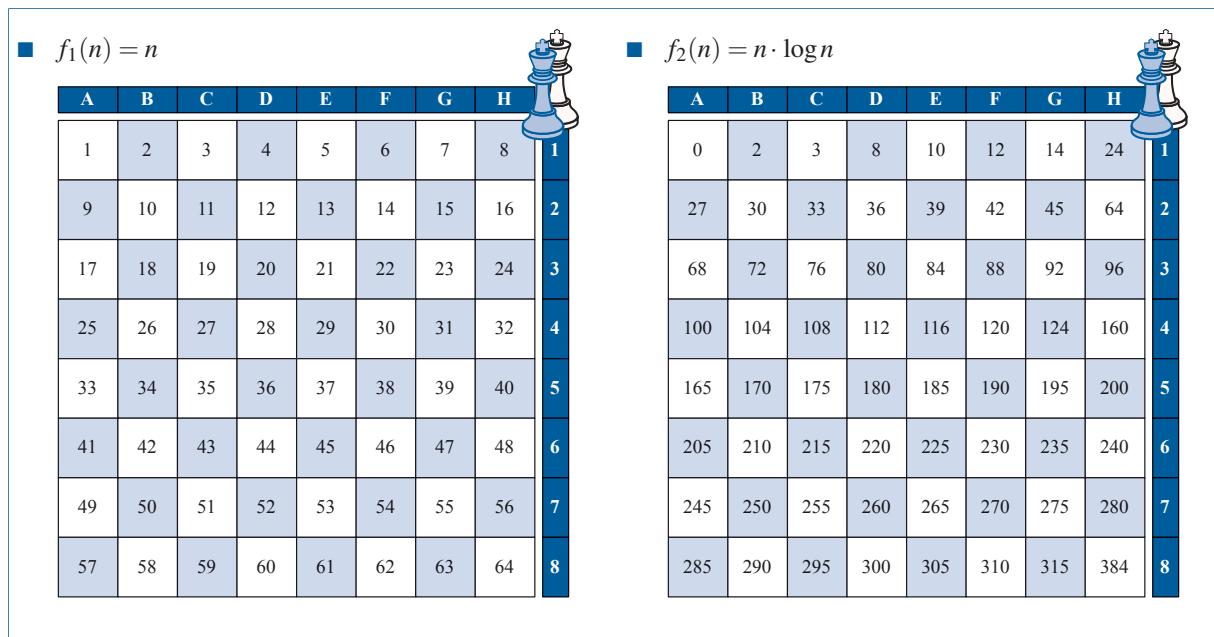


Abbildung 7.8: Die Wachstumsfunktionen n und $n \cdot \log n$ im Vergleich

■ $O(n \log n)$ (Linear-logarithmisches Wachstum)

In der Praxis kommen viele Algorithmen zum Einsatz, deren Laufzeit proportional mit dem Produkt $n \cdot \log n$ zunimmt. Beispiele sind die Sortierung einer Liste (Heapsort [107], Mergesort [62]), die Berechnung der schnellen Fourier-Transformation (*Fast Fourier Transform*, kurz FFT) oder die Bestimmung der kürzesten Route für die Pkw-Navigation.

■ $O(n^k)$ (Polynomielles Wachstum)

Die asymptotische Entwicklung der Laufzeit bzw. des Platzbedarfs wird in diesen Algorithmen durch ein Polynom begrenzt. Ein bekanntes Beispiel für einen polynomiellen Laufzeitalgorithmus ist die Multiplikation zweier $n \times n$ -Matrizen. Auch das schon häufiger erwähnte PRIME-Problem lässt sich in Polynomialzeit lösen und fällt damit in diese Algorithmenklasse.

■ $O(k^n)$ (Exponentielles Wachstum)

Die Laufzeit bzw. der Platzbedarf dieser Algorithmen nimmt exponentiell mit der Eingabelänge n zu und wächst damit stärker als jedes Polynom. Die enorme Wachstumsgeschwindigkeit der Expo-

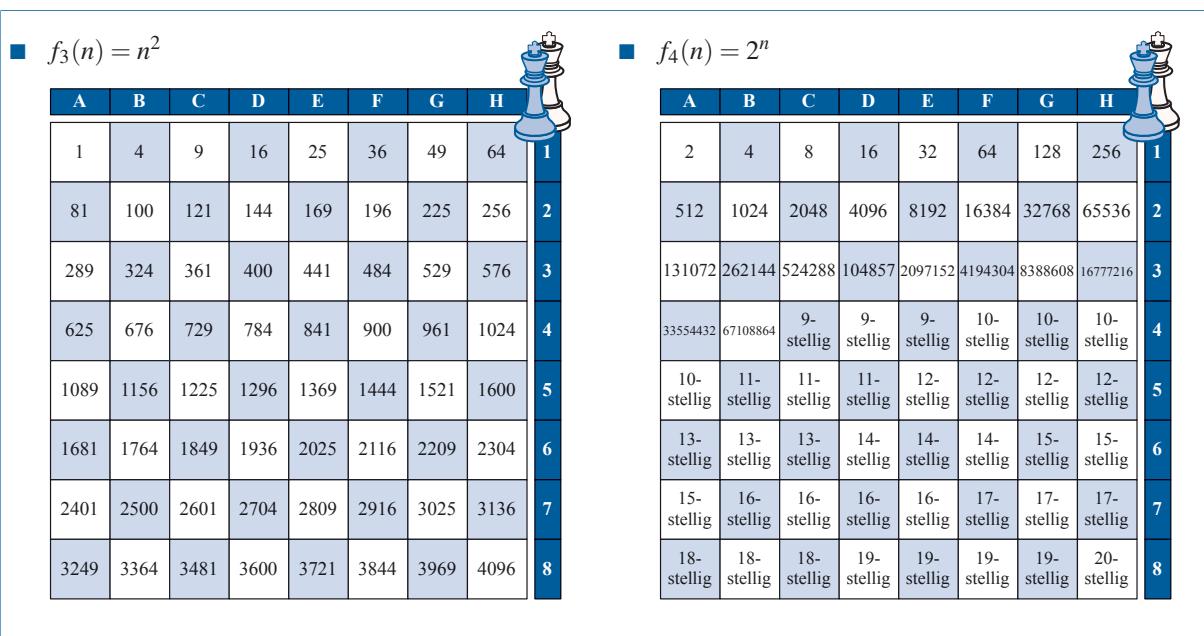


Abbildung 7.9: Die Wachstumsfunktionen n^2 und 2^n im Vergleich

nentialfunktion schränkt den praktische Nutzen dieser Algorithmen drastisch ein; oft scheinen Algorithmen aus dieser Klasse schon für kleine Eingabegrößen augenscheinlich nicht mehr zu terminieren.

Wir wollen unser Augenmerk auf zwei Besonderheiten der aufgelisteten Wachstumsfunktionen richten. Die erste Beobachtung betrifft das linear-logarithmische Wachstum. Da die Laufzeit dieser Algorithmen überproportional mit der Eingabegröße n anwächst, scheint deren praktische Verwertbarkeit auf den ersten Blick fraglich zu sein. Bei genauerer Betrachtung stellt sich diese Wachstumsklasse jedoch als erstaunlich harmlos heraus – selbst große Eingaben lassen sich in der Praxis sehr effizient verarbeiten. Um Ihnen ein Gefühl für das Wachstumsverhalten zu vermitteln, ist in Abbildung 7.8 das linear-logarithmische Wachstum dem linearen Wachstum gegenübergestellt. Dass die überproportionale Zunahme so moderat ausfällt, verdanken wir der außerordentlich geringen Geschwindigkeit, mit der die Logarithmus-Funktion gegen unendlich strebt.

Ein weiteres erstaunliches Ergebnis offenbart Abbildung 7.9. Die Ge- genüberstellung der beiden Komplexitätsklassen $O(n^2)$ und $O(2^n)$

zeigt, wie stark sich das polynomielle und das exponentielle Wachstum wirklich unterscheiden. Die Exponentialfunktion steigt so schnell an, dass der genaue Funktionswert bereits für vergleichsweise kleine n nicht mehr exakt in die Felder eingetragen werden kann.

Die exponentielle Wachstumsgeschwindigkeit unterscheidet sich so eklatant von der polynomiellen Wachstumsgeschwindigkeit, dass die meisten theoretischen Informatiker hier die Trennlinie zwischen den praktisch lösbar und praktisch unlösbar Problemen ziehen. Grob gesprochen gilt ein Problem P als praktisch unlösbar, wenn alle Algorithmen für P ein exponentielles Laufzeitverhalten aufzeigen (vgl. Abbildung 7.10).

In der theoretischen Informatik ist der Unterschied zwischen den polynomiellen und den exponentiellen Wachstumsklassen von so großer Bedeutung, dass wir ihren Eigenschaften im nächsten Abschnitt noch etwas tiefer auf den Zahn fühlen wollen.

7.2.1 P und NP

Über die Laufzeit eines Algorithmus haben wir bereits viel gesprochen, bisher aber versäumt, eine präzise Definition für diesen Begriff bereitzustellen. Dies wollen wir an dieser Stelle nachholen und das formale Gedankengerüst der Turing-Maschine entsprechend erweitern.



Definition 7.6 (Laufzeit von Turing-Maschinen)

Mit T sei eine akzeptierende Mehrband-Turing-Maschine gegeben. Über die Hilfsmenge A (*Accept*) mit

$$A(\omega) := \{n \mid T \text{ terminiert nach } n \text{ Schritten in } z \in E\}$$

definieren wir die *Laufzeitfunktion* $t_T : \Sigma^* \rightarrow \mathbb{N}$ wie folgt:

$$t_T(\omega) := \begin{cases} \min A(\omega) & \text{falls } A(\omega) \neq \emptyset \\ \perp & \text{falls } A(\omega) = \emptyset \end{cases}$$

Definition 7.6 enthält einige Besonderheiten, die eine nähere Betrachtung verdienen. Zunächst fällt auf, dass der Laufzeitbegriff für akzeptierende Turing-Maschinen formuliert ist. Dies stellt keine Einschränkung im eigentlichen Sinne dar, da wir gezeigt haben, dass zwischen der Entscheidbarkeit einer Sprache, der Berechenbarkeit einer Funktion

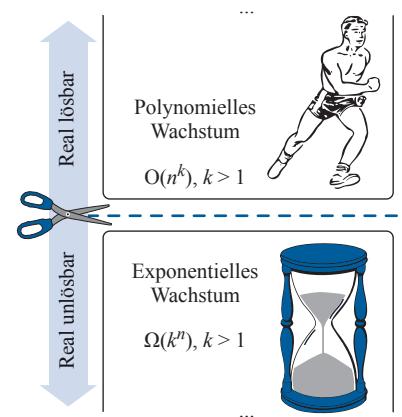


Abbildung 7.10: Der Übergang von polynomiellen zu exponentiellen Wachstumsfunktionen bildet in der theoretischen Informatik die Grenze zwischen real lösbar und real unlösbar Problemen.

Die Ergebnisse dieses Abschnitts haben uns dazu gebracht, polynomiale Probleme als praktisch lösbar und exponentielle Probleme als praktisch unlösbar anzusehen. Wir wollen diese Sichtweise auf den Prüfstand stellen und uns mit der Frage beschäftigen, ob die imaginär gezogene Trennlinie überhaupt existiert oder ob es vielleicht Wachstumsfunktionen gibt, die stärker als jedes Polynom, aber schwächer als jede Exponentialfunktion steigen. In der Tat haben wir in Tabelle 7.4 mit $n^{\log_a n}$ eine solche Funktion bereits kennen gelernt. Da $\log_a n$ mit wachsendem n gegen unendlich strebt, gilt

$$n^{\log_a n} = \omega(n^k)$$

für alle $k \in \mathbb{N}$. Andererseits gilt

$$n^{\log_a n} = a^{\log_a(n^{\log_a n})} = a^{(\log_a n)^2}$$

Die Funktion wächst langsamer als jede Exponentialfunktion der Form k^n und wir erhalten die Beziehung:

$$n^{\log_a n} = o(k^n)$$

Algorithmen mit solchen Komplexitäten sind in der Realität jedoch so selten anzutreffen, dass sie praktisch kaum eine Rolle spielen. Dies ist der Grund, warum wir die Trennlinie des praktisch Lösbarren auch weiterhin zwischen polynomiellem und exponentiell Wachstumsraten ziehen dürfen.

Beachten Sie, dass nicht jedes Problem, für das ein polynomielles Algorithmus existiert, auch wirklich einfach lösbar ist. In der Praxis sind die zu verarbeitenden Eingaben oft so groß, dass bereits diejenigen Algorithmen an ihre Grenze stoßen, die aus theoretischer Sicht „nur“ ein quadratisches Laufzeitverhalten zeigen. Viele Praktiker sehen daher im Übergang von $\Theta(n \log n)$ nach $\Theta(n^2)$ die wahre kritische Grenze, die zwischen real anwendbaren und faktisch nutzlosen Algorithmen unterscheidet.

und der Lösung eines Ja-Nein-Problems kein prinzipieller Unterschied besteht. Alle erzielten Ergebnisse lassen sich in diese Richtung verallgemeinern und wir können mit demselben Begriffsinstrumentarium über die Zeitkomplexität von Problemen oder Funktionen sprechen.

Zudem fällt auf, dass wir mit T explizit eine Turing-Maschine mit mehreren Bändern zugrunde gelegt haben. Die vorgenommene Verallgemeinerung ist der Tatsache geschuldet, dass Rechenschritte, die auf modernen Computeranlagen als elementar angesehen werden können, in Turings Basismodell nur umständlich, d. h. mit einer anderen Laufzeitkomplexität, durchgeführt werden können. Aus dem Originalmodell abgeleitete Laufzeitergebnisse lassen sich hierdurch nur sehr eingeschränkt auf die Realität übertragen. Mehrband-Turing-Maschinen ähneln modernen Computerarchitekturen dagegen in vielerlei Hinsicht, so dass sich die theoretischen Ergebnisse nahezu eins zu eins auf die realen Gegebenheiten übertragen lassen.

Auf der Laufzeitfunktion t_T aufbauend definieren wir die Komplexitätsklasse $\text{TIME}(f(n))$:



Definition 7.7 (Klasse TIME)

Eine Sprache L liegt in $\text{TIME}(f(n))$, falls eine *deterministische* Mehrband-Turing-Maschine T mit den folgenden Eigenschaften existiert:

- $\mathcal{L}(T) = L$
- Für alle Wörter $\omega \in L$ gilt $t_T(\omega) \leq f(|\omega|)$

Während t_T die Laufzeit einer Turing-Maschine für spezielle Wörter beschreibt, stellt Definition 7.7 einen Zusammenhang zwischen der Anzahl der Berechnungsschritte und der Länge der Eingabe her. Damit sind wir in der Lage, sämtliche Wachstumsbetrachtungen aus den vorherigen Abschnitten auf Turing-Maschinen zu übertragen.

Ist Ihnen aufgefallen, dass wir die Beziehung $t_T(\omega) \leq f(|\omega|)$ nur für die Wörter $\omega \in L$ gefordert haben? Dies hat zur Konsequenz, dass die Laufzeitkomplexität ausschließlich durch jene Wörter bestimmt wird, die T akzeptiert. Dass sich die Turing-Maschine für ein Wort $\omega \notin L$ beliebig viel Zeit lassen darf, läuft unserer Vorstellung zuwider, die wir mit dem Begriff der Laufzeit verbinden. Intuitiv erwarten wir, dass die Turing-Maschine für alle Wörter $\omega \in \Sigma^*$ innerhalb der gegebenen Zeitschranke zu einem Ergebnis kommt und nicht nur für jene aus der Sprache L .

Tatsächlich können wir eine solche Maschine auf einfache Weise konstruieren, wenn wir Voraussetzen, dass die betrachteten Laufzeitfunktionen *zeitkonstruierbar* sind. Was dies im Detail bedeutet und welche Konsequenzen sich daraus ergeben, wollen wir jetzt ergründen.



Definition 7.8 (Zeitkonstruierbare Funktion)

Eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ heißt *zeitkonstruierbar*, wenn eine deterministische Mehrband-Turing-Maschine existiert, die unter Eingabe von ω nach genau $f(|\omega|)$ Schritten terminiert.

Sämtliche Laufzeitfunktionen, die in den folgenden Betrachtungen eine Rolle spielen, sind *zeitkonstruierbar*. Dies folgt unmittelbar aus den folgenden Abschlusseigenschaften, die wir ohne Beweis akzeptieren wollen:

- Die Funktionen n und $n \log n$ sind *zeitkonstruierbar*.
- Sei $c \in \mathbb{N}^+$. Ist f *zeitkonstruierbar*, so sind es auch $c \cdot f, f + c$.
- Sind f und g *zeitkonstruierbar*, so sind es auch $f + g, f \cdot g, f^g, f \circ g$.

Sei nun $f(n)$ eine *zeitkonstruierbare Funktion* und L eine Sprache aus $\text{TIME}(f(n))$. Dann existiert per Definition eine Turing-Maschine T mit $\mathcal{L}(T) = L$, die alle Wörter $\omega \in L$ in maximal $f(|\omega|)$ Schritten akzeptiert. Aufgrund der Zeitkonstruierbarkeit von f existiert eine zweite Turing-Maschine, die nach $f(|\omega|)$ Schritten hält. Wie in Abbildung 7.11 skizziert, können wir diese Maschine im Sinne einer Uhr verwenden und mit T zu einem Entscheider für L verschmelzen. Die Funktionsweise ist simpel: Die neue Maschine zählt im Hintergrund die Anzahl der Berechnungsschritte und weist das Eingabewort nach mehr als $f(|\omega|)$ Schritten zurück; ω kann dann nicht zu L gehören.

Auf die gezeigt Weise können wir jede Turing-Maschine, die alle Wörter ω aus der Sprache L in maximal $f(|\omega|)$ Schritten entscheidet, zu einer Turing-Maschine umbauen, die *jedes* Wort $\omega \in \Sigma^*$ in maximal $f(|\omega|)$ Schritten entscheidet. Dies ist der Grund, warum wir die Bedingung $t_T(\omega) \leq f(|\omega|)$ nur für die Wörter $\omega \in L$ fordern mussten.



Satz 7.5

Sei f eine beliebige *zeitkonstruierbare Funktion*. Dann ist jede Sprache $L \in \text{TIME}(f(n))$ entscheidbar.

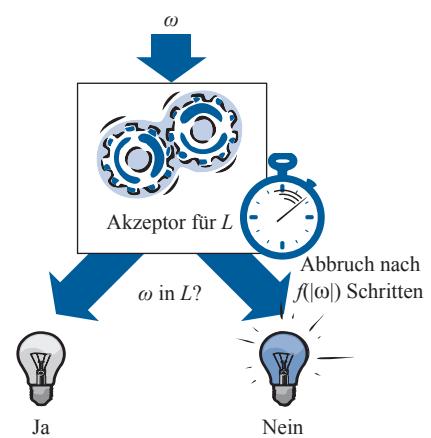


Abbildung 7.11: Ist f eine *zeitkonstruierbare Funktion*, so können wir jede Turing-Maschine, die eine Sprache L aus der Menge $\text{TIME}(f(n))$ akzeptiert, zu einer Turing-Maschine umbauen, die L entscheidet. Im Kern der Konstruktion steht die Idee, die Maschine um eine Uhr anzureichern. Diese wird durch eine parallel simulierte Turing-Maschine implementiert, die alle ausgeführten Berechnungsschritte mitzählt und das Eingabewort nach mehr als $f(|\omega|)$ Berechnungsschritten verwirft. Die akzeptierte Sprache wird hierdurch nicht verändert, da ein Wort ω , das nach $f(|\omega|)$ Schritten noch nicht akzeptiert wurde, nicht zu L gehören kann.

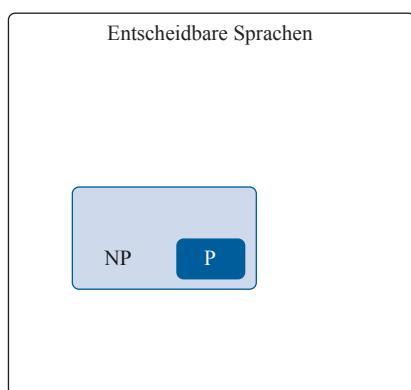


Abbildung 7.12: Inklusionsbeziehungen der bisher eingeführten Komplexitätsklassen

In der Definition von $\text{TIME}(f(n))$ wird explizit gefordert, dass die zugrunde liegende Turing-Maschine deterministisch ist. Heben wir diese Beschränkung auf, so gelangen wir ohne Umwege zur nichtdeterministischen Komplexitätsklasse $\text{NTIME}(f(n))$:



Definition 7.9 (Klasse NTIME)

Eine Sprache L liegt in $\text{NTIME}(f(n))$, falls eine *nichtdeterministische* Mehrband-Turing-Maschine T mit den folgenden Eigenschaften existiert:

- $\mathcal{L}(T) = L$
- Für alle Wörter $\omega \in L$ gilt $t_T(\omega) \leq f(|\omega|)$

Da jede deterministische Turing-Maschine im formalen Sinne auch nichtdeterministisch ist, folgt sofort die Inklusionsbeziehung

$$\text{TIME}(f(n)) \subseteq \text{NTIME}(f(n)) \quad (7.4)$$

Auf Basis von TIME und NTIME lassen sich weitere Komplexitätsklassen ableiten, indem wir $f(n)$ durch konkrete Funktionen ersetzen. Von besonderem Interesse sind diejenigen Sprachen L , die von Turing-Maschinen in polynomieller Laufzeit akzeptiert werden:



Definition 7.10 (P, NP)

Die polynomiellen Komplexitätsklassen P und NP sind definiert als

$$P := \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k)$$

$$NP := \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$$

Aus Gleichung (7.4) folgt unmittelbar die Inklusionsbeziehung

$$P \subseteq NP \quad (7.5)$$

Landläufig wird P auch als die Klasse der *effizient entscheidbaren Sprachen*, der *effizient berechenbaren Funktionen* oder der *effizient lösbarer Probleme* bezeichnet. Abbildung 7.12 stellt die Inklusionsbeziehungen der bisher eingeführten Komplexitätsklassen grafisch gegenüber.

Auch wenn der Unterschied zwischen den Klassen P und NP auf den ersten Blick unscheinbar wirkt, könnten seine Auswirkungen kaum größer sein. So existieren viele praktische Problemstellungen, die mithilfe nichtdeterministischer Algorithmen sehr einfach gelöst werden können, sich einer effizienten deterministischen Lösung aber vehement zu entziehen scheinen.

Abbildung 7.13 demonstriert das Gesagte am Beispiel des Problems SAT, dem wir bereits im Einführungskapitel in Abschnitt 1.2.5 begegnet sind. Hinter SAT verbirgt sich die Frage, ob für eine gegebene aussagenlogische Formel F eine erfüllende Belegung der Variablen existiert oder nicht. Nichtdeterministisch ist das Problem auf einfache Weise lösbar, indem wir zunächst die richtige Kombination von Wahrheitswerten erraten und anschließend zeigen, dass F für diese Belegung tatsächlich den Wahrheitswert 1 annimmt. Der Algorithmus ist nicht nur einfach, sondern auch effizient. Bezeichnet n die Länge der Formel F , so müssen wir für maximal n Variablen die Belegung erraten und F anschließend auswerten. Beide Schritte lassen sich mit linearem Zeitaufwand erledigen.

Natürlich sind wir auch in der Lage, einen Algorithmus zu formulieren, der das SAT-Problem deterministisch löst. Wie in Abbildung 7.13 (unten) gezeigt, schreiben wir hierzu ein Programm, das sämtliche Wahrheitswertkombinationen erzeugt und F jedes Mal aufs Neue auf Erfüllbarkeit prüft. Leider müssen wir für einen Ausdruck mit n Variablen 2^n verschiedene Kombinationen in Betracht ziehen, so dass die Laufzeit unseres Algorithmus exponentiell zunimmt. In der Praxis werden wir mit dieser Methode nur für sehr kleine Ausdrücke in der Lage sein, das Erfüllbarkeitsproblem zu entscheiden.

Die Art und Weise, wie wir das SAT-Problem nichtdeterministisch gelöst haben, lässt sich für viele weitere Problemstellungen verallgemeinern. In Frage kommen alle Probleme, die über einen großen, meist exponentiell wachsenden Suchraum verfügen und zudem die Eigenschaft besitzen, dass sich eine gefundene Lösung mit geringem Aufwand als solche verifizieren lässt. Deterministischen Algorithmen bleibt in der Regel nichts anderes üblich, als den Suchraum systematisch zu durchkämmen (*Brute-Force-Methode*). Wächst dieser exponentiell mit der Eingabegröße an, so erhalten wir einen Algorithmus, der in der Praxis nur für sehr kleine Eingaben ein akzeptables Laufzeitverhalten zeigt.

Das im Jahre 1959 von Dana Scott und Michael Rabin eingeführte Konzept der nichtdeterministischen Berechnung überwindet dieses Problem. Wie in Abschnitt 6.4 ausführlich dargelegt, akzeptiert eine nichtdeterministische Turing-Maschine ein Wort genau dann, wenn mindes-

■ Nichtdeterministische Lösung

solveSAT.ndet

```

1 // guess
2 x1 := 0 or 1
3 ...
4 xn := 0 or 1
5
6 // verify
7 if (F(x1,...,xn) ≡ 1)
8   return true;
9 else
10  return false;
11
12

```

■ Deterministische Lösung

solveSAT.det

```

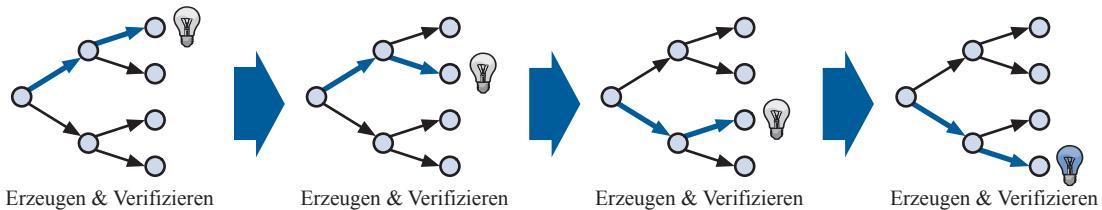
1 // brute force attack
2 forall x1 in {0,1} {
3   ...
4     forall xn in {0,1} {
5       if (F(x1,...,xn) ≡ 1)
6         return true;
7     }
8   ...
9 }
10 return false;
11
12

```

Abbildung 7.13: Nichtdeterministische und deterministische Implementierung zur Lösung des SAT-Problems

Deterministische Lösungsstrategie

Der Suchraum wird systematisch durchkämmt (Brute-Force-Methode).



Nichtdeterministische Lösungsstrategie

Der richtige Berechnungspfad wird geraten und die Lösung anschließend verifiziert.



Abbildung 7.14: Deterministische und nondeterministische Lösungsstrategien im Vergleich

tens eine Berechnungsfolge existiert, die in einem akzeptierenden Zustand endet. Die Definition erlaubt ganz bewusst, dass weitere Pfade existieren, die zu einer unendlichen Berechnungsfolge führen oder in einem nichtakzeptierenden Zustand enden. Aus algorithmischer Sicht entspricht dieses Konzept einem Programm nach dem oben diskutierten Schema. Die potenzielle Lösung wird erraten und am Ende nur noch auf Gültigkeit überprüft. Abbildung 7.14 stellt die deterministische Lösungsstrategie der nondeterministischen nach Scott und Rabin grafisch gegenüber.

Wie nutzbringend die nondeterministische Berechnung ist, verdeutlicht auch das in Abschnitt 1.2.5 diskutierte Hamilton-Problem. Für einen gegebenen Graphen G galt es zu entscheiden, ob ein Rundweg existiert, der jeden Knoten exakt einmal passiert. Auch hier steht ein exponentieller Suchraum der Eigenschaft gegenüber, dass sich für einen gegebenen Pfad mit wenig Aufwand entscheiden lässt, ob er das Hamilton-Problem löst oder nicht. Folgerichtig können wir ein nondeterministisches Lösungsverfahren konstruieren, das im ersten Schritt

einen geschlossenen Linienzug errät und anschließend die Hamilton-Eigenschaft überprüft. Abbildung 7.15 demonstriert die Lösungsstrategie anhand eines konkreten Beispiels.

7.2.2 PSPACE und NPSPACE

Die Klassen P und NP haben wir über die Laufzeitkomplexität von Turing-Maschinen definiert. Der Bandplatz, den eine Maschine während einer Berechnung belegt, wurde bisher vollständig außen vor gelassen. Hierfür existieren mit PSPACE und NPSPACE eigene Komplexitätsklassen, die der gleichen Kernidee folgen. Bevor wir die Klassen formal definieren, wollen wir zunächst festlegen, was wir genau unter dem Platzbedarf einer (nichtdeterministischen) Turing-Maschine verstehen wollen.



Definition 7.11 (Platzbedarf von Turing-Maschinen)

Mit T sei eine akzeptierende Mehrband-Turing-Maschine gegeben.
 Über die Hilfsmenge A (*Accept*) mit

$$A(\omega) := \{n \mid T \text{ benutzt } n \text{ Zellen und terminiert in } z \in E\}$$

definieren wir die *Bandplatzfunktion* $s_T : \Sigma^* \rightarrow \mathbb{N}$ wie folgt:

$$s_T(\omega) := \begin{cases} \min A(\omega) & \text{falls } A(\omega) \neq \emptyset \\ \perp & \text{falls } A(\omega) = \emptyset \end{cases}$$

In der gleichen Weise, wie wir auf der Laufzeitfunktion t_T die Laufzeitklassen TIME und NTIME definiert haben, bauen wir auf der Bandplatzfunktion s_T die Platzklassen SPACE und NSPACE auf.

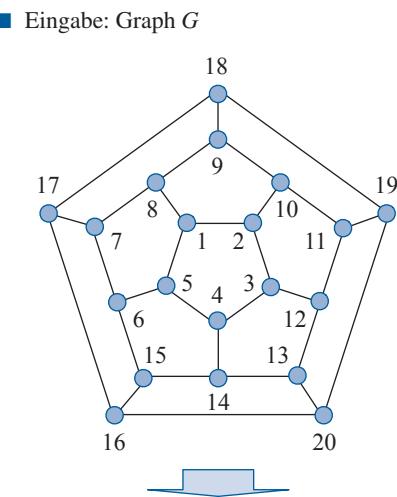


Definition 7.12 (SPACE und NSPACE)

Eine Sprache L liegt in $\text{SPACE}(f(n))$, falls eine *deterministische* Mehrband-Turing-Maschine T mit den folgenden Eigenschaften existiert:

- $\mathcal{L}(T) = L$
 - Für alle Wörter $\omega \in L$ gilt $s_T(\omega) \leq f(|\omega|)$

L liegt in $\text{NSPACE}(f(n))$, wenn eine *nichtdeterministische* Mehrband-Turing-Maschine mit den genannten Eigenschaften existiert.



- ## ■ Raten

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 14 \rightarrow$
 $13 \rightarrow 12 \rightarrow 11 \rightarrow 10 \rightarrow 9 \rightarrow$
 $8 \rightarrow 7 \rightarrow 17 \rightarrow 18 \rightarrow 19 \rightarrow$
 $20 \rightarrow 16 \rightarrow 15 \rightarrow 6 \rightarrow 5 \rightarrow 1$



- ## ■ Verifizieren

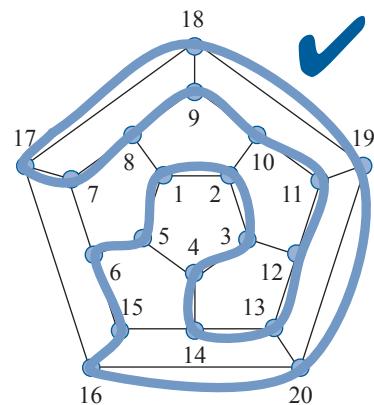


Abbildung 7.15: Nichtdeterministisch kann das Hamilton-Problem auf einfache Weise gelöst werden.

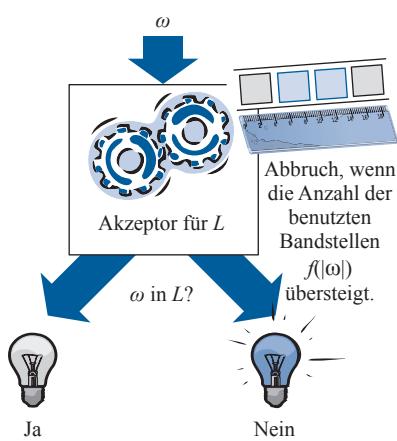


Abbildung 7.16: Ist f eine platzkonstruierbare Funktion, so können wir jede Turing-Maschine, die eine Sprache L aus der Menge $\text{SPACE}(f(n))$ akzeptiert, zu einer Turing-Maschine umbauen, die L entscheidet. Im Kern der Konstruktion steht die Idee, die Maschine um einen Wächter anzureichern, der den Bandplatz überwacht und das Eingabewort verwirft, wenn die Anzahl der benutzten Bandstellen den Wert $f(|\omega|)$ übersteigt.

Genau wie die Laufzeitkomplexität wird auch die Platzkomplexität ausschließlich durch die Wörter $\omega \in L$ bestimmt. Dass wir hierbei keinen Fehler machen, hat den gleichen Grund wie im Fall der Laufzeit. Können wir die Sprachzugehörigkeit $\omega \in L$ mit $f(|\omega|)$ Bandstellen entscheiden, so gelingt dies auch für die Wörter $\omega \notin L$. Wir müssen jetzt lediglich anstelle einer Uhr eine Wächtermaschine mitlaufen lassen, die den verwendeten Bandplatz überwacht und das Eingabewort ω zurückweist, falls mehr als $f(|\omega|)$ Bandstellen beschrieben wurden. Ein solcher Wächter lässt sich immer dann konstruieren, wenn die Funktion $f(n)$ *platzkonstruierbar* ist (Abbildung 7.16).

Definition 7.13 (Platzkonstruierbare Funktion)

Eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ heißt *platzkonstruierbar*, wenn eine deterministische Mehrband-Turing-Maschine existiert, die unter Eingabe von ω genau $f(|\omega|)$ Bandstellen verändert und danach terminiert.

Wie schon im Fall der zeitkonstruierbaren Funktionen sind nahezu alle in der Praxis angetroffenen Funktionen auch platzkonstruierbar. Tatsächlich lässt sich zeigen, dass jede zeitkonstruierbare Funktion immer auch platzkonstruierbar ist. Die Umkehrung dieser Aussage gilt jedoch nicht.

Jetzt sind alle Voraussetzungen geschaffen, um die Platzklassen PSPACE und NPSPACE einzuführen:

Definition 7.14 (PSPACE, NPSPACE)

Die polynomiellen Komplexitätsklassen PSPACE und NPSPACE sind definiert als

$$\text{PSPACE} := \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k)$$

$$\text{NPSPACE} := \bigcup_{k \in \mathbb{N}} \text{NSPACE}(n^k)$$

Obwohl die Platzklassen PSPACE und NPSPACE einen völlig anderen Aspekt als die Zeitklassen P und NP betrachten, lässt sich zwischen beiden eine Verbindung herstellen. Nimmt der Speicherplatzbedarf eines Algorithmus z. B. quadratisch mit der Eingabelänge n zu, so werden zum Beschreiben des Speichers mindestens n^2 Rechenschritte benötigt.

Kurzum: Die Zeitkomplexität eines Algorithmus kann niemals kleiner sein als seine Platzkomplexität.

Ein anderes wichtiges Resultat bewies der US-amerikanische Computerwissenschaftler Walter Savitch im Jahre 1970. Er konnte zeigen, dass jede von einer nichtdeterministischen Turing-Maschine akzeptierte Sprache mit einer deterministischen Turing-Maschine akzeptiert werden kann, deren Platzbedarf nur quadratisch höher liegt [87]. Folgerichtig geht die Eigenschaft des polynomiellen Platzbedarfs durch die Übersetzung in eine deterministische Maschine nicht verloren und wir erhalten auf einen Schlag die Beziehung $\text{PSPACE} = \text{NPSPACE}$. Zusammengefasst ergibt sich die folgende Inklusionskette (vgl. Abbildung 7.17):

Satz 7.6 (Klassenhierarchie)

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE}$$

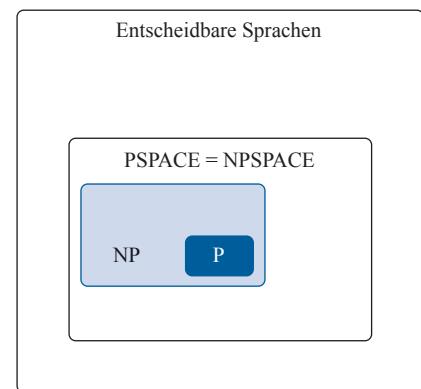


Abbildung 7.17: Inklusionsbeziehungen der bisher eingeführten Komplexitätsklassen

Ob zwischen den Klassen P und NP eine echte Inklusionsbeziehung besteht, ist gegenwärtig ungeklärt. In Abschnitt 7.3 werden wir uns näher damit beschäftigen, wie beide Klassen zusammenhängen.

7.2.3 EXP und NEXP

In Analogie zu den polynomiellen Komplexitätsklassen P und NP definieren wir die exponentiellen Komplexitätsklassen EXP und NEXP:

Definition 7.15 (EXP, NEXP)

Die exponentiellen Komplexitätsklassen EXP und NEXP sind wie folgt definiert:

$$\text{EXP} := \bigcup_{c,k \in \mathbb{N}} \text{TIME}(c^{n^k})$$

$$\text{NEXP} := \bigcup_{c,k \in \mathbb{N}} \text{NTIME}(c^{n^k})$$

In den Klassen EXP und NEXP sind somit alle Sprachen enthalten, die in höchstens exponentieller Laufzeit durch eine deterministische bzw. eine nichtdeterministische Turing-Maschine entschieden werden

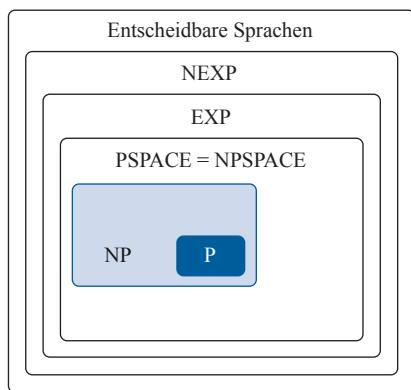


Abbildung 7.18: Inklusionsbeziehungen der bisher eingeführten Komplexitätsklassen

können. Offensichtlich gilt zwischen EXP und seiner nichtdeterministischen Variante NEXP die folgende Inklusionsbeziehung:

$$\text{EXP} \subseteq \text{NEXP} \quad (7.6)$$

Wir wollen nun versuchen, die Klassen EXP und NEXP in Bezug zu den bisher eingeführten Klassen zu stellen. Zunächst halten wir fest, dass Wachstumsraten existieren, die jenseits des Exponentiellen liegen. So haben wir in Tabelle 7.4 mit $O(2^{2^n})$ bereits eine Funktion kennen gelernt, die stärker wächst als die in Definition 7.15 gewählten Exponentialfunktionen. Hieraus folgt, dass die Klassen EXP und NEXP echte Teilmengen der entscheidbaren Sprachen sind.

Weniger offensichtlich ist die Beziehung, die zwischen der Speicherplatzklasse PSPACE und der Laufzeitklasse EXP besteht. Um diese Beobachtung zu vertiefen, nehmen wir an, T sei eine Turing-Maschine, deren Platzverbrauch nur polynomiell mit der Größe der Eingabe ω wächst. Da wir die Anzahl der belegten Bandzellen von T durch ein Polynom $p(|\omega|)$ nach oben abschätzen können, gelingt uns gleichzeitig eine quantitative Aussage über die Konfigurationen, in denen sich T befinden kann. Da eine Konfiguration durch den aktuellen Zustand ($|S|$ Möglichkeiten), die Kopfposition ($p(|\omega|)$ Möglichkeiten) und den Bandinhalt ($|\Pi|^{p(|\omega|)}$ Möglichkeiten) eindeutig bestimmt ist, können maximal

$$p(|\omega|) \cdot |S| \cdot |\Pi|^{p(|\omega|)} \quad (7.7)$$

verschiedene Konfigurationen existieren. Da die Turing-Maschine deterministisch arbeitet, kann sie nur terminieren, wenn sie niemals dieselbe Konfiguration zweimal einnimmt. Damit ist der Ausdruck (7.7) gleichermaßen eine Abschätzung für die Anzahl der Konfigurationsübergänge. Da sich (7.7) selbst durch einen Term der Form c^{n^k} abschätzen lässt, ist die Laufzeit von T exponentiell nach oben begrenzt. Mit anderen Worten: PSPACE liegt in EXP. In Kombination mit der Inklusionsbeziehung (7.6) erhalten wir das folgende Ergebnis:

Satz 7.7

$$\text{PSPACE} \subseteq \text{EXP} \subseteq \text{NEXP}$$

Abbildung 7.18 fasst die bisher herausgearbeiteten Ergebnisse grafisch zusammen.

7.2.4 Komplementäre Komplexitätsklassen

Zu jeder Komplexitätsklasse C existiert eine komplementäre Komplexitätsklasse, die wir als $\text{co}C$ bezeichnen.



Definition 7.16 (Komplementäre Komplexitätsklasse)

Mit C sei eine beliebige Komplexitätsklasse gegeben. Die Menge

$$\text{co}C := \{L \mid \bar{L} \in C\}$$

ist die *komplementäre Komplexitätsklasse* von C .

Eine Sprache L gehört demnach genau dann zur Klasse $\text{co}C$, wenn ihr Komplement \bar{L} zu C gehört.

Im Folgenden wollen wir die Eigenschaften der Komplementärklasse genauer untersuchen. Hierzu nehmen wir zunächst an, dass mit C eine deterministische Komplexitätsklasse gegeben ist. In diesem Fall existiert für jede Sprache L eine deterministische Turing-Maschine T , die für alle Eingaben ω nach endlich vielen Berechnungsschritten terminiert. Ist $\omega \in L$, so liefert T die Antwort „ja“, ansonsten „nein“. Invertieren wir die Menge der Endzustände, so erhalten wir eine Maschine T' , die das Komplement \bar{L} entscheidet. Da sich das Laufzeitverhalten der Maschine nicht verändert hat, liegt T' in der gleichen Komplexitätsklasse wie T . Aus $L \in C$ folgt damit stets die Beziehung $\bar{L} \in C$, so dass die Mengen $\text{co}C$ und C exakt die gleichen Elemente enthalten müssen. Damit haben wir einen Beweis für den folgenden Satz erbracht:



Satz 7.8

Für jede *deterministische* Komplexitätsklasse C gilt $C = \text{co}C$.

Ersetzen wir C in Satz 7.8 durch die deterministischen Komplexitätsklassen P oder PSPACE, so erhalten wir das folgenden Ergebnis:

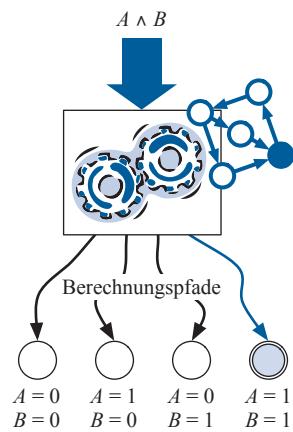


Korollar 7.1

Für die Komplexitätsklassen P und PSPACE gilt:

$$P = \text{co}P \text{ und } \text{PSPACE} = \text{coPSPACE}$$

Turing-Maschine T



T nach der Zustandsinvertierung

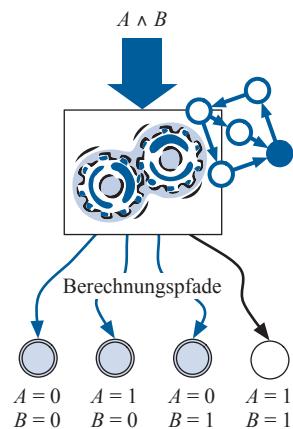


Abbildung 7.19: Vorsicht bei nichtdeterministischen Turing-Maschinen! Im Gegensatz zu deterministischen Maschinen reicht es nicht aus, die Menge der Endzustände zu invertieren, um einen Akzeptor für die Sprache $\bar{\mathcal{L}}(T)$ zu erhalten.

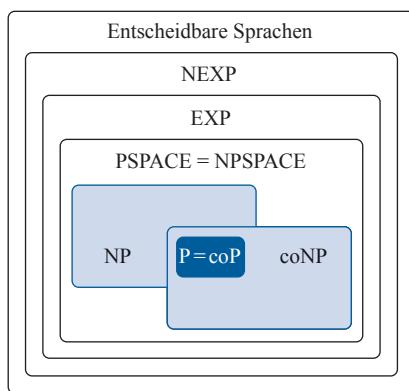


Abbildung 7.20: Inklusionsbeziehungen der bisher eingeführten Komplexitätsklassen

Auf nichtdeterministische Komplexitätsklassen lassen sich die getätigten Überlegungen nicht übertragen. Invertieren wir die Menge der Endzustände einer nichtdeterministischen Turing-Maschine, so erhalten wir – anders als im deterministischen Fall – keinen Akzeptor für die Komplementsprache. Der Grund hierfür liegt in der Definition des Nichtdeterminismus verborgen. Um ein Wort zu akzeptieren, reicht es aus, dass ein einziger Berechnungspfad in einen Endzustand führt; andere Pfade dürfen dagegen in beliebigen Zuständen terminieren.

Abbildung 7.19 demonstriert das Problem am Beispiel einer nichtdeterministischen Turing-Maschine zur Lösung von SAT. Für die Eingabe $A \wedge B$ ergeben sich vier verschiedene Berechnungspfade, die den vier verschiedenen Variablenkombinationen von A und B entsprechen. Die Formel $A \wedge B$ wird als erfüllbar akzeptiert, da ein Berechnungspfad existiert, der in einem Endzustand terminiert. Dieser Pfad entspricht der Kombination $A = B = 1$. Invertieren wir alle Endzustände, so entstehen drei Berechnungspfade, die erneut in einem Endzustand terminieren. Folgerichtig wird die Formel $A \wedge B$ weiterhin akzeptiert.

Beachten Sie, dass es genauso falsch wäre, aus den angestellten Überlegungen die Beziehung $NP \neq coNP$ zu folgern. Wir haben lediglich gezeigt, dass wir ein nichtdeterministisches Entscheidungsverfahren für eine Sprache L – anders als im deterministischen Fall – nicht durch die simple Invertierung der Endzustände zu einem Entscheidungsverfahren für die Komplementmenge \bar{L} umbauen können. Obwohl es als unwahrscheinlich gilt, ist es nicht ausgeschlossen, dass ein solcher Umbau auf anderem Wege funktioniert. Die Frage, ob die Gleichung $NP = coNP$ gilt oder nicht, ist bis heute ungeklärt und gehört zu den wichtigsten ungelösten Problemen der modernen theoretischen Informatik.

Ein interessantes Ergebnis erhalten wir, wenn wir die Klasse P mit der Klasse coNP in Bezug setzen. Hierzu sei L ein Element aus P. Dann ist \bar{L} per Definition ein Element von coP. Da nach Satz 7.8 die Beziehung $\bar{L} \in P$ gilt, folgt aus der Inklusionseigenschaft $P \subseteq NP$ sofort $\bar{L} \in NP$. Damit ist $\bar{L} \in coNP$. \bar{L} ist aber nichts anderes als das Element L selbst, so dass wir effektiv die Inklusion $P \subseteq coNP$ gezeigt haben (Abbildung 7.20). Zusammen mit der Beziehung $P \subseteq NP$ erhalten wir den folgenden Satz:

Satz 7.9

Es gilt die Inklusionsbeziehung $P \subseteq (NP \cap coNP)$

7.3 NP-Vollständigkeit

7.3.1 Polynomielle Reduktion

Erinnern Sie sich noch an das Prinzip der *Problemreduktion*, mit dem wir in Kapitel 6 unter anderem die Unentscheidbarkeit des Post'schen Korrespondenzproblems zeigen konnten? Die Grundidee war verblüffend einfach: Wir hatten ein Problem L auf ein Problem L' reduziert, indem wir zeigten, dass mit einer Lösungsstrategie für L' auch L gelöst werden kann. War L ein bekanntermaßen unentscheidbares Problem, so folgte unmittelbar, dass auch L' unentscheidbar sein musste.

Auch im Bereich der Komplexitätstheorie lässt sich das Reduktionsprinzip gewinnbringend einsetzen. Hier wird es verwendet, um Komplexitätsaussagen einer Problemklasse an andere zu vererben. Den Schlüssel hierzu bildet der Begriff der *polynomiellen Reduzierbarkeit*, den wir an dieser Stelle formal einführen wollen:



Definition 7.17 (Polynomielle Reduzierbarkeit)

Mit $L \subseteq \Sigma^*$ und $L' \subseteq \Gamma^*$ seien zwei Sprachen gegeben. L ist genau dann auf L' polynomiell reduzierbar, geschrieben als $L \leq_{poly} L'$, falls eine totale Funktion $f : \Sigma^* \rightarrow \Gamma^*$ mit den folgenden Eigenschaften existiert:

- f ist in polynomieller Zeit berechenbar
- $\omega \in L \Leftrightarrow f(\omega) \in L'$

Wissen wir, dass eine Sprache L' in polynomieller Zeit entscheidbar ist, dann gilt diese Eigenschaft auch für jede Sprache L mit $L \leq_{poly} L'$. Wie in Abbildung 7.21 gezeigt, wird die Frage $\omega \in L$ zunächst auf die äquivalente Frage $\omega' \in L'$ abgebildet und erst dann entschieden. Beide Schritte weisen eine polynomielle Laufzeit von $O(n^{k_1})$ bzw. $O(n^{k_2})$ auf, so dass wir die ursprüngliche Frage $\omega \in L$ mit der Komplexität

$$O(n^{k_1} + n^{k_2}) = O(n^{\max\{k_1, k_2\}})$$

und damit ebenfalls in polynomieller Zeit beantworten können.

Im Folgenden werden wir nicht nur von polynomiell entscheidbaren Sprachen, sondern auch vermehrt von *polynomiell lösbar* Problemen reden. Die Redensart ist legitim, da wir bereits mehrfach festgestellt haben, dass zwischen der Entscheidbarkeit einer Sprache und der Lösung eines Ja-Nein-Problems kein prinzipieller Unterschied besteht.

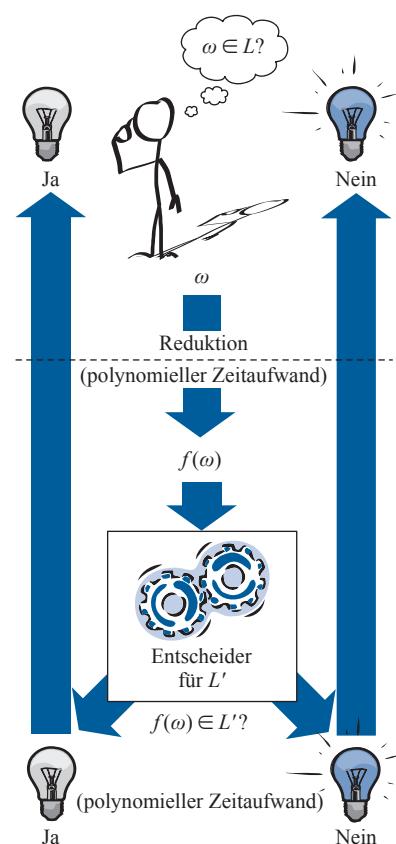
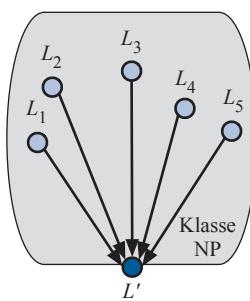
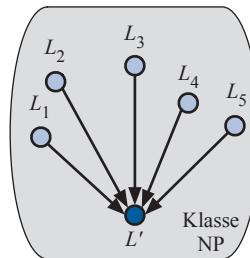


Abbildung 7.21: Prinzip der polynomiellen Reduzierbarkeit



Ein Problem L' ist *NP-hart*, wenn sich alle Probleme aus der Klasse NP darauf reduzieren lassen. L' kann selbst in NP liegen, muss es aber nicht.



Ein Problem L' ist *NP-vollständig*, wenn sich alle Probleme aus der Klasse NP darauf reduzieren lassen und L' selbst in NP liegt.

Abbildung 7.22: Zusammenhang zwischen NP-harten und NP-vollständigen Problemen

7.3.2 P-NP-Problem

Mithilfe der polynomiellen Reduzierbarkeit sind wir in der Lage, zwei prominente Problemklassen der Komplexitätstheorie zu definieren:



Definition 7.18 (NP-hart, NP-vollständig)

Eine Sprache $L' \subseteq \Sigma^*$ heißt

- *NP-hart*, falls für alle $L \in \text{NP}$ die Beziehung $L \leq_{\text{poly}} L'$ gilt.
- *NP-vollständig*, falls L' NP-hart und ein Element von NP ist.

NPC bezeichnet die Klasse aller NP-vollständigen Probleme.

Wörtlich gesprochen ist ein Problem genau dann NP-hart, wenn sich alle nichtdeterministisch polynomiell lösbar Probleme darauf reduzieren lassen (vgl. Abbildung 7.22 oben). Ist es darüber hinaus selbst nichtdeterministisch polynomiell lösbar, so sprechen wir von einem NP-vollständigen Problem (vgl. Abbildung 7.22 unten). Abbildung 7.23 zeigt, wie sich die Komplexitätsklasse NPC in die bisher herausgearbeitete Klassenhierarchie einfügt.

Ein Beweis der NP-Vollständigkeit besteht immer aus zwei Teilen:

- Es ist zu zeigen, dass das untersuchte Problem in der Klasse NP liegt. Dies kann durch die Angabe eines nichtdeterministischen Algorithmus erfolgen, der zunächst eine Antwort rät und anschließend in polynomieller Zeit überprüft, ob wir eine Lösung vor uns haben.
- Es ist zu zeigen, dass sich alle Probleme aus der Klasse NP auf das untersuchte Problem reduzieren lassen (vgl. Abbildung 7.24). Dies kann durch die polynomielle Reduktion eines beliebigen anderen Problems aus der Klasse NPC geschehen.

NP-vollständige Probleme sind aus dem folgenden Grund von Interesse: Gelingt es uns, für ein einziges dieser Probleme zu zeigen, dass es *deterministisch mit polynomiellem Aufwand* gelöst werden kann, so wären auf einen Schlag alle anderen NP-vollständigen Probleme ebenfalls deterministisch polynomiell lösbar. Wir würden damit nichts weniger als die Beziehung $P = NP$ beweisen und hätten das berühmte *P-NP-Problem* positiv beantwortet. Auf einen Schlag hätten wir einer Vielzahl von Problem ihren Schrecken genommen, die sich gegenwärtig einer effizienten Lösung entziehen.

Wenngleich das P-NP-Problem bis heute ungelöst ist, vermuten die meisten Experten, dass die Mengen P und NP verschieden sind. Ein Grund hierfür ist die große Anzahl an Problemen, die in der Vergangenheit als NP-vollständig identifiziert wurden. Wäre $P = NP$, so ließen sich all diese Probleme in polynomieller Zeit auf realen Computeranlagen lösen. So verlockend diese Möglichkeit auch klingt: Mit jedem neu entdeckten NP-vollständigen Problem scheint sie ein Stück weit unwahrscheinlicher zu werden. Trotzdem ist eine positive Lösung des P-NP-Problems keinesfalls ausgeschlossen und es wäre nicht das erste Mal, dass die mehrheitliche Expertenmeinung im Bereich der theoretischen Informatik widerlegt werden würde.

7.3.3 Satz von Cook

Dass überhaupt NP-harte oder NP-vollständige Probleme existieren, ist keinesfalls selbstverständlich. Halten wir uns an Definition 7.18, so können wir ein Problem L' als NP-hart identifizieren, indem wir ein anderes NP-hartes Problem darauf reduzieren. Damit wir die Reduktionstechnik anwenden können, muss also stets ein anderes Problem mit der gewünschten Eigenschaft existieren. Aus dieser Henne-Ei-Situation gibt es für uns nur einen Ausweg: Wir müssen die NP-Härte für irgend ein Problem direkt beweisen. Ist dieser Schritt gelungen, so lässt sich die Eigenschaft mithilfe der Reduktionstechnik auf weitere Probleme übertragen.

Im Jahre 1971 gelang es dem amerikanischen Computerwissenschaftler Stephen Cook, die NP-Vollständigkeit des SAT-Problems zu zeigen [24]. Weil sein Beweis auf die Reduktionstechnik verzichtet, kommt er ohne die Annahme aus, dass andere NP-vollständige Probleme existieren. Der Satz von Cook ist der Grundstein, auf dem alle anderen Vollständigkeitsbeweise aufbauen.



Satz 7.10 (Satz von Cook)

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) ist NP-vollständig.

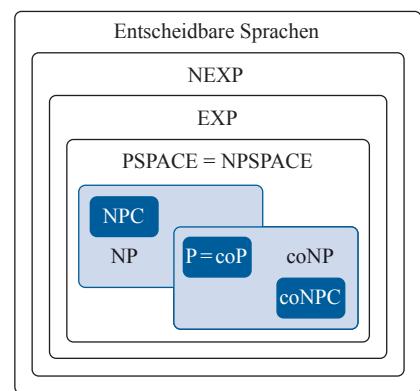
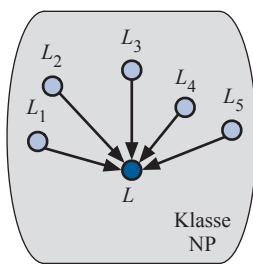
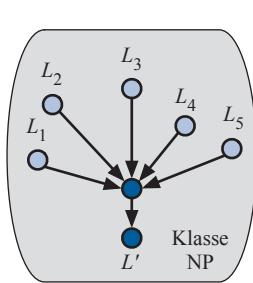


Abbildung 7.23: Inklusionsbeziehungen der eingeführten Komplexitätsklassen

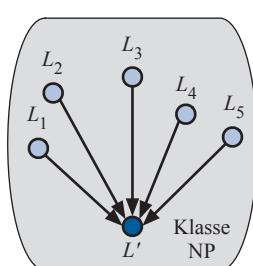
Der Vollständigkeitsbeweis besteht aus zwei Teilen. Zum einen müssen wir zeigen, dass SAT in NP liegt, und zum anderen, dass SAT ein NP-hartes Problem ist. Den ersten Teil haben wir in Abschnitt 7.2.1 erledigt. Dort haben wir herausgearbeitet, dass sich SAT nichtdeterministisch in linearer Zeit lösen lässt. Die eigentliche Schwierigkeit im Beweis des



Ist L NP-vollständig, so lassen sich alle Probleme aus NP auf L reduzieren.



Lässt sich L polynomiell auf $L' \in \text{NP}$ reduzieren, so folgt aus der Transitivität, ...



... dass sich alle Probleme aus NP auch auf L' reduzieren lassen. L' ist damit ebenfalls NP-vollständig.

Abbildung 7.24: Die NP-Vollständigkeit eines Problems lässt sich durch die polynomielle Reduktion eines anderen NP-vollständigen Problems beweisen.

Satzes von Cook besteht im Nachweis der NP-Härte, schließlich müssen wir uns davon überzeugen, dass sich ausnahmslos alle Probleme der Klasse NP polynomiell auf SAT reduzieren lassen.

In den folgenden Betrachtungen bezeichnet L ein beliebiges Problem aus NP. Per Definition existiert eine nichtdeterministische Turing-Maschine T , die L in polynomieller Zeit entscheidet. Wir werden nun zeigen, dass wir für jedes Eingabewort ω eine Formel F mit der nachstehenden Eigenschaft konstruieren können:

$$T(\omega) \text{ terminiert in einem Endzustand} \Leftrightarrow F(\omega) \text{ ist erfüllbar}$$

Gelingt es uns, F in polynomieller Zeit zu konstruieren, so sind wir am Ziel: Wir haben L polynomiell auf SAT reduziert.

Bevor wir die Struktur der Formel F im Detail diskutieren, führen wir zunächst die Variablen ein, die zu ihrer Konstruktion benötigt werden. Auf der obersten Ebene lassen sich diese in drei Kategorien einteilen:

- $B_{i,k,\sigma}$: Variablen zur Codierung des Bandinhalts
 $B_{i,k,\sigma} = 1 \Leftrightarrow$ Nach i Schritten steht an Position k das Zeichen σ
- $S_{i,s}$: Variablen zur Codierung des aktuellen Zustands
 $S_{i,s} = 1 \Leftrightarrow T$ befindet sich nach i Schritten im Zustand s
- $P_{i,k}$: Variablen zur Codierung der aktuellen Kopfposition
 $P_{i,k} = 1 \Leftrightarrow T$ befindet sich nach i Schritten auf Bandposition k

Beachten Sie, dass die Laufzeit der Turing-Maschine T polynomiell beschränkt ist. Ist $n = |\omega|$ die Länge der Eingabe, so wird die Anzahl der Arbeitsschritte von T durch ein Polynom $p(n)$ begrenzt. Auch wenn der exakte Wert von $p(n)$ in unserer Betrachtung keine Rolle spielt, ist die Abschätzung wichtig. Sie besagt, dass die Indizes i, s im Intervall $[0; p(n)]$ liegen müssen. Zudem kann sich der Schreib-Lese-Kopf maximal $p(n)$ Positionen nach links bzw. rechts bewegen, d. h., wir müssen den Bandinhalt ausschließlich auf dem endlichen Teilstück $-p(n), \dots, p(n)$ betrachten. Wie gewohnt gehen wir davon aus, dass der Schreib-Lese-Kopf initial über der Position 0 steht. Insgesamt zeigt die Betrachtung, dass F zwar eine große Menge an Variablen enthält, deren Anzahl aber nur polynomiell mit der Eingabelänge n wächst.

Um nicht im Nebel der Theorie zu versinken, wollen wir das Gesagte am Beispiel der Sprache

$$L := \{\#^n \mid n \in \mathbb{N}^+\}$$

genauer untersuchen. Die Sprache wurde bewusst ausgewählt, da sie durch eine bestechend einfache Turing-Maschine mit nur zwei Zuständen erkannt werden kann (vgl. Abbildung 7.25). Sie besteht aus dem Eingabealphabet $\Sigma = \{\#\}$, dem Bandalphabet $\Pi = \{\#, \square\}$, dem Startzustand s_0 und dem (einzig) Endzustand s_1 . Die Funktionsweise der Turing-Maschine ist äußerst simpel. Vom Startzustand geht sie direkt in den Endzustand über, falls das aktuelle Bandzeichen gleich $\#$ ist; in diesem Fall gilt $\omega \in L$. Wird stattdessen das Blank-Symbol \square gelesen, terminiert die Maschine in s_0 . Da die Berechnung immer nach einem Schritt beendet ist, ist die Laufzeit durch das Polynom $p(n) = 1$ begrenzt. Wie die Auflistung in der unteren Hälfte von Abbildung 7.25 zeigt, werden trotzdem 22 Variablen benötigt, um die Turing-Maschine auf eine SAT-Instanz abzubilden.

Nachdem wir die verschiedenen Variablen zusammen mit ihrer Bedeutung eingeführt haben, können wir uns der eigentlichen Konstruktion von F zuwenden. Wie in Abbildung 7.26 im Detail dargestellt, gleicht die Formel auf der obersten Ebene einer viergliedrigen Konjunktion:

$$F := S \wedge R \wedge T \wedge A$$

Die Teilausdrücke besitzen die folgende Bedeutung:

■ S (Startbedingung)

Dieser Teilausdruck beschreibt die initiale Konfiguration der Turing-Maschine. T befindet sich im Startzustand s_0 , alle Zellen des relevanten Bandabschnitts, die nicht mit Zeichen des Eingabeworts ω belegt sind, enthalten ein Blank-Symbol und der Schreib-Lese-Kopf steht auf Position 0.

■ R (Randbedingungen)

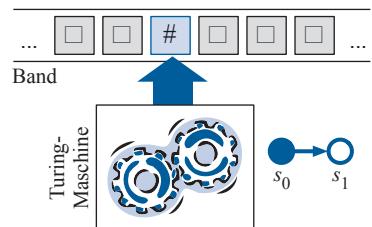
Dieser Teilausdruck beschreibt generelle Konsistenz-eigenschaften, die eine Turing-Maschine zu jedem Zeitpunkt erfüllen muss. Die Eigenschaften werden mithilfe dreier Teilausdrücke R_1, R_2, R_3 formuliert, die im Einzelnen die folgende Bedeutung besitzen:

- R_1 : Eine Turing-Maschine befindet sich zu jedem Zeitpunkt in genau einem Zustand.
- R_2 : Zu jedem Zeitpunkt befindet sich der Schreib-Lese-Kopf an genau einer Position.
- R_3 : Zu jedem Zeitpunkt befindet sich genau ein Symbol an jeder Bandstelle.

Mehrere Formeln aus Abbildung 7.26 nutzen den Ausdruck

$$\text{Onehot}(A_1, A_2, \dots),$$

■ Turing-Maschine



$$S = \{s_0, s_1\}$$

$$\Sigma = \{\#\}$$

$$\Pi = \{\#, \square\}$$

$$E = \{s_1\}$$

■ Übergangstabelle

#	\square
$(s_1, \#, \circlearrowleft)$	—
—	—

■ Bandvariablen

$B_{0,-1,\square}$	$B_{0,0,\square}$	$B_{0,1,\square}$
$B_{0,-1,\#}$	$B_{0,0,\#}$	$B_{0,1,\#}$
$B_{1,-1,\square}$	$B_{1,0,\square}$	$B_{1,1,\square}$
$B_{1,-1,\#}$	$B_{1,0,\#}$	$B_{1,1,\#}$

■ Zustandsvariablen

S_{0,s_0}	S_{0,s_1}
S_{1,s_0}	S_{1,s_1}

■ Positionsvariablen

$P_{0,-1}$	$P_{0,0}$	$P_{0,1}$
$P_{1,-1}$	$P_{1,0}$	$P_{1,1}$

Abbildung 7.25: Erster Schritt in der Konstruktion einer SAT-Instanz. Aus der Beschreibung der Turing-Maschine wird die Menge der Variablen ermittelt, die zur Codierung benötigt werden.

Der Begriff der NP-Vollständigkeit wurde im Jahre 1971 durch den Computerwissenschaftler Stephen Arthur Cook eingeführt. 1939 in Buffalo, New York, geboren, führte ihn seine akademische Laufbahn über Michigan, Harvard und Berkeley an die University of Toronto, wo er noch heute lehrt. In seiner berühmten Publikation „*The Complexity of Theorem Proving Procedures*“ zeigte er die NP-Vollständigkeit von SAT und der vereinfachten Variante 3SAT. 1982 wurde Cook für seine wegweisende Arbeit mit dem Turing-Award ausgezeichnet:

„For his advancement of our understanding of the complexity of computation in a significant and profound way. His seminal paper, *The Complexity of Theorem Proving Procedures*, presented at the 1971 ACM SIGACT Symposium on the Theory of Computing, laid the foundations for the theory of NP-Completeness. The ensuing exploration of the boundaries and nature of NP-complete class of problems has been one of the most active and important research activities in computer science for the last decade.“



Unabhängig von Cook entwickelte der im ukrainischen Dnipropetrovsk geborene und später in die USA emigrierte Mathematiker Leonid Levin einen zur NP-Vollständigkeit äquivalenten Begriff. In seiner 1973 publizierten Arbeit wies er die Eigenschaft für insgesamt 6 Probleme nach; darunter auch SAT [65]. Aus diesem Grund ist der Satz von Cook in der Literatur auch als Cook-Levin-Theorem bekannt.

um die geschilderten Eigenschaften zu beschreiben. Dieser steht stellvertretend für eine aussagenlogische Formel, die genau dann wahr ist, wenn eine und nur eine der Variablen A_1, A_2, \dots mit 1 belegt wird. In der Übungsaufgabe auf Seite 154 hatten Sie Gelegenheit herauszufinden, wie sich eine solche Funktion mit den Operatoren \wedge , \vee und \neg konstruieren lässt.

■ *T* (Transitionsbedingungen)

Die Teilformel T codiert die möglichen Konfigurationsübergänge und setzt sich wiederum aus der Konjunktion dreier Teilausdrücke T_1 , T_2 und T_3 zusammen. Die Bedeutung dieser Terme liest sich wie folgt:

- T_1 : Die Maschine geht in die Folgekonfiguration über.
- T_2 : Bereits terminierte Maschinen sind inaktiv.
- T_3 : Nur der Schreib-Lese-Kopf kann den Bandinhalt ändern.

■ *A* (Akzeptanzbedingung)

Dieser Teilterm codiert die Bedingung, dass die Turing-Maschine das Eingabewort ω genau dann akzeptiert, wenn sie in einem Endzustand anhält. Da die Laufzeit von T durch das Polynom $p(n)$ beschränkt ist, muss der Endzustand in weniger als $p(n)$ Berechnungsschritten erreicht werden. Da sich die Konfiguration einer terminierten Maschine nicht mehr ändert, erreicht T genau dann einen Endzustand, wenn sich die Maschine zum Zeitpunkt $p(n)$ in einem solchen befindet.

Abbildung 7.27 demonstriert die Formelkonstruktion für die weiter oben eingeführte Beispiel-Turing-Maschine. Haben wir F_ω nach dem dargelegten Schema für ein Eingabewort ω aufgestellt, so lässt sich aus jeder Berechnungsfolge, die T unter Eingabe von ω in einem Endzustand terminieren lässt, eine erfüllende Belegung für F_ω ableiten. Umgekehrt gestattet die Konstruktion von F_ω , dass wir aus jeder erfüllenden Belegung eine Berechnungsfolge ableiten können, die T in einen Endzustand führt. Kurzum: T akzeptiert das Eingabewort ω genau dann, wenn für F_ω eine erfüllende Belegung existiert.

Abbildung 7.29 zeigt den Zusammenhang zwischen der Akzeptanz eines Wortes ω durch die Turing-Maschine T und der Erfüllbarkeit von F_ω anhand zweier konkreter Berechnungssequenzen. Im linken Beispiel wird die Maschine mit dem Eingabewort $\#$ gestartet. Die Maschine terminiert nach dem ersten Bearbeitungsschritt in einem Endzustand und die Formel $F_\#$ wird durch die korrespondierende Variablenbelegung erfüllt. Im rechten Beispiel wird T auf das leere Wort angewendet.

■ Finale SAT-Instanz

$$F := S \wedge R \wedge T \wedge A$$

■ Startbedingung

$$S := S_{0,s_0} \wedge P_{0,0} \bigwedge_{k=-p(n)}^{-1} B_{0,k,\square} \bigwedge_{k=0}^{n-1} B_{0,k,\omega_k} \bigwedge_{k=n}^{p(n)} B_{0,k,\square}$$

■ Randbedingungen: $R := R_1 \wedge R_2 \wedge R_3$ mit

$$\begin{aligned} R_1 &:= \bigwedge_{i=0}^{p(n)} \text{Onehot}(S_{i,s_0}, S_{i,s_1}, \dots) \\ R_2 &:= \bigwedge_{i=0}^{p(n)} \text{Onehot}(P_{i,-p(n)}, \dots, P_{i,0}, \dots, P_{i,p(n)}) \\ R_3 &:= \bigwedge_{i=0}^{p(n)} \bigwedge_{k=-p(n)}^{p(n)} \text{Onehot}(B_{i,k,\sigma_1}, B_{i,k,\sigma_2}, \dots) \end{aligned}$$

■ Transitionsbedingungen: $T := T_1 \wedge T_2 \wedge T_3$ mit

$$\begin{aligned} T_1 &:= \bigwedge_{\substack{i,k,s,\sigma \\ \delta(s,\sigma) \neq \emptyset}} (S_{i,s} \wedge P_{i,k} \wedge B_{i,k,\sigma}) \rightarrow \left[\bigvee_{\substack{(s',\sigma',\leftarrow) \\ \in \delta(s,\sigma)}} (S_{i+1,s'} \wedge P_{i+1,k-1} \wedge B_{i+1,k,\sigma'}) \right. \\ &\quad \left. \bigvee_{\substack{(s',\sigma',\rightarrow) \\ \in \delta(s,\sigma)}} (S_{i+1,s'} \wedge P_{i+1,k+1} \wedge B_{i+1,k,\sigma'}) \bigvee_{\substack{(s',\sigma',\odot) \\ \in \delta(s,\sigma)}} (S_{i+1,s'} \wedge P_{i+1,k} \wedge B_{i+1,k,\sigma'}) \right] \\ T_2 &:= \bigwedge_{\substack{i,k,s,\sigma \\ \delta(s,\sigma) = \emptyset}} (S_{i,s} \wedge P_{i,k} \wedge B_{i,k,\sigma}) \rightarrow (S_{i+1,s} \wedge P_{i+1,k} \wedge B_{i+1,k,\sigma}) \\ T_3 &:= \bigwedge_{i=0}^{p(n)} \bigwedge_{k=-p(n)}^{p(n)} \bigwedge_{\sigma \in \Pi} (\overline{P_{i,k}} \wedge B_{i,k,\sigma}) \rightarrow B_{i+1,k,\sigma} \end{aligned}$$

■ Akzeptanzbedingung

$$A := \bigvee_{z \in E} S_{p(n),z}$$

Abbildung 7.26: Codierung von Turing-Maschinen als SAT-Instanz

■ Finale SAT-Instanzen (für $\omega = \#$ und $\omega = \varepsilon$)

$$\begin{aligned} F_{\#} &= S_{\#} \wedge (R_1 \wedge R_2 \wedge R_3) \wedge (T_1 \wedge T_2 \wedge T_3) \wedge A \\ F_{\varepsilon} &= S_{\varepsilon} \wedge (R_1 \wedge R_2 \wedge R_3) \wedge (T_1 \wedge T_2 \wedge T_3) \wedge A \end{aligned}$$

■ Startbedingung

$$\begin{aligned} S_{\#} &= S_{0,s_0} \wedge P_{0,0} \wedge B_{0,-1,\square} \wedge B_{0,0,\#} \wedge B_{0,1,\square} \\ S_{\varepsilon} &= S_{0,s_0} \wedge P_{0,0} \wedge B_{0,-1,\square} \wedge B_{0,0,\square} \wedge B_{0,1,\square} \end{aligned}$$

■ Randbedingungen

$$\begin{aligned} R_1 &= (S_{0,s_0} \vee S_{0,s_1}) \wedge (\overline{(S_{0,s_0} \wedge S_{0,s_1})} \wedge (S_{1,s_0} \vee S_{1,s_1}) \wedge \overline{(S_{1,s_0} \wedge S_{1,s_1})}) \\ R_2 &= (P_{0,-1} \vee P_{0,0} \vee P_{0,1}) \wedge (\overline{(P_{0,-1} \wedge P_{0,0})} \wedge \overline{(P_{0,-1} \wedge P_{0,1})} \wedge \overline{(P_{0,0} \wedge P_{0,1})} \\ &\quad \wedge (P_{1,-1} \vee P_{1,0} \vee P_{1,1}) \wedge (\overline{P_{1,-1} \wedge P_{1,0}}) \wedge (\overline{P_{1,-1} \wedge P_{1,1}}) \wedge (\overline{P_{1,0} \wedge P_{1,1}})) \\ R_3 &= (B_{0,-1,\square} \vee B_{0,-1,\#}) \wedge (\overline{B_{0,-1,\square} \wedge B_{0,-1,\#}}) \wedge (B_{0,0,\square} \vee B_{0,0,\#}) \wedge (\overline{B_{0,0,\square} \wedge B_{0,0,\#}}) \wedge \\ &\quad (B_{0,1,\square} \vee B_{0,1,\#}) \wedge (\overline{B_{0,1,\square} \wedge B_{0,1,\#}}) \wedge (B_{1,-1,\square} \vee B_{1,-1,\#}) \wedge (\overline{B_{1,-1,\square} \wedge B_{1,-1,\#}}) \wedge \\ &\quad (B_{1,0,\square} \vee B_{1,0,\#}) \wedge (\overline{B_{1,0,\square} \wedge B_{1,0,\#}}) \wedge (B_{1,1,\square} \vee B_{1,1,\#}) \wedge (\overline{B_{1,1,\square} \wedge B_{1,1,\#}}) \end{aligned}$$

■ Transitionsbedingungen

$$\begin{aligned} T_1 &= ((S_{0,s_0} \wedge P_{0,-1} \wedge B_{0,-1,\#}) \rightarrow (S_{1,s_1} \wedge P_{1,-1} \wedge B_{1,-1,\#})) \wedge \\ &\quad ((S_{0,s_0} \wedge P_{0,0} \wedge B_{0,0,\#}) \rightarrow (S_{1,s_1} \wedge P_{1,0} \wedge B_{1,0,\#})) \wedge \\ &\quad ((S_{0,s_0} \wedge P_{0,1} \wedge B_{0,1,\#}) \rightarrow (S_{1,s_1} \wedge P_{1,1} \wedge B_{1,1,\#})) \wedge \\ T_2 &= ((S_{0,s_0} \wedge P_{0,-1} \wedge B_{0,-1,\square}) \rightarrow (S_{1,s_0} \wedge P_{1,-1} \wedge B_{1,-1,\square})) \wedge \\ &\quad ((S_{0,s_0} \wedge P_{0,0} \wedge B_{0,0,\square}) \rightarrow (S_{1,s_0} \wedge P_{1,0} \wedge B_{1,0,\square})) \wedge \\ &\quad ((S_{0,s_0} \wedge P_{0,1} \wedge B_{0,1,\square}) \rightarrow (S_{1,s_0} \wedge P_{1,1} \wedge B_{1,1,\square})) \wedge \\ &\quad ((S_{0,s_1} \wedge P_{0,-1} \wedge B_{0,-1,\#}) \rightarrow (S_{1,s_1} \wedge P_{1,-1} \wedge B_{1,-1,\#})) \wedge \\ &\quad ((S_{0,s_1} \wedge P_{0,0} \wedge B_{0,0,\#}) \rightarrow (S_{1,s_1} \wedge P_{1,0} \wedge B_{1,0,\#})) \wedge \\ &\quad ((S_{0,s_1} \wedge P_{0,1} \wedge B_{0,1,\#}) \rightarrow (S_{1,s_1} \wedge P_{1,1} \wedge B_{1,1,\#})) \wedge \\ &\quad ((S_{0,s_1} \wedge P_{0,-1} \wedge B_{0,-1,\square}) \rightarrow (S_{1,s_1} \wedge P_{1,-1} \wedge B_{1,-1,\square})) \wedge \\ &\quad ((S_{0,s_1} \wedge P_{0,0} \wedge B_{0,0,\square}) \rightarrow (S_{1,s_1} \wedge P_{1,0} \wedge B_{1,0,\square})) \wedge \\ &\quad ((S_{0,s_1} \wedge P_{0,1} \wedge B_{0,1,\square}) \rightarrow (S_{1,s_1} \wedge P_{1,1} \wedge B_{1,1,\square})) \wedge \\ T_3 &= ((\overline{P_{0,-1}} \wedge B_{0,-1,\square}) \rightarrow B_{1,-1,\square}) \wedge ((\overline{P_{0,-1}} \wedge B_{0,-1,\#}) \rightarrow B_{1,-1,\#}) \wedge \\ &\quad ((\overline{P_{0,0}} \wedge B_{0,0,\square}) \rightarrow B_{1,0,\square}) \wedge ((\overline{P_{0,0}} \wedge B_{0,0,\#}) \rightarrow B_{1,0,\#}) \wedge \\ &\quad ((\overline{P_{0,1}} \wedge B_{0,1,\square}) \rightarrow B_{1,1,\square}) \wedge ((\overline{P_{0,1}} \wedge B_{0,1,\#}) \rightarrow B_{1,1,\#}) \end{aligned}$$

■ Akzeptanzbedingung

$$A = S_{1,s_1}$$

Abbildung 7.27: Konstruktion zweier SAT-Instanzen F_{ω} und F_{ε} für unsere Beispiel-Turing-Maschine

■ Finale SAT-Instanzen (für $\omega = \#$ und $\omega = \varepsilon$)

$$\begin{aligned} F_{\#} &= S_{\#} \wedge (R_1 \wedge R_2 \wedge R_3) \wedge (T_1 \wedge T_2 \wedge T_3) \wedge A \\ F_{\varepsilon} &= S_{\varepsilon} \wedge (R_1 \wedge R_2 \wedge R_3) \wedge (T_1 \wedge T_2 \wedge T_3) \wedge A \end{aligned}$$

■ Startbedingung

$$\begin{aligned} S_{\#} &= S_{0,s_0} \wedge P_{0,0} \wedge B_{0,-1,\square} \wedge B_{0,0,\#} \wedge B_{0,1,\square} \\ S_{\varepsilon} &= S_{0,s_0} \wedge P_{0,0} \wedge B_{0,-1,\square} \wedge B_{0,0,\square} \wedge B_{0,1,\square} \end{aligned}$$

■ Randbedingungen

$$\begin{aligned} R_1 &= (S_{0,s_0} \vee S_{0,s_1}) \wedge (\overline{S_{0,s_0}} \vee \overline{S_{0,s_1}}) \wedge (S_{1,s_0} \vee S_{1,s_1}) \wedge (\overline{S_{1,s_0}} \vee \overline{S_{1,s_1}}) \\ R_2 &= (P_{0,-1} \vee P_{0,0} \vee P_{0,1}) \wedge (\overline{P_{0,-1}} \vee \overline{P_{0,0}}) \wedge (\overline{P_{0,-1}} \vee \overline{P_{0,1}}) \wedge (\overline{P_{0,0}} \vee \overline{P_{0,1}}) \wedge \\ &\quad (P_{1,-1} \vee P_{1,0} \vee P_{1,1}) \wedge (\overline{P_{1,-1}} \vee \overline{P_{1,0}}) \wedge (\overline{P_{1,-1}} \vee \overline{P_{1,1}}) \wedge (\overline{P_{1,0}} \vee \overline{P_{1,1}}) \\ R_3 &= (B_{0,-1,\square} \vee B_{0,-1,\#}) \wedge (\overline{B}_{0,-1,\square} \vee \overline{B}_{0,-1,\#}) \wedge (B_{0,0,\square} \vee B_{0,0,\#}) \wedge (\overline{B}_{0,0,\square} \vee \overline{B}_{0,0,\#}) \wedge \\ &\quad (B_{0,1,\square} \vee B_{0,1,\#}) \wedge (\overline{B}_{0,1,\square} \vee \overline{B}_{0,1,\#}) \wedge (B_{1,-1,\square} \vee B_{1,-1,\#}) \wedge (\overline{B}_{1,-1,\square} \vee \overline{B}_{1,-1,\#}) \wedge \\ &\quad (B_{1,0,\square} \vee B_{1,0,\#}) \wedge (\overline{B}_{1,0,\square} \vee \overline{B}_{1,0,\#}) \wedge (B_{1,1,\square} \vee B_{1,1,\#}) \wedge (\overline{B}_{1,1,\square} \vee \overline{B}_{1,1,\#}) \end{aligned}$$

■ Transitionsbedingungen

$$\begin{aligned} T_1 &= (\overline{S_{0,s_0}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\#}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\#}} \vee P_{1,-1}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\#}} \vee B_{1,-1,\#}) \wedge \\ &\quad (\overline{S_{0,s_0}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\#}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\#}} \vee P_{1,0}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\#}} \vee B_{1,0,\#}) \wedge \\ &\quad (\overline{S_{0,s_0}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\#}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\#}} \vee P_{1,1}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\#}} \vee S_{1,s_1} \vee B_{1,1,\#}) \\ T_2 &= (\overline{S_{0,s_0}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\square}} \vee S_{1,s_0}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\square}} \vee P_{1,-1}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\square}} \vee B_{1,-1,\square}) \wedge \\ &\quad (\overline{S_{0,s_0}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\square}} \vee S_{1,s_0}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\square}} \vee P_{1,0}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\square}} \vee B_{1,0,\square}) \wedge \\ &\quad (\overline{S_{0,s_0}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\square}} \vee S_{1,s_0}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\square}} \vee P_{1,1}) \wedge (\overline{S_{0,s_0}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\square}} \vee B_{1,1,\square}) \wedge \\ &\quad (\overline{S_{0,s_1}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\#}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\#}} \vee P_{1,-1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\#}} \vee B_{1,-1,\#}) \wedge \\ &\quad (\overline{S_{0,s_1}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\#}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\#}} \vee P_{1,0}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\#}} \vee B_{1,0,\#}) \wedge \\ &\quad (\overline{S_{0,s_1}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\#}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\#}} \vee P_{1,1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\#}} \vee B_{1,1,\#}) \wedge \\ &\quad (\overline{S_{0,s_1}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\square}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\square}} \vee P_{1,-1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,-1}} \vee \overline{B_{0,-1,\square}} \vee B_{1,-1,\square}) \wedge \\ &\quad (\overline{S_{0,s_1}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\square}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\square}} \vee P_{1,0}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,0}} \vee \overline{B_{0,0,\square}} \vee B_{1,0,\square}) \wedge \\ &\quad (\overline{S_{0,s_1}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\square}} \vee S_{1,s_1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\square}} \vee P_{1,1}) \wedge (\overline{S_{0,s_1}} \vee \overline{P_{0,1}} \vee \overline{B_{0,1,\square}} \vee B_{1,1,\square}) \\ T_3 &= (P_{0,-1} \vee \overline{B_{0,-1,\square}} \vee B_{1,-1,\square}) \wedge (P_{0,-1} \vee \overline{B_{0,-1,\#}} \vee B_{1,-1,\#}) \wedge \\ &\quad (P_{0,0} \vee \overline{B_{0,0,\square}} \vee B_{1,0,\square}) \wedge (P_{0,0} \vee \overline{B_{0,0,\#}} \vee B_{1,0,\#}) \wedge \\ &\quad (P_{0,1} \vee \overline{B_{0,1,\square}} \vee B_{1,1,\square}) \wedge (P_{0,1} \vee \overline{B_{0,1,\#}} \vee B_{1,1,\#}) \end{aligned}$$

■ Akzeptanzbedingung

$$A = S_{1,s_1}$$

Abbildung 7.28: Transformation der SAT-Instanzen F_{ω} und F_{ε} in konjunktive Normalform

Hier verharrt die Maschine im Startzustand s_0 und F_e wird durch die korrespondierende Variablenbelegung nicht erfüllt.

Da uns das entwickelte Konstruktionsschema ermöglicht, jedes Problem der Klasse NP auf SAT zu reduzieren, sind wir einer fertigen Beweisskizze für den Satz von Cook bereits sehr nahe. Der fehlende Baustein ist der Nachweis, dass die Reduktion polynomiell erfolgt, d. h., dass die konstruierte Formel F nur polynomiell mit der Eingabelänge n wächst. Von dieser Eigenschaft können wir uns überzeugen, indem wir die Länge der vier Teilterme S , R , T und A genauer untersuchen. Es gelten die folgenden Beziehungen:

$$\begin{array}{ll} |S| = O(p(n)) & |R| = O(p(n)^3) \\ |T| = O(p(n)^2) & |A| = O(1) \end{array}$$

Die Gesamtformel F lässt sich mit dem Aufwand $O(p(n)^3)$ konstruieren und wächst damit in der Tat nur polynomiell mit der Eingabelänge n . Damit sind wir am Ziel und haben SAT als NP-vollständiges Problem identifiziert, ohne auf die Reduktionstechnik zurückzugreifen.

7.3.4 Reduktionsbeweise

Die mühsame Arbeit des vorherigen Abschnitts war nicht umsonst, schließlich können wir unser erworbene Wissen über SAT jetzt einsetzen, um weitere Probleme als NP-vollständig zu identifizieren. Hierzu werden wir auf die weiter oben skizzierte Reduktionstechnik zurückgreifen und zeigen, dass sich SAT in polynomieller Zeit auf die untersuchten Probleme reduzieren lässt.

Als erstes Beispiel betrachten wir mit 3SAT eine abgeschwächte Variante von SAT. Formal ist dieses Problem wie folgt definiert:



Definition 7.19 (3SAT)

Das Problem 3SAT lautet wie folgt:

- Gegeben: n aussagenlogische Klauseln mit maximal 3 Literalen
- Gefragt: Gibt es eine erfüllende Belegung?

Genau wie SAT stellt 3SAT die Frage nach der Erfüllbarkeit einer aussagenlogischen Formel. Während SAT jedoch beliebige Eingaben zulässt,

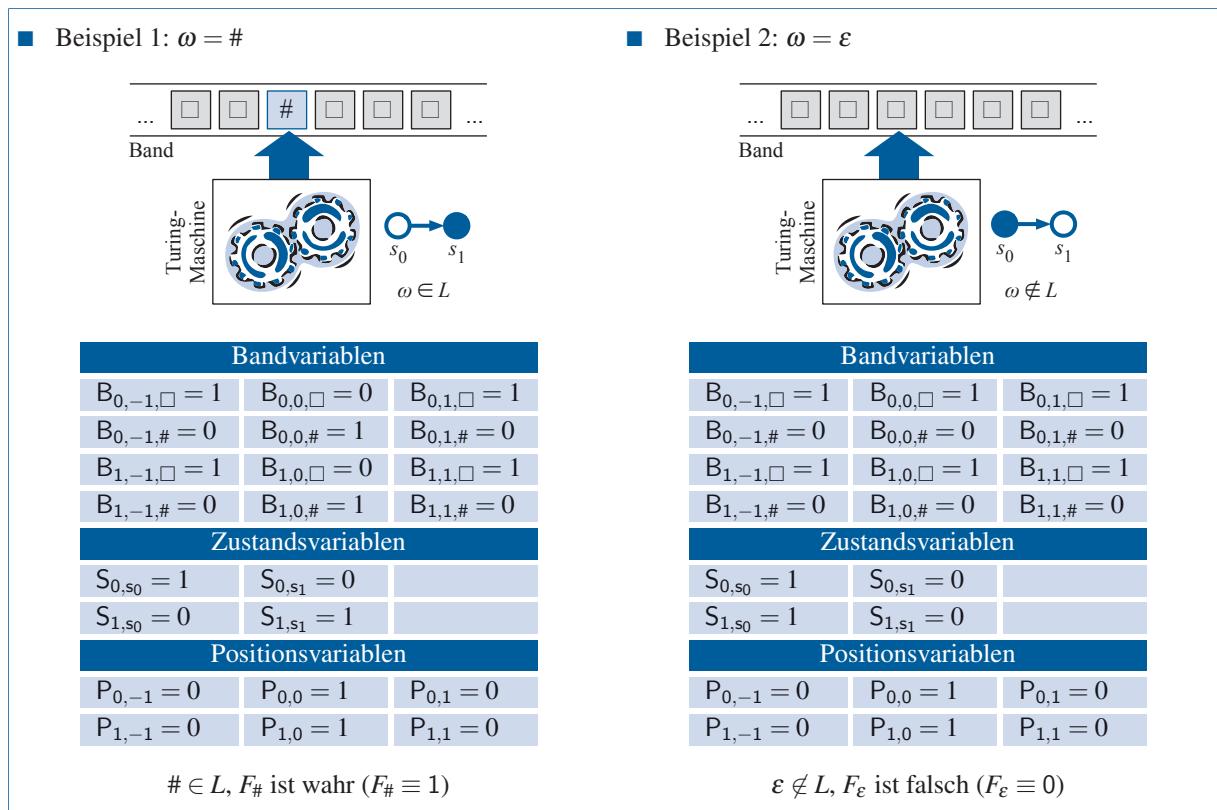


Abbildung 7.29: Zwei konkrete Beispiele, die den Zusammenhang zwischen T und F_ω demonstrieren. Die Turing-Maschine T akzeptiert das Eingabewort ω genau dann, wenn die Formel F_ω erfüllbar ist.

muss die Formel für 3SAT in der Form

$$\{L_{11}, L_{12}, L_{13}\} \wedge \{L_{21}, L_{22}, L_{23}\} \wedge \dots \wedge \{L_{n1}, L_{n2}, L_{n3}\}$$

vorliegen. Beachten Sie, dass wir in Definition 7.19 nicht gefordert haben, dass eine Klausel genau 3 Literale besitzen muss. Hierdurch ist z. B. auch die Formel

$$\{A, \neg B\} \wedge \{\neg A\}$$

eine 3SAT-Instanz.

Bevor wir die Reduktion von SAT auf 3SAT vollziehen, wollen wir einen erneuten Blick auf den im vorherigen Abschnitt durchgeführten Beweis werfen. Wir haben gezeigt, wie sich eine nichtdeterministische

Der amerikanische Computerwissenschaftler Richard Manning Karp erkannte als einer der Ersten, wie bedeutend Cooks Entdeckung über die NP-Vollständigkeit von SAT wirklich war. In seiner 1972 publizierten Arbeit stellte er 20 weitere Probleme vor, die er mithilfe der Reduktionstechnik als NP-vollständig identifizierte. Zu diesen gehörte unter anderem auch das CLIQUE-Problem, das uns in diesem Abschnitt zur Demonstration der Reduktionstechnik dient. Für seine bedeutenden Arbeiten wurde Karp im Jahre 1985 mit dem Turing-Award ausgezeichnet:

„For his continuing contributions to the theory of algorithms including the development of efficient algorithms for network flow and other combinatorial optimization problems, the identification of polynomial-time computability with the intuitive notion of algorithmic efficiency, and, most notably, contributions to the theory of NP-completeness. Karp introduced the now standard methodology for proving problems to be NP-complete which has led to the identification of many theoretical and practical problems as being computationally difficult.“



Neben seinen Errungenschaften im Gebiet der theoretischen Informatik leistete Karp wichtige Beiträge im Gebiet der Algorithmentechnik. Zu den wichtigsten Arbeiten gehören der Edmonds-Karp-Algorithmus zur Berechnung des maximalen Flusses in Netzwerken und der Rabin-Karp-Algorithmus für die Hash-basierte Textsuche.

Turing-Maschine T polynomiell auf eine aussagenlogische Formel F reduzieren lässt, die genau dann erfüllbar ist, wenn T in einem Endzustand terminiert. Betrachten wir die Struktur der konstruierten Formel F genauer, so wird klar, dass wir in Wirklichkeit ein stärkeres Ergebnis bewiesen haben. Auf der obersten Ebene besitzt F eine konjunktive Struktur und auch die Teilterme sind diesbezüglich weitgehend regulär aufgebaut.

Ersetzen wir den Implikationsoperator $G \rightarrow H$ durch den äquivalenten Ausdruck $\overline{G} \vee H$, schieben die Negationen mithilfe der De Morgan'schen Regeln in die Teilterme und wenden auf die verbleibenden Ausdrücke das Distributivgesetz an, so lässt sich F in eine Klausel-form überführen, die nur polynomiell länger wird als die ursprüngliche Formel F . Abbildung 7.28 zeigt die entstehende Klauselmenge für unsere weiter oben diskutierte Beispielmaschine. Damit ist die NP-Vollständigkeit von 3SAT bereits dann bewiesen, wenn wir es schaffen, eine Klauselmenge der Form

$$\{L_{11}, \dots, L_{1i_1}\}, \dots, \{L_{21}, \dots, L_{2i_2}\}, \dots, \{L_{n1}, \dots, L_{ni_n}\} \quad (7.8)$$

so auf eine äquivalente 3SAT-Instanz abzubilden, dass die Länge nur polynomiell zunimmt.

Um dies zu erreichen, bedienen wir uns eines Ergebnisses, das Sie im Übungsteil auf Seite 154 herausarbeiten durften. Dort sollten Sie zeigen, dass die Formel

$$(A_1 \vee A_2 \vee A_3 \vee A_4)$$

genau dann erfüllbar ist, wenn die Formel

$$(A_1 \vee A_2 \vee B) \wedge (\neg B \vee A_3 \vee A_4)$$

erfüllbar ist. Wenden wir dieses Schema rekursiv an, so können wir die Menge (7.8) wie folgt in eine erfüllbarkeitsäquivalente Klauselmenge übertragen:

$$\{L_{11}, L_{12}, L'\}, \{\neg L', L_{13}, L''\}, \{\neg L'', L_{14}, L'''\}, \dots \quad (7.9)$$

Die erzeugte Menge ist eine Instanz von 3SAT, in der sich die Gesamtzahl der Literale nur linear erhöht hat. Mit der vorgenommenen Transformation ist es uns gelungen, die ursprüngliche Klauselmenge polynomiell auf eine Instanz von 3SAT zu reduzieren und hierdurch die Beziehung $SAT \leq_{poly} 3SAT$ zu zeigen. Damit sind wir am Ziel und haben den folgenden Satz bewiesen:



Satz 7.11 (Satz von Cook)

Das Problem 3SAT ist NP-vollständig.

Auch wenn 3SAT in der praktischen Informatik kaum eine Rolle spielt, ist seine Bedeutung für die theoretische Informatik umso größer. Genauso wie SAT können wir 3SAT dazu verwenden, um andere Probleme als NP-vollständig zu identifizieren; aufgrund der einfachen Klauselstruktur lässt sich die Reduktion aber meist einfacher durchführen als im Falle von SAT. Am Beispiel des CLIQUE-Problems wollen wir eine entsprechende Reduktion demonstrieren.



Definition 7.20 (CLIQUE)

Das Problem CLIQUE lautet wie folgt:

- Gegeben: Graph G mit Knotenmenge V und Kantenmenge E .
- Gefragt: Existieren n Knoten v_1, \dots, v_n , die paarweise durch eine Kante aus E verbunden sind?

Eine Menge $C = \{v_1, \dots, v_n\}$ mit dieser Eigenschaft heißt Clique der Größe n .

Interpretieren wir die Knoten eines Graphen als Menschen und die Kanten als Bekanntschaftsbeziehungen, so entspricht eine Clique der Größe m einer Gruppe von m Menschen, in der jeder jeden kennt.

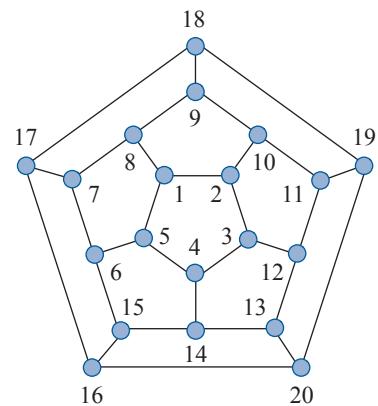
Abbildung 7.30 demonstriert die Beziehung anhand zweier Beispielgraphen. Während der erste Graph ausschließlich Cliques der Größe 2 enthält, sind im zweiten Graphen mit $\{2, 3, 5\}$, $\{3, 5, 6\}$, $\{3, 4, 6\}$ und $\{5, 6, 8\}$ vier verschiedene 3er-Cliques vorhanden.

Wir werden nun zeigen, dass wir mit CLIQUE ein weiteres NP-vollständiges Problem vor uns haben. Dass es in der Komplexitätsklasse NP liegt, ist offensichtlich: Um festzustellen, ob eine Clique der Größe n existiert, müssen wir lediglich eine geeignete Knotenmenge v_1, \dots, v_n raten und anschließend überprüfen, ob alle Knoten paarweise durch eine Kante verbunden sind. Die Überprüfung ist in polynomieller Zeit durchführbar, da der Aufwand lediglich quadratisch mit n zunimmt.

Den Rest des Beweises erledigen wir durch die Reduktion von 3SAT auf CLIQUE. Hierzu müssen wir eine n -elementige Klauselmenge

$$M := \{ \{L_{11}, L_{12}, L_{13}\}, \{L_{21}, L_{22}, L_{23}\}, \dots, \{L_{n1}, L_{n2}, L_{n3}\} \}$$

■ Beispiel 1



■ Beispiel 2

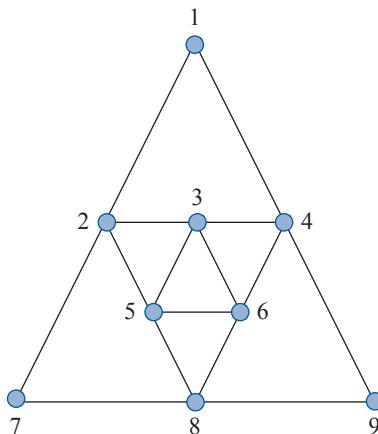
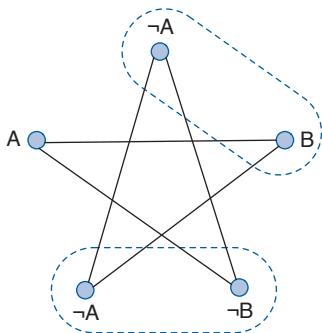


Abbildung 7.30: Eine Clique der Größe k ist eine Menge von k paarweise verbundenen Knoten. Während der obere Graph ausschließlich Cliques der Größe 2 enthält, besitzt der untere Graph 4 Cliques der Größe 3.

■ Beispiel 1

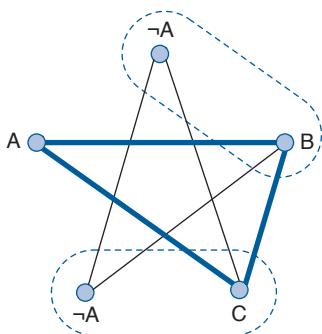
$$M := \{\{A\}, \{\neg A, B\}, \{\neg A, \neg B\}\}$$



Der Graph enthält keine Clique der Größe 3 $\Rightarrow M$ ist unerfüllbar

■ Beispiel 2

$$M := \{\{A\}, \{\neg A, B\}, \{\neg A, C\}\}$$



Der Graph enthält eine Clique der Größe 3 $\Rightarrow M$ ist erfüllbar

Erfüllende Belegung:
 $I(A) = I(B) = I(C) = 1$

Abbildung 7.31: Transformation einer n -elementigen Klauselmenge M in einen Graphen G . M ist genau dann erfüllbar, wenn G eine Clique der Größe n enthält.

polynomiell auf einen Graphen $G = (V, E)$ mit der folgenden Eigenschaft abbilden:

$$M \text{ ist erfüllbar} \Leftrightarrow G \text{ enthält eine Clique der Größe } n$$

Abbildung 7.31 demonstriert anhand zweier Beispiele, wie wir einen Graphen mit dieser Eigenschaft erhalten können. Zunächst generieren wir für jedes Literal einen Knoten v . Anschließend verbinden wir zwei Knoten v_1 und v_2 genau dann mit einer Kante, falls

- L_1 und L_2 nicht derselben Klausel angehören und
- L_1 und L_2 gleichzeitig erfüllbar sind.

Zwei Literale L_1 und L_2 sind genau dann gleichzeitig erfüllbar, wenn sie nicht komplementär zueinander sind. Die zweite Bedingung ist damit äquivalent zu der Aussage $L_1 \neq \neg L_2$.

Um den Zusammenhang zwischen M und G zu verstehen, unterscheiden wir zwei Fälle. Ist die Menge M erfüllbar, so existiert in jeder Klausel mindestens ein Literal, das zu 1 auswertet. Aufgrund der gewählten Konstruktion von G sind die zugehörigen Knoten allesamt durch Kanten miteinander verbunden, so dass in G eine Clique der Größe n existiert. Die Umkehrung der Schlussrichtung gilt ebenfalls. Existiert eine Clique der Größe n , so können wir aus den zugehörigen Knoten eine Variablenbelegung für M ableiten. Repräsentiert ein Knoten v ein Literal der Form $L = \neg A$, so setzen wir $A = 0$. Gilt dagegen $L = A$, so wählen wir $A = 1$. Da die Literale einer Clique alle gleichzeitig erfüllbar sind, erhalten wir als Ergebnis eine erfüllende Belegung für M . Kurzum: M ist genau dann erfüllbar, wenn der konstruierte Graph eine Clique der Größe n enthält. Damit sind wir am Ziel und haben den folgenden Satz bewiesen:

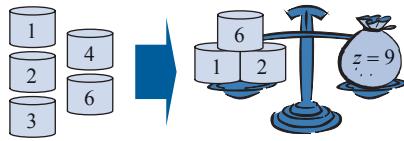


Das Problem CLIQUE ist NP-vollständig.

Abbildung 7.32 enthält eine Auswahl weitere Probleme, die sich mithilfe der Reduktionstechnik als NP-vollständig entlarven lassen. Eine ausführliche Darstellung, wie die Reduktionen im Einzelnen durchgeführt werden können, findet sich in [89].

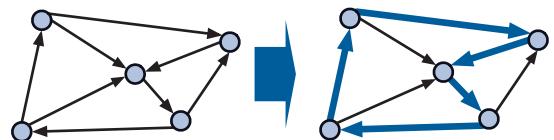
■ SELECT

Gegeben seien eine Folge natürlicher Zahlen $x_1, \dots, x_n \in \mathbb{N}$ und eine weitere natürliche Zahl $z \in \mathbb{N}$. Existiert eine Auswahl von Elementen x_{i_1}, \dots, x_{i_k} , deren Summe z ergibt? Mit anderen Worten: Existiert eine Teilmenge $S \subseteq \{1, \dots, n\}$ mit $\sum_{i \in S} x_i = z$?



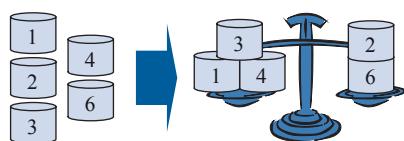
■ HAMILTON (gerichtet)

Gegeben sei ein *gerichteter* Graph $G = (V, E)$. Lassen sich die Knoten derart umsortieren, dass die neu geordnete Menge $V' = (v_1, \dots, v_n)$ für $1 \leq i < n$ die Beziehung $(v_i, v_{i+1}) \in E$ und für $i = n$ die Beziehung $(v_i, v_1) \in E$ erfüllt?



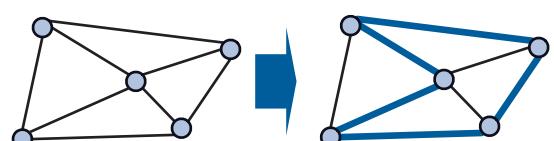
■ PARTITION

Gegeben sei eine Folge natürlicher Zahlen x_1, \dots, x_n . Lassen sich die Zahlen so aufteilen, dass die Summe beider Partitionen gleich ist? Mit anderen Worten: Existiert eine Teilmenge $S \subseteq \{1, \dots, n\}$ mit $\sum_{i \in S} x_i = \sum_{i \notin S} x_i$?



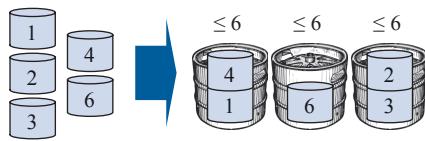
■ HAMILTON (ungerichtet)

Gegeben sei ein *ungerichteter* Graph $G = (V, E)$. Lassen sich die Knoten derart umsortieren, dass die neu geordnete Menge $V' = (v_1, \dots, v_n)$ für $1 \leq i < n$ die Beziehung $(v_i, v_{i+1}) \in E$ und für $i = n$ die Beziehung $(v_i, v_1) \in E$ erfüllt?



■ BIN PACKING

Gegeben seien k Container mit dem gleichen Fassungsvermögen V . Für n Zahlen x_1, \dots, x_n ist zu entscheiden, ob sich diese auf die Container verteilen lassen, ohne das Fassungsvermögen zu überschreiten. Mit anderen Worten: Existiert eine Zuordnung $f : \{1, \dots, n\} \rightarrow \{1, \dots, k\}$ mit $\sum_{f(i)=j} x_i < V$?



■ TRAVELING SALESMAN

Gegeben sei eine Menge von Städten $\{S_1, \dots, S_n\}$ sowie eine Straßenkarte, auf der die Entfernungen zwischen S_i und S_j verzeichnet sind. Für eine gegebene Konstante k gilt es zu entscheiden, ob alle Städte auf einem Rundweg besucht werden können, der die Gesamtstrecke k nicht überschreitet.

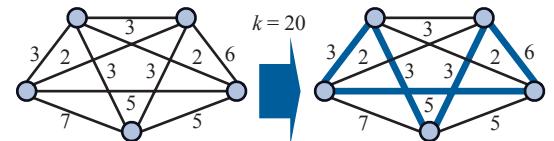


Abbildung 7.32: Auswahl weiterer NP-vollständiger Probleme

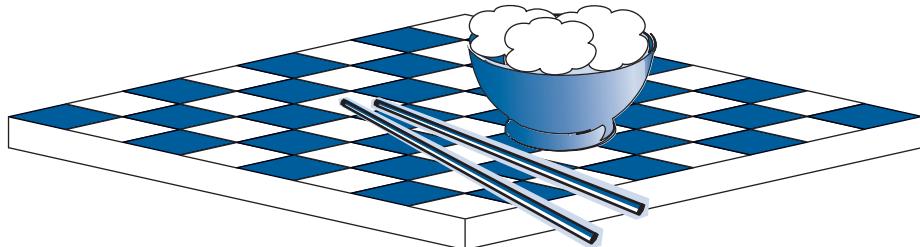
7.4 Übungsaufgaben

Aufgabe 7.1



Webcode
7890

Erinnern Sie sich an die alte chinesische Legende von dem Bauer, der dem Kaiser eine großen Dienst erwies? Der Bauer nahm ein Schachbrett zur Hand und bat den Kaiser darum, auf das erste Feld ein Reiskorn zu legen und die Zahl der Reiskörner mit jedem Feld zu verdoppeln. Der Kaiser wollte ihm jeden Wunsch gewähren und willigte ein.



- Erklären Sie, warum der königliche Mathematiker das Bewusstsein verlor.
- Hätte der Kaiser einwilligen sollen, wenn sich der Bauer für eine polynomiale Zunahme der Reiskörner entschieden hätte?

Aufgabe 7.2



Webcode
7587

Ihre Aufgabe ist es, die Beladung des abgebildeten Transportflugzeugs zu organisieren. Als Fracht stehen Ihnen vier verschiedene Containertypen zur Verfügung, die sich in Volumen und Wert voneinander unterscheiden.



- Wie muss das Flugzeug beladen werden, um einen größtmöglichen Wert zu transportieren? Achten Sie in Ihrer Lösung darauf, das maximale Ladungsvolumen von 20 Einheiten nicht zu überschreiten.
- Funktioniert der von Ihnen verwendete Algorithmus auch dann, wenn die Volumina in Form reeller Zahlen gegeben wären?

Die abgebildeten Java-Funktionen durchsuchen ein absteigend sortiertes Integer-Array `a` nach dem Element `key`. Ist die Suche erfolgreich, so liefert die Funktion die Listenposition von `key` zurück. Ist `key` nicht in der Liste enthalten, so ist der Funktionswert gleich `-1`.

Aufgabe 7.3**Weocode****7375****linSearch.java**

```
public static int
linSearch(int[] a, int key)
{
    int lo = 0;
    int hi = a.length;

    for (lo = 0; lo < hi; lo++) {
        int val = a[lo];

        if (val == key)
            return lo;
    }
    return -1;
}
```

binSearch.java

```
1  public static int
2  binSearch(int[] a, int key)
3  {
4      int lo = 0;
5      int hi = a.length;
6
7      while (lo < hi) {
8          int mid = (lo + hi) >> 1;
9          int val = a[mid];
10
11         if (val < key)
12             lo = mid + 1;
13         else if (val > key)
14             hi = mid;
15         else
16             return mid;
17     }
18     return -1;
19 }
```

Die linke Implementierung verwendet eine *lineare Suche* und vergleicht alle Array-Elemente von `a` nacheinander mit dem Element `key`. Sobald das Element gefunden wurde, terminiert der Algorithmus vorzeitig und liefert den aktuellen Stand der Schleifenvariablen `lo` zurück.

Die zweite Implementierung verwendet das Prinzip der *binären Suche*. Über die Variablen `lo` und `hi` wird ein Suchintervall definiert, in dem sich das Element `key` befinden muss. Zu Beginn erstreckt sich das Intervall über sämtliche Array-Elemente und wird anschließend iterativ verkleinert. Hierzu wird das Element in der Intervallmitte bestimmt (Position `mid`) und mit der Variablen `key` verglichen. Ist das Element kleiner als `key`, so muss sich das gesuchte Element in der rechten Hälfte befinden. Ist es dagegen größer, so kann es ausschließlich in der linken Hälfte liegen. Auf diese Weise wird das Suchintervall kontinuierlich halbiert, bis das Element `key` gefunden wird oder das Suchintervall zur leeren Menge degradiert.

- Welche Laufzeitkomplexitäten besitzt die lineare Suche?
- Welche Laufzeitkomplexitäten besitzt die binäre Suche?
- Ist die binäre Suche der linearen Suche in allen Fällen überlegen?

Aufgabe 7.4
Webcode
7761

Auf Seite 116 ist das Strukturbild eines vollständig aufgebauten 4-Bit-Carry-look-ahead-Addierers dargestellt. Ihre Aufgabe ist es, die asymptotische Wachstumsrate des Flächenbedarfs eines n -Carry-look-ahead-Addierers zu bestimmen.

- Welche Flächenkomplexität besitzt der Addierer unter der Annahme, dass alle Logikgatter die gleiche Größe haben und der Flächenbedarf der Signalleitungen vernachlässigbar ist?
- Welche Flächenkomplexität besitzt der Addierer unter der Annahme, dass der Flächenbedarf eines Logikgatters proportional mit der Anzahl seiner Eingänge wächst?

Geben Sie Ihre Antworten in der Notation des O-Kalküls an.

Aufgabe 7.5
Webcode
7469

Exponentielle Wachstumsraten sind der Feind eines jeden Programmierers und nur wenige Menschen sind in der Lage, sie präzise abzuschätzen. Wie schwer uns der Umgang mit dem Exponentiellen wirklich fällt, demonstriert das folgende, aus der Volkswirtschaftslehre entliehene Beispiel. Dargestellt ist eine vergleichende Hochrechnung des Wirtschaftswachstums der drei Länder Indien, Indonesien und Japan für die Jahre 1890 bis 1990:



In Indien wuchs das Bruttoinlandsprodukt (BIP) in diesem Zeitraum durchschnittlich ca. 0,65 % pro Jahr. In Indonesien betrug das Wachstum zur gleichen Zeit durchschnittlich ca. 1 % und in Japan ca. 3 % pro Jahr.

- Schätzen Sie ohne Rechnung, um welchen Faktor sich das Bruttoinlandsprodukt in den angesprochenen 100 Jahren in jedem Land verändert hat.
- Rechnen Sie die kumulierte Wachstumsrate exakt aus und vergleichen Sie das Ergebnis mit Ihrer Schätzung.

Mit der o-Notation haben Sie eine Möglichkeit kennen gelernt, starke obere asymptotische Schranken anzugeben. So drückt die Schreibweise $f(n) = o(g(n))$ aus, dass f asymptotisch langsamer wächst als g . Formal haben wir die o-Notation wie folgt definiert:

$$\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c \cdot f(n) \leq g(n)$$

Können wir diese Definition durch die folgende Alternative ersetzen, ohne die definierten Komplexitätsklassen zu verändern?

a) $\exists n_0 \in \mathbb{N} \forall c \in \mathbb{R}^+ \forall n \geq n_0 c \cdot f(n) \leq g(n)$

Wie wirken sich die nachstehenden Veränderungen aus?

b) $\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n > n_0 c \cdot f(n) \leq g(n)$

c) $\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 c \cdot f(n) < g(n)$

d) $\forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n > n_0 c \cdot f(n) < g(n)$

Aufgabe 7.6

Webcode
7049

In Kapitel 3 haben Sie mit dem Tableaukalkül ein Semi-Entscheidungsverfahren für die Prädikatenlogik erster Stufe kennen gelernt. Wir sind in dieser Aufgabe an der Laufzeit des Tableau-Verfahrens interessiert. Um diese abschätzen zu können, definieren wir zunächst die Funktion $t(F)$:

$$t(F) := \begin{cases} n & : \text{Unter Eingabe von } F \text{ terminiert das Verfahren nach } n \text{ Schritten} \\ 0 & : \text{Unter Eingabe von } F \text{ terminiert das Verfahren nicht} \end{cases}$$

Um die Laufzeit in Relation zur Eingabelänge zu setzen, definieren wir die Wachstumsfunktion $f(n)$ wie folgt:

$$f(n) := \max\{t(F) \mid |F| = n\}$$

$|F|$ bezeichnet die Länge der prädikatenlogischen Formel F . Damit entspricht der Funktionswert $f(n)$ der maximalen Anzahl an Berechnungsschritten, die das Tableaux-Verfahren im Terminierungsfall für Eingaben der Länge n benötigt. Wir wollen die Frage klären, welcher Komplexitätsklasse wir $f(n)$ zuordnen können.

Aufgabe 7.7

Webcode
7199

a) Versuchen Sie, das asymptotische Wachstum von $f(n)$ nach oben abzuschätzen, indem Sie eine Funktion $g(n)$ mit $f(n) = O(g(n))$ angeben.

b) Obwohl die gesuchte Funktion $g(n)$ existieren muss, war Ihre Suche mit Sicherheit nicht erfolgreich. Erklären Sie, warum Sie keine Chance hatten, die Funktion $g(n)$ zu finden.

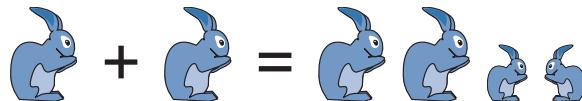
Aufgabe 7.8

Webcode
7490

Vielleicht erinnern Sie sich an das Anagramm

1 1 2 3 5 8 13 21 LEONARDO DA VINCI! THE MONA LISA!

aus dem packenden Thriller *Sakrileg*. Die Zahlen sind Teil der *Fibonacci-Folge*, die der italienische Mathematiker Leonardo da Pisa, gennant Fibonacci, in der ersten Hälfte des dreizehnten Jahrhunders aufstellte. Was die meisten Leser von Dan Brown's Roman nicht wissen: Die Folge ist das Ergebnis von Fibonaccis Untersuchungen über das Wachstum von Kaninchenpopulationn. Er ging dabei von folgenden Annahmen aus: Zu Beginn existiert ein einziges Paar geschlechtsreifer Kaninchen. Jedes neugeborene Paar wird im zweiten Monat geschlechtsreif und jedes geschlechtsreife Paar gebärt pro Monat ein weiteres Paar.



Offensichtlich lässt sich die Folge mit der nachstehenden Funktion berechnen:

fibonacci.c

```
int fibonacci( int n )
{
    if ( n == 1 || n == 2 ) return 1;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

1
2
3
4
5

Die Laufzeit, die der Funktionsaufruf $\text{fibonacci}(n)$ konsumiert, bezeichnen wir mit $T(n)$. Die rekursive Struktur erlaubt es uns, das asymptotische Wachstum von $T(n)$ in Form einer *Rekurrenzgleichung* anzugeben:

$$T(1) = 1, \quad T(2) = 1, \quad T(n) = T(n-1) + T(n-2)$$

- a) Welcher Zusammenhang besteht zwischen $T(n)$ und $\text{fibonacci}(n)$?
- b) Versuchen Sie, $T(n)$ mit einer geschlossenen Formel zu berechnen.
- c) Bestimmen Sie mit der ermittelten Formel die Laufzeitkomplexität des Algorithmus.
- d) Kann die dynamische Programmierung helfen, die Laufzeit zu verbessern?

In dieser Aufgabe geht es um die Berechnung des Produkts zweier quadratischer Matrizen $X, Y \in \mathbb{R}^{n \times n}$. Um die Betrachtungen nicht komplizierter als nötig zu gestalten, sei n eine Zweierpotenz. In diesem Fall lassen sich X und Y und die Produktmatrix Z in vier gleich große Fragmente zerlegen:

$$X := \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix}, \quad Y := \begin{pmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{pmatrix}, \quad Z := \begin{pmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{pmatrix}$$

Basierend auf dieser Zerlegung können wir Z wie folgt ausrechnen:

$$Z_{11} := X_{11} \cdot Y_{11} + X_{12} \cdot Y_{21}$$

$$Z_{12} := X_{11} \cdot Y_{12} + X_{12} \cdot Y_{22}$$

$$Z_{21} := X_{21} \cdot Y_{11} + X_{22} \cdot Y_{21}$$

$$Z_{22} := X_{21} \cdot Y_{12} + X_{22} \cdot Y_{22}$$

Im Jahre 1969 konnte der deutsche Mathematiker Volker Strassen zeigen, dass sich die Berechnung optimieren lässt [94]. Der *Strassen-Algorithmus* beginnt mit der Vorberechnung mehrerer Hilfsmatrizen:

$$H_1 := (X_{11} + X_{22}) \cdot (Y_{11} + Y_{22})$$

$$H_2 := (X_{21} + X_{22}) \cdot Y_{11}$$

$$H_3 := X_{11} \cdot (Y_{12} - Y_{22})$$

$$H_4 := X_{22} \cdot (Y_{21} - Y_{11})$$

$$H_5 := (X_{11} + X_{12}) \cdot Y_{22}$$

$$H_6 := (X_{21} - X_{11}) \cdot (Y_{11} + Y_{12})$$

$$H_7 := (X_{12} - X_{22}) \cdot (Y_{21} + Y_{22})$$

Anschließend werden die vier Komponenten der Produktmatrix wie folgt erzeugt:

$$Z_{11} := H_1 + H_4 - H_5 + H_7$$

$$Z_{12} := H_3 + H_5$$

$$Z_{21} := H_2 + H_4$$

$$Z_{22} := H_1 - H_2 + H_3 + H_6$$

- Zeigen Sie, dass der Strassen-Algorithmus tatsächlich das Produkt $X \cdot Y$ berechnet.
- Bestimmen Sie die asymptotische Laufzeitkomplexität des Strassen-Algorithmus. Vergleichen Sie die von Ihnen ermittelte Komplexitätsklasse mit jener der rekursiven, aber unoptimierten Variante.
- Ist der Strassen-Algorithmus immer besser als die Standardmethode, die Sie aus dem Mathematikunterricht kennen? Sehen Sie eine Möglichkeit, beide Algorithmen zu kombinieren?

Aufgabe 7.9

Webcode
7789

Aufgabe 7.10

**Webcode
7437**

In Abschnitt 2.4.1 haben wir bewiesen, dass für alle n mit $n \geq 4$ die Abschätzung

$$2^n < n! < n^n$$

gilt. Welche der folgenden Beziehungen lassen sich aus diesem Ergebnis ableiten?

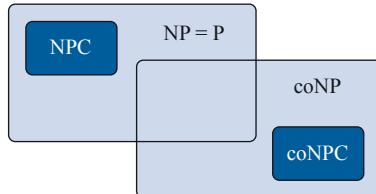
- a) $2^n = O(n!)$
- b) $2^n = \Omega(n^n)$
- c) $2^n = o(n!)$
- d) $2^n = \omega(n^n)$
- e) $n! = O(2^n)$
- f) $n! = \Omega(n^n)$
- g) $n! = o(2^n)$
- h) $n! = \omega(n^n)$
- i) $n^n = O(2^n)$
- j) $n^n = \Omega(n!)$
- k) $n^n = o(2^n)$
- l) $n^n = \omega(n!)$

Aufgabe 7.11

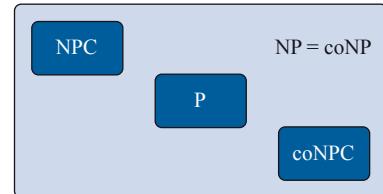
**Webcode
7189**

In diesem Kapitel haben Sie mehrere Komplexitätsklassen zusammen mit ihren Inklusionsbeziehungen kennen gelernt. Wie die einzelnen Klassen genau in Beziehung stehen, ist heute noch nicht in allen Einzelheiten bekannt. Welche der abgebildeten Inklusionskonstellationen sind aus heutiger Sicht möglich?

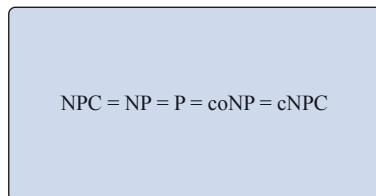
■ Konstellation 1



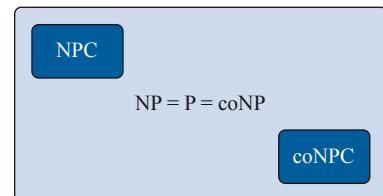
■ Konstellation 3



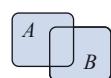
■ Konstellation 2



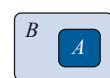
■ Konstellation 4



Interpretieren Sie die Venn-Diagramme nach dem folgenden Schema:



$A \setminus B \neq \emptyset, B \setminus A \neq \emptyset, A \cap B \neq \emptyset$



$A \subset B, B \setminus A \neq \emptyset$

In dieser Aufgabe wollen wir 3SAT auf Klauselmengen mit einer beliebigen, aber festen Anzahl von Literalen erweitern:

Aufgabe 7.12

Webcode
7955

Definition 7.21 (k -SAT)

Das Problem k -SAT lautet wie folgt:

- Gegeben: n aussagenlogische Klauseln mit jeweils k Literalen.
- Gefragt: Gibt es eine erfüllende Belegung?

Beweisen Sie die folgenden Aussagen:

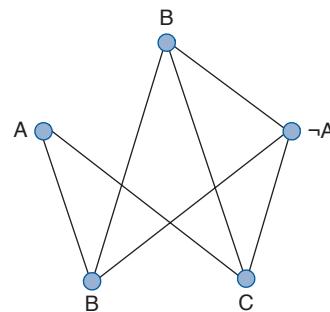
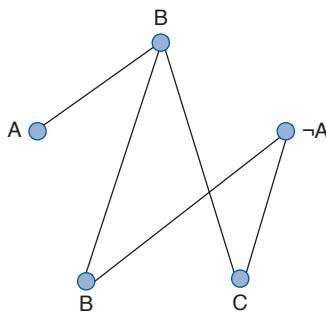
- a) k -SAT ist für $k \geq 3$ NP-vollständig.
- b) k -SAT ist für $k < 3$ polynomiell deterministisch lösbar.

In diesem Kapitel haben Sie gelernt, wie die NP-Vollständigkeit von CLIQUE durch die Reduktion von 3SAT bewiesen werden kann. Ausgehend von einer n -elementigen Klauselmenge M haben wir einen Graphen G konstruiert, der genau dann eine Clique der Größe n enthält, wenn M erfüllbar ist. Die resultierenden Graphen könnten z. B. wie folgt aussehen:

Aufgabe 7.13

Webcode
7230

 ■ G_1

 ■ G_2


- a) Ihre Aufgabe ist es, den Konstruktionsprozess umzukehren. Geben Sie für G_1 und G_2 die Ausgangsmengen M_1 und M_2 an.
- b) Sind M_1 und M_2 für alle Graphen G_1 und G_2 immer eindeutig bestimmt?

Aufgabe 7.14**Webcode
7609**

In diesem Kapitel haben wir den Begriff der zeitkonstruierbaren Funktion und den Begriff der platzkonstruierbaren Funktion eingeführt.

- Begründen Sie, warum jede zeitkonstruierbare Funktion auch platzkonstruierbar ist.
- Geben Sie eine Funktion an, die platzkonstruierbar, aber nicht zeitkonstruierbar ist.

Aufgabe 7.15**Webcode
7617**

Satz 7.5 besagt, dass die Menge $\text{TIME}(f(n))$ ausschließlich entscheidbare Sprachen enthält, wenn die Funktion f zeitkonstruierbar ist. Wir wollen versuchen, die Voraussetzung dieses Satzes abzuschwächen.

- Bleibt der Satz wahr, wenn wir die Voraussetzung der Zeitkonstruierbarkeit durch die schwächere Voraussetzung der Berechenbarkeit ersetzen?
- Bleibt der Satz wahr, wenn wir die Voraussetzung der Zeitkonstruierbarkeit ersatzlos streichen?

Aufgabe 7.16**Webcode
7345**

Welche der folgenden Aussagen sind richtig?

- $P = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k)$
- $P = \bigcup_{k, a_0, \dots, a_k \in \mathbb{N}} \text{TIME}(a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \dots + a_1 \cdot n + a_0)$
- $\text{EXP} = \bigcup_{c, k \in \mathbb{N}} \text{TIME}(c^{n^k})$
- $\text{EXP} = \bigcup_{k \in \mathbb{N}} \text{TIME}(k^{n^k})$

Anhang

A Notationsverzeichnis

B Abkürzungsverzeichnis

C Glossar

Literaturverzeichnis

Namensverzeichnis

Sachwortverzeichnis



A Notationsverzeichnis

Mathematik

\emptyset oder $\{\}$	Leere Menge
$M, M_i, N, N_i \dots$	Menge
$\{a, b, c, \dots\}$	Menge mit den Elementen a, b, c, \dots
$a \in M$	a ist enthalten in M
$a \notin M$	a ist nicht enthalten in M
\overline{M}	Komplementärmenge
$M_1 \setminus M_2$	Differenzmenge
$M_1 \subseteq M_2$ oder $M_2 \supseteq M_1$	M_1 ist Teilmenge von M_2
$M_1 \subset M_2$ oder $M_2 \supset M_1$	M_1 ist Teilmenge von M_2 und $M_1 \neq M_2$
$M_1 \cup M_2$	Vereinigungsmenge von M_1 und M_2
$M_1 \cap M_2$	Schnittmenge von M_1 und M_2
$M_1 \times M_2$	Kartesisches Produkt (Kreuzprodukt) von M_1 und M_2
M^n	$M \times M \times \dots \times M$ (n -mal)
$ M $	Kardinalität von M
2^M	Potenzmenge (Menge aller Teilmengen von M)
\mathbb{N}	Menge der natürlichen Zahlen $(0, 1, 2, 3, \dots)$
\mathbb{N}^+	Menge der positiven ganzen Zahlen $(1, 2, 3, 4, \dots)$
\mathbb{Z}	Menge der ganzen Zahlen
\mathbb{Q}	Menge der rationalen Zahlen (Brüche)
\mathbb{R}	Menge der reellen Zahlen
$x \bmod y$	Ganzzahlig Divisionsrest von $\frac{x}{y}$
R, S, \dots	Relation
$x \sim_R y$	x und y stehen in der Relation R zueinander
$x \not\sim_R y$	x und y stehen nicht in der Relation R zueinander
$R \cdot S$	Relationenprodukt
R^{-1}	Inverse Relation
R^+	Transitive Hülle von R
R^*	Reflexiv transitive Hülle von R
$[x]_\sim$	Äquivalenzklasse $(= \{y \mid x \sim y\})$
f, g, h, \dots	Funktion
$\text{succ}(x)$	Nachfolger der natürlichen Zahl x ($\text{succ}(x) = x + 1$)
$a \uparrow^n b$	Up-Arrow-Notation zur Beschreibung großer Zahlen
$\text{ack}(n, m)$	Ackermann-Funktion
π	Kreiszahl oder (Cantor'sche) Paarungsfunktion

Logik

0	Logisch falsch (<i>false</i>)
1	Logisch wahr (<i>true</i>)
$A_1, A_2, \dots, A, B, C, \dots, X, Y, Z$	Aussagenlogische Variable
$\neg A$ oder \bar{A}	Negation (nicht A)
$(\neg)A$ oder L	Literal (entweder A oder $\neg A$)
$A \wedge B$	Konjunktion (A und B)
$A \vee B$	Disjunktion (A oder B)
$A \rightarrow B$	Implikation (aus A folgt B)
$A \leftrightarrow B$ oder $A \oplus B$	Antivalenz (entweder A oder B)
$A \leftrightarrow B$	Äquivalenz (A genau dann, wenn B)
F, G, H, \dots	Formeln
$F = G$	Gleichheit (F und G sind syntaktisch gleich)
$F \equiv G$	Äquivalenz (F und G sind logisch äquivalent)
$F \equiv_E G$	Erfüllbarkeitsäquivalenz (F ist genau dann erfüllbar, wenn G erfüllbar ist)
I	Interpretation oder Belegung
$I \models F$	Modellrelation (I ist ein Modell für F)
\vdash_K oder \vdash	Ableitungsrelation (im Kalkül K)
$\{A, B, C\}$	Klausel (entspricht $A \vee B \vee C$)
\square	Leere Klausel (entspricht 0)
x, y, z, \dots	Prädikatenlogische Variable
$f(x), g(x, f(y)), \dots$	Prädikatenlogischer Term
P, Q, R, \dots	Prädikat
\forall, \exists	Allquantor („für alle ...“), Existenzquantor („es gibt ein ...“)
$\Sigma = (U, I)$	Prädikatenlogische Signatur
U	Grundmenge, Universum, Individuenbereich
I	Interpretation
$HU(F)$	Herbrand-Universum der Formel F
$G(F)$	Grundinstanzen der Formel F

Formale Sprachen

G	Grammatik
V	Variablenmenge (Menge aller Nonterminale)
A, B, C, \dots	Nonterminale
Σ	Terminalalphabet
$\sigma, \sigma_0, \sigma_1, \dots$	Terminale
ε	Leeres Wort
Σ^*	Mit Symbolen aus Σ bildbare Zeichenketten (Kleene'sche Hülle)
Σ^+	Entspricht $\Sigma^* \setminus \{\varepsilon\}$
ω	Wort ($\omega \in \Sigma^*$)

$ \omega $	Wortlänge (Anzahl der Symbole in ω)
P	Produktionenmenge
L, L_1, L_2, \dots	Sprache
$\mathcal{L}(G)$	Von G erzeugte Sprache
\Rightarrow_G	Von G induzierte Ableitungsrelation
\mathcal{L}_n	Menge der Typ- n -Sprachen
Reg_Σ	Menge der regulären Ausdrücke

Endliche Automaten

A, A_1, A_2, \dots	Automat
$S = \{s_0, s_1, \dots\}$	Zustandsmenge
Σ	Eingabealphabet
$\sigma, \sigma_0, \sigma_1, \dots$	Eingabezeichen
ω	Eingabewort
δ	Zustandsübergangsfunktion
E	Menge der Endzustände ($E \subseteq S$)
\rightarrow_A	Übergangsrelation
\sim_k	k -Äquivalenz (zwischen Zuständen)
\sim	Bisimulation ($\sim \subseteq S \times S$)
ε	Leeres Wort bzw. ε -Übergang
$\ s\ _\varepsilon$	ε -Hülle des Zustands s
K, K_1, K_2, \dots	Kellerautomat
Γ	Kellaralphabet
γ	Kellerzeichen
\perp	Kellerbodensymbol
T, T_1, T_2, \dots	Transduktor
Π	Ausgabealphabet
π, π_0, π_1, \dots	Ausgabezeichen
λ	Ausgabefunktion
P, P_1, P_2, \dots	Petri-Netz
κ	Konfiguration
I	Inzidenzmatrix
Ψ	Parikh-Vektor
Z	Zellmenge (eines zellulären Automaten)
v	Nachbarschaftsfunktion

Berechenbarkeitstheorie

$\text{succ}(x_i)$	Nachfolgerberechnung (+1)
$\text{pred}(x_i)$	(Gesättigte) Vorgängerberechnung ($\max\{0, x_i - 1\}$)

\coloneqq	Zuweisungsoperator
$:$	Kompositionsoperator
loop do end	Loop-Schleifenkonstrukt
while do end	While-Schleifenkonstrukt
if goto	Bedingter Sprung
halt	Stoppbefehl
v	Speichervektor
δ	(Zustands)übergangsfunktion
$p_i^n(a_1, \dots, a_n)$	Projektion ($p_i^n(a_1, \dots, a_n) := a_i$)
μ	Rekursionsoperator
T, T_1, T_2, \dots	Turing-Maschine
$S = \{s_0, s_1, \dots\}$	Zustandsmenge
Σ	Eingabealphabet
Π	Bandalphabet
ω, v	Bandinhalt (als Wort)
$\sigma, \sigma_0, \sigma_1, \dots, \rho, \rho_0, \rho_1, \dots$	Bandinhalt (einzelne Zeichen)
\square	Blank-Symbol
$\leftarrow, \rightarrow, \circlearrowleft$	Kopfänderung einer Turing-Maschine (links, rechts, nicht bewegen)
π	Codierung, (Cantor'sche) Paarungsfunktion
$\lceil T \rceil$	Gödelnummer der Turing-Maschine T
χ, χ'	Charakteristische Funktion, partielle charakteristische Funktion
$L \leq L'$	L ist reduzierbar auf L'
$B(n)$	Biberfunktion
L_D	Diagonalsprache

Komplexitätstheorie

$f(n) = O(g(n))$	f wächst höchstens so schnell wie g
$f(n) = \Omega(g(n))$	f wächst mindestens so schnell wie g
$f(n) = \Theta(g(n))$	f wächst genauso schnell wie g
$f(n) = o(g(n))$	f wächst langsamer als g
$f(n) = \omega(g(n))$	f wächst schneller als g
t_T	Laufzeitfunktion
s_T	Bandplatzfunktion
P	Deterministisch in polynomieller Zeit entscheidbare Probleme
NP	Nichtdeterministisch in polynomieller Zeit entscheidbare Probleme
PSPACE	Deterministisch in polynomiellem Platzbedarf entscheidbare Probleme
NPSPACE	Nichtdeterministisch in polynomiellem Platzbedarf entscheidbare Probleme
EXP	Deterministisch in exponentieller Zeit entscheidbare Probleme
NEXP	Nichtdeterministisch in exponentieller Zeit entscheidbare Probleme
coP, coNP, ...	Komplementäre Komplexitätsklasse
$L \leq_{poly} L'$	L ist polynomiell reduzierbar auf L'

B

Abkürzungsverzeichnis

ACM	Association for Computing Machinery
AKS	Agrawal-Kayal-Saxena(-Primzahltest)
BCD	Binary-Coded Decimal
BIP	Bruttoinlandsprodukt
BPCP	Binäres Post'sches Korrespondenzproblem
CMI	Clay Mathematics Institute
COBOL	COmmon Business-Oriented Language
CTL	Computation Tree Logic
CYK	Cocke-Younger-Kasami(-Algorithmus)
DEA	Deterministischer endlicher Automat
DET	Deterministischer endlicher Transduktor
DF	Disjunktive Form
DNF	Disjunktive Normalform
DNA, DNS	Desoxyribonukleinsäure
DT	Deduktionstheorem
DVD	Digital Versatile Disc
ENIAC	Electronic Numerical Integrator And Computer
EXP	(Deterministisch) exponentiell
FFT	Fast Fourier Transformation
FIFO	First In, First Out
GNU	GNU is not Unix
GUI	Graphical User Interface
HOL	Higher-Order Logic
HTML	HyperText Markup Language
HU	Herbrand-Universum
KF	Konjunktive Form
KNF	Konjunktive Normalform
LBA	Linear beschränkter Akzeptor
LIFO	Last In, First Out
LTL	Linear Temporal Logic
MP	Modus Ponens
MPCP	Modifiziertes Post'sches Korrespondenzproblem
NEA	Nichtdeterministischer endlicher Akzeptor
NEXP	Nichtdeterministisch exponentiell
NP	Nichtdeterministisch polynomiell
NPC	NP Complete (NP-vollständig)

NPSPACE	Nichtdeterministisch polynomieller Platzverbrauch
P	(Deterministisch) polynomiell
PCP	Post'sches Korrespondenzproblem
PDA	Pushdown Automaton (Kellerautomat)
PROLOG	PROgramming in LOGic
PSPACE	(Deterministisch) polynomieller Platzverbrauch
RAM	Random Access Machine oder Random Access Memory
ROM	Read-Only Memory
RNA, RNS	Ribonukleinsäure
RSA	Rivest-Shamir-Adleman(-Kryptosystem)
WZK	Wolf-Ziege-Kohlkopf(-Problem)
SAT	(Boolean) SATisfiability (problem)
SIGACT	Special Interest Group on Algorithms and Computation Theory
TM	Turing-Maschine
UC	University of California
ZA	Zellulärer Automat
ZF	Zermelo-Fraenkel(-Mengenlehre)
ZFC	Zermelo-Fraenkel(-Mengenlehre) mit Auswahlaxiom (axiom of Choice)

C Glossar

3SAT

7.3.4

Spezielle Variante des SAT -Problems. Für eine gegebene Menge von Klauseln mit je drei Literalen ist zu entscheiden, ob eine erfüllende Belegung existiert. Genau wie SAT gehört auch diese Fragestellung zu den NP-vollständigen Problemen. Aufgrund seiner Einfachheit wird 3SAT gerne dazu verwendet, um andere Probleme mit dem Mittel der $\text{polynomialen Reduktion}$ ebenfalls als NP-vollständig zu identifizieren.

Abzählbarkeit

2.3.3

Eine Menge M heißt abzählbar, wenn sie die gleiche Mächtigkeit wie die Menge der natürlichen Zahlen besitzt. Dies ist genau dann der Fall, wenn jedes Element von M eineindeutig einer natürlichen Zahl zugeordnet werden kann. Jede aufzählbare Menge mit unendlich vielen Elementen ist auch abzählbar, nicht jedoch umgekehrt.

Ackermann-Funktion

2.3.2

Beispiel einer Funktion, die While-berechenbar , aber nicht Loop-berechenbar ist. Ihre Existenz beweist, dass die Loop-Sprache nicht an die Stärke der meisten anderen in diesem Buch diskutierten $\text{Berechnungsmodelle}$ heran kommt.

Akzeptor

5.2

Überbegriff für eine Reihe $\text{endlicher Automaten}$, die sich weiter in deterministische (DEA) und nichtdeterministische (NEA) Akzeptoren untergliedern lassen. In beiden Fällen bestehen die Automaten aus einer Menge von Zuständen, einem Eingabealphabet, einer Zustandsübergangsfunktion, einer Reihe von End- oder Finalzuständen sowie einem dedizierten Startzustand. In jedem Verarbeitungsschritt nimmt der Automat ein einzelnes Eingabzeichen entgegen und geht in einen neuen Zustand über. Die Eingabesequenz wird genau dann akzeptiert, wenn die Verarbeitung in einem Finalzustand endet.

Allgemeingültigkeit

3.1.1

Eine allgemeingültige Formel besitzt die Eigenschaft, dass jede Interpretation ein Modell ist. Eine solche Formel ist damit immer wahr, unabhängig davon, wie wir ihre Bestandteile interpretieren. Der Begriff wird synonym mit dem Begriff der Tautologie verwendet.

Allgemeinster Unifikator

3.2.3.1

Spezieller Unifikator mit der Eigenschaft, dass sich alle Unifikatoren durch die Anwendung einer weiteren Substitution erzeugen lassen.

Antinomie

1.2.1

Spezielle Art des logischen Widerspruchs, in der die Richtigkeit einer Aussage ihre eigene Falschheit zur Folge hat und umgekehrt. In formalen Systemen führen Antinomien zu Inkonsistenzen und damit zur völligen Unbrauchbarkeit der zugrunde liegenden Axiome und Schlussregeln. Zu den bekanntesten Vertretern gehören das Barbier-Paradoxon sowie die $\text{Russell'sche Antinomie}$ der naiven Mengenlehre.

Äquivalenzproblem

4.1

Wichtige Problemstellung aus dem Bereich der formalen Sprachen . Hinter dem Äquivalenzproblem verbirgt sich die Frage, ob zwei Sprachen L_1 und L_2 aus den gleichen Wörtern bestehen. Mit anderen Worten: Gilt $L_1 = L_2$?

Asymptotisches Wachstum

7.1.1

Beschreibt den Werteverlauf einer Funktion $f(n)$ für den Fall $n \rightarrow \infty$. Wie sich die Funktion $f(n)$ auf einem beliebigen, aber endlichen Anfangsstück verhält, spielt für das asymptotische Wachstum keine Rolle. Von konstanten Faktoren wird ebenfalls abstrahiert, so dass die Wachstumsanalyse zu einer Einteilung von Funktionen in verschiedene $\text{Komplexitätsklassen}$ führt. Das asymptotische Wachstum ist die Grundla-

ge, um das Laufzeitverhalten und den Speicherplatzverbrauch von Algorithmen auf einer abstrakten Ebene zu beschreiben.

Aussagenlogik

3.1

Die Aussagenlogik beschäftigt sich mit *atomaren Aussagen* („Es regnet“, „Die Straße ist nass“) und den Beziehungen, die zwischen solchen Aussagen bestehen („Wenn es regnet, dann ist die Straße nass“). Die Bedeutung der Aussagenlogik ist zweigeteilt. Zum einen ist sie als Teilmenge in nahezu allen anderen Logiken enthalten, unter anderem auch in der *Prädikatenlogik* und den *Logiken höherer Stufe*. Zum anderen spielt sie eine wichtige Rolle im Hardware-Entwurf, da sich jede kombinatorische Digitalschaltung mithilfe aussagenlogischer Formeln beschreiben lässt.

Automatenminimierung

5.2.2

Verfahren, um die Anzahl der Zustände eines *endlichen Automaten* zu verringern, ohne die nach außen sichtbare Funktion zu verändern. Ein Automat heißt reduziert, wenn kein funktional äquivalenter Automat existiert, der weniger Zustände besitzt.

Automatensynthese

5.6.3

Bezeichnet die Umsetzung eines *endlichen Automaten* in ein Schaltungsmodell, das aus Speicherelementen und Logikgattern besteht. Die Automatensynthese ist ein wesentlicher Arbeitsschritt im Entwurfsprozess digitaler Hardware-Schaltungen.

Axiom

3.1.3.1

Zentraler Bestandteil eines *Hilbert-Kalküls*. Axiome bilden den Ausgangspunkt eines formalen Beweises und müssen per Definition nicht selbst abgeleitet werden. In einem formalen Beweis wird die zu zeigende Aussage durch die sukzessive Anwendung fest definierter *Schlussregeln* aus den Axiomen deduziert.

Backus-Naur-Form

4.4.2.2

Spezielle Notation zur Beschreibung *kontextfreier Sprachen*. Die Backus-Naur-Form wurde erstmals zur Spezifikation der Sprache Algol60 eingesetzt und hat sich heute als De-facto-Standard für die Syntaxdefinition von Programmiersprachen etabliert.

Barbier-Paradoxon

2.5

Bildhafte *Antinomie*, die das Problem der Selbstbezüglichkeit anschaulich demonstriert. Der Barbier von Sevilla rasiert alle Männer von Sevilla, die sich nicht selbst rasieren. Die intuitive Annahme, dass der Barbier sich entweder selbst oder nicht selbst rasiert, wird durch einen Ringschluss stets zu Widerspruch geführt. Das Paradoxon entspricht im Kern der *Russell'schen Antinomie*, mit der die Mathematik zu Beginn des zwanzigsten Jahrhunderts in ihre bislang größte Krise gestürzt wurde.

Berechenbarkeit

6.1

Eine Funktion f heißt berechenbar, falls ein systematisches Verfahren existiert, das für alle Eingaben x nach endlich vielen Schritten terminiert und den Funktionswert $f(x)$ als Ausgabe liefert. Was wir unter einem systematischen Verfahren zu verstehen haben, wird durch die Definition eines *Berechnungsmodells* formal festgelegt. Der Begriff der berechenbaren Funktion ist eng mit dem Begriff der *Entscheidbarkeit* gekoppelt.

Berechenbarkeitstheorie

Kapitel 6

Teilgebiet der theoretischen Informatik, das sich mit den formalen Grundlagen und den Grenzen der algorithmischen Methode befasst. Im Gegensatz zur *Komplexitätstheorie* steht die prinzipielle *Berechenbarkeit* einer Funktion im Vordergrund und nicht die Effizienz der Lösung. Eine zentrale Erkenntnis der Berechenbarkeitstheorie ist die Existenz unberechenbarer Funktionen (*Halteproblem*).

Berechnungsmodell

6.1

Formales Konstrukt, um den Begriff der *Berechenbarkeit* mathematisch präzise zu erfassen. In der Vergangenheit wurden zahlreiche Berechnungsmodelle postuliert, die sich von außen betrachtet erheblich voneinander unterscheiden. Einige besitzen durch und durch mathematischen Charakter, während sich andere sehr nahe an der Hardware-Architektur realer Computersysteme orientieren. Die *primitiv-rekursiven Funktionen*, die *μ -rekursiven Funktionen* und das Lambda-Kalkül gehören zur ersten Gruppe, die *Loop-, Goto-* und *While-Sprache* sowie die *Turing-Maschine* und die *Register-Maschine* zur zweiten.

Bisimulation

☞ 5.2.2

Spezielle Äquivalenzrelation auf der Zustandsmenge eines *endlichen Automaten*. Zwei Zustände s_1 und s_2 sind genau dann bisimulativ zueinander, wenn sich der Automat für alle zukünftigen Eingaben gleich verhält, unabhängig davon, ob er sich in s_1 oder s_2 befindet. Die Bestimmung bisimulativer Zustände ist eine Kernaufgabe in der *Automatenminimierung*.

Charakteristische Funktion

☞ 6.3

Mathematisches Konstrukt, das den Zusammenhang zwischen der *Berechenbarkeit* einer Funktion und der *Entscheidbarkeit* einer Menge herstellt. Eine Menge M ist genau dann entscheidbar, wenn die charakteristische Funktion χ_M berechenbar ist. Ein Spezialfall ist die partielle charakteristische Funktion, die direkt zum Begriff der *Semi-Entscheidbarkeit* führt.

Chomsky-Hierarchie

☞ 4.2

Rangfolge von Klassen formaler *Grammatiken*. Anhand der Struktur der Produktionsregeln wird eine Grammatik einer von vier Klassen zugeordnet. Typ-0-Grammatiken unterliegen keinerlei Einschränkung und definieren die Sprachklasse der rekursiv aufzählbaren Sprachen. Typ-1- und Typ-2-Grammatiken erzeugen die *kontextsensitiven* und die *kontextfreien Sprachen*. *Reguläre Sprachen* werden von Typ-3-Grammatiken erzeugt. Zwischen den Sprachklassen besteht eine Inklusionsbeziehung, d. h., jede Typ-3-Sprache ist auch eine Typ-2-Sprache und so fort. Ferner existieren in jeder Klasse Sprachen, die in der eingebetteten Klasse nicht enthalten sind.

Chomsky-Normalform

☞ 4.4.2.1

Spezielle Darstellungsform für Typ-2-Grammatiken. Zu jeder *kontextfreien Sprache* L existiert eine *Grammatik* in Chomsky-Normalform, die L erzeugt. Die Chomsky-Normalform ist eng verwandt mit der *Greibach-Normalform*.

Church'sche These

☞ 6.2

Von Alonzo Church aufgestellte These über den Berechenbarkeitsbegriff. Sie besagt, dass die Klasse der Turingberechenbaren Funktionen mit der Klasse der intuitiv be-

rechenbaren Funktionen übereinstimmt. Mit anderen Worten: Jede Funktion, die überhaupt in irgendeiner Weise berechenbar ist, kann auch durch eine *Turing-Maschine* berechnet werden. Die These wird durch die Beobachtung gestützt, dass alle hinreichend berechnungsstarken Modelle wie die *While-Sprache*, die *Goto-Sprache*, die *μ -Rekursion* oder die *Registermaschine* exakt den gleichen Berechenbarkeitsbegriff definieren.

Co-Komplexität

☞ 7.2.4

Zu jeder *Komplexitätsklasse* C existiert die komplementäre Komplexitätsklasse $\text{co}C$. Eine Sprache L gehört genau dann zu $\text{co}C$, wenn das Komplement \bar{L} zu C gehört. Die co-Komplexität spielt nur im Bereich nichtdeterministischer Komplexitätsklassen eine Rolle. Für jede deterministische Komplexitätsklasse C gilt die Beziehung $C = \text{co}C$.

CYK-Algorithmus

☞ 4.4.4

Nach John Cocke, Daniel Younger und Tadao Kasami benannter Algorithmus zur Lösung des *Wortproblems kontextfreier Sprachen*. Der Algorithmus basiert auf dem Verfahren der *dynamischen Programmierung*. Ausgangspunkt ist eine *Grammatik* G in *Chomsky-Normalform*.

DEA

☞ 5.2

Deterministischer endlicher *Akzeptor*. Im Gegensatz zu seinem nichtdeterministischen Pendant (*NEA*) ist der ausführende Zustandsübergang für jedes Eingabezeichen eindeutig definiert. Die von DEAs und NEAs akzeptierte Sprachklasse ist identisch und entspricht der Klasse der *regulären Sprachen*.

Diagonalisierung

☞ 2.3.3

Mathematisches Beweisverfahren, um Aussagen über die Mächtigkeit zweier Mengen zu verifizieren. Unter anderem kann mit der Methode die Gleichmächtigkeit von \mathbb{N} und \mathbb{Q} gezeigt (erstes Cantor'sche Diagonalargument) oder die Menge \mathbb{R} als überabzählbar entlarvt werden (zweites Cantor'sches Diagonalargument). In Kapitel 6 wurde die Diagonalisierung eingesetzt, um die *Unentscheidbarkeit* des allgemeinen *Halteproblems* zu beweisen.

Dirichlet'sches Schubfachprinzip
☞ 3.1.1

Mathematische Schlussregel aus dem Bereich der diskreten Mathematik. Plakativ beschreibt sie den folgenden Sachverhalt: Werden $n+1$ Gegenstände auf n Fächer verteilt, so enthält mindestens ein Fach mehr als einen Gegenstand. In der Literatur wird das Dirichlet'sche Schubfachprinzip auch als *Taubenschlagprinzip* bezeichnet, in Anlehnung an den englischen Begriff *pigeonhole principle*.

Disjunktive Form
☞ 3.1.2

Eine Logikformel liegt in disjunktiver Form vor, wenn sie als Disjunktion von Konjunktionen aufgebaut ist.

Disjuktive Normalform
☞ 3.1.2

Eine Logikformel liegt in disjunktiver Normalform vor, wenn sie als Disjunktion von ☞ Mintermen aufgebaut ist und alle Minterme paarweise verschieden sind.

Dynamische Programmierung
☞ 4.4.4

Spezielles Konstruktionsprinzip für Algorithmen. Die dynamische Programmierung kann immer dann eingesetzt werden, wenn sich die optimale Lösung eines Problems aus den optimalen Lösungen seiner Teilprobleme zusammensetzen lässt. Ein bekannter Algorithmus, der nach diesem Prinzip arbeitet, ist der ☞ CYK-Algorithmus zur Lösung des ☞ Wortproblems ☞ kontextfreier Sprachen. Auch das ☞ Rucksackproblem lässt sich mit dem Prinzip der dynamischen Programmierung effizient lösen.

Endlicher Automat
☞ Kapitel 5

Mathematisches Modell zur Beschreibung zustandsbasierter Systeme. Entsprechend ihrer Funktionsweise werden endliche Automaten in ☞ Akzeptoren und ☞ Transduktoren eingeteilt. Akzeptoren sind ein wichtiges Hilfsmittel für die Analyse ☞ formaler Sprachen. Transduktoren sind die theoretische Grundlage für die Modellierung jeglicher Computer-Hardware.

Endlichkeitsproblem
☞ 4.1

Wichtige Problemstellung aus dem Bereich der ☞ formalen Sprachen. Hinter dem Endlichkeitsproblem verbirgt sich die

Frage, ob eine Sprache L über einem endlichen Wortschatz verfügt. Mit anderen Worten: Gilt $|L| < \infty$?

Entscheidbarkeit
☞ 6.3

Eine Menge M heißt entscheidbar, falls die ☞ charakteristische Funktion ☞ berechenbar ist. Für eine entscheidbare Menge M existiert ein systematisches Verfahren, das für alle Eingaben ω nach endlicher Zeit beantwortet, ob ω ein Element von M ist oder nicht. Heute wissen wir, dass zahllose ☞ unentscheidbare Mengen existieren. Kurzum: Der algorithmischen Methode sind unüberwindbare Grenzen gesetzt.

Epsilon-Übergang
☞ 5.3.3

Spontaner Zustandsübergang in einem ☞ endlichen Automaten, der kein Eingabezeichen konsumiert. Epsilon-Übergänge (kurz ε -Übergänge) erleichtern die Konstruktion von ☞ Akzeptoren für viele ☞ reguläre Sprachen, führen aber zu keiner grundlegenden Veränderung des Automatenmodells. Jeder ε -Automat lässt sich in einen äquivalenten Automaten übersetzen, der keine ε -Übergänge mehr besitzt.

Erfüllbarkeit
☞ 3.1.1

Eine erfüllbare Formel besitzt die Eigenschaft, dass mindestens eine ☞ Interpretation auch ein ☞ Modell ist. Eine wichtige Teilmenge der erfüllbaren Formeln sind die ☞ Tautologien.

Erfüllbarkeitsäquivalenz
☞ 3.2.2

Zwei Formeln heißen erfüllbarkeitsäquivalent, wenn aus der ☞ Erfüllbarkeit der einen die Erfüllbarkeit der anderen folgt. Zwei äquivalente Formeln sind immer auch erfüllbarkeitsäquivalent, aber nicht umgekehrt.

EXP
☞ 7.2.3

Die ☞ Komplexitätsklasse EXP enthält alle Sprachen, die von deterministischen ☞ Turing-Maschinen in exponentieller Zeit entschieden werden können.

Formale Sprache
☞ 4.1

Menge von Wörtern, die über einem endlichen Alphabet Σ gebildet werden. Formale Sprachen lassen sich mithilfe von ☞ Grammatiken generativ erzeugen und anhand der

☞ **Chomsky-Hierarchie** in verschiedene Klassen einteilen. Wichtige Fragestellungen in der Theorie der formalen Sprachen sind das ☞ **Wortproblem**, das ☞ **Leerheitsproblem**, das ☞ **Äquivalenzproblem** und das ☞ **Endlichkeitsproblem**.

Formales System

☞ 3.1.3

In diesem Buch wird der Begriff des formalen Systems synonym mit dem Begriff des ☞ **Kalküls** verwendet. Leider hat sich diese Terminologie in der Literatur nicht einheitlich durchgesetzt. So wird ein formales System in einigen Büchern nur dann als Kalkül bezeichnet, wenn es spezielle Eigenschaften wie z. B. die Forderung nach einer endlichen Anzahl von ☞ **Axiomen** oder ☞ **Schlussregeln** erfüllt.

Gilmore-Algorithmus

☞ 3.2.3

Beweisverfahren für Formeln der ☞ **Prädikatenlogik**. Der Algorithmus zeigt die ☞ **Allgemeingültigkeit** einer Formel F , indem schrittweise die Menge der ☞ **Grundinstanzen** von $\neg F$ approximiert wird. Ist die erzeugte Teilmenge im aussagenlogischen Sinne ☞ **unerfüllbar**, so ist nach dem ☞ **Satz von Herbrand** auch die Formel $\neg F$ unerfüllbar. Aus der Unerfüllbarkeit von F folgt sofort die Allgemeingültigkeit von F .

Gödelisierung

☞ 6.1.5.6

Eine injektive Funktion $c : M \rightarrow \mathbb{N}$ heißt Gödelisierung, wenn die Bildmenge $c(M)$ ☞ **entscheidbar** und neben c auch die Umkehrfunktion c^{-1} ☞ **berechenbar** ist. $c(x)$ heißt die **Gödelnummer** des Elements x , geschrieben als $\langle x \rangle$. Die Gödelisierung erlaubt es, Aussagen über Objekte einer ☞ **abzählbaren** Menge M in Aussagen über die natürlichen Zahlen zu übersetzen.

Goto-Programm

☞ 6.1.3

Programm, verfasst in der fiktiven ☞ **Goto-Sprache**. Der bedingte Sprungbefehl (If-Goto) ist die einzige Möglichkeit, den Kontrollfluss zu steuern. Nach der ☞ **Church'schen These** lässt sich jede intuitiv berechenbare Funktion mithilfe eines Goto-Programms berechnen.

Goto-Sprache

☞ 6.1.3

Menge aller ☞ **Goto-Programme**. Nach der ☞ **Church'schen These** sind alle intuitiv berechenbaren Funktionen f Goto-berechenbar, d. h., es existiert ein Goto-Programm, das x als

Eingabe entgegennimmt und $f(x)$ als Ausgabe liefert. Die Goto-Sprache besitzt die gleiche Berechnungsstärke wie die ☞ **While-Sprache**, die ☞ **Turing-Maschine**, die ☞ **Registermaschine** und die ☞ **μ -rekursiven Funktionen**.

Grammatik

☞ 4.1

Instrumentarium zur generativen Beschreibung ☞ **formaler Sprachen**. Die Wörter einer Sprache werden aus einem Startsymbol durch die Anwendung verschiedener Produktionsregeln abgeleitet. Abhängig von der Struktur der einzelnen Produktionen werden Grammatiken in vier disjunkte Klassen eingeteilt. Es entsteht die sogenannte ☞ **Chomsky-Hierarchie**.

Greibach-Normalform

☞ 4.7

Verallgemeinerung der Bildungsregeln ☞ **regulärer Grammatiken**. Es lässt sich zeigen, dass jede von einer Greibach-Grammatik erzeugte Sprache kontextfrei ist. Umgekehrt existiert zu jeder ☞ **kontextfreien Sprache** L mit $\epsilon \notin L$ eine erzeugende Grammatik in Greibach-Normalform. Eine weitere wichtige Darstellungsvariante für Typ-2-Sprachen ist die ☞ **Chomsky-Normalform**.

Grundinstanz

☞ 3.2.3

Sei F eine Formel der ☞ **Prädikatenlogik**. Eine Grundinstanz entsteht, indem alle Variablen durch Terme ersetzt werden, die ausschließlich Funktions- und Konstantensymbole aus F enthalten.

Halteproblem

☞ 6.5

Synonym für die Frage, ob für jede ☞ **Turing-Maschine** T und jedes Eingabewort ω algorithmisch entschieden werden kann, ob T unter Eingabe von ω terminiert. Die ☞ **Berechenbarkeitstheorie** lehrt uns, dass das Halteproblem ☞ **unentscheidbar** ist und ein solches Entscheidungsverfahren aus fundamentalen Überlegungen heraus nicht existieren kann. Mithilfe der ☞ **Reduktionstechnik** lassen sich weitere ☞ **unentscheidbare** Probleme identifizieren. Hierzu gehören das ☞ **Halteproblem auf leerem Band**, das ☞ **spezielle Halteproblem** und das ☞ **Post'sche Korrespondenzproblem**. Die Verallgemeinerung des Halteproblems führt auf direkten Weg zum berühmten ☞ **Satz von Rice**.

Halteproblem auf leerem Band

☞ 6.5

Synonym für die Frage, ob für jede ☞Turing-Maschine T algorithmisch entschieden werden kann, ob T unter Eingabe von ϵ terminiert. Das Halteproblem auf leerem Band ist ☞unentscheidbar.

Hamilton-Problem

☞ 1.2.5

Bekanntes Problem aus der Graphentheorie. Untersucht wird die Frage, ob in einem Graphen G ein geschlossener Weg existiert, der alle Knoten genau einmal besucht. Das Hamilton-Problem ist ☞NP-vollständig und damit Teil einer großen Klasse schwer zu lösender Probleme. Für die ☞Komplexitätstheorie ist es von Interesse, da eine geringfügige Änderung der Aufgabenstellung zu einem Problem führt, das in linearer Zeit gelöst werden kann (☞Königsberger Brückenproblem).

Herbrand-Interpretation

☞ 3.2.3

Spezielle ☞Interpretation für Formeln der ☞Prädikatenlogik, in der die Variablen- und Funktionssymbole durch Elemente des ☞Herbrand-Universums interpretiert werden.

Herbrand-Universum

☞ 3.2.3

Das Herbrand-Universum einer prädikatenlogischen Formel F ist die Menge aller variablenfreier Terme, die mit den Funktionssymbolen von F gebildet werden können.

Herbrand-Modell

☞ 3.2.3

Ist eine ☞Herbrand-Interpretation ein ☞Modell einer Formel F , so sprechen wir von einem Herbrand-Modell. Alle prädikatenlogischen Kalküle beruhen auf der Eigenschaft, dass eine Formel F genau dann erfüllbar ist, wenn sie ein Herbrand-Modell besitzt.

Hilbert-Kalkül

☞ 3.1.3

Spezielle ☞Kalküle, die sich an der traditionellen mathematischen Beweisführung orientieren. In einem Hilbert-Kalkül werden wahre Aussagen aus einer Menge von Axiomen durch die sukzessive Anwendung fest definierter Schlussregeln abgeleitet. Ein Beweis ist eine Folge von ☞Tautologien, an deren Ende die Behauptung steht. Hilbert-Kalküle sind von hoher theoretischen Interesse, spielen in der Praxis dagegen

kaum eine Rolle. Hier kommen fast ausschließlich ☞Widerspruchskalküle zum Einsatz, in denen sich die ☞Allgemeingültigkeit einer Formel mit weniger Aufwand beweisen lässt.

Induktionsaxiom

☞ 2.3.1

Name des fünften ☞Peano-Axioms. Seine Aussage lautet wie folgt: Enthält eine Teilmenge M von \mathbb{N} die Zahl 1 und zu jedem Element n auch ihren Nachfolger n' , so gilt $M = \mathbb{N}$. Aus dem Induktionsaxiom erwächst das Beweisprinzip der ☞vollständigen Induktion.

Interpretation

☞ 3.1.1

Spezielle Abbildung, die den Symbolen einer Logikformel eine Bedeutung zuordnet. Interpretationen sind das Bindeglied zwischen der ☞Syntax und der ☞Semantik einer ☞Logik.

Inzidenzmatrix

☞ 5.7

Matrix der Größe $m \times n$, die das Schaltverhalten eines ☞Petri-Netzes mit m Stellen und n Transitionen beschreibt. Für jede Stelle existiert eine eigene Zeile und für jede Transition eine eigene Spalte. Der Wert (i, j) besagt, wie sich die Anzahl der Marken in der Stelle S_i ändert, wenn die Transition T_j schaltet. Durch die Multiplikation mit dem ☞Parikh-Vektor lässt sich die Folgekonfiguration berechnen.

Kalkül

☞ 3.1.3

Regelsystem, mit dem die Allgemeingültigkeit einer Logikformel auf ☞syntaktischer Ebene bewiesen werden kann. Die Durchführung eines Beweises verläuft rein maschinell und kommt ohne Metawissen über ☞Interpretationen oder ☞Modelle aus. In der ☞Aussagenlogik und ☞Prädikatenlogik spielen die ☞Hilbert-Kalküle, der ☞Resolutionskalkül und der ☞Tableaukalkül eine hervorgehobene Rolle. Die beiden letztgenannten fallen in die Klasse der ☞Widerspruchskalküle. In diesem Buch wird der Begriff des Kalküls synonym mit dem Begriff des ☞formalen Systems verwendet.

Kardinalzahl

☞ 2.3.3

Maß für die ☞Mächtigkeit einer Menge. Die Kardinalzahl einer endlichen Menge mit n Elementen entspricht n . Unendliche Mengen lassen sich bez. ihrer Mächtigkeit in Äquivalenzklassen einteilen, die durch die Kardinalzahlen

$\aleph_0, \aleph_1, \dots$ repräsentiert werden. \aleph (Aleph) ist der erste Buchstabe des hebräischen Alphabets.

Kellerautomat

5.5

Spezieller **endlicher Automat**, der neben einer endlichen Zustandsmenge einen **Kellerspeicher** besitzt. Durch den hinzugefügten Speicher gewinnen Kellerautomaten im Vergleich zu DEAs oder NEAs an Ausdrucksstärke hinzu. Die Klasse der von Kellerautomaten akzeptierten Sprachen entspricht der Klasse der **kontextfreien Sprachen**.

Kellerspeicher

5.5

Unendlich großer Speicher, der nach dem FIFO-Prinzip arbeitet. FIFO steht für *First In, First Out* und bedeutet, dass immer nur das oberste Kellerzeichen manipuliert werden kann.

Klausel

3.1.2

Eine Klausel ist eine Menge von **Literalen** $\{(\neg)A_1, \dots, (\neg)A_i\}$ und steht stellvertretend für die Formel $(\neg)A_1 \vee \dots \vee (\neg)A_i$. Die leere Klausel \square repräsentiert den Wahrheitswert 0.

Kleene'sche Normalform

6.1.3

Spezielle Darstellungsform für **While-Programme**, die mit einer einzigen While-Schleife auskommen. Nach dem **Satz von Kleene** existiert für jedes While-Programm ein äquivalentes Programm in Kleene'scher Normalform.

Komplexitätsklasse

7.1.1

In der **Komplexitätstheorie** werden Funktionen anhand ihres **asymptotischen Wachstums** in verschiedene Komplexitätsklassen eingeteilt. Für deren Beschreibung wird die **O-Notation** verwendet. Komplexitätsklassen werden eingesetzt, um die Laufzeit und den Platzverbrauch von Algorithmen zu kategorisieren. Von besonderer Bedeutung sind die Klassen **P**, **NP**, **EXP**, **NEXP**, **PSPACE** und **NPSPACE**. Zu jeder Komplexitätsklasse C existiert eine komplementäre Komplexitätsklasse $\text{co}C$ (**Co-Komplexität**).

Komplexitätstheorie

7 Kapitel 7

Teilgebiet der theoretischen Informatik, das sich mit der Frage beschäftigt, wie sich Algorithmen für sehr große Eingaben

verhalten. Hierzu werden Algorithmen anhand ihres Speicherplatzbedarfs und Zeitverbrauchs in verschiedene Komplexitätsklassen eingeteilt, die Rückschlüsse auf das asymptotische Wachstum der untersuchten Parameter zulassen. Im Gegensatz zur **Berechenbarkeitstheorie**, die Fragen nach der puren Existenz von Berechnungsverfahren beantwortet, stellt die Komplexitätstheorie die praktische Verwertbarkeit von Algorithmen in den Vordergrund.

Konfiguration

5.2

Momentaufnahme eines **endlichen Automaten** oder einer **Turing-Maschine**. Der Konfigurationsbegriff ist ein technisches Hilfsmittel, um Zustandsübergänge und die hieraus resultierende Ableitungsrelation in mathematisch präziser Form zu charakterisieren.

Königsberger Brückenproblem

1.2.5

Bekanntes Problem aus der Graphentheorie. Untersucht wird die Frage, ob in einem Graphen G ein geschlossener Weg existiert, der alle Kanten genau einmal besucht. Leonhard Euler konnte zeigen, dass das Problem in linearer Zeit gelöst werden kann. Für die **Komplexitätstheorie** ist es von Interesse, da eine geringfügige Änderung der Aufgabenstellung ein **NP-vollständiges** Problem entstehen lässt (**Hamilton-Problem**).

Konjunktive Form

3.1.2

Eine Logikformel liegt in konjunktiver Form vor, wenn sie als Konjunktion von Disjunktionen aufgebaut ist.

Konjunktive Normalform

3.1.2

Eine Logikformel liegt in konjunktiver Normalform vor, wenn sie als Konjunktion von **Maxtermen** aufgebaut ist und alle Maxterme paarweise verschieden sind.

Kontextfreie Grammatik

4.2

Grammatik mit der Eigenschaft, dass die linke Seite einer Produktionsregel ausschließlich aus einer einzigen Variablen besteht. In der Nomenklatur der **Chomsky-Hierarchie** werden kontextfreie Grammatiken als Typ-2-Grammatiken bezeichnet.

Kontextfreie Sprache

☞ 4.2

Eine Sprache L heißt kontextfrei, falls eine ☞kontextfreie Grammatik existiert, die L erzeugt. Die Menge der kontextfreien Sprachen entspricht der Menge der von ☞Kellerautomaten akzeptierten Sprachen.

Kontextsensitive Grammatik

☞ 4.2

Grammatik mit der Eigenschaft, dass die Anwendung einer Produktion niemals zu einer Verkürzung der abgeleiteten Zeichenkette führt. In der Nomenklatur der ☞Chomsky-Hierarchie werden kontextsensitive Grammatiken als Typ-1-Grammatiken bezeichnet.

Kontextsensitive Sprache

☞ 4.2

Eine Sprache L heißt kontextsensitiv, falls eine ☞kontextsensitive Grammatik existiert, die L erzeugt. Die Menge der kontextsensitiven Sprachen entspricht der Menge der Sprachen, die von linear beschränkten ☞Turing-Maschinen akzeptiert werden.

Landau-Symbole

☞ 7.1.1

Bezeichnung für die Symbolmenge $\{O, \Omega, \Theta, o, \omega\}$. Die Landau-Symbole werden in der ☞O-Notation verwendet, um das ☞asymptotische Wachstum von Funktionen zu beschreiben.

Leerheitsproblem

☞ 4.1

Wichtige Problemstellung aus dem Bereich der ☞formalen Sprachen. Hinter dem Leerheitsproblem verbirgt sich die Frage, ob eine gegebene Sprache L mindestens ein Wort enthält. Mit anderen Worten: Gilt $L \neq \emptyset$?

Linksableitung

☞ 4.1

Ableitungssequenz mit der Eigenschaft, dass in jedem Schritt das am weitesten links stehende Nonterminal ersetzt wurde.

Literal

☞ 3.1.2

Bezeichnung für eine atomare Formel oder deren Negation. In der Aussagenlogik hat ein Literal damit die Form A oder $\neg A$, wobei A eine beliebige aussagenlogische Variable bezeichnet.

Logik

☞ Kapitel 3

Teilbereich der Mathematik, der sich mit grundlegenden Fragestellungen mathematischer Theorien beschäftigt. Ferner wird der Begriff für die Beschreibung von formalen Systemen verwendet, in denen die ☞Syntax und die ☞Semantik strikten Regeln folgen. In der Vergangenheit wurden verschiedene Logiken postuliert, die sich sowohl in ihrem Erscheinungsbild als auch in ihrer Ausdrucksstärke erheblich voneinander unterscheiden. Wichtige Vertreter sind die ☞Aussagenlogik, die ☞Prädikatenlogik sowie die ☞Logiken höherer Stufe.

Logik höherer Stufe

☞ 3.3.2

Erweiterung der ☞Prädikatenlogik, die unter anderem stark genug ist, um die natürlichen Zahlen zu beschreiben. Im Gegensatz zur Prädikatenlogik dürfen in Logiken höherer Stufe auch Teilmengen des Grundbereichs, d. h. auch Prädikate quantifiziert werden. Erst hierdurch wird es möglich, das für die Formalisierung der natürlichen Zahlen unabdingbare ☞Induktionsaxiom innerhalb der Logik auszudrücken. Logiken höherer Stufe werden in denjenigen Bereichen der formalen Hard- und Software-Verifikation eingesetzt, die eine hohe Ausdrucksstärke benötigen.

Loop-Programm

☞ 6.1.1

Programm, verfasst in der fiktiven ☞Loop-Sprache. Die Loop-Schleife ist die einzige Möglichkeit, den Kontrollfluss zu steuern. Anders als in einem ☞While-Programm steht die Anzahl der auszuführenden Iterationen vor dem Schleifen-eintritt fest und kann danach nicht mehr verändert werden.

Loop-Sprache

☞ 6.1.1

Menge aller ☞Loop-Programme. Die Loop-Sprache ist berechnungsschwächer als die ☞While-Sprache oder die ☞Goto-Sprache. So lässt sich beispielsweise die ☞Ackermann-Funktion mithilfe eines ☞While-Programms oder eines ☞Goto-Programms berechnen, nicht jedoch mit einem Loop-Programm.

Mächtigkeit

☞ 2.3.3

Mathematisches Konstrukt, das quantitative Aussagen über die Anzahl der Elemente beliebig großer Mengen gestattet. Die Mächtigkeit endlicher Mengen wird mit der Anzahl ih-

rer Elemente gleichgesetzt. Unendliche Mengen werden bez. ihrer Eigenschaft untersucht, bijektiv auf andere Mengen mit bekannter Mächtigkeit abbildbar zu sein. Auf diese Weise entsteht eine Hierarchie verschiedener Unendlichkeiten, die sich mithilfe von \aleph -Kardinalzahlen eindeutig beschreiben lassen.

Maxterm

☞ 3.1.2

Aussagenlogische Formel, die für genau eine Variablenbelegung falsch wird. Ein Maxterm einer Funktion mit n Variablen besteht aus n disjunktiv verknüpften \neg -Literalen.

Mealy-Automat

☞ 5.6.4

Spezieller \rightarrow -Transdukt, der die Ausgabe sowohl aus dem aktuellen Zustand als auch aus der aktuellen Eingabe berechnet. Mealy-Automaten werden aufgrund dieser Eigenschaft auch als Übergangsautomaten bezeichnet.

Mehrdeutigkeitsproblem

☞ 4.1

Wichtige Problemstellung aus dem Bereich der \rightarrow -formalen Sprachen. Hinter dem Mehrdeutigkeitsproblem verbirgt sich die Frage, ob die Ableitungssequenzen einer Grammatik G stets eindeutig sind. In mehrdeutigen Grammatiken existiert mindestens ein Wort ω , das sich durch unterschiedliche Regelanwendungen aus dem Startsymbol S ableiten lässt.

Minterm

☞ 3.1.2

Aussagenlogische Formel, die für genau eine Variablenbelegung wahr wird. Ein Minterm einer Funktion mit n Variablen besteht aus n konjunktiv verknüpften \neg -Literalen.

Modell

☞ 3.1.1

Spezielle \rightarrow -Interpretation, die eine gegebene logische Formel wahr werden lässt.

Modus ponens

☞ 3.1.3

Elementare Schlussregel der mathematischen Logik, die sich in Wörtern wie folgt umschreiben lässt: Wenn die Aussage A wahr ist und aus A die Aussage B folgt, so ist auch B wahr. Der Modus ponens ist die bevorzugte Schlussregel in den meisten \rightarrow -Hilbert-Kalkülen.

Moore-Automat

☞ 5.6.4

Spezieller \rightarrow -Transdukt, der die aktuelle Ausgabe ausschließlich aus dem aktuellen Zustand berechnet. Moore-Automaten werden aufgrund dieser Eigenschaft auch als Zustandsautomaten bezeichnet.

μ -rekursive Funktion

☞ 6.1.4

Kleinste Menge, die alle \rightarrow -primitiv-rekursiven Funktionen enthält und außerdem unter der Anwendung des μ -Operators abgeschlossen ist. Die μ -rekursiven Funktionen besitzen die gleiche Berechnungsstärke wie die \rightarrow -While-Sprache, die \rightarrow -Goto-Sprache, die \rightarrow -Turing-Maschine und die \rightarrow -Registermaschine. Nach der \rightarrow -Church'schen These ist jede intuitiv berechenbare Funktion auch μ -rekursiv.

NEA

☞ 5.3

Nichtdeterministischer endlicher \rightarrow -Akzeptor. Im Gegensatz zu seinem deterministischen Pendant (\rightarrow -DEA) können für die gleiche Eingabe mehrere mögliche Zustandsübergänge definiert sein und damit mehrere verschiedene Berechnungsfolgen für die gleiche Eingabe existieren. Mit dem \rightarrow -Potenzmengenautomaten existiert zu jedem NEA ein DEA, der die gleiche \rightarrow -formale Sprache akzeptiert. Die von DEAs und NEAs akzeptierten Sprachklassen sind identisch und entsprechen der Klasse der \rightarrow -regulären Sprachen.

Negationsnormalform

☞ 3.2.2

Eine Formel liegt in Negationsnormalform vor, wenn das Negationszeichen nur vor atomaren Formeln auftaucht. In der \rightarrow -Prädikatenlogik wird diese Darstellung als Zwischenschritt in der Erzeugung der \rightarrow -Pränex-Form verwendet.

NEXP

☞ 7.2.3

Die \rightarrow -Komplexitätsklasse NEXP enthält alle Sprachen, die von nichtdeterministischen \rightarrow -Turing-Maschinen in exponentieller Zeit entschieden werden können.

NP

☞ 7.2.1

Die \rightarrow -Komplexitätsklasse NP enthält alle Sprachen, die von nichtdeterministischen \rightarrow -Turing-Maschinen in polynomieller Zeit entschieden werden können. NP ist eine Obermenge von

P. Ob es sich dabei um eine echte Obermenge handelt ($NP \setminus P \neq \emptyset$), ist Gegenstand des derzeit ungelösten **P-NP-Problems**.

NP-hart

☞ 7.3.2

Ein Problem ist NP-hart, wenn sich sämtliche Probleme der Komplexitätsklasse **NP** durch **polynomiale Reduktion** darauf abbilden lassen. Ein NP-hartes Problem kann, muss aber nicht selbst in der Klasse NP liegen.

NPSPACE

☞ 7.2.2

Die **Komplexitätsklasse NPSPACE** enthält alle Sprachen, die von nichtdeterministischen **Turing-Maschinen** mit polynomiellem Bandplatzverbrauch entschieden werden können. Aus dem **Satz von Savitch** folgt, dass die Klasse NSPACE mit der Klasse **PSPACE** übereinstimmt.

NP-vollständig

☞ 7.3.2

Ein Problem ist NP-vollständig, wenn es **NP-hart** ist und selbst in der Klasse **NP** liegt. NP-vollständige Probleme gelten als die schwierigsten Probleme innerhalb von NP, da sich alle anderen Probleme durch **polynomiale Reduktion** darauf abbilden lassen.

O-Notation

☞ 7.1.1

Standardschreibweise für die Benennung von **Komplexitätsklassen**. Das große 'O' ist eines von fünf **Landau-Symbolen** und der Namensgeber dieser Notation.

P

☞ 7.2.1

Die **Komplexitätsklasse P** enthält alle Sprachen, die von deterministischen **Turing-Maschinen** in polynomieller Zeit entschieden werden können. P ist eine Teilmenge von **NP**. Ob es sich dabei um eine echte Teilmenge handelt ($NP \setminus P \neq \emptyset$), ist Gegenstand des derzeit ungelösten **P-NP-Problems**.

Paarungsfunktion

☞ 2.3.3

Die Cantor'sche Paarungsfunktion bildet die Elemente der Menge $\mathbb{N} \times \mathbb{N}$ bijektiv auf die Menge \mathbb{N} ab. Ihre Existenz beweist, dass die Menge \mathbb{N}^k für alle $k \in \mathbb{N}$ die gleiche **Mächtigkeit** besitzt wie die natürlichen Zahlen selbst.

Parikh-Vektor

☞ 5.7

Vektordarstellung einer Sequenz von nacheinander schaltenden Transitionen eines **Petri-Netzes**. Durch die Multiplikation mit der **Inzidenzmatrix** lässt sich die Folgekonfiguration eines Petri-Netzes berechnen.

Partielle Funktion

☞ 2.2

Funktion, die nicht für alle Elemente ihres Definitionsbereichs einen definierten Funktionswert besitzt. In der klassischen Mathematik ist dieser Begriff unbekannt – dort sind alle Funktionen per Definition **total**. In der theoretischen Informatik werden partielle Funktionen für die Beschreibung von Algorithmen eingesetzt, die für gewisse Eingabewerte nicht terminieren.

Peano-Axiome

☞ 2.3.1

Formale Beschreibung der natürlichen Zahlen, die aus insgesamt fünf Axiomen besteht. Die Peano-Axiome dienen uns heute als Grundlage für den berechenbarkeitstheoretischen Umgang mit den natürlichen Zahlen. Unter anderem folgt aus Peanos fünftem Axiom, dem **Induktionsaxiom**, dass die **Prädikatenlogik** erster Stufe zu schwach ist, um die natürlichen Zahlen zu formalisieren.

Petri-Netz

☞ 5.7

Formales Modell zur Beschreibung nebenläufiger Systeme. Genau wie **endliche Automaten** arbeiten Petri-Netze zustandsbasiert, verfügen jedoch über deutlich komplexere Übergangsmechanismen. Streng unterschieden werden **Bedingungen** und **Ereignisse**. Erstere werden durch **Stellen**, letztere durch **Transitionen** beschrieben. In einem Petri-Netz wird der aktuelle Zustand eines Systems durch **Marken** modelliert. Schaltet eine Transition, so wird eine Marke aus jeder Eingabestelle entfernt und jeder Ausgangsstelle eine zusätzliche Marke hinzugefügt.

P-NP-Problem

☞ 7.3.2

Hinter diesem Problem verbirgt sich die Frage, ob jede Sprache, die durch eine nichtdeterministische Turing-Maschine in polynomieller Zeit entschieden werden kann, auch von einer deterministischen Turing-Maschine in polynomieller Zeit entschieden werden kann. Das P-NP-Problem gilt als das wichtigste offene Problem der Theoretischen Informatik.

tigste der bis dato offenen Probleme der theoretischen Informatik, da seine Lösung weitreichende Konsequenzen für nahezu alle Teilbereiche der Informatik hat. Wäre $P = NP$, so ließen sich viele Algorithmen in polynomieller Zeit lösen, für die heute ausschließlich Algorithmen mit exponentiellem Zeitaufwand existieren. Unter anderem wären viele kryptografische Systeme auf einen Schlag angreifbar. In der großen Zahl der bisher entdeckten NP -vollständigen Probleme sehen viele Experten einen Hinweis darauf, dass P und NP voneinander verschieden sind. Einen formalen Beweis für diese Vermutung konnte jedoch noch niemand erbringen.

Polynomielle Reduktion

7.3.1

Wichtiges Beweisprinzip der $\text{Komplexitätstheorie}$. Im Rahmen eines Reduktionsbeweises wird gezeigt, dass sich ein Problem lösen lässt, indem die Fragestellung in polynomieller Zeit auf ein anderes Problem abgebildet wird, dessen (polynomielle) Lösung bereits bekannt ist. Durch den geschickten Einsatz dieser Technik lassen sich die Komplexitätseigenschaften vieler Probleme auf weitere Fragestellungen übertragen. Der Begriff ist eng verwandt mit der Reduzierbarkeit aus dem Bereich der $\text{Berechenbarkeitstheorie}$.

Post'sches Korrespondenzproblem

6.5.4

Für eine Folge von Wortpaaren $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ gilt es zu entscheiden, ob eine Indexsequenz i_1, \dots, i_k existiert, so dass die Konkatenation von x_{i_1}, \dots, x_{i_k} die gleiche Zeichenkette hervorbringt wie die Konkatenation von y_{i_1}, \dots, y_{i_k} . Genau wie das Halteproblem ist auch das $\text{Post'sche Korrespondenzproblem}$ unentscheidbar. Die Eigenschaft, auf viele andere Problemstellungen reduzierbar zu sein, macht das Post'sche Korrespondenzproblem zu einem der wichtigsten Hilfsmittel im Bereich der $\text{Berechenbarkeitstheorie}$.

Potenzmengenautomat

5.3.2

Spezieller Automat, der die Potenzmenge der Zustände eines anderen Automaten als Zustandsraum verwendet. Der Potenzmengenautomat ist der Schlüssel, um die Äquivalenz von DEAs und NEAs zu beweisen. Die Konstruktion ermöglicht es, jeden NEA in einen DEA zu übersetzen, der die gleiche Sprache akzeptiert.

Prädikatenlogik

3.2

Die Prädikatenlogik erweitert die Aussagenlogik um mehr-

stellige Prädikate sowie um die Quantoren \forall („für alle“) und \exists („es existiert“). Viele Aspekte des logischen Schließens, die in der Aussagenlogik nicht ausgedrückt werden können, lassen sich mithilfe prädikatenlogischer Formeln formal beschreiben. Die Prädikatenlogik ist die Grundlage der Programmiersprache Prolog .

Pränex-Form

3.2.2

Eine Formel liegt in Pränex-Form vor, wenn alle Quantoren links von den anderen Formelbestandteilen stehen. Ausgehend von der $\text{Negationsnormalform}$ lässt sich jede Formel der Prädikatenlogik in eine äquivalente Formel in Pränex-Form übersetzen.

Primitive Rekursion

6.1.4

Eine Funktion $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ ist nach dem Schema der *primiven Rekursion* aufgebaut, wenn sie die folgende Form besitzt:

$$f(m, n) = \begin{cases} g(n) & \text{falls } m = 0 \\ h(f(m-1, n), m-1, n) & \text{falls } m > 0 \end{cases}$$

Primitiv-rekursive Funktion

6.1.4

Kleinste Menge von Funktionen, die die Nullfunktion, die Nachfolgerfunktion und die Projektion enthält und bez. Komposition und $\text{primitiver Rekursion}$ abgeschlossen ist. Die Klasse der primitiv-rekursiven Funktionen besitzt die gleiche Berechnungsstärke wie die Loop-Sprache . Mit anderen Worten: Jede primitiv-rekursive Funktion lässt sich mit einem Loop-Programm berechnen und jede Loop-berechenbare Funktion ist ihrerseits primitiv-rekursiv. Die primitiv-rekursiven Funktionen sind vollständig in der Menge der μ -rekursiven Funktionen enthalten.

Prolog

3.2.4

Logische Programmiersprache, die auf dem prädikatenlogischen Resolutionskalkül basiert. Prolog-Programme bestehen aus einer Problembeschreibung in Form von Logikformeln. Wird eine Anfrage gestellt, versucht der Prolog-Interpreter, die Antwort selbstständig aus der Problembeschreibung zu deduzieren.

PSPACE**7.2.2**

Die **Komplexitätsklasse PSPACE** enthält alle Sprachen, die von deterministischen **Turing-Maschinen** mit polynomiellem Bandplatzverbrauch entschieden werden können. Aus dem **Satz von Savitch** folgt, dass die Klasse PSPACE mit der Klasse **NPSPACE** übereinstimmt.

Pumping-Lemma**4.3.2**

Hilfsmittel, um zu beweisen, dass eine Menge $L \subseteq \Sigma^*$ keine **reguläre Sprache** oder keine **kontextfreie Sprache** ist. Das Pumping-Lemma nutzt die Eigenschaft dieser Sprachklassen aus, dass mit jedem hinreichend langen Wort $\omega \in L$ auch diejenigen Wörter ω_i in L enthalten sind, die durch das Duplizieren gewisser Teilsequenzen entstehen.

Rechtsableitung**4.1**

Ableitungssequenz mit der Eigenschaft, dass in jedem Schritt das am weitesten rechts stehende Nonterminal ersetzt wurde.

Reduktion**6.5**

Wichtiges Beweisprinzip der **Berechenbarkeitstheorie**. Im Rahmen eines Reduktionsbeweises wird gezeigt, dass sich ein Problem lösen lässt, indem die Fragestellung auf ein anderes Problem abgebildet wird, dessen Lösung bereits bekannt ist. Durch den geschickten Einsatz dieser Technik lassen sich die Berechenbarkeitseigenschaften vieler Probleme auf weitere Fragestellungen übertragen. Der Begriff ist eng verwandt mit der **polynomiellen Reduzierbarkeit** aus dem Bereich der **Komplexitätstheorie**.

Registermaschine**6.1.6.1**

Spezielles **Berechnungsmodell**, das der Architektur realer Computersysteme sehr nahe kommt. Eine Registermaschine setzt sich aus dem Eingabeband, dem Ausgabeband und der Zentraleinheit zusammen. Letztere besteht aus dem Speicher, dem Programm und zwei Registern. Die Programmanweisungen werden über den Befehlszähler adressiert und sequentiell abgearbeitet. Arithmetische Berechnungen werden im Akkumulator durchgeführt. Die Registermaschine besitzt die gleiche Berechnungsstärke wie die **While-Sprache**, die **Goto-Sprache**, die **Turing-Maschine** und die **μ -rekursiven Funktionen**. Nach der **Church'schen These** lässt

sich jede intuitiv berechenbare Funktion mithilfe einer **Registermaschine** berechnen.

Regulärer Ausdruck**4.3.4**

Alternative Beschreibungsform für **reguläre Sprachen**. Reguläre Ausdrücke sind der De-facto-Standard für die Spezifikation von Suchmustern.

Reguläre Grammatik**4.2**

Kontextfreie Grammatik mit der zusätzlichen Eigenschaft, dass die rechte Seite einer Produktion entweder aus dem leeren Wort ε oder einem Terminalsymbol, gefolgt von einem Nonterminal, besteht. In der Nomenklatur der **Chomsky-Hierarchie** werden kontextfreie Grammatiken als Typ-3-Grammatiken bezeichnet.

Reguläre Sprache**4.2**

Eine Sprache L heißt regulär, falls eine **reguläre Grammatik** existiert, die L erzeugt. Die Menge der regulären Sprachen stimmt mit der Menge der von **DEAs** und **NEAs** akzeptierten Sprachen überein.

Resolutionskalkül**3.1.3**

Spezielles **Widerspruchskalkül** der **Aussagenlogik** und **Prädikatenlogik**. Um die **Allgemeingültigkeit** einer Formel F zu zeigen, wird die negierte Formel $\neg F$ in eine Menge von **Klauseln** übersetzt. Anschließend werden in einem iterativen Prozess neue Resolventen abgeleitet. Wird die leere Klausel \square deduiert, so terminiert das Verfahren. In diesem Fall ist gezeigt, dass $\neg F$ kein **Modell** besitzt und F demnach eine **Tautologie** sein muss.

Robinson-Algoritmus**3.2.3.1**

Systematisches Verfahren, um eine Menge prädikatenlogischer Formeln zu **unifizieren**. Der Algorithmus durchsucht die Formeln zeichenweise von links nach rechts. Abweichungen werden durch Substitutionen korrigiert. Sind die Eingabeformeln unifizierbar, so lässt sich der Unifikator durch die Verkettung der berechneten Substitutionen erzeugen. Der Algorithmus von Robinson berechnet den **allgemeinsten Unifikator** der Eingabemenge.

Rucksackproblem

☞ 7.1

Ein Rucksack ist so mit Gegenständen zu bepacken, dass der erzielte Gesamtwert maximal wird, ohne das Fassungsvermögen zu überschreiten. In der hier vorgestellten Variante sind für jeden Gegenstandstyp beliebig viele Exemplare vorhanden und die Werte und Volumina ganzzahlig. Unter diesen Voraussetzungen lässt sich das Rucksackproblem effizient mit dem Mittel der ☞dynamischen Programmierung lösen. Eine leichte Abwandlung der Problemstellung lässt dagegen ein ☞NP-vollständiges Problem entstehen.

Russell'sche Antinomie

☞ 1.2.1

Fundamentaler Widerspruch in der Cantor'schen Mengenlehre. Unter der Annahme, dass sich eine Menge M entweder selbst enthält oder nicht selbst enthält, definierte der britische Mathematiker Bertrand Russell die Menge aller Mengen, die sich nicht selbst als Element enthalten. Genau wie im Falle des ☞Barbier-Paradoxons führt diese Definition einen fundamentalen Widerspruch herbei. Erst der formale axiomatische Aufbau der Mengenlehre durch Ernst Zermelo und Abraham Fraenkel konnte die entdeckte Inkonsistenz im Cantor'schen Begriffsgerüst entgültig beheben.

SAT

☞ 7.3.3

Frage nach der ☞Erfüllbarkeit einer Formel der ☞Aussagenlogik. Zu den Sternstunden der theoretischen Informatik gehört der Beweis, dass SAT ein ☞NP-vollständiges Problem ist (☞Satz von Cook).

Satz von Cantor

☞ 2.3.3

Gleich mehrere zentrale Ergebnisse der Mathematik werden mit Georg Cantors Namen verbunden. Hierzu gehört der Nachweis, dass \mathbb{N} und \mathbb{N}^2 die gleiche ☞Mächtigkeit besitzen (zu zeigen über das erste Cantor'sche Diagonalisierungsargument), wie auch der Satz über die ☞Überabzählbarkeit der reellen Zahlen (zu zeigen über das zweite Cantor'sche Diagonalisierungsargument).

Satz von Cook

☞ 7.3.3

Im Jahre 1971 gelang es Stephen Cook, die ☞NP-Vollständigkeit von ☞SAT zu beweisen, ohne auf das Prinzip der ☞polynomiellen Reduktion zurückzugreifen. Der Satz ist

in zweierlei Hinsicht von Bedeutung. Zum einen beweist er, dass NP-vollständige Probleme wirklich existieren. Zum anderen lassen sich durch polynomielle Reduktion andere Fragestellungen als NP-vollständig identifizieren. Hierzu gehören unter anderem das ☞Hamilton-Problem sowie die Probleme CLIQUE, SELECT, PARTITION, BIN PACKING und TRAVELING SALESMAN. Für die Durchführung der Reduktion wird zumeist auf die vereinfachte Variante ☞3SAT zurückgegriffen, die ebenfalls NP-vollständig ist.

Satz von Herbrand

☞ 3.2.3

Der Satz von Herbrand stellt einen wichtigen Zusammenhang zwischen der ☞Prädikatenlogik und der ☞Aussagenlogik her. Er besagt, dass eine Formel F in ☞Skolem-Form genau dann ein ☞Herbrand-Modell besitzt, wenn alle endlichen Teilmengen der ☞Grundinstanzen von F im aussagenlogischen Sinne erfüllbar sind.

Satz von Kleene

☞ 6.1.2

Besagt, dass sich jede While-berechenbare Funktion mit einem ☞While-Programm berechnen lässt, das eine einzige Schleife besitzt. Die Anzahl der Schleifen, die für die Umsetzung eines Algorithmus benötigt werden, ist somit unabhängig von dessen Komplexität.

Satz von Rabin und Scott

☞ 5.3.2

Besagt, dass zu jedem nichtdeterministischen endlichen Automaten ein deterministischer endlicher Automat existiert, der die gleiche Sprache akzeptiert.

Satz von Rice

☞ 6.5.2

Im Jahre 1953 gelang es Henry Gordon Rice, einen Zusammenhang zwischen dem ☞Halteproblem und einer beliebigen nichttrivialen Eigenschaft von Turing-Maschinen herzustellen. Aus dem Satz von Rice folgt unmittelbar, dass es unmöglich ist, irgendeine nichttriviale Eigenschaft von Turing-Maschinen algorithmisch zu überprüfen.

Satz von Savitch

☞ 7.2.2

Besagt, dass jedes von einer nichtdeterministischen ☞Turing-Maschine lösbar Problem mit einer deterministischen Turing-Maschine gelöst werden kann, deren Platzbedarf nur quadratisch höher liegt.

Schlussregel
☞ 3.1.3.1

Zentraler Bestandteil eines ☞*Hilbert-Kalküls*. Die Schlussregeln eines Kalküls definieren, wie sich aus bestehenden Aussagen neue Aussagen ableiten lassen. In einem formalen Beweis wird die zu zeigende Behauptung durch die sukzessive Anwendung der Schlussregeln aus den ☞*Axiomen* deduziert.

Semantik
☞ 3.1.1

Während die ☞*Syntax* den strukturellen Aufbau von Objekten beschreibt, befasst sich die Semantik mit deren Bedeutung. Im Bereich der natürlichen Sprache definiert die Semantik, welche Elemente der realen Welt sich hinter den gesprochenen Wörtern verbergen.

Semi-Entscheidbarkeit
☞ 6.3

Eine Menge M heißt semi-entscheidbar, falls die partielle ☞*charakteristische Funktion* ☞*berechenbar* ist. Für eine semi-entscheidbare Menge $M \subseteq \Sigma^*$ existiert ein systematisches Verfahren, das für alle Elemente $\omega \in M$ nach endlicher Zeit die Mengenzugehörigkeit bestätigt. Ist $\omega \notin M$, so kommt die Berechnung zu keinem Ende.

Skolem-Form
☞ 3.2.2

Eine prädikatenlogische Formel liegt in Skolem-Form vor, wenn sie die ☞*Pränex-Form* besitzt und keine Existenzquantoren mehr enthält. Jede Formel der ☞*Prädikatenlogik* lässt sich in eine ☞*erfüllbarkeitsäquivalente* Formel in Skolem-Form übersetzen.

Spezielles Halteproblem
☞ 6.5

Synonym für die Frage, ob für jede ☞*Turing-Maschine* T algorithmisch entschieden werden kann, ob T unter Eingabe der eigenen Gödelnummer $\langle T \rangle$ terminiert (☞*Gödelisierung*). Das spezielle Halteproblem ist ☞*unentscheidbar*.

Strukturelle Induktion
☞ 2.4.2

Variante der ☞*vollständigen Induktion*, mit der sich Aussagen über rekursiv definierte Strukturen beweisen lassen. Hierzu wird die Aussage zunächst für alle Basisfälle explizit bewiesen und anschließend gezeigt, dass sich deren Gültigkeit auf zusammengesetzte Objekte vererbt.

Syntax
☞ 3.1.1

Die Syntax befasst sich mit dem strukturellen Aufbau von Objekten. Im Bereich der natürlichen Sprache definiert die Syntax die Regeln, nach denen einzelne Wörter zu grammatisch korrekten Sätzen kombiniert werden können. Auf der syntaktischen Ebene werden Objekte als sinnleere Symbolketten verstanden. Erst die ☞*Semantik* weißt den einzelnen Objekten eine Bedeutung zu.

Syntaxbaum
☞ 4.1

Baumförmige Darstellung der Ableitungssequenzen einer ☞*Grammatik*. Ein Syntaxbaum wird so konstruiert, dass alle inneren Knoten mit Nonterminalen und alle Blätter mit Terminalsymbolen markiert sind. In der Baumdarstellung geht die Information über die Reihenfolge der Produktionsanwendungen verloren, so dass verschiedene Ableitungssequenzen zu demselben Syntaxbaum führen können.

Tableaukalkül
☞ 3.1.3.3

Spezielles ☞*Widerspruchskalkül* der ☞*Aussagenlogik* und ☞*Prädikatenlogik*. Um die ☞*Allgemeingültigkeit* einer Formel F zu zeigen, wird aus den Teilformeln von $\neg F$ eine Baumstruktur – das sogenannte *Tableau* – erzeugt. Anhand spezieller Abschlussbedingungen lassen sich offene und geschlossene Zweige identifizieren. Sind alle Zweige geschlossen, so besitzt $\neg F$ kein Modell und F ist als Tautologie identifiziert.

Tautologie
☞ 3.1.1

Bezeichnung für eine uneingeschränkt wahre Aussage. Der Begriff wird synonym mit dem Begriff der ☞*Allgemeingültigkeit* verwendet.

Theoretische Informatik
☞ Kapitel 1 – 7

Untersucht die mathematischen Methoden und Modelle, die sich hinter der Fassade der modernen Hardware- und Software-Technik verbergen. Wichtige Teilbereiche der theoretischen Informatik sind die ☞*Logik*, die Theorie der ☞*formalen Sprachen*, die Theorie der ☞*endlichen Automaten* sowie die ☞*Berechenbarkeits-* und die ☞*Komplexitätstheorie*.

Totale Funktion

☞ 2.2

Funktion, die für alle Elemente ihres Definitionsbereichs einen festgelegten Funktionswert besitzt. In der klassischen Mathematik sind alle Funktionen per Definition total und daher nicht explizit als solche gekennzeichnet. In der theoretischen Informatik existiert zusätzlich der Begriff der ☞*partiellen Funktion*.

Transduktoren

☞ 5.6

Neben den ☞*Akzeptoren* die zweite große Untergruppe ☞*endlicher Automaten*. Transduktoren bestehen aus einer Menge von Zuständen, einem Eingabe- und einem Ausgabealphabet, einer Zustandsübergangsfunktion sowie einem dedizierten Startzustand. In jedem Verarbeitungsschritt nimmt der Automat ein einzelnes Eingabezeichen entgegen und übersetzt dieses in ein Ausgabezeichen. Die Verarbeitung endet, wenn das letzte Eingabezeichen eingelesen wurde. Die Übersetzung von Transduktoren in digitale Hardware-Schaltungen ist Gegenstand der ☞*Automatensynthese*.

Turing-Berechenbarkeit

☞ 6.1.5

Eine Funktion f heißt Turing-berechenbar, falls eine ☞*Turing-Maschine* existiert, die f berechnet. Nach der ☞*Church'schen These* ist die Klasse der Turing-berechenbaren Funktionen mit der Klasse der intuitiv berechenbaren Funktionen identisch.

Turing-Maschine

☞ 6.1.5

Mathematisches Modell, um den Berechenbarkeitsbegriff formal zu erfassen. Grundlage ist ein eindimensionales Band, das sich aus unendlich vielen, nebeneinander angeordneten Zellen zusammensetzt. Die Maschine verfügt über einen Schreib-Lese-Kopf, der zu jeder Zeit über einer bestimmten Zelle positioniert ist. In jedem Bearbeitungsschritt kann eine Turing-Maschine das aktuell betrachtete Symbol durch ein anderes ersetzen und den Schreib-Lese-Kopf verschieben. Die ausgeführten Aktionen gehen mit einem potenziellen Wechsel des inneren Zustands einher. Anders als z. B. im Falle des ☞*Transduktors* werden alle Lese- und alle Schreiboperationen auf dem gleichen Band ausgeführt. Turings Maschinenmodell besitzt die gleiche Berechnungsstärke wie die ☞*While-Sprache*, die ☞*Goto-Sprache*, die ☞*Registersmaschine* und die ☞ *μ -rekursiven Funktionen*. Nach der

☞ *Church'schen These* lässt sich jede intuitiv berechenbare Funktion mithilfe einer Turing-Maschine berechnen.

Überabzählbarkeit

☞ 2.3.3

Eine Menge M mit unendlich vielen Elementen heißt überabzählbar, wenn es nicht möglich ist, M bijektiv auf die Menge der natürlichen Zahlen abzubilden. Mit dem Mittel der ☞*Diagonalisierung* kann unter anderem die Überabzählbarkeit der reellen Zahlen gezeigt werden. Der Begriff ist eng verwandt mit dem Begriff der ☞*Abzählbarkeit*.

Unberechenbarkeit

☞ 6.1

Eine Funktion f heißt unberechenbar, falls kein systematisches Verfahren existiert, das für alle Eingaben x nach endlich vielen Schritten terminiert und den Funktionswert $f(x)$ als Ausgabe liefert. Was wir unter einem systematischen Verfahren zu verstehen haben, wird durch die Definition eines ☞*Berechnungsmodells* formal festgelegt. Der Begriff der unberechenbaren Funktion ist eng mit dem Begriff der ☞*Unentscheidbarkeit* gekoppelt.

Unentscheidbarkeit

☞ 6.3

Eine Menge M heißt unentscheidbar, falls die ☞*charakteristische Funktion* ☞*unberechenbar* ist. Für eine unentscheidbare Menge M existiert kein systematisches Verfahren, das für alle Eingaben ω nach endlicher Zeit beantwortet, ob ω ein Element von M ist oder nicht. Viele praktische Fragestellungen, zu denen auch das ☞*Halteproblem* und das ☞*Post'sche Korrespondenzproblem* gehören, wurden in der Vergangenheit als unentscheidbar identifiziert.

Unerfüllbarkeit

☞ 3.1.1

Eine unerfüllbare Formel besitzt die Eigenschaft, kein einziges ☞*Modell* zu besitzen. Eine solche Formel ist damit immer falsch, unabhängig davon, wie wir ihre Bestandteile interpretieren.

Unifikation

☞ 3.2.3.1

Methode, die für eine Menge prädikatenlogischer Formeln einen ☞*Unifikator* berechnet. Ein bekannter Vertreter ist der ☞*Robinson-Algorithmus* zur Berechnung des ☞*allgemeinsten Unifikators*.

Unifikator
☞ 3.2.3.1

Eine Menge prädikatenlogischer Formeln $\{F_1, \dots, F_n\}$ heißt unifizierbar, falls eine Substitution σ existiert mit $\sigma F_1 = \dots = \sigma F_n$. Die Substitution σ wird als Unifikator bezeichnet.

Universelle Turing-Maschine
☞ 6.1.5

Spezielle Turing-Maschine, die in der Lage ist, jede andere Turing-Maschine zu simulieren. Hierzu wird die zu simulierende Maschine ☞ gödелиERT und zusammen mit dem Eingabewort auf das Band geschrieben.

Up-Arrow-Notation
☞ 2.3.2

Von Donald E. Knuth eingeführte Schreibweise, mit der sich arithmetische Verknüpfungen in einem einheitlichen Schema darstellen lassen. Unter anderem ermöglicht die \uparrow -Notation, Operatoren wie die Hyperpotenz aufzuschreiben, für die in der klassischen Mathematik kein natives Symbol existiert.

Vollständige Induktion
☞ 2.4.1

Neben dem direkten und dem indirekten Beweis ist die vollständige Induktion die dritte klassische Beweistechnik der Mathematik. Sie ist immer dann anwendbar, wenn eine parametrisierte Aussage $A(n)$ für alle natürlichen Zahlen n bewiesen werden soll. Ein Induktionsbeweis erfolgt in drei Schritten: Zunächst wird im Induktionsanfang die Aussage für einen oder mehrere Basisfälle bewiesen. Im nächsten Schritt erfolgt die Annahme, dass die Aussage für ein gewisses n und alle kleineren Werte bewiesen sei (Induktionsannahme). Gelingt im Anschluss der Beweis, dass aus der Gültigkeit von $A(n)$ die Gültigkeit von $A(n+1)$ folgt, so ist die Aussage für alle n bewiesen. Eine mit der vollständigen Induktion verwandte Beweistechnik ist die ☞ strukturelle Induktion.

While-Programm
☞ 6.1.2

Programm, verfasst in der fiktiven ☞ While-Sprache. Die While-Schleife ist die einzige Möglichkeit, den Kontrollfluss zu steuern. Nach der ☞ Church'schen These lässt sich jede intuitiv berechenbare Funktion mithilfe eines While-Programms berechnen.

While-Sprache
☞ 6.1.2

Menge aller ☞ While-Programme. Nach der ☞ Church'schen These sind alle intuitiv berechenbaren Funktionen f While-berechenbar, d. h., es existiert ein While-Programm, das x als Eingabe entgegennimmt und $f(x)$ als Ausgabe liefert. Die While-Sprache besitzt die gleiche Berechnungsstärke, wie die ☞ Goto-Sprache, die ☞ Turing-Maschine, die ☞ Registermaschine und die ☞ μ -rekursiven Funktionen.

Widerspruchskalkül
☞ 3.1.3

Spezieller ☞ Kalkül, der die ☞ Allgemeingültigkeit einer Formel über die ☞ Unerfüllbarkeit der negierten Aussage beweist. Bekannte Vertreter sind der ☞ Resolutionskalkül und der ☞ Tableaukalkül.

Wortproblem
☞ 4.1

Wichtige Problemstellung aus dem Bereich der ☞ formalen Sprachen. Hinter dem Wortproblem verbirgt sich die Frage, ob ein bestimmtes Wort ω in einer Sprache L enthalten ist. Mit anderen Worten: Gilt $\omega \in L$?

Zellulärer Automat
☞ 5.8

Zustandsbasiertes System, das aus einer großen Anzahl simultan arbeitender Elementarautomaten, den sogenannten Zellen, besteht. Zu jedem Zeitpunkt befinden sich diese in einem von endlich vielen Zuständen. In jedem Schaltschritt geht eine Zellen in eine Folgekonfiguration über, die sowohl durch ihren eigenen Zustand als auch durch die Zustände ihrer Nachbarn bestimmen wird. Hierdurch stehen die Zellen in ständiger Interaktion. Eingesetzt wird das Modell vor allem zur Beschreibung dynamischer, selbstorganisierender Systeme.

Literaturverzeichnis

- [1] Ackermann, W.: Zum Hilbert'schen Aufbau der reellen Zahlen. In: *Mathematische Annalen* 99 (1928), S. 118–133
- [2] Agrawal, M.; Kayal, N.; Saxena, N.: PRIMES is in P. In: *Annals of Mathematics* 160 (2004), Nr. 2, S. 781–793
- [3] Aho, A. V.; Lam, M. S.; Sethi, R.; Ullman, J. D.: *Compilers. Principles, Techniques, and Tools*. Amsterdam: Addison-Wesley, 2006
- [4] Amos, M.: *Theoretical and Experimental DNA Computation*. Berlin, Heidelberg, New York: Springer-Verlag, 2005
- [5] Bachmann, P.: *Zahlentheorie. Versuch einer Gesamtdarstellung dieser Wissenschaft in ihren Haupttheilen. Zweiter Theil*. Leipzig: Teubner Verlag, 1894
- [6] Bauer, F. L.: *Entzifferte Geheimnisse, Methoden und Maximen der Kryptographie*. Berlin, Heidelberg, New York: Springer-Verlag, 2000
- [7] Biggs, N. L.; Lloyd, E. K.; Wilson, R. J.: *Graph Theory 1736–1936*. Oxford: Oxford University Press, 1986
- [8] Bois-Reymond, E. H. D.: *Über die Grenzen des Naturerkennens*. Saarbrücken: VDM Verlag Dr. Müller, 2006
- [9] Boole, G.: *An Investigation of the Laws of Thought*. London: Walton and Maberley, 1854. – Nachgedruckt in [10]
- [10] Boole, G.; Corcoran, J.: *The Laws of Thought (Reprint)*. New York: Prometheus Books, 2003
- [11] Kapitel The Busy Beaver Game and the Meaning of Life. In: Brady, A. H.: *The Universal Turing Machine: A Half Century Survey*. Oxford: Oxford University Press, 1991, S. 259–277
- [12] C. H. Edwards, Jr.: *The Historical Development of the Calculus*. Berlin, Heidelberg, New York: Springer-Verlag, 1994
- [13] Cantor, G.: Beiträge zur Begründung der transfiniten Mengenlehre. In: *Mathematische Annalen* 46 (1895), Nr. 4, S. 481–512
- [14] Cantor, G.; Zermelo, E. (Hrsg.): *Gesammelte Abhandlungen mathematischen und philosophischen Inhalts*. Hildesheim: Georg Olms Verlag, 1966
- [15] Casiro, F.: Das Hotel Hilbert. In: *Spektrum der Wissenschaft Spezial* 2 (2005), S. 76–80
- [16] Chomsky, N.: Three models for the description of language. In: *IEEE Transactions on Information Theory* 2 (1956), Nr. 3, S. 113–124
- [17] Chomsky, N.: *Syntactic Structures*. Berlin: Mouton de Gruyter, 2002
- [18] Church, A.: A Note on the Entscheidungsproblem. In: *Journal of Symbolic Logic* 1 (1936), Nr. 1, S. 40–41
- [19] Church, A.: An Unsolvable Problem of Elementary Number Theory. In: *American Journal of Mathematics* 58 (1936), Nr. 2, S. 345–363
- [20] Church, A.: A Formulation of the Simple Theory of Types. In: *Journal of Symbolic Logic* 5 (1940), S. 56–68
- [21] Cobham, A.: The intrinsic computational difficulty of functions. In: Bar-Hillel, Y. (Hrsg.): *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*. Amsterdam: North Holland, 1964, S. 24–30
- [22] Cocke, J.: *Programming languages and their compilers: Preliminary notes*. New York: Courant Institute of Mathematical Sciences, New York University, 1969
- [23] Cohen, P.: The Independence of the Continuum Hypothesis. In: *Proceedings of the National Academy of Sciences of the United States of America* Bd. 50. Washington, DC: National Academy of Sciences, 1963, S. 1143–1148
- [24] Cook, S. A.: The Complexity of Theorem-Proving Procedures. In: *Proceedings of the 3rd Annual ACM*

- Symposium on Theory of Computing.* New York: ACM Press, 1971, S. 151–158
- [25] Dedekind, R.: *Was sind und was sollen die Zahlen?* Braunschweig, 1918
- [26] Edmonds, J.: Paths, Trees, and Flowers. In: *Canadian Journal of Mathematics* 17 (1965), Nr. 3, S. 449–467
- [27] Engesser, H. (Hrsg.); Claus, V. (Hrsg.); Schwill, A. (Hrsg.): *Duden Informatik.* Mannheim: Dudenverlag, 1988
- [28] Ertel, W.: *Angewandte Kryptographie.* München: Hanser-Verlag, 2007
- [29] Euklid; Thaer, Clemens (Hrsg.): *Die Elemente. Buch I - XIII.* Frankfurt: Verlag Harri Deutsch, 2003 (Ostwalds Klassiker)
- [30] Euler, L.: Solutio problematis ad geometriam situs pertinentis. In: *Commentarii academiae scientiarum Petropolitanae* 8 (1741), S. 128–140
- [31] Floyd, R. W.: Algorithm 97: Shortest path. In: *Communications of the ACM* 5 (1962), June, Nr. 6, S. 345
- [32] Fraenkel, A.: Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. In: *Mathematische Annalen* 86 (1922), S. 230–237
- [33] Frege, G.: *Grundgesetze der Arithmetik, Begriffsschriftlich abgeleitet.* Bd. 2. Jena: Verlag Hermann Pohle, 1903
- [34] Frege, G.: *Grundgesetze der Arithmetik, Begriffsschriftlich abgeleitet.* Bd. 2. Darmstadt: Wissenschaftliche Buchgesellschaft, 1962
- [35] Frege, G.: *Begriffsschrift und andere Aufsätze.* Hildesheim: Verlag Olms, 2007
- [36] Gödel, K.: *Über die Vollständigkeit des Logikkalküls,* Universität Wien, Diss., 1929
- [37] Gödel, K.: The consistency of the axiom of choice and of the generalized continuum-hypothesis. 24 (1938), S. 556–557
- [38] Gordon, M. J. C.; Melham, T. F.: *Introduction to HOL: A theorem proving environment for higher-order logic.* Cambridge: Cambridge University Press, 1993
- [39] Gray, F.: *U.S. Patent No. 2.632.058.* 1953
- [40] Hally, M.: *Electronic Brains: Stories from the Dawn of the Computer Age.* Washington, D.C.: Joseph Henry Press, 2005
- [41] Harel, D.: *First-Order Dynamic Logic.* Berlin, Heidelberg, New York: Springer-Verlag, 1979 (Lecture Notes in Computer Science)
- [42] Hartmanis, J.; Stearns, R. E.: On the computational complexity of algorithms. In: *Transactions of the American Mathematical Society* 117 (1965), S. 285–306
- [43] Herbrand, J.: *Recherches sur la théorie de la démonstration,* University of Paris, Diss., 1930
- [44] Hermes, H.: *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit.* Berlin, Heidelberg, New York: Springer-Verlag, 1961
- [45] Heuser, H.: *Lehrbuch der Analysis I.* Wiesbaden: Teubner Verlag, 2006
- [46] Hilbert, D.: *Grundlagen der Geometrie.* Leipzig: Teubner Verlag, 1899
- [47] Hilbert, D.: Über das Unendliche. In: *Mathematische Annalen* 95 (1926), Nr. 1, S. 161–190
- [48] Hodges, A.: *Enigma.* Berlin, Heidelberg, New York: Springer-Verlag, 1994
- [49] Hoffmann, D. W.: *Grundlagen der Technischen Informatik.* München: Hanser-Verlag, 2007
- [50] Hoffmann, D. W.: *Grenzen der Mathematik. Eine Reise durch die Kerngebiete der mathematischen Logik.* Heidelberg: Spektrum Akademischer Verlag, 2011
- [51] Hofstadter, D. R.: *Gödel, Escher, Bach: Ein endloses geflochtenes Band.* Stuttgart: Klett-Cotta, 2006
- [52] Hopcroft, J. E.; Ullman, J. D.; Motwani, R.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* München: Pearson Studium, 2003
- [53] Immermann, N.: Nondeterministic space is closed under complementation. In: *SIAM Journal on Computing* 17 (1988), Nr. 5, S. 935–938
- [54] Kapitel Reducability Among Combinatorial Problems. In: Karp, R. M.: *Complexity of Computer Computations.* Plenum Press, 1972, S. 85–103
- [55] Kasami, T.: An efficient recognition and syntax analysis algorithm for context free languages. (1965), Nr. AF CRL-65-758
- [56] Kellerer, H.; Pferschy, U.; Pisinger, D.: *Knapsack Problems.* Berlin, Heidelberg, New York: Springer-Verlag, 2007

- [57] Kernighan, B. W.: *Programmieren in C*. München: Hanser Fachbuchverlag, 1990
- [58] Kernighan, B. W.; Pike, R. P.: *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice Hall, 1984
- [59] Kleene, S. C.: Lambda-definability and recursiveness. In: *Duke Mathematical Journal* 2 (1936), S. 340–353
- [60] Kleene, S. C.: *Representation of events in nerve nets and finite automata. Project RAND Research Memorandum RM-704*. Santa Monica, CA: RAND Corporation, 1951
- [61] Knuth, D. E.: Mathematics and Computer Science: Coping with Finiteness. In: *Science Magazine* 194 (1976), Nr. 4271, S. 1235–1242
- [62] Knuth, D. E.: *The Art of Computer Programming, Volume 3: Sorting and Searching*. Redwood City, CA: Addison Wesley Longman Publishing Co., Inc., 1998
- [63] Kowalski, R. A.: The early years of logic programming. In: *Communications of the ACM* 31 (1988), Nr. 1, S. 38–43
- [64] Landau, E.: *Handbuch der Lehre von der Verteilung der Primzahlen*. Leipzig: Teubner Verlag, 1909
- [65] Levin, L.: Universal Sequential Search Problems. In: *Problems of Information Transmission* 9 (1973), Nr. 3, S. 265–266
- [66] Levin, L.: Randomness Conservation Inequalities: Information and Independence in Mathematical Theories. In: *Information and Control* 61 (1984), April, Nr. 1, S. 15–37
- [67] Mandelbrot, B.: *Die Fraktale Geometrie der Natur*. Basel: Birkhäuser-Verlag, 1987
- [68] RWTH Aachen, IRT: *Netlab homepage*. <http://www.irt.rwth-aachen.de>. Version: 2008
- [69] McCulloch, W.; Pitts, W.: A Logical Calculus of the Ideas Immanent in Nervous Activity. In: *Bulletin of Mathematical Biophysics* 5/115 (1943)
- [70] Mealy, G. H.: A Method for Synthesizing Sequential Circuits. In: *Bell Systems Technical Journal* 34 (1955), September, S. 1045–1079
- [71] Mendelson, E.: *Introduction to Mathematical Logic*. 4th edition. Boca Raton, FL: Chapman & Hall, CRC Press, 1997
- [72] Menzel, W.; Schmitt, P. H.: *Formale Systeme. Vorlesungsskript Wintersemester 94/95*. Universität Karlsruhe, 1994
- [73] Minsky, M. L.: Size and Structure of Universal Turing Machines Using Tag Systems. In: *Recursive Function Theory: Proceedings, Symposium in Pure Mathematics* Bd. 5. Providence: American Mathematical Society, 1962, S. 229–238
- [74] Modrow, E.: *Theoretische Informatik mit Delphi*. Norderstedt: Books on Demand, 2005
- [75] Moore, E. F.: Gedanken-Experiments on Sequential Machines. In: Shannon, C. E. (Hrsg.); McCarthy, J. (Hrsg.): *Automata Studies*. Princeton, NJ: Princeton University Press, 1956, S. 129–153
- [76] Myhill, J.: Finite automata and the representation of events / Wright Air Development Command. 1957 (57-624). – Forschungsbericht. – Technical Report
- [77] Nerode, A.: Linear Automaton Transformations. In: *Proceedings of the AMS* Bd. 9, American Mathematical Society, 1951, S. 541–544
- [78] Neumann, J. von; Burks, A. W. (Hrsg.): *Theory of self-reproducing automata*. Urbana: University of Illinois Press, 1966
- [79] Nielsen, M. A.; Chuang, I. L.: *Quantum Computation and Quantum Information*. Cambridge: Cambridge University Press, 2000 (Cambridge Series on Information & the Natural Sciences)
- [80] Peano, G.: *Arithmetices principia, nova methodo exposita*. Bocca: Augustae Taurinorum, 1889
- [81] Petri, C. A.: Kommunikation mit Automaten. In: *Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik* (1962)
- [82] Post, E. L.: Formal reductions of the combinatorial decision problem. In: *American Journal of Mathematics* 65 (1943), Nr. 2, S. 197–215
- [83] Rabin, M.; Scott, D.: Finite automata and their decision problems. In: *IBM Journal of Research and Development* 3 (1959), S. 114–125
- [84] Rado, T.: On Non-Computable Functions. In: *The Bell System Technical Journal* 41 (1962), Nr. 3, S. 877–884
- [85] Rechenberg, P.: *Technisches Schreiben. (Nicht nur) für Informatiker*. München: Hanser-Verlag, 2006
- [86] Robinson, S.: New Method Said to Solve Key Problem in Math. (2002), August, 8

- [87] Savitch, W. J.: Relationships Between Nondeterministic and Deterministic Tape Complexities. In: *Journal of Computer and System Sciences* 4 (1970), Nr. 2, S. 177–192
- [88] Schöning, U.: *Logik für Informatiker*. Heidelberg: Spektrum Akademischer Verlag, 2000
- [89] Schöning, U.: *Theoretische Informatik – kurzgefasst*. Heidelberg: Spektrum Akademischer Verlag, 2001
- [90] Sedgewick, R.: *Algorithmen*. München: Pearson Studium, 2002
- [91] Sierpinski, W.: Sur une nouvelle courbe qui remplit toute une aire plaine. In: *Bull. Acad. Sci. Cracovie Serie A* (1912), S. 462–478
- [92] Solovay, R. M.; Strassen, V.: A fast Monte-Carlo test for primality. In: *SIAM Journal on Computing* 6 (1977), Nr. 1, S. 84–85
- [93] Stern, N.: Who Invented the First Electronic Digital Computer? In: *Annals of the History of Computing* 2 (1980), October, Nr. 4, S. 375–376
- [94] Strassen, V.: Gaussian Elimination is not Optimal. In: *Numerische Mathematik* 14 (1969), Nr. 3, S. 354–356
- [95] Szelepcsenyi, R.: The method of forced enumeration for nondeterministic automata. In: *Acta Informatica* 26 (1988), Nr. 3, S. 279–284
- [96] Thue, A.: Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. In: *Skrifter utgit av Videnskapsselskapet i Kristiania* I.10 (1914)
- [97] Turing, A. M.: On computable numbers with an application to the Entscheidungsproblem. In: *Proceedings of the London Mathematical Society* 2 (1936), Juli – September, Nr. 42, S. 230–265
- [98] Turing, A. M.: Computability and Lambda-Definability. In: *Journal of Symbolic Logic* 2 (1937), Nr. 4, S. 153–163
- [99] Turing, A. M.: Computing Machinery and Intelligence. In: *Mind* 59 (1950), S. 433–460
- [100] Venn, J.: *Symbolic Logic*. London: MacMillan Publishing, 1881
- [101] Venn, J.: *Symbolic Logic (Reprint)*. Providence, RI: AMS Chelsea Publishing, American Mathematical Society, 2007
- [102] Viterbi, A. J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In: *IEEE Transactions on Information Theory* 13 (1967), Nr. 2, S. 260–269
- [103] Vossen, G.; Witt, K.-U.: *Grundkurs Theoretische Informatik*. Wiesbaden: Vieweg-Verlag, 2006
- [104] Warshall, S.: A Theorem on Boolean Matrices. In: *Journal of the ACM* 9 (1962), January, Nr. 1, S. 11–12
- [105] Wermke, D. (Hrsg.); Kunkel-Razum, K. (Hrsg.); Scholze-Stubenrecht, W. (Hrsg.): *Duden, Die deutsche Rechtschreibung*. Mannheim: Dudenverlag, 2004
- [106] Wikipedia: *Ogdens Lemma*. Webartikel. http://de.wikipedia.org/wiki/Ogdens_Lemma
- [107] Williams, J. W. J.: Algorithm 232: Heapsort. In: *Communications of the ACM* 7 (1964), S. 347–348
- [108] Wolfram, S.: *A New Kind of Science*. Champaign, IL: Wolfram Media, Inc., 2002
- [109] Wolfram, S.: The Prize Is Won; The Simplest Universal Turing Machine is Proved. In: *Wolfram Blog* (2007), October. <http://blog.wolfram.com>
- [110] Younger, D. H.: Recognition and Parsing of Context-Free Languages in Time n^3 . In: *Information and Control* 10 (1967), Nr. 2, S. 189–208
- [111] Zermelo, E.: Untersuchungen über die Grundlagen der Mengenlehre. In: *Mathematische Annalen* 65 (1908), S. 261–281
- [112] Zermelo, E.: Über Grenzzahlen und Mengenbereiche. In: *Fundamenta Mathematicae* 16 (1930), S. 29–47

Namensverzeichnis

A

- Ackermann, Wilhelm F., 56
Adleman, Leonard M., 27
Agrawal, Manindra, 33
Allen, Frances E., 27

B

- Bachman, Charles W., 27
Backus, John W., 27, 181
Bar-Hillel, Yehoshua, 185
Binet, Jacques P. M., 79
Blum, Manuel, 27
Boole, George, 43, 44
Brahmagupta, 54
Brooks, Frederick P. Jr., 27
Brown, Dan, 390

C

- Cantor, Georg F. L. P., 17, 38, 415
Carbató, Fernando J., 27
Cerf, Vinton G., 27
Chomsky, A. Noam, 25, 168, 179
Church, Alonzo, 277, 300, 301, 308, 405
Clarke, Edmund M., 27
Cocke, John, 27, 186, 405
Codd, Edgar F., 27
Cohen, Paul, 64
Cohen, Paul J., 39
Collatz, Lothar, 78
Cook, Stephen A., 27, 32, 373, 376, 415

D

- da Pisa, Leonardo, 390
Dahl, Ole-Johan, 27

- De Morgan, Augustus, 44
Dedekind, J. W. Richard, 52, 54
Descartes, René, 58
Dijkstra, Edsger W., 27
Dirichlet, Peter G. L., 91
Du Bois-Reymond, Emil H., 19

E

- Eckert, J. Presper, 23
Emerson, E. Allen, 27
Engelbart, Douglas, 27
Euklid von Alexandria, 54
Euler, Leonhard, 31, 74

F

- Feigenbaum, Edward, 27
Fibonacci, 390
Floyd, Robert W., 27
Fraenkel, Abraham A. H., 17, 39, 415
Frege, Gottlob, 16

G

- Garey, Michael, 32
Gauß, J. Carl Friedrich, 58
Gray, Jim, 27
Gödel, Kurt, 149

H

- Hamilton, Sir William, 30
Hamming, Richard, 27
Hartmanis, Juris, 27
Herbrand, Jacques, 125, 415
Hilbert, David, 15
Hoare, C. Antony R., 27

- Hofstadter, Douglas R., 35
Hopcroft, John E., 27
Horn, Alfred, 142

I

- Iverson, Kenneth E., 27

J

- Johnson, David, 32

K

- Kahan, William, 27
Kahn, Robert E., 27
Karp, Richard M., 27, 32, 382
Kasami, Tadao, 186, 405
Kay, Alan, 27
Kayal, Neeraj, 33
Kleene, Stephen C., 26, 264, 268, 300, 301, 409, 415
Knuth, Donald E., 27, 56, 418

L

- Lampson, Butler W., 27
Landau, Edmund G. H., 350, 410
Leibniz, Gottfried W., 54, 58
Levin, Leonid, 32
Liskov, Barbara, 27

M

- Mauchly, John W., 23
McCarthy, John, 27, 300
McCulloch, Warren S., 26
Mealy, George H., 26

Milner, Robin, 27
Minsky, Marvin L., 27, 294, 295
Moore, Edward F., 26
Myhill, John R., 174

N

Naur, Peter, 27
Nerode, Anil, 174
Neumann, John von, 71
Newell, Allen, 27
Newton, Sir Isaac, 54
Nygaard, Kristen, 27

P

Peano, Giuseppe, 52
Perlis, Alan J., 27
Petri, Carl A., 238
Pitts, Walter, 26
Pnueli, Amir, 27
Post, Emil L., 326, 413

R

Rabin, Michael O., 26, 27, 211, 363, 415
Radó, Tibor, 337

Reddy, Raj, 27
Rice, Henry G., 324, 415
Ritchie, Dennis M., 27
Rivest, Ronald L., 27
Rosser, J. Barkley, 301
Russell, Bertrand A. W., 17, 415

S

Savitch, Walter, 367, 415
Saxena, Nitin, 33
Scott, Dana S., 26, 27, 211, 363, 415
Shamir, Adi, 27
Sifakis, Joseph, 27
Simon, Herbert A., 27
Smith, Alex, 295
Solovay, Robert M., 32
Stearns, Richard E., 27
Stirling, James, 73
Strassen, Volker, 32, 391
Sutherland, Ivan, 27

T

Tarjan, Robert E., 27
Thacker, Charles P., 27
Thompson, Kenneth L., 27

Thue, Axel, 169
Tolkien, John R. R., 138
Turing, Alan M., 20, 21, 277, 279, 301

V

Venn, John, 41
Viterbi, Andrew J., 186

W

Whitehead, Alfred N., 17
Wilkes, Maurice V., 27
Wilkinson, J. H., 27
Wirth, Niklaus E., 27
Wolfram, Stephen, 244, 294, 295

Y

Yao, Andrew Chi-Chih, 27
Younger, Daniel H., 186, 405

Z

Zermelo, Ernst F. F., 17, 39, 71, 415
Zuse, Konrad, 22

Sachwortverzeichnis

Symbole

μ -Operator, 274
 μ -Rekursion, 274
 μ -rekursive Funktion, 275, 411
3SAT, 380, 403
4er-Nachbarschaft, 244
8er-Nachbarschaft, 244

A

Abbildung, 50
Ableitungsrelation, 96
Abschwächungsregel, 98
Absolute Adressierung, 296
Abstraktion, 300
Abtrennungsregel, 17
Abzählbare Sprache, 310
Abzählbarkeit, 59, 310, 403
Ackermann-Funktion, 56, 403
Addierer
 Carry-look-ahead-, 116
 Carry-ripple-, 114
Adressierung
 absolute, 296
 indirekte, 296
 unmittelbare, 296
Äquivalenz, 87
Akkumulator, 296
AKS-Algorithmus, 33
Akzeptierende Turing-Maschine, 312
Akzeptierender Automat, 203, 403
Akzeptor, 203, 403
 Minimierung, 206
 Turing-, 312
Algorithmische Komplexität, 342
Algorithmus

CYK-, 186, 405
effektiver, 27
effizienter, 27
Gilmore-, 128, 407
Las-Vegas-, 32
Monte-Carlo-, 32
randomisierter, 32
rekursiver, 271
Robinson-, 131, 414
Strassen-, 391
Allgemeingültigkeit, 85, 403
 prädikatenlogische, 122
Allgemeiner Unifikator, 131, 403
Alphabet, 162
Antinomie, 403
 Russell'sche, 17, 39, 415
Antivalenzoperator, 83
Äquivalenz, 87
 -klasse, 44
 -operator, 83
 -problem, 163, 403
 -relation, 49
Arbeitsband, 292
Asymptotische Komplexität, 350
Asymptotisches Wachstum, 403
Atomare Aussage, 82, 404
Atomare Formel, 83
Aufzählbare Sprache, 310
Aufzählbarkeit, 310
Ausdruck
 regulärer, 26, 176, 217, 414
Auszabealphabet
 von Transduktoren, 230
Auszabeband, 296
Ausgabeschaltnetz, 233
Aussage
 atomare, 82, 404
Aussagenlogik, 23, 82, 404
 Normalformen, 91
Auswahlaxiom, 39
Automat
 äquivalenter, 206
 akzeptierender, 203, 403
 DEA, 204, 405
 deterministischer, 204
 endlicher, 25, 201, 406
 Keller-, 223, 409
 linearer, 244
 Mealy-, 203, 234, 411
 Moore-, 203, 234, 411
 NEA, 209, 411
 nichtdeterministischer, 208
 Potenzmengen-, 211, 413
 Produkt-, 219
 reduzierter, 206
 übersetzender, 203, 230
 zellulärer, 243, 252, 418
Automatenminimierung, 404
 von Akzeptoren, 206
 von Transduktoren, 231
Automatensynthese, 233, 404
Automatentheorie, 25, 201
Axiom, 35, 404
Axiome
 von Peano, 52

B

Backtracking, 145
Backus-Naur-Form, 181, 404
 erweiterte, 181
Bandalphabet
 von Turing-Maschinen, 279
Bandplatzfunktion, 365

Bar-Hillel-Theorem, 185
 Barbier-Paradoxon, 34, 404
 Basis, 306
 BCD-Code, 249
 Befehlszähler, 296
 Belegung, 84
 Berechenbarkeit, **254**, 302, 404
 Goto-, 266
 Loop-, 256
 Turing-, 281, 417
 While-, 261
 Berechenbarkeitstheorie, 253, 404
 Berechnungsmodell, 254, 404
 Beweis
 direkter, 66
 durch Widerspruch, 66
 induktiver, 65
 Beweistheorie
 aussagenlogische, 96
 prädikatenlogische, 124
 Biberfunktion, 337
 Bijektive Funktion, 51
 Bild, 51
 Binärbaum, 68
 balancierter, 68
 saturierter, 68
 Binäre Codierung, 282
 Binäre Suche, 387
 Binomialkoeffizient, 79
 Binomischer Lehrsatz, 79
 Bisimulation, **206**, **231**, 405
 Blättermenge, 68
 Blank-Symbol, 279
 Boolesche Algebra, 43
 Boolesche Funktion, 85
 Brute-Force-Methode, 363

C

Cantor'sche Paarungsfunktion, 61, 75
 Cantor-Maschine, 312
 Carry bit, 114
 Carry-look-ahead-Addierer, 116
 Carry-ripple-Addierer, 114
 CD, 248
 Charakteristische Funktion, 309, 405
 Chomsky-Hierarchie, 25, **168**, 405

Chomsky-Normalform, 179, 405
 Church'sche These, 254, **302**, 308, 405
 Church-Rosser-Eigenschaft, 301
 CLIQUE, 383
 Co-Komplexität, 369, 405
 COBOL, 24
 Code
 einschrittiger, 231
 Codierung
 binäre, 282
 unäre, 282
 Collatz-Funktion, 78
 Cook
 Satz von, **373**, 383
 Cook, Satz von, 415
 CYK-Algorithmus, 186, 405

D

Datenspur, 287
 DEA, 204, 405
 Deadlock, 242
 Dedekind'scher Schnitt, 54
 Deduktion, 81
 Deduktionsbeweis, 66
 Definition
 rekursive, 65
 Definitionsmenge, 50
 Deklarative Programmierung, 138
 Deterministischer Automat, 204
 Diagonalsierung, 38, 405
 Diagonalsierungsargument, 62
 Diagonalsprache, 338
 Differenzmenge, 42
 Dirichlet'sches Schubfachprinzip, 90,
 406
 Disjunktion, 82
 Disjunktive Form, 406
 Disjunktive Minimalform, 95
 Disjunktive Normalform, 95, 406
 kanonische, 93
 Distributivgesetz, 42, 98
 Divide and conquer, 356
 DNA computing, 308
 DNF, 94
 DVD, 248
 Dyck-Sprache, 165, 227

Dynamische Logik, 295
 Dynamische Programmierung, **186**,
 346, 406

E

Ebene
 Meta-, 82
 Objekt-, 82
 Einband-Turing-Maschine, 277
 Eingabealphabet
 von ε -Automaten, 213
 von DEAs, 204
 von Kellerautomaten, 224
 von NEAs, 209
 von Transduktoren, 230
 von Turing-Maschinen, 279

Eingabeband, 296
 Einschrittiger Code, 231
 Element, 38
 Elementaroperatoren, 89
 Endlicher Automat, 25, **201**, 406
 Endlichkeitsproblem, 162, 406
 Endrekursion, 272
 Endzustand
 von ε -Automaten, 213
 von DEAs, 204
 von NEAs, 209
 von Turing-Maschinen, 279

Enigma, 23
 Entscheidbare Sprache, 313
 Entscheidbarkeit, 19, **309**, 406
 Semi-, 309, 416

Epsilon-Übergang, 212, 213, 406
 Erfüllbarkeit, 406

 aussagenlogische, 85
 prädikatenlogische, 122

Erfüllbarkeitsäquivalenz, 123, 406

Erreichbarkeitsanalyse, 241

Euklidische Axiome, 15

Euler-Kreis, 29

EXP, 367, 406

F

Faktorisierungsregel, 134
 Faktum

- in Prolog, 138
 Ferritkernspeicher, 24
 Fibonacci-Folge, 390
 Finalzustand
 von ε -Automaten, 213
 von DEAs, 204
 von NEAs, 209
 von Turing-Maschinen, 279
 Fixpunkt, 208
 -operator, 301
 Fleißiger Biber, 337
 Flipflop, 233
 Formale Sprache, 25, 162, 406
 Formales System, 12, 35, 407
 Formel
 aussagenlogische, 82
 bereinigte, 119
 erfüllbarkeitsäquivalente, 123
 geschlossene, 119
 prädikatenlogische, 119
 Formulario-Projekt, 53
 FORTRAN, 24
 Funktion, 44, 50
 μ -rekursive, 275, 411
 Ackermann-, 56, 403
 bijektive, 51
 boolesche, 85
 charakteristische, 309, 405
 injektive, 51
 partielle, 51, 258, 412
 platzkonstruierbare, 366
 primitiv-rekursive, 269, 413
 surjektive, 51
 totale, 51, 417
 unberechenbare, 319
 zeitkonstruierbare, 361
 Funktions
 -variable, 147
 Funktionstabelle, 85
 Funktionswert, 51
- G**
- Gegenbeispiel, 109
 Generative Grammatik, 25
 Gilmore-Algorithmus, 128, 407
 Gleichheitsrelation, 47
- Gödelsierung, 291, 407
 Gödelnummer, 291, 321, 339, 407
 Goto-Berechenbarkeit, 266
 Goto-Programm, 264, 407
 Goto-Sprache, 264, 407
 Grad
 von Polynomen, 353
 Grammatik, 162–164, 407
 eindeutige, 167
 generative, 25
 kontextfreie, 168, 179, 409
 kontextsensitive, 168, 191, 410
 mehrdeutige, 167
 rechtslineare, 170
 reguläre, 168, 170, 414
 Gray-Code, 231
 Greibach-Normalform, 197, 407
 Grundinstanz, 127, 407
 Grundlagenkrise, 14
 Grundmenge, 120
 Grundsubstitution, 120
- H**
- Halteproblem, 21, 319, 407
 allgemeines, 320
 auf leerem Band, 322, 408
 spezielles, 339, 416
 Hamilton-Kreis, 30
 Hamilton-Problem, 364, 408
 Hardware-Entwurf, 112
 Haskell, 300
 Head recursion, 272
 Herbrand
 Satz von, 126, 415
 Herbrand-Interpretation, 126, 408
 Herbrand-Modell, 126, 408
 Herbrand-Universum, 125, 408
 Hilbert-Kalkül, 98, 408
 Hilbert-Wüste, 76
 Hilberts Hotel, 61
 Horn-Formel, 143
 Hülle
 Kleene'sche, 162
 reflexiv-transitive, 47, 48
 transitive, 47
 Huffman-Normalform, 234
- I**
- Identität, 47
 Ignorabimus, 19
 Imaginäre Einheit, 71
 Implikation, 82
 Indirekte Adressierung, 296
 Individuenbereich, 120
 Induktion
 strukturelle, 68, 416
 vollständige, 66, 418
 Induktionsaxiom, 408
 Induktiver Beweis, 65
 Injektive Funktion, 51
 Instanzen, 99
 Interpretation, 84, 120, 408
 Herbrand-, 126, 408
 Inverses Element, 42
 Inzidenzmatrix, 239, 408
 Irrationale Zahl, 54
 Isomorphie, 212
 Iterationslemma, 185
- K**
- Kalkül, 12, 35, 96, 408
 Hilbert-, 98, 408
 Lambda-, 300
 Resolutions-, 104, 130, 414
 Tableau-, 109, 135, 416
 Widerspruchs-, 97, 418
 Kapazität, 241
 Kardinalität, 38, 58
 Kardinalzahl, 64, 408
 Kartesisches Produkt, 44
 KDNF, 93
 Kellaralphabet, 224
 Kellerautomat, 223, 409
 deterministischer, 228, 229
 Kellerspeicher, 409
 Kettenregel, 180
 KKNF, 93
 Klausel, 95, 409
 leere, 95
 Klauseldarstellung, 95
 Kleene
 Satz von, 264, 415

- K**
- Kleene'sche Hülle, 162
 - Kleene'sche Normalform, **268**, 304, 409
 - KNF, 94
 - Königsberger Brückenproblem, 29, 409
 - Kollisionsfreiheit, 135
 - Kommutativgesetz, 42
 - Komplementärautomat, 218
 - Komplementärmenge, 42
 - Komplexe Zahl, 71
 - Komplexitätsklasse, 29, **356**, 409
 - komplementäre, 369, 405
 - Komplexitätstheorie, 341, 409
 - Komposition, 270
 - Konfiguration, 409
 - globale, 244
 - lokale, 244
 - von ε -Automaten, 213
 - von DEAs, 205
 - von Kellerautomaten, 225
 - von NEAs, 210
 - von Petri-Netzen, 239
 - von Turing-Maschinen, 280
 - Konjunktion, 82
 - Konjunktive Form, 409
 - Konjunktive Minimalform, 95
 - Konjunktive Normalform, 95, 409
 - kanonische, 93
 - Konklusion, 98
 - Kontextfreie Grammatik, 168, **179**, 409
 - Kontextfreie Sprache, **179**, 226, 410
 - Kontextsensitive Grammatik, 168, **191**, 410
 - Kontextsensitive Sprache, 191, 410
 - Kontinuum, 64
 - Kontinuumshypothese, 64
 - Kontraposition, 98
 - Konversionsregel, 300
 - Kopfrekursion, 272
 - Kopfzelle, 293
 - Korrektheit, 96
 - Korrespondenzproblem, 326, 413
 - binäres, 340
 - modifiziertes, 328
- L**
- Lambda-Kalkül, 300
- M**
- Lambda-Term, 300
 - Landau-Symbole, 349, 410
 - Las-Vegas-Algorithmus, 32
 - Last-In-First-Out, 224
 - Latch, 233
 - Laufzeitfunktion, 359
 - LBA-Problem
 - erstes, 318
 - zweites, 318
 - Lebendigkeit
 - von Petri-Netzen, 241
 - Leere Klausel, 95
 - Leere Menge, 39
 - Leerheitsproblem, 162, 410
 - LIFO, 224
 - Lineare Rekursion, 272
 - Lineare Suche, 387
 - Linearer Automat, 244
 - Linksableitung, 165, 410
 - Literal, 92, 410
 - Logik, 81, 410
 - Aussagenlogik, 23, **82**, 404
 - dynamische, 295
 - höherer Stufe, 147, 410
 - Prädikatenlogik, 117, 413
 - Logikminimierung, 95
 - Logische Folgerung, 87
 - Logische Programmierung, 138
 - Logizismus, 16, 35
 - Loop-Berechenbarkeit, 256
 - Loop-Programm, 254, 410
 - Loop-Sprache, 254, 410
- N**
- Mächtigkeit, 58, 410
 - Makro, 257
 - Marke, 238, 412
 - Markenindex, 265
 - Markierungsgleichung, 241
 - Maschinenkomposition, 288
 - Matrix, 122
 - Maxterm, 92, 411
 - Mealy-Automat, 203, **234**, 411
 - Mehrband-Turing-Maschine, 286
 - Mehrdeutigkeitsproblem, 411
 - Mehrspur-Turing-Maschine, 286
 - Menge, 38
 - disjunkte, 41
 - leere, 39
 - Null-, 40
 - unentscheidbare, 319
 - wohlgeordnete, 72
 - Mengenalgebra, 42
 - Mengenlehre, 16
 - axiomatische, 39
 - Cantor'sche, 38
 - Fraenkel-, 39
 - Zermelo-Fraenkel-, 39
 - Mengenoperation, 41
 - Metaebene, 82
 - Millenium-Probleme, 32
 - Minimalform, 95
 - disjunktive, 95
 - konjunktive, 95
 - Minimierung, 404
 - Logik-, 95
 - von Akzeptoren, 206
 - von Transduktoren, 231
 - Minterm, 92, 411
 - Miranda, 300
 - ML, 300
 - Modell, **84**, **121**, 411
 - Herbrand-, 126, 408
 - Modellrelation, **84**, 120, 121
 - Modus barbara, 150
 - Modus ponens, 17, **98**, 411
 - Monte-Carlo-Algorithmus, 32
 - Moore-Automat, 203, **234**, 411
 - Moore-Nachbarschaft, 244

NEXP, 367, 411
 Nichtdeterministischer Automat, 208
 Nichtterminal, 163
 Nonterminal, 163, 164
 Normalform, 92, 179
 aussagenlogische, 91
 Chomsky-, 179, 405
 disjunktive, 406
 Greibach-, 197, 407
 Huffman-, 234
 kanonische
 disjunktive, 93
 konjunktive, 93
 Kleene'sche, **268**, 304, 409
 konjunktive, 409
 Negations-, 122, 411
 prädikatenlogische, 122
 NP, 359, 411
 NP-hart, 372, 412
 NP-vollständig, 31, **371**, 412
 NPSPACE, 365, 412
 Nullmenge, **40**

O
 O-Kalkül, 349
 O-Notation, 349, 412
 Obermenge, 41
 Objektebene, 82
 ODER-Operator, 82
 Ogdens Lemma, 185
 Operator, 51, 55
 Operatorensystem
 vollständiges, 89
 Orakel, 364
 Ordnung
 lineare, 50
 partielle, 50
 totale, 50
 Ordnungsrelation, 50

P
 P, 359, 412
 P-NP-Problem, 372, 412
 Paarungsfunktion, **61**, 258, 412
 Cantor'sche, 61, 75

Palindromsprache, 196, 226
 Parikh-Vektor, 239, 412
 Paritätsbit, 246
 Paritätscode, 246
 Partielle Funktion, **51**, 258, 412
 Partition, 44
 Peano-Axiome, 52, 412
 Petri-Netz, 238, 412
 Pfad
 geschlossener, 109
 offener, 109
 vollständiger, 110
 widerspruchsfreier, 109
 widersprüchlicher, 109
 Phrasenstrukturgrammatik, 168
 Phrasenstruktursprache, 193
 Pigeonhole principle, 90, 406
 Platzkonstruierbare Funktion, 366
 Polynom, 353
 Polynomielle Reduktion, 371, 413
 Pop-Operation, 224
 Positionsspur, 287
 Post'sche Tag-Maschine, 295
 Post'sches Korrespondenzproblem, 326,
 413
 binäres, 340
 modifiziertes, 328
 Potenzmenge, 44
 Potenzmengenautomat, 211, 413
 Prädikat, 117
 -variable, 147
 Prädikatenlogik, 117, 413
 Normalformen, 122
 zweiter Stufe, 147
 Prämisse, 98
 Pränex-Form, 122, 413
 Primitiv-rekursive Funktion, 269, 413
 Primitive Rekursion, 269, 413
 Principia Mathematica, 17, 18
 Problem
 unentscheidbares, 319
 Produktautomat, 219
 Produktion, 164
 Programm, 296
 Goto-, 264, 407
 Loop-, 254, 410
 While-, 260, 418
 Programmierung

deklarative, 138
 dynamische, **186**, **346**, 406
 logische, 138
 Prolog, 413
 PSPACE, 365, 414
 Pumping-Lemma, 185, 414
 für kontextfreie Sprachen, 182
 für reguläre Sprachen, 172
 Push-Operation, 224

Q

Quantenrechner, 308

R

Rabin und Scott
 Satz von, 210, 415
 Rad des Theodorus, 65
 Random access machine, 296
 Randomisierter Algorithmus, 32
 Rationale Zahl, 53
 Rechtsableitung, 165, 414
 Rechtslineare Grammatik, 170
 Reduktion, 414
 polynomielle, 371, 413
 Reduktionsbeweis, 325, 380
 Reelle Zahl, 54
 Regel, 35, 164
 in Prolog, 138
 Registermaschine, 296, 414
 verallgemeinerte, 296
 Reguläre Grammatik, 168, **170**, 414
 Reguläre Sprache, **170**, 216, 414
 Regulärer Ausdruck, 26, **176**, 217, 414
 Rekurrenzgleichung, 390
 Rekursion, 65
 μ -, 274
 lineare, 272
 primitive, 269, 413
 verschachtelte, 272
 verzweigende, 272
 wechselseitige, 272
 Rekursiv aufzählbare Sprache, 310
 Rekursive Definition, 65
 Relation, 44
 Ableitungs-, 96

Äquivalenz-, 49
inverse, 47
Ordnungs-, 50

Relationenattribut, 45

Relationenprodukt, 47

Resolutionsbaum, 105

Resolutionskalkül, 414

- aussagenlogisches, 104
- prädikatenlogisches, 130

Resolutionsregel, 105

Resolvente, 105

Rice

- Satz von, 322, 415

Robinson-Algorithmus, 131, 414

Rucksackproblem, 186, 342, 415

Russell'sche Antinomie, 17, 39, 415

S

SAT, 415

Satz

- von Cantor, 64, 415
- von Cook, 373, 383, 415
- von Herbrand, 126, 415
- von Kleene, 264, 415
- von Myhill-Nerode, 174, 221
- von Rabin und Scott, 210, 415
- von Rice, 322, 415
- von Savitch, 367, 415

Savitch

- Satz von, 367, 415

Schaltnetz, 233

- Ausgabe-, 233

- Übergangs-, 233

Schaltwerk, 25, 233

Scheme, 300

Schleife

- While-, 260

Schleifensatz, 185

Schlussregel, 416

Schnittmenge, 41

Schubfachprinzip, 90, 406

Selbstabbildung, 51

Semantik, 82, 416

Semi-entscheidbare Sprache, 313

Semi-entscheidbarkeit, 313

Semi-Entscheidbarkeit, 309, 416

Semi-Thue-System, 169
Sicherheit

- von Petri-Netzen, 241

Sierpinski-Dreieck, 245, 252
Skolem-Form, 123, 416
Speicher, 296
Speichervektor, 255
Spezielles Halteproblem, 339, 416
Sprache

- abzählbar, 310
- Diagonal-, 338
- entscheidbare, 313
- formale, 25, 162, 406
- Goto-, 264, 407
- inhärent mehrdeutige, 167
- kontextfreie, 179, 226, 410
- kontextsensitive, 191, 410
- Loop-, 254, 410
- Palindrom-, 196, 226
- Phrasenstruktur-, 193
- reguläre, 170, 216, 414
- rekursiv aufzählbare, 310
- semi-entscheidbare, 313
- unentscheidbare, 319
- While-, 260, 418

Stack, 224, 259
Stapel, 224, 259
Startkonfiguration, 245
Startsymbol, 164
Startvariable, 164
Startzustand

- von ϵ -Automaten, 213
- von DEAs, 204
- von Kellerautomaten, 224
- von NEAs, 209
- von Transduktoren, 230
- von Turing-Maschinen, 279

Stirling-Zahl, 73
Strassen-Algorithmus, 391
Strukturelle Induktion, 68, 416
Substitution, 119
Substitutionstheorem, 89
Suche

- binäre, 387
- lineare, 387

Surjektive Funktion, 51
Syllogismus, 150
Syntax, 82, 416

Syntaxbaum, 167, 416
Synthese

- von Automaten, 233, 404, 416

T

Tableau

- geschlossenes, 110
- offenes, 110
- vollständiges, 110
- widerspruchsfreies, 110

Tableaukalkül, 416

- aussagenlogisches, 109
- prädikatenlogisches, 135

Tag-Maschine, 295

Tail recursion, 272

Taubenschlagprinzip, 90, 406

Tautologie, 416

- aussagenlogische, 85
- prädikatenlogische, 122

Teile-und-herrsche-Prinzip, 356

Teilformel, 83

Teilmenge, 41

Terminal, 163

Terminalalphabet, 164

Terminierungsmenge, 261

Thue-System, 169

Totale Funktion, 51, 417

Totalordnung, 50

Trägermenge, 41

Transduktor, 203, 230, 248, 417

- Minimierung, 231

Transistor, 24

Turing-Akzeptor, 312

Turing-Berechenbarkeit, 281, 417

Turing-Bombe, 23

Turing-Maschine, 20, 277, 417

- akzeptierende, 312
- Einband-, 277
- einseitig beschränkte, 285
- Komposition, 288
- linear beschränkte, 285
- Mehrband-, 286
- Mehrspur-, 286
- universelle, 289, 418
- zelluläre, 293

Turing-Test, 279

U

- Überabzählbarkeit, 59, 417
Übergangsfunktion, 255
Übergangsrelation, 314
Übergangsschaltnetz, 233
Übersetzender Automat, 203, 230
Umkehrabbildung, 51
Unäre Codierung, 282
Unberechenbarkeit, 319, 417
UND-Operator, 82
Unendlichkeit, 57
Unentscheidbarkeit, 319, 417
Unerfüllbarkeit, 85, 417
Unifikation, 130, 417
Unifikator, 130, 418
 allgemeinster, 131, 403
Universalmenge, 41
Universelle Turing-Maschine, 289, 418
Universum, 120
 Herbrand-, 125, 408
Unmittelbare Adressierung, 296
Untermenge, 41
Up-Arrow-Notation, 56, 418
Urbild, 51

V

- Variable, 82, 164
 Funktions-, 147
 Prädikat-, 147
Venn-Diagramm, 41, 42
Vereinigungsmenge, 41
Verklemmungsfreiheit
 von Petri-Netzen, 242

Verschachtelte Rekursion, 272

- Verzweigende Rekursion, 272
Vollständige Induktion, 66, 418
Vollständiges Operatorensystem, 89
Vollständigkeit, 18, 96
Von-Neumann-Nachbarschaft, 244

W

- Wahrheitstabelle, 85
Wahrheitstafel, 85
Wechselseitige Rekursion, 272
While-Berechenbarkeit, 261
While-Programm, 260, 418
While-Schleife, 260
While-Sprache, 260, 418
Widerspruchsbeweis, 66
Widerspruchsfreiheit, 19
Widerspruchskalkül, 97, 418
Wohlordnung, 72
Wort, 162
Wortproblem, 162, 418
Wurzelschnecke, 65

X

- XOR-Operator, 83

Z

- Z3, 22
Zahl
 ganze, 39, 53
 große, 55
 irrationale, 54

- natürliche, 39, 52
nichtnegative, 39
positive, 39
rationale, 53
reelle, 54

Zeichen, 162
Zeitkonstruierbare Funktion, 361

- Zelle, 243
Zellmenge, 243
Zelluläre Turing-Maschine, 293
Zellulärer Automat, 243, 252, 418
Zentraleinheit, 296
Zermelo-Fraenkel-Mengenlehre, 39
Zermelo-Mengenlehre, 39
Zielmenge, 50
Zustand, 202
 äquivalenter, 206
Zustandsband, 292
Zustandsmenge
 von ε -Automaten, 213
 von DEAs, 204
 von Kellerautomaten, 224
 von NEAs, 209
 von Transduktoren, 230
 von Turing-Maschinen, 279
 von zellulären Automaten, 243

Zustandsübergangsdiagramm, 202
Zustandsübergangsfunktion

- von ε -Automaten, 213
 von DEAs, 204
 von Kellerautomaten, 224
 von NEAs, 209
 von Transduktoren, 230
 von Turing-Maschinen, 279
 von zellulären Automaten, 243

eBooks zum sofortigen Download

www.ciando.com

find
out!

Know How ist nur einen Klick entfernt.

- eBooks können Sie sofort per Download beziehen.
- eBooks sind ganz oder kapitelweise erhältlich.
- eBooks bieten eine komfortable Volltextsuche.

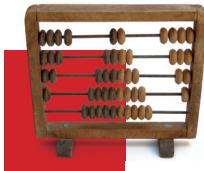


 **ciando**
eBooks



DIE NEUE LEHRBUCHGENERATION //

- Für das Bachelor-Studium geeignet.
- Umfassende verständliche Einführung in die wichtigsten Teilgebiete.
- Grundlegende Konzepte, Methoden und Ergebnisse.
- Zahlreiche Beispiele und Übungsaufgaben.



THEORETISCHE INFORMATIK // Das Buch führt umfassend in das Gebiet der theoretischen Informatik ein und behandelt den Stoffumfang, der für das Bachelor-Studium an Universitäten und Fachhochschulen in den Fächern Informatik und Informationstechnik benötigt wird. Die Darstellung und das didaktische Konzept verfolgen das Ziel, einen durchweg praxisnahen Zugang zu den mitunter sehr theoretisch geprägten Themen zu schaffen. Theoretische Informatik muss nicht trocken sein. Sie kann Spaß machen und genau dies versucht das Buch zu vermitteln. Die verschiedenen Methoden und Verfahren werden anhand konkreter Beispiele eingeführt und durch zahlreiche Querverbindungen wird gezeigt, wie die fundamentalen Ergebnisse der theoretischen Informatik die moderne Informationstechnologie prägen.

Das Buch behandelt die Themengebiete: Logik und Deduktion, Automatentheorie, formale Sprachen, Entscheidbarkeitstheorie, Berechenbarkeitstheorie und Komplexitätstheorie. Die Lehrinhalte aller Kapitel werden durch zahlreiche Übungsaufgaben komplettiert, so dass sich die Lektüre neben der Verwendung als studienbegleitendes Lehrbuch auch bestens zum Selbststudium eignet. Die 2. Auflage wurde aktualisiert und um neue Aufgaben erweitert.

DAS EXTRA ZUM BUCH // Die Lösungen zu den Übungsaufgaben, eine Linkssammlung sowie weiterführende Informationen bieten einen vielfältigen Zusatznutzen und stehen im Internet zur Verfügung: <http://www.dirkwhoffmann.de/TH>

Systemvoraussetzungen für eBook-inside: Internet-Verbindung und eBookreader Adobe Digital Editions.

Prof. Dr. Dirk W. HOFFMANN betreut die Lehrgebiete Technische Informatik, Embedded Software und Multimedia-Technik an der Fakultät für Informatik der Hochschule Karlsruhe.

HANSER

www.hanser.de/computer

€ 39,90 [D] | € 41,10 [A]
ISBN 978-3-446-42639-9

9 783446 426399

Studierende der Informatik
und Informationstechnik