

it
informatik



John E. Hopcroft
Rajeev Motwani
Jeffrey D. Ullman

Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit

3., aktualisierte Auflage

Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit

**John E. Hopcroft,
Rajeev Motwani,
Jeffrey D. Ullman**

Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit

3., aktualisierte Auflage



ein Imprint von Pearson Education
München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

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.

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.

Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.

Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Autoren dankbar.

Authorized translation from the English language edition, entitled Introduction to Automata Theory, Languages, and Computation, 3rd Edition by John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright © 2007; German language edition published by Pearson Education Deutschland GmbH, Copyright © 2011.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Produktbezeichnungen und weitere Stichworte und sonstige Angaben, die in diesem Buch verwendet werden, sind als eingetragene Marken geschützt. Da es nicht möglich ist, in allen Fällen zeitnah zu ermitteln, ob ein Markenschutz besteht, wird das ®-Symbol in diesem Buch nicht verwendet.

10 9 8 7 6 5 4 3 2 1
13 12 11

ISBN 978-3-86894-082-4 (print); 978-3-86326-509-0 (PDF); 978-3-86326-072-9 (ePUB)

© 2011 by Pearson Studium
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
www.pearson-studium.de

Übersetzung: Sigrid Richter und Ingrid Tokar für transit, München
Fachlektorat: Prof. Dr. Walter Hower
Programmleitung: Birger Peil, bpeil@pearson.de
Development: Alice Kachnij, akachnij@pearson.de
Korrektorat: Ivonne Domnick
Einbandgestaltung: Thomas Arlt, tarlt@adesso21.net
Titelbild: Corbis, Deutschland
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: text&form GbR, Fürstenfeldbruck
Druck und Verarbeitung: Kösel, Krugzell (www.KoeselBuch.de)
Printed in Germany

Inhaltsübersicht

Vorwort	17
Vorwort zur deutschen Auflage	21
Kapitel 1 Automaten: Die Grundlagen und Methoden ...	23
Kapitel 2 Endliche Automaten	61
Kapitel 3 Reguläre Ausdrücke und Sprachen	113
Kapitel 4 Eigenschaften regulärer Sprachen	159
Kapitel 5 Kontextfreie Grammatiken und Sprachen	205
Kapitel 6 Pushdown-Automaten	259
Kapitel 7 Eigenschaften kontextfreier Sprachen	297
Kapitel 8 Einführung in Turing-Maschinen	355
Kapitel 9 Unentscheidbarkeit	419
Kapitel 10 Nicht handhabbare Probleme	469
Kapitel 11 Zusätzliche Problemklassen	529
Literaturverzeichnis	577
Stichwortverzeichnis	587

Inhaltsverzeichnis

Vorwort	17
Vorwort zur deutschen Auflage	21
Kapitel 1 Automaten: Die Grundlagen und Methoden	23
1.1 Wozu dient das Studium der Automatentheorie?	25
1.1.1 Einführung in endliche Automaten	25
1.1.2 Strukturelle Repräsentationen	27
1.1.3 Automaten und Komplexität	28
1.2 Einführung in formale Beweise	28
1.2.1 Deduktive Beweise	29
1.2.2 Reduktion auf Definitionen	32
1.2.3 Andere Formen von Sätzen	34
1.2.4 Sätze, die keine Wenn-dann-Aussagen zu sein scheinen	37
1.3 Weitere Formen von Beweisen	37
1.3.1 Beweise der Äquivalenz von Mengen	38
1.3.2 Die Umkehrung	39
1.3.3 Beweis durch Widerspruch	41
1.3.4 Gegenbeispiele	41
1.4 Induktive Beweise	43
1.4.1 Induktive Beweise mit ganzen Zahlen	44
1.4.2 Allgemeinere Formen der Induktion mit ganzen Zahlen	47
1.4.3 Strukturelle Induktion	48
1.4.4 Gegenseitige Induktion	51
1.5 Die zentralen Konzepte der Automatentheorie	53
1.5.1 Alphabete	54
1.5.2 Zeichenreihen	54
1.5.3 Sprachen	56
1.5.4 Probleme	57
Zusammenfassung von Kapitel 1	59

Kapitel 2	Endliche Automaten	61
2.1	Eine informelle Darstellung endlicher Automaten	63
2.1.1	Die Grundregeln	63
2.1.2	Das Protokoll	64
2.1.3	Die Automaten dazu befähigen, Eingaben zu ignorieren	66
2.1.4	Das gesamte System aus Automaten darstellen	68
2.1.5	Mithilfe des Produktautomaten die Gültigkeit des Protokolls überprüfen	70
2.2	Deterministische endliche Automaten	71
2.2.1	Definition eines deterministischen endlichen Automaten	71
2.2.2	Wie ein DEA Zeichenreihen verarbeitet	72
2.2.3	Einfachere Notationen für DEAs	74
2.2.4	Die Übergangsfunktion auf Zeichenreihen erweitern	75
2.2.5	Die Sprache eines DEA	79
2.2.6	Übungen zum Abschnitt 2.2	79
2.3	Nichtdeterministische endliche Automaten	82
2.3.1	Eine informelle Sicht auf nichtdeterministische endliche Automaten	83
2.3.2	Definition nichtdeterministischer endlicher Automaten	85
2.3.3	Die erweiterte Übergangsfunktion	86
2.3.4	Die Sprache eines NEA	87
2.3.5	Äquivalenz deterministischer und nichtdeterministischer endlicher Automaten	88
2.3.6	Ein ungünstiger Fall für die Teilmengenkonstruktion	93
2.3.7	Übungen zum Abschnitt 2.3	95
2.4	Eine Anwendung: Textsuche	97
2.4.1	Zeichenreihen in Texten finden	97
2.4.2	Nichtdeterministische endliche Automaten für die Textsuche	98
2.4.3	Ein DEA, um die Menge von Schlüsselwörtern zu erkennen	99
2.4.4	Übungen zum Abschnitt 2.4	101
2.5	Endliche Automaten mit ε -Übergängen	101
2.5.1	Verwendungen von ε -Übergängen	102
2.5.2	Die formale Notation eines ε -NEA	103
2.5.3	ε -Hüllen	104
2.5.4	Erweiterte Übergänge und Sprachen für ε -NEAs	105
2.5.5	ε -Übergänge eliminieren	107
2.5.6	Übungen zum Abschnitt 2.5	110
	Zusammenfassung von Kapitel 2	111

Kapitel 3	Reguläre Ausdrücke und Sprachen	113
3.1	Reguläre Ausdrücke	114
3.1.1	Die Operatoren regulärer Ausdrücke	115
3.1.2	Reguläre Ausdrücke bilden	117
3.1.3	Auswertungsreihenfolge der Operatoren regulärer Ausdrücke	120
3.1.4	Übungen zum Abschnitt 3.1	121
3.2	Endliche Automaten und reguläre Ausdrücke	122
3.2.1	Von DEAs zu regulären Ausdrücken	122
3.2.2	DEA durch die Eliminierung von Zuständen in reguläre Ausdrücke umwandeln	128
3.2.3	Reguläre Ausdrücke in Automaten umwandeln	134
3.2.4	Übungen zum Abschnitt 3.2	138
3.3	Anwendungen regulärer Ausdrücke	140
3.3.1	Reguläre Ausdrücke in Unix	140
3.3.2	Lexikalische Analyse	142
3.3.3	Textmuster finden	144
3.3.4	Übungen zum Abschnitt 3.3	146
3.4	Algebraische Gesetze für reguläre Ausdrücke	147
3.4.1	Assoziativität und Kommutativität	147
3.4.2	Identitäten und Annihilatoren	148
3.4.3	Distributivgesetze	149
3.4.4	Das Idempotenzgesetz	150
3.4.5	Gesetze bezüglich der Hüllenbildung	150
3.4.6	Gesetze für reguläre Ausdrücke entdecken	151
3.4.7	Test eines für reguläre Ausdrücke geltenden Gesetzes der Algebra	154
3.4.8	Übungen zum Abschnitt 3.4	156
	Zusammenfassung von Kapitel 3	157
Kapitel 4	Eigenschaften regulärer Sprachen	159
4.1	Beweis der Nichtregularität von Sprachen	160
4.1.1	Das Pumping-Lemma für reguläre Sprachen	161
4.1.2	Anwendungen des Pumping-Lemmas	162
4.1.3	Übungen zum Abschnitt 4.1	164
4.2	Abschluss-Eigenschaften regulärer Sprachen	166
4.2.1	Abgeschlossenheit regulärer Sprachen bezüglich Boolescher Operationen	166
4.2.2	Spiegelung	173
4.2.3	Homomorphismus	174
4.2.4	Inverser Homomorphismus	176
4.2.5	Übungen zum Abschnitt 4.2	182

4.3	Entscheidbarkeits-Eigenschaften regulärer Sprachen	185
4.3.1	Wechsel zwischen Repräsentationen	186
4.3.2	Prüfen, ob eine reguläre Sprache leer ist	189
4.3.3	Zugehörigkeit zu einer regulären Sprache prüfen	190
4.3.4	Übungen zum Abschnitt 4.3	191
4.4	Äquivalenz und Minimierung von Automaten	191
4.4.1	Prüfen, ob Zustände äquivalent sind	192
4.4.2	Prüfen, ob reguläre Sprachen äquivalent sind	195
4.4.3	Minimierung von DEAs	198
4.4.4	Warum minimierte DEAs unschlagbar sind	201
4.4.5	Übungen zum Abschnitt 4.4	203
	Zusammenfassung von Kapitel 4	204
Kapitel 5 Kontextfreie Grammatiken und Sprachen		205
5.1	Kontextfreie Grammatiken	206
5.1.1	Ein informelles Beispiel	206
5.1.2	Definition kontextfreier Grammatiken	208
5.1.3	Ableitungen mithilfe einer Grammatik	210
5.1.4	Links- und rechtsseitige Ableitungen	213
5.1.5	Die Sprache einer Grammatik	215
5.1.6	Satzformen	216
5.1.7	Übungen zum Abschnitt 5.1	217
5.2	Parse-Bäume	219
5.2.1	Parse-Bäume aufbauen	219
5.2.2	Der Ergebnis eines Parse-Baums	221
5.2.3	Inferenz, Ableitungen und Parse-Bäume	222
5.2.4	Von Inferenzen zu Bäumen	223
5.2.5	Von Bäumen zu Ableitungen	225
5.2.6	Von Ableitungen zu rekursiven Inferenzen	228
5.2.7	Übungen zum Abschnitt 5.2	230
5.3	Anwendungen kontextfreier Grammatiken	231
5.3.1	Parser	231
5.3.2	Der YACC-Parsergenerator	234
5.3.3	Markup-Sprachen	235
5.3.4	XML und Dokumenttypdefinitionen	238
5.3.5	Übungen zum Abschnitt 5.3	244
5.4	Mehrdeutigkeit von Grammatiken und Sprachen	245
5.4.1	Mehrdeutige Grammatiken	246
5.4.2	Mehrdeutigkeit aus Grammatiken tilgen	248
5.4.3	Linksseitige Ableitungen als Möglichkeit zur Beschreibung von Mehrdeutigkeit	251
5.4.4	Inhärente Mehrdeutigkeit	252
5.4.5	Übungen zum Abschnitt 5.4	255
	Zusammenfassung von Kapitel 5	256

Kapitel 6	Pushdown-Automaten	259
6.1	Definition des Pushdown-Automaten	260
6.1.1	Informelle Einführung	260
6.1.2	Die formale Definition von Pushdown-Automaten	262
6.1.3	Eine grafische Notation für PDAs	264
6.1.4	Unmittelbare Beschreibungen eines PDA	265
6.1.5	Übungen zum Abschnitt 6.1	269
6.2	Die Sprachen eines PDA	270
6.2.1	Akzeptanz durch Endzustand	270
6.2.2	Akzeptanz durch leeren Stack	272
6.2.3	Vom leeren Stack zum Endzustand	272
6.2.4	Vom Endzustand zum leeren Stack	276
6.2.5	Übungen zum Abschnitt 6.2	278
6.3	Äquivalenz von PDAs und kontextfreien Grammatiken	279
6.3.1	Von Grammatiken zu PDAs	280
6.3.2	Von PDAs zu Grammatiken	283
6.3.3	Übungen zum Abschnitt 6.3	288
6.4	Deterministische Pushdown-Automaten	289
6.4.1	Definition eines deterministischen PDA	290
6.4.2	Reguläre Sprachen und deterministische PDAs	291
6.4.3	DPDAs und kontextfreie Sprachen	292
6.4.4	DPDAs und mehrdeutige Grammatiken	293
6.4.5	Übungen zum Abschnitt 6.4	294
	Zusammenfassung von Kapitel 6	295
Kapitel 7	Eigenschaften kontextfreier Sprachen	297
7.1	Normalformen kontextfreier Grammatiken	298
7.1.1	Eliminierung unnützer Symbole	298
7.1.2	Berechnung der erzeugenden und erreichbaren Symbole	301
7.1.3	ϵ -Produktionen eliminieren	302
7.1.4	Einheitsproduktionen eliminieren	306
7.1.5	Chomsky-Normalform	311
7.1.6	Übungen zum Abschnitt 7.1	316
7.2	Das Pumping-Lemma für kontextfreie Sprachen	319
7.2.1	Die Größe von Parse-Bäumen	319
7.2.2	Aussage des Pumping-Lemmas	320
7.2.3	Anwendungen des Pumping-Lemmas für kontextfreie Sprachen	323
7.2.4	Übungen zum Abschnitt 7.2	326
7.3	Abschluss-Eigenschaften kontextfreier Sprachen	328
7.3.1	Substitutionen	328
7.3.2	Anwendungen des Substitutions-Theorems	331
7.3.3	Spiegelung	332
7.3.4	Durchschnitt mit einer regulären Sprache	332

7.3.5	Inverse Homomorphismen	337
7.3.6	Übungen zum Abschnitt 7.3	339
7.4	Entscheidbarkeits-Eigenschaften kontextfreier Sprachen	341
7.4.1	Komplexität der Umwandlung von kfGs in PDAs und umgekehrt	342
7.4.2	Ausführungszeit der Umwandlung in Chomsky- Normalform	343
7.4.3	Prüfen, ob eine kontextfreie Sprache leer ist	345
7.4.4	Die Zugehörigkeit zu einer kontextfreien Sprache prüfen ...	347
7.4.5	Vorschau auf unentscheidbare kFL-Probleme	351
7.4.6	Übungen zum Abschnitt 7.4	352
	Zusammenfassung von Kapitel 7	353
Kapitel 8 Einführung in Turing-Maschinen		355
8.1	Probleme, die Computer nicht lösen können	356
8.1.1	Programme, die »Hello, World« ausgeben	357
8.1.2	Der hypothetische »Hello, World«-Tester	359
8.1.3	Ein Problem auf ein anderes Problem reduzieren	362
8.1.4	Übungen zum Abschnitt 8.1	365
8.2	Die Turing-Maschine	366
8.2.1	Das Streben danach, alle mathematischen Fragen zu entscheiden	367
8.2.2	Die Notation der Turing-Maschine	368
8.2.3	Unmittelbare Beschreibungen für Turing-Maschinen	369
8.2.4	Übergangsdiagramme für Turing-Maschinen	373
8.2.5	Die Sprache einer Turing-Maschine	376
8.2.6	Turing-Maschinen und das Halteproblem	377
8.2.7	Übungen zum Abschnitt 8.2	378
8.3	Programmiertechniken für Turing-Maschinen	379
8.3.1	Speicher im Zustand	380
8.3.2	Mehrere Spuren	381
8.3.3	Unterprogramme	383
8.3.4	Übungen zum Abschnitt 8.3	386
8.4	Erweiterungen für die einfache Turing-Maschine	386
8.4.1	Turing-Maschinen mit mehreren Bändern	386
8.4.2	Äquivalenz zwischen ein- und mehrbändigen TMn	388
8.4.3	Ausführungszeit und die Viele-Bänder-in-eins-Konstruktion	390
8.4.4	Nichtdeterministische Turing-Maschinen	391
8.4.5	Übungen zum Abschnitt 8.4	393
8.5	Beschränkte Turing-Maschinen	396
8.5.1	Turing-Maschinen mit semi-unendlichen Bändern	397
8.5.2	Maschinen mit mehreren Stacks	400
8.5.3	Zählermaschinen	403

8.5.4	Die Leistungsfähigkeit von Zählernmaschinen	404
8.5.5	Übungen zum Abschnitt 8.5	406
8.6	Turing-Maschinen und Computer	407
8.6.1	Eine Turing-Maschine mit einem Computer simulieren	407
8.6.2	Einen Computer mit einer Turing-Maschine simulieren	409
8.6.3	Laufzeitvergleich zwischen Computern und Turing-Maschinen	413
	Zusammenfassung von Kapitel 8	416
Kapitel 9	Unentscheidbarkeit	419
9.1	Eine nicht rekursiv aufzählbare Sprache	421
9.1.1	Binärzeichenreihen aufzählen	421
9.1.2	Codes für Turing-Maschinen	422
9.1.3	Die Diagonalisierungssprache	423
9.1.4	Der Beweis, dass L_d nicht rekursiv aufzählbar ist	425
9.1.5	Übungen zum Abschnitt 9.1	425
9.2	Ein unentscheidbares Problem, das rekursiv aufzählbar ist	426
9.2.1	Rekursive Sprachen	426
9.2.2	Komplemente rekursiver und rekursiv aufzählbarer Sprachen	427
9.2.3	Die universelle Sprache	430
9.2.4	Unentscheidbarkeit der universellen Sprache	433
9.2.5	Übungen zum Abschnitt 9.2	434
9.3	Unentscheidbare Probleme über Turing-Maschinen	436
9.3.1	Reduktionen	436
9.3.2	Turing-Maschinen, die die leere Sprache akzeptieren	438
9.3.3	Der Satz von Rice und Eigenschaften der rekursiv aufzählbaren Sprachen	441
9.3.4	Probleme bezüglich Spezifikationen von Turing-Maschinen	444
9.3.5	Übungen zum Abschnitt 9.3	444
9.4	Das Postsche Korrespondenz-Problem	446
9.4.1	Definition des Postschen Korrespondenz-Problems	446
9.4.2	Das »modifizierte« PKP	449
9.4.3	Fertigstellung des Beweises der PKP-Unentscheidbarkeit	452
9.4.4	Übungen zum Abschnitt 9.4	458
9.5	Andere unentscheidbare Probleme	459
9.5.1	Probleme bei Programmen	459
9.5.2	Unentscheidbarkeit der Mehrdeutigkeit kontextfreier Grammatiken	459
9.5.3	Das Komplement einer Listensprache	462
9.5.4	Übungen zum Abschnitt 9.5	465
	Zusammenfassung von Kapitel 9	466

Kapitel 10 Nicht handhabbare Probleme	469
10.1 Die Klassen \mathcal{P} und \mathcal{NP}	471
10.1.1 Mit polynomialem Zeitaufwand lösbare Probleme	471
10.1.2 Beispiel: Der Kruskal-Algorithmus	472
10.1.3 Nichtdeterministischer polynomialer Zeitaufwand	476
10.1.4 Ein NP-Beispiel: Das Problem des Handlungsreisenden	477
10.1.5 Polynomzeit-Reduktionen	478
10.1.6 NP-vollständige Probleme	480
10.1.7 Übungen zum Abschnitt 10.1	482
10.2 Ein NP-vollständiges Problem	483
10.2.1 Das Erfüllbarkeitsproblem	484
10.2.2 SAT-Instanzen repräsentieren	485
10.2.3 NP-Vollständigkeit des SAT-Problems	486
10.2.4 Übungen zum Abschnitt 10.2	493
10.3 Ein eingeschränktes Erfüllbarkeitsproblem	493
10.3.1 Normalformen für Boolesche Ausdrücke	494
10.3.2 Ausdrücke in KNF konvertieren	495
10.3.3 NP-Vollständigkeit von CSAT	498
10.3.4 NP-Vollständigkeit von 3SAT	503
10.3.5 Übungen zum Abschnitt 10.3	504
10.4 Weitere NP-vollständige Probleme	505
10.4.1 NP-vollständige Probleme beschreiben	506
10.4.2 Das Problem unabhängiger Mengen	506
10.4.3 Das Problem der Knotenüberdeckung	511
10.4.4 Das Problem des gerichteten Hamiltonschen Kreises	512
10.4.5 Ungerichtete Hamiltonsche Kreise und das Problem des Handlungsreisenden	519
10.4.6 Zusammenfassung NP-vollständiger Probleme	521
10.4.7 Übungen zum Abschnitt 10.4	521
Zusammenfassung von Kapitel 10.	526
Kapitel 11 Zusätzliche Problemklassen	529
11.1 Komplemente von Sprachen, die in \mathcal{NP} enthalten sind	531
11.1.1 Die Sprachklasse $\text{Co-}\mathcal{NP}$	531
11.1.2 NP-vollständige Probleme und $\text{Co-}\mathcal{NP}$	532
11.1.3 Übungen zum Abschnitt 11.1	533
11.2 Probleme, die mit polynomialem Speicherplatz lösbar sind	534
11.2.1 Turing-Maschinen mit polynomialer Platzbegrenzung	534
11.2.2 Beziehung von \mathcal{PS} und \mathcal{NPS} zu früher definierten Klassen	535
11.2.3 Deterministischer und nichtdeterministischer polynomialer Speicherplatz	537

11.3 Ein für \mathcal{PS} vollständiges Problem	540
11.3.1 PS-Vollständigkeit	540
11.3.2 Quantifizierte Boolesche Formeln	541
11.3.3 Quantifizierte Boolesche Formeln auswerten	542
11.3.4 PS-Vollständigkeit des QBF-Problems	544
11.3.5 Übungen zum Abschnitt 11.3	550
11.4 Sprachklassen basierend auf Randomisierung	550
11.4.1 Quicksort: Ein Beispiel für einen zufallsabhängigen Algorithmus	551
11.4.2 Ein auf Zufallsabhängigkeit basierendes Modell einer Turing-Maschine	552
11.4.3 Die Sprache einer zufallsabhängigen Turing-Maschine	554
11.4.4 Die Klasse \mathcal{RP}	556
11.4.5 In \mathcal{RP} enthaltene Sprachen erkennen	558
11.4.6 Die Klasse \mathcal{ZPP}	559
11.4.7 Beziehung zwischen \mathcal{RP} und \mathcal{ZPP}	560
11.4.8 Beziehungen zu den Klassen \mathcal{P} und \mathcal{NP}	562
11.5 Die Komplexität des Primzahltests	562
11.5.1 Die Bedeutung des Primzahltests	563
11.5.2 Einführung in Modular-Arithmetik	565
11.5.3 Die Komplexität modular-arithmetischer Berechnungen	567
11.5.4 Zufallsabhängig-polynomiales Primzahl-Testen	568
11.5.5 Nichtdeterministische Primzahltests	570
11.5.6 Übungen zum Abschnitt 11.5	573
Zusammenfassung von Kapitel 11	574

Literaturverzeichnis	577
-----------------------------------	------------

Stichwortverzeichnis	587
-----------------------------------	------------

Vorwort

Im Vorwort des 1979 erschienenen Vorgängers dieses Buches wunderten sich Hopcroft und Ullman über die Tatsache, dass das Interesse an dem Thema Automaten gegenüber dem Zeitpunkt, zu dem sie ihr erstes Buch verfassten (1969), förmlich explodiert war. In der Tat enthielt das Buch von 1979 viele Themen, die in dem früheren Werk nicht vertreten waren, und es war etwa doppelt so dick. Wenn man das vorliegende Buch mit dem von 1979 vergleicht, stellt man fest, dass dieses Buch, wie die Autos in den Siebzigern, »außen größer, aber innen kleiner« ist. Das klingt nach einem Rückschritt, aber wir sind aus verschiedenen Gründen mit diesen Änderungen zufrieden. Erstens war die Automaten- und Sprachtheorie 1979 noch ein Bereich aktiver Forschung. Das damalige Buch sollte unter anderem dazu dienen, an der Mathematik interessierte Studierende dazu zu ermutigen, zu diesem Feld beizutragen. Heute gibt es kaum Forschungsprojekte, die sich direkt mit der Automatentheorie befassen (im Gegensatz zu deren Anwendungen), und daher besteht für uns kaum Veranlassung, den knappen, sehr mathematischen Stil des 1979 erschienenen Buches beizubehalten. Zweitens hat sich die Rolle der Automaten- und Sprachtheorie in den letzten beiden Jahrzehnten geändert. 1979 war die Automatentheorie größtenteils ein fortgeschrittenes Thema für Studierende höherer Semester, und wir nahmen an, unsere Leser seien Studierende fortgeschrittener Semester, insbesondere die Leser der hinteren Kapitel des Buches. Heute gehört dieses Thema zum Curriculum der Anfangssemester. Daher darf der Inhalt dieses Buches weniger Vorkenntnisse seitens der Studierenden voraussetzen und muss mehr Hintergrundinformationen und Argumentationsdetails bieten als das frühere Werk. Eine dritte Änderung besteht darin, dass die Informatik in den letzten drei Jahrzehnten in einem fast unvorstellbaren Maß gewachsen ist. Während es 1979 häufig eine Herausforderung darstellte, das Curriculum mit Material zu füllen, das unserer Einschätzung nach die nächste technologische Welle überdauern würde, streiten sich heute viele Teildisziplinen um einen Platz im begrenzten Raum des Grundstudium-Curriculums.

Viertens wurde die Informatik zu einem stärker berufsorientierten Thema, und bei vielen Studierenden ist ein starker Pragmatismus feststellbar. Wir glauben weiterhin, dass bestimmte Aspekte der Automatentheorie grundlegende Werkzeuge für verschiedene neue Disziplinen sind und dass die in typischen Kursen zum Thema Automatentheorie enthaltenen theoretischen, das Bewusstsein erweiternden Übungen, nach wie vor ihren Wert haben, ganz gleich, wie sehr die Studierenden es auch bevorzugen mögen, sich mit sofort monetär umsetzbarer Technologie zu befassen. Um dem Gebiet allerdings einen dauerhaften Platz auf der Themenliste des Informatikstudierenden sicherzustellen, sind wir der Meinung, dass es notwendig ist, die Anwendungen neben der Mathematik stärker hervorzuheben. Daher haben wir eine Reihe der eher

abstrakten Themen des früheren Buches durch Beispiele ersetzt, die zeigen, wie diese Ideen heute Anwendung finden. Wenngleich die Anwendungen der Automaten- und Sprachtheorie im Compilerbau jetzt so bekannt sind, dass sie normalerweise in einem Compilerkurs behandelt werden, gibt es verschiedene neuere Anwendungen, einschließlich »Model-Checking«-Algorithmen zur Verifizierung von Protokollen und Dokumentbeschreibungssprachen, die kontextfreien Grammatiken nachempfunden sind.

Die letzte Erklärung für die gleichzeitige Vergrößerung und Verkleinerung dieses Buches besteht darin, dass wir heute die Vorteile der von Don Knuth und Les Lamport entwickelten Satzsysteme TeX und LaTeX nutzen können. Insbesondere LaTeX ermutigt zu einem »offenen« Satzstil, der Bücher zwar umfangreicher, aber einfacher lesbar macht. Wir schätzen die Arbeit beider Herren.

Verwendung des Buches

Dieses Buch eignet sich für einen halb- oder einsemestrigen Kurs auf Eingangsniveau oder später. An der Universität von Stanford haben wir das Material in dem Kurs CS154, einem Kurs zum Thema Automaten- und Sprachtheorie, verwendet. Es handelt sich um einen halbsemestrigen Kurs, den sowohl Rajeev als auch Jeff abgehalten haben. Aufgrund der zeitlichen Beschränkung wird in diesem Kurs Kapitel 11 nicht behandelt, und einige Materialien späterer Kapitel, wie z. B. die schwierigere Polynom-Zeit-Reduktion aus Abschnitt 10.4, werden ebenfalls weggelassen. Die Website zu diesem Buch (siehe unten) enthält Aufzeichnungen und Vorlesungsbeschreibungen für verschiedene Angebote des Kurses CS154.

Vor einigen Jahren stellten wir fest, dass viele Studierende höherer Semester nach Stanford kamen und Kurse in Automatentheorie absolviert hatten, in denen die Theorie der Nichthandhabbarkeit fehlte. Da der Lehrkörper der Universität von Stanford der Auffassung ist, dass jeder Informatiker diese Konzepte zu einem Grad kennen muss, der über die Ebene der Aussage »NP-vollständig bedeutet, dass es zu lange dauert« hinausgeht, wird ein weiterer Kurs, CS154N, angeboten, den Studierende belegen können, um nur die Kapitel 8, 9 und 10 abzudecken. Tatsächlich nehmen sie in etwa am letzten Drittel von CS154 teil, um die Anforderungen von CS154N zu erfüllen. Auch heute noch finden wir in jedem Halbsemester Studierende, die diese Option nutzen. Da es wenig Zusatzaufwand erfordert, empfehlen wir dieses Vorgehen.

Voraussetzungen

Um dieses Buch am besten erschließen zu können, sollten die Studierenden vorher bereits einen Kurs belegt haben über Diskrete Mathematik, z. B. Graphen, Bäume, Logik und Beweis-Techniken. Wir nehmen zudem an, dass die Studierenden mehrere Programmierkurse absolviert haben und mit üblichen Datenstrukturen, Rekursion und der Rolle der wichtigsten Systemkomponenten wie Compilern vertraut sind. Diese Voraussetzungen sollten in den ersten Semestern eines typischen Informatikstudien-ganges erworben worden sein.

Übungen

Das Buch enthält ausgiebig Übungen zu fast jedem Abschnitt. Wir kennzeichnen schwierigere Übungen oder Teile von Übungen mit einem Ausrufezeichen. Die schwierigsten Übungen haben zwei Ausrufezeichen.

Einige Übungen oder Teile davon sind mit Sternchen markiert. Für diese Übungen sind Lösungen auf der Website des Buches bereitgestellt. Beachten Sie, dass in einigen Fällen eine Übung *B* die Modifikation oder Anpassung Ihrer Lösung zu einer anderen Übung *A* erfordert. Falls zu bestimmten Teilen von *A* Lösungen existieren, dann können Sie davon ausgehen, dass es auch für die entsprechenden Teile von *B* Lösungen gibt.

Unterstützung im World Wide Web

Die Web-Adresse des Buches lautet:

<http://www-db.stanford.edu/~ullman/ialc.html>

Hier finden sich Lösungen zu den mit Sternen gekennzeichneten Übungen, Errata und unterstützende Materialien. Wir hoffen die Skripte bei jedem Anbieten des Kurses CS154 zur Verfügung stellen zu können, einschließlich Hausarbeiten, Lösungen und Prüfungen.



Danksagung

Eine Ausarbeitung von Craig Silverstein über »how to do proofs« hatte Einfluss auf einen Teil des in Kapitel 1 dargelegten Materials. Kommentare und Errata zu Entwürfen der zweiten Auflage gingen ein von: Zoe Abrams, George Candea, Haowen Chen, Byong-Gun Chun, Jeffrey Shallit, Bret Taylor, Jason Townsend und Erik Uzureau.

Wir erhielten auch viele E-Mails, welche auf Fehler in der zweiten Auflage dieses Buches hinwiesen, und diese wurden online auf den Fehlerseiten zu dieser Ausgabe gewürdigt. Jedoch möchten wir hier die folgenden Leute erwähnen, die eine Vielzahl bedeutsamer Fehler offenlegten: Zeki Bayram, Sebastian Hick, Kang-Rae Lee, Christian Lemburg, Nezam Mahdavi-Amiri, Dave Maier, A. P. Marathe, Mark Meuleman, Mustafa Sait-Ametov, Alexey Sarytchev, Jukka Suomela, Rod Topor, Po-Lian Tsai, Tom Whaley, Aaron Windsor und Jacinth H. T. Wu.

Die Hilfe all dieser Menschen wird dankend anerkannt. Verbleibende Fehler sind natürlich uns zuzurechnen.

J. E. H.
R. M.
J. D. U.
Ithaca NY und Stanford CA
Februar 2006

Vorwort zur deutschen Auflage

Die Computerwissenschaften lassen sich grob in vier Bereiche gliedern: Die Theoretische, Technische und Praktische Informatik sowie die Anwendungs-Informatik. Hierbei stellt die *Theoretische Informatik* (TI) das Fundament dar. Konzeptuell noch weiter vorgelagert ist die *Diskrete Mathematik*. Ohne eine solche Unterfütterung ist ein Informatikstudium nicht geerdet. TI ist an Universitäten sowohl im Grund- als auch im Hauptstudium selbstverständlich und findet sich ebenso in vielen Fachhochschul-Studiengängen wieder.

TI erscheint für viele Studierende als unbequemes Pflichtfach. Mit der richtigen Einstellung dazu, bietet es jedoch auch eine Chance. Es ist schließlich innerhalb der Informatik dasjenige Gebiet, das am meisten zur Allgemeinbildung beiträgt und zugleich über die größte »Halbwertszeit« des Wissensbestands verfügt. Ohne TI ist ein sinnvolles Studium der Computerwissenschaften nicht möglich. Die Grundlagen zu diesem Wissensgebiet in deutscher Sprache darzubieten, ist die Idee der vorliegenden Übersetzung des Original-Klassikers.

Der inhaltliche Rahmen des Studiums gestaltet sich recht homogen; der Haupt-Kanon besteht zumeist aus den Gebieten *Komplexitätstheorie*, *Formale Sprachen*, *Automatentheorie* und *Unberechenbarkeit*. Für das letztgenannte Thema hat sich leider der Begriff *Berechenbarkeit* etabliert, obwohl es dort gerade darum geht, welche Abgründe sich hinter dieser (mathematisch begründbaren) Grenze, also im Unberechenbaren, auftun. (Wären diese Probleme berechenbar, würde man sich für die Komplexität ihrer Lösungsverfahren interessieren – womit man beim erstgenannten Thema landet; manche Fragestellungen lassen sich jedoch nicht algorithmisieren und bleiben tatsächlich unberechenbar.)

Das amerikanische Original dieses Werkes ist seit Jahrzehnten der Klassiker unter den TI-Büchern. Bereits zu meiner Studienzeit in den 1980er-Jahren gehörte die Erstauflage, noch im 2er-Autorenteam Hopcroft/Ullman, zur Basisliteratur. Es folgte später die etwas umgestaltete Zweitaufgabe, an der Rajeev Motwani als weiterer Autor mitarbeitete. Die anschließende dritte Auflage bildet in Bezug auf das damals aktive 3er-Autorenteam nun den Abschluss — Rajeev ist, laut Bericht der Stanford University, inzwischen leider verstorben.

Diese dritte Auflage liegt nun in deutscher Fassung vor. Im Vergleich zur deutschsprachigen Vorgängerausgabe wurden viele Korrekturen vorgenommen, z. B. bei den *Genau-dann-wenn/iff*-Formulierungen. Des Weiteren wurden offiziell dokumentierte Errata (einschließlich denen der englischsprachigen Ausgabe) behoben und hoffentlich keine neuen eingearbeitet. Einige Stellen wurden über das Original hinausgehend mit *-Fußnoten versehen und mit eigenen Erläuterungen zur weiteren Verdeutlichung ergänzt.

Sowohl aus Sicht der Studierenden als auch der Lehrenden lässt sich dieses Werk in der vorgegebenen Kapitelfolge bearbeiten. Nach dem einführenden Kapitel 1 ist jedoch genauso gut eine leichte Modifizierung denkbar: Man geht zunächst zu den Kapiteln 3/4, 5/7 und 9 mit den Sprachklassen 3, 2 und 0. Danach widmet man sich den korrespondierenden Automatentypen in den Kapiteln 2, 6 und 8. (Eigentlich könnte auch Klasse bzw. Typ 1 erwähnt sein, aber es ist ja nun mal eine Übersetzung.) Abschließend gelangt man zu den Schlusskapiteln 10 und 11. Ebenso könnte man zuerst die berechenbaren Teile behandeln, z. B. in der Kapitelreihenfolge 3/4/2, 5/7/6 sowie 10 und danach die unberechenbare Turing-Maschine zusammen mit der Beleuchtung der prinzipiellen Unentscheidbarkeit in den Kapiteln 8 und 9 bearbeiten. Zuletzt folgt Kapitel 11.

Der Einstieg ist benutzungsfreundlich; so haben wir z. B. im ersten Abschnitt 1.1.1, in Ergänzung zum Original, bei der Einführung in das Automatenkonzept alle möglichen Übergangsfälle auf dem Weg zur Akzeptanz des Schlüsselworts then dokumentiert.



Die Companion Website (CWS) dieses Buchs steht unter

<http://www.pearson-studium.de/>

Am schnellsten gelangen Sie von dort zu den Online-Inhalten, wenn Sie in das Feld »Schnellsuche« die Titelnnummer **4082** eingeben. Auf der CWS finden Sie für Dozenten alle Buchabbildungen auf Folien zum Einsatz in Lehrveranstaltungen und für Studenten Lösungen zu den markierten Übungsaufgaben im Buch.

Ein solches Unterfangen dieser Größenordnung zu stemmen, ist eine Ehre und eine Herausforderung zugleich. »Nobody is perfect« – Verbesserungsvorschläge sind herzlich willkommen.

Wir begeben uns nun auf hohe See – gutes Gelingen mit dieser Neuauflage!

WHo

Automaten: Die Grundlagen und Methoden

1

1.1	Wozu dient das Studium der Automatentheorie?	25
1.2	Einführung in formale Beweise	28
1.3	Weitere Formen von Beweisen	37
1.4	Induktive Beweise	43
1.5	Die zentralen Konzepte der Automatentheorie ...	53
	Zusammenfassung von Kapitel 1	59

ÜBERBLICK


» Mit dem Begriff »Automatentheorie« ist das Studium abstrakter Rechengерäte oder »Maschinen« gemeint. In den dreißiger Jahren, als es noch keine Computer gab, studierte A. Turing eine abstrakte Maschine, die über sämtliche Fähigkeiten heutiger Computer verfügte, zumindest was deren Rechenleistung betraf. Turing wollte die Grenze zwischen dem, was eine Rechenmaschine berechnen kann, und dem, was sie nicht berechnen kann, genau beschreiben. Seine Schlussfolgerungen trafen nicht nur auf seine abstrakten *Turing-Maschinen* zu, sondern auch auf die heutigen realen Maschinen.

In den vierziger und fünfziger Jahren wurden von einigen Forschern einfachere Maschinen untersucht, die heute als »endliche Automaten« bezeichnet werden. Diese Automaten, die ursprünglich zur Simulation von Gehirnfunktionen vorgesehen waren und auf die wir in Abschnitt 1.1 eingehen werden, haben sich für verschiedene andere Zwecke als außerordentlich nützlich erwiesen. In den späten fünfziger Jahren begann zudem der Linguist N. Chomsky, formale »Grammatiken« zu studieren. Diese Grammatiken sind zwar streng genommen keine Maschinen, weisen aber eine enge Verwandtschaft zu abstrakten Automaten auf und dienen heute als Grundlage einiger wichtiger Softwarekomponenten, zu denen auch Teile von Compilern gehören.

1969 führte S. Cook Turings Untersuchung der Frage fort, was berechnet werden kann und was nicht. Cook konnte die Probleme, die sich effizient mit Computern lösen lassen, von jenen Problemen trennen, die prinzipiell lösbar sind, deren Lösung jedoch in der Praxis so zeitaufwändig ist, dass Computer sich lediglich zur Lösung sehr kleiner Beispiele solcher Probleme eignen. Diese Klasse von Problemen wird als »nicht handhabbar« (englisch: intractable) oder »NP-hart« bezeichnet. Es ist sehr unwahrscheinlich, dass die exponentielle Steigerung der Rechengeschwindigkeit, die bei der Computerhardware erzielt worden ist (»Moore's Gesetz«), sich bemerkenswert auf unsere Fähigkeit auswirken wird, umfangreiche Beispiele solcher nicht handhabbaren Probleme berechnen zu können.

All diese theoretischen Entwicklungen wirken sich direkt auf die Tätigkeit der Informatiker von heute aus. Einige dieser Konzepte, wie endliche Automaten und bestimmte Arten formaler Grammatiken, werden im Design und im Aufbau wichtiger Arten von Software verwendet. Andere Konzepte, wie Turing-Maschinen, helfen uns dabei zu verstehen, was wir von unserer Software erwarten können. Insbesondere aus der Theorie der nicht handhabbaren Probleme können wir ableiten, ob es wahrscheinlich ist, dass wir ein Problem direkt angehen und ein Programm zu seiner Lösung schreiben können (weil es nicht zur Klasse der nicht handhabbaren Probleme gehört), oder ob wir eine Möglichkeit finden müssen, das nicht handhabbare Problem zu umgehen: durch eine Approximation, die Verwendung einer Heuristik oder durch eine andere Methode, um die Menge an Zeit zu beschränken, die das Programm zur Lösung des Problems aufwendet.

In diesem einführenden Kapitel beginnen wir mit einer sehr abstrakten Betrachtung des Gegenstands der Automatentheorie und ihrer Anwendungen. Ein Großteil des Kapitels ist der Untersuchung von Beweistechniken und Tricks zur Entdeckung von Beweisen gewidmet. Wir gehen auf deduktive Beweise, Umformulierungen, Beweise

durch Widerspruch, Beweise durch Induktion und andere wichtige Konzepte ein. Im letzten Abschnitt werden Konzepte vorgestellt, die in der Automatentheorie von zentraler Bedeutung sind: Alphabete, Zeichenreihen und Sprachen. 

1.1 Wozu dient das Studium der Automatentheorie?

Es gibt verschiedene Gründe, warum das Studium von Automaten und Komplexität einen wichtigen Teil des Kerns der Informatik bildet. Dieser Abschnitt stellt eine Einführung in die grundlegende Motivation dar und skizziert überdies die Hauptthemen, die in diesem Buch behandelt werden.

1.1.1 Einführung in endliche Automaten

Endliche Automaten sind ein nützliches Modell für viele wichtige Arten von Hardware und Software. Wir werden in Kapitel 2 und den nachfolgenden Kapiteln Beispiele dafür zeigen, wie die Konzepte verwendet werden. Für den Augenblick soll es genügen, einige der wichtigsten Arten aufzulisten:

- 1.** Software zum Entwurf und zur Überprüfung des Verhaltens digitaler Schaltkreise.
- 2.** Die »lexikalische Analysekomponente« von typischen Compilern, d. h. die Compilerkomponente, die den Eingabetext in logische Einheiten aufschlüsselt, wie z. B. Bezeichner, Schlüsselwörter und Satzzeichen.
- 3.** Software zum Durchsuchen umfangreicher Texte, wie Sammlungen von Webseiten, um Vorkommen von Wörtern, Ausdrücken oder anderer Muster zu finden.
- 4.** Software zur Verifizierung aller Arten von Systemen, die eine endliche Anzahl verschiedener Zustände besitzen, wie Kommunikationsprotokolle oder Protokolle zum sicheren Datenaustausch.

Obwohl wir bald die präzise Definition von Automaten unterschiedlicher Typen vorstellen, wollen wir unsere informelle Einführung mit einer kurzen Beschreibung dessen beginnen, was ein endlicher Automat ist und tut. Es gibt viele Systeme oder Komponenten, wie die oben aufgezählten, die so betrachtet werden können, dass sie sich zu jedem gegebenen Zeitpunkt in einem von einer endlichen Anzahl von »Zuständen« befinden. Zweck eines Zustands ist es, den relevanten Teil der Geschichte eines Systems festzuhalten. Da es lediglich eine endliche Anzahl von Zuständen gibt, kann im Allgemeinen nicht die gesamte Geschichte beschrieben werden. Das System muss daher sorgfältig entworfen werden, sodass das Wichtige beschrieben und gespeichert und das Unwichtige vergessen wird. Lediglich mit einer endlichen Anzahl von Zuständen zu arbeiten, bietet den Vorteil, dass wir das System mit einer fixen Menge von Ressourcen implementieren können. Wir können es beispielsweise als Schaltkreis in Hardware oder als einfaches Programm implementieren, das Entscheidungen anhand einer begrenzten Menge von Daten oder basierend auf der Position der Anweisung im Code trifft.

Beispiel 1.1 Ein Kippschalter ist der vielleicht einfachste nicht triviale endliche Automat. Dieses Gerät weiß, wann es sich im Zustand *Ein* oder *Aus* befindet, und es ermöglicht dem Benutzer, einen Schalter zu drücken, der, abhängig vom Zustand des Kippschalters, eine unterschiedliche Wirkung hat. Wenn sich der Kippschalter im Zustand *Aus* befindet, dann wird er durch das Drücken des Schalters in den Zustand *Ein* versetzt, und wenn sich der Kippschalter im Zustand *Ein* befindet, dann wird er durch das Drücken des Schalters in den Zustand *Aus* versetzt.

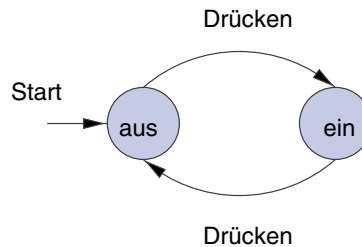


Abbildung 1.1: Als endlicher Automat dargestellter Kippschalter

Abbildung 1.1 zeigt die Darstellung des Kippschalters als endlichen Automaten. Wie bei allen endlichen Automaten werden die Zustände durch Kreise repräsentiert. In diesem Beispiel haben wir die Zustände *Ein* und *Aus* genannt. Die Pfeile zwischen den Zuständen werden mit den »Eingaben« beschriftet, die die auf das System wirkenden externen Einflüsse darstellen. Hier sind beide Pfeile mit der Eingabe *Drücken* beschriftet, die für das Drücken des Schalters durch den Benutzer steht. Die beiden Pfeile zeigen an, dass das System beim Empfang der Eingabe *Drücken* in den jeweils anderen Zustand wechselt, unabhängig davon, in welchem Zustand es sich davor befand.

Einer der Zustände wird als »Startzustand« ausgewählt, d. h. als der Zustand, in dem sich das System ursprünglich befindet. In unserem Beispiel ist *Aus* der Startzustand, und wir bezeichnen den Startzustand konventionell durch das Wort *Start* sowie einen Pfeil, der auf den betreffenden Zustand zeigt.

Es ist oft notwendig, einen oder mehrere Zustände als »finale« oder »akzeptierende« Zustände zu kennzeichnen. Wird nach einer Reihe von Eingaben in einen dieser Zustände gewechselt, dann zeigt dies an, dass die Eingabesequenz in irgendeiner Weise gültig ist. Wir hätten zum Beispiel den Zustand *Ein* in Abbildung 1.1 als akzeptierenden Zustand betrachten können, da das Gerät, das von diesem Kippschalter gesteuert wird, in diesem Zustand eingeschaltet ist. Gemäß Konvention werden akzeptierende Zustände durch einen doppelten Kreis bezeichnet. In Abbildung 1.1 haben wir auf eine solche Bezeichnung verzichtet. ■

Beispiel 1.2 Gelegentlich kann das, was von einem Zustand beschrieben wird, sehr viel komplizierter sein als die Wahl zwischen *Ein* und *Aus*. Abbildung 1.2 zeigt einen anderen endlichen Automaten, der Teil einer lexikalischen Analysekomponente sein könnte. Dieser Automat hat die Aufgabe, das Schlüsselwort *then* zu erkennen. Daher benötigt er fünf Zustände, die jeweils eine der verschiedenen Positionen, die bislang erreicht worden sind, im Wort *then* repräsentieren. Diese Positionen entsprechen den

Vorsilben des Wortes und reichen von einer leeren Zeichenreihe (d. h. bislang wurde kein Buchstabe des Wortes erkannt) bis zum vollständigen Wort.

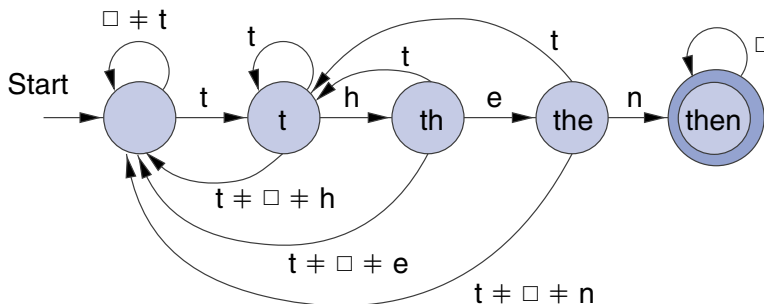


Abbildung 1.2: Explizite (gegenüber Original erweiterte) Darstellung (aller Buchstaben-Fälle); $\square \in \text{Alphabet}$

Die fünf Zustände in Abbildung 1.2 sind nach der bisher erkannten Vorsilbe von then benannt. Als Eingaben werden Buchstaben empfangen. Wir können uns vorstellen, dass die lexikalische Analysekomponente die einzelnen Buchstaben des Programms, das gerade kompiliert wird, nacheinander prüft und dass der nächste Buchstabe die Eingabe des endlichen Automaten bildet. Der Startzustand ist gleich der leeren Zeichenreihe, und jeder Zustand wird mit dem nächsten Buchstaben von then in den Zustand überführt, der dem nächstgrößeren Präfix von then entspricht. Der Zustand namens then wird erreicht, wenn die Eingabe dem Wort then genau entspricht. Wir können diesen Zustand als den einzigen akzeptierenden betrachten, da dieser Automat das Vorkommen des Wortes then erkennen soll. ■

1.1.2 Strukturelle Repräsentationen

Es gibt zwei wichtige Begriffsbildungen, die nicht zur Beschreibung von Automaten dienen, aber eine wichtige Rolle im Studium von Automaten und deren Anwendungen spielen.

1. Grammatiken sind nützliche Modelle zum Entwurf von Software, die Daten mit einer rekursiven Struktur verarbeitet. Das bekannteste Beispiel ist ein »Parser«, also die Komponente eines Compilers, die mit den rekursiv verschachtelten Elementen einer typischen Programmiersprache umgeht, wie arithmetische Ausdrücke, Bedingungsausdrücke usw. Beispielsweise besagt die Grammatikregel $E \Rightarrow E + E$, dass ein Ausdruck gebildet werden kann, indem zwei beliebige Ausdrücke durch ein Pluszeichen miteinander verbunden werden. Diese Grammatikregel ist typisch für die Art und Weise, wie in wirklichen Programmiersprachen Ausdrücke gebildet werden. Wir stellen diese so genannten kontextfreien Grammatiken in Kapitel 5 vor.

2. Reguläre Ausdrücke bezeichnen auch die Struktur von Daten, insbesondere von Text-Zeichenreihen. Wie wir in Kapitel 3 sehen werden, entsprechen die Muster, die durch reguläre Ausdrücke beschrieben werden, genau dem, was durch end-

liche Automaten beschrieben werden kann. Diese Ausdrücke sind ganz anders geartet als Grammatiken, und wir wollen uns hier mit einem einfachen Beispiel begnügen. Der im Unix-Stil formulierte reguläre Ausdruck '[A-Z] [a-z]*[] [A-Z] [A-Z]' repräsentiert mit einem Großbuchstaben beginnende Wörter, denen ein Leerzeichen und zwei Großbuchstaben folgen. Dieser Ausdruck beschreibt Textmuster, bei denen es sich um einen amerikanischen Städtenamen samt Staatenkürzel handeln könnte, wie z. B. Ithaca NY. Er deckt aus zwei Wörtern bestehende Städtenamen, wie Palo Alto CA, nicht ab; diese könnten jedoch durch einen komplizierteren regulären Ausdruck erfasst werden:

$$'[A-Z][a-z]*([] [A-Z][a-z]*)*[] [A-Z][A-Z]'$$

Zur Interpretation solcher Ausdrücke müssen wir nur wissen, dass [A-Z] den Bereich von Textzeichen zwischen dem Großbuchstaben »A« und dem Großbuchstaben »Z« repräsentiert, dass [] zur Darstellung eines einzelnen Leerzeichens dient und dass das Zeichen * für »eine beliebige Anzahl« des voranstehenden Ausdrucks steht. Runde Klammern werden zur Gruppierung der Komponenten des Ausdrucks verwendet; sie repräsentieren keine Zeichen des beschriebenen Textes.

1.1.3 Automaten und Komplexität

Automaten sind für das Studium der Grenzen der Berechenbarkeit von zentraler Bedeutung. Wie wir in der Einführung zu diesem Kapitel erwähnt haben, sind zwei wichtige Fragen zu untersuchen:

- 1.** Was können Computer überhaupt leisten? Man bezeichnet dies als die Frage der »Entscheidbarkeit«, und Probleme, die sich mithilfe eines Computers lösen lassen, werden »entscheidbar« genannt. Dieses Thema wird in Kapitel 9 behandelt.
- 2.** Was können Computer effizient leisten? Dies wird als Frage der »Handhabbarkeit« bezeichnet, und Probleme, die von einem Computer innerhalb einer Zeitspanne gelöst werden können, die nicht stärker anwächst als eine langsam wachsende Funktion der Größe der Eingabe, werden »handhabbar« genannt. Häufig gehen wir davon aus, dass alle polynomialen¹ Funktionen »langsam wachsen«, während Funktionen, die schneller wachsen als polynomiale Funktionen, als zu schnell wachsend betrachtet werden. Dieses Thema wird in Kapitel 10 näher untersucht.

1.2 Einführung in formale Beweise

Wer früher als Abiturient die Geometrie von Flächen gelernt hat, hat wahrscheinlich detaillierte »deduktive Beweise« führen müssen, in denen die Wahrheit einer Behauptung durch eine detaillierte Folge von Schritten und Schlüssen bewiesen wurde. Die Geometrie hat selbstverständlich praktische Aspekte (z. B. müssen Sie die

¹ »polynomiellen« (ebenso geläufig)

Formel zur Berechnung der Fläche eines Rechtecks kennen, um für ein Zimmer die richtige Menge an Teppichboden kaufen zu können), doch stellte das Studium formaler Beweismethoden einen mindestens ebenso wichtigen Grund für die Aufnahme dieses Zweigs der Mathematik in den Lehrplan dar.

Auch wenn es gut ist, dass man sich der Wahrheit einer Aussage bewusst ist, die man verwenden muss, werden wichtige Beweistechniken heute in den Schulen kaum mehr unterrichtet. Von Beweisen sollte jedoch jeder Informatiker etwas verstehen. Einige Informatiker vertreten sogar den extremen Standpunkt, dass ein formaler Beweis der Korrektheit eines Programms Hand in Hand mit dessen Programmierung gehen sollte. Wir bezweifeln, dass es produktiv wäre, so vorzugehen. Auf der anderen Seite gibt es jene, die behaupten, der Beweis wäre in der Disziplin der Informatik fehl am Platz. Dieses Lager proklamiert häufig das Motto »Wenn man nicht sicher ist, ob ein Programm korrekt ist, dann sollte man es starten und sehen, ob es richtig läuft«.

Unsere Position befindet sich irgendwo zwischen diesen beiden Extremen. Das Testen von Programmen ist sicherlich unabdingbar, hat jedoch seine Grenzen, da man Programme nicht mit jeder möglichen Eingabe testen kann. Wenn es sich um ein kompliziertes Programm handelt (angenommen, eine verzwickte Rekursion oder Iteration), dann ist es jedoch noch wichtiger, dass Sie verstehen, was in einer Schleife oder einer rekursiven Funktion passiert, da Sie sonst nicht in der Lage sein werden, richtig zu programmieren. Und wenn Tests ergeben, dass der Programmcode nicht korrekt ist, müssen Sie ihn auf jeden Fall korrigieren.

Um die Iteration oder Rekursion zu korrigieren, müssen Sie eine Induktionsannahme formulieren und Schlussfolgerungen ziehen können. Das Vorgehen, das zum Verständnis der Arbeitsweise eines korrekten Programms führt, entspricht im Grunde genommen genau dem Vorgehen zum Beweisen von Sätzen durch Induktion. Abgesehen davon, dass Sie damit Modelle erhalten, die für bestimmte Typen von Software nützlich sind, ist dies der Grund, warum formale Beweise zu einem traditionellen Bestandteil jedes Kurses in der Automatentheorie wurden. Die Automatentheorie eignet sich vielleicht mehr als andere Kernbereiche der Informatik für natürliche und interessante Beweise, die sowohl *deduktiver* (eine Folge von berechtigten Schritten) als auch *induktiver* (rekursive Beweise parametrisierter Aussagen, in denen die Aussage selbst mit »niedrigeren« Parameterwerten verwendet wird) Natur sein können.

1.2.1 Deduktive Beweise

Wie oben erwähnt, besteht ein **deduktiver Beweis** aus einer Folge von Aussagen, deren Wahrheit von einer Ausgangsaussage, der so genannten **Hypothese** oder **gegebenen Behauptung**, zu einer **Konklusion** führt. Jeder Beweisschritt muss sich nach einer akzeptierten logischen Regel aus den gegebenen Fakten oder aus vorangegangenen Aussagen dieses deduktiven Beweises ergeben.

Die Hypothese kann wahr oder falsch sein, was in der Regel von den Werten ihrer Parameter abhängt. Häufig setzt sich die Hypothese aus mehreren voneinander unabhängigen Aussagen zusammen, die durch den logischen Operator UND miteinander

verknüpft sind. In diesem Fall wird jede einzelne dieser Aussagen als **Hypothese** oder **gegebene Behauptung** bezeichnet.

Der Satz, dass die Folge der Beweisschritte von einer Hypothese H zu einer Konklusion K führt, entspricht der Aussage »Wenn H , dann K «. Man sagt in diesem Fall, dass K aus H abgeleitet ist. Ein Beispielsatz der Form »wenn H , dann K «, soll diese Punkte veranschaulichen.

Satz 1.1 Wenn $x \geq 4$, dann $2^x \geq x^2$.

Es ist nicht allzu schwierig, sich informell davon zu überzeugen, dass Satz 1.1 wahr ist, aber der formale Beweis erfordert Induktion und wird für Beispiel 1.7 aufgespart. Erstens ist zu beachten, dass » $x \geq 4$ « die Hypothese H darstellt. Diese Hypothese hat einen Parameter x und ist daher weder wahr noch falsch. Die Wahrheit der Hypothese hängt dagegen vom Wert des Parameters x ab; z. B. ist H wahr für $x = 6$ und falsch für $x = 2$.

Analog stellt » $2^x \geq x^2$ « die Konklusion K dar. Auch diese Aussage enthält den Parameter x und ist für bestimmte Werte von x wahr und für andere nicht. Beispielsweise ist K für $x = 3$ falsch, da $2^3 = 8$ und 8 kleiner ist als $3^2 = 9$. Andererseits ist K für $x = 4$ wahr, da $2^4 = 4^2 = 16$. Für $x = 5$ ist die Aussage auch wahr, da $2^5 = 32$ und 32 größer oder gleich $5^2 = 25$ ist.

Vielleicht erkennen Sie das intuitive Argument, aus dem wir schließen können, dass die Konklusion $2^x \geq x^2$ immer dann wahr ist, wenn $x \geq 4$. Wir haben bereits gesehen, dass die Konklusion für $x = 4$ wahr ist. Sobald x größer als 4 wird, verdoppelt sich die linke Seite 2^x jedes Mal, wenn x um 1 erhöht wird. Die rechte Seite x^2 wächst allerdings nur im Verhältnis $\left(\frac{x+1}{x}\right)^2$. Wenn $x \geq 4$, dann kann $\frac{x+1}{x}$ nicht größer als 1,25 sein und folglich kann $\left(\frac{x+1}{x}\right)^2$ nicht größer als 1,5625 sein. Da $1,5625 < 2$, wächst die linke Seite 2^x bei jedem Anstieg von x über 4 stärker an als die rechte Seite x^2 . Folglich gilt, dass x um einen beliebigen Betrag erhöht werden kann, solange wir mit einem Wert von $x = 4$ beginnen, mit dem die Ungleichung bereits erfüllt ist.

Wir haben hiermit einen zwar informellen, aber akkuraten Beweis des Satzes 1.1 zu Ende geführt. Wir werden zu diesem Beweis zurückkehren und ihn in Beispiel 1.17 präzisieren, nachdem wir »induktive« Beweise vorgestellt haben.

Satz 1.1 umfasst wie alle interessanten Sätze eine unendliche Anzahl von miteinander in Beziehung stehenden Fakten, hier die Aussage »wenn $x \geq 4$, dann $2^x \geq x^2$ « für alle ganzen Zahlen x . In der Tat ist die Annahme, dass x eine ganze Zahl ist, gar nicht erforderlich. Da im Beweis aber davon die Rede war, dass x beginnend mit $x = 4$ jeweils um 1 erhöht wird, wurde hier eigentlich nur der Fall berücksichtigt, dass x eine ganze Zahl ist.

Satz 1.1 kann zur Ableitung anderer Sätze herangezogen werden. Im nächsten Beispiel betrachten wir einen vollständigen deduktiven Beweis eines einfachen Satzes, in dem Satz 1.1 zur Anwendung kommt.

Satz 1.2 Wenn x die Summe der Quadrate von vier positiven ganzen Zahlen ist, dann gilt $2^x \geq x^2$.

Beweis ■ Intuitiv denken wir uns den Beweis so, dass x mindestens gleich 4 sein muss, wenn die Hypothese – also dass x die Summe der Quadrate von vier positiven ganzen Zahlen ist – wahr ist. Daher gilt hier die Hypothese von Satz 1.1, und weil wir an den Wahrheitsgehalt dieses Satzes glauben, können wir behaupten, auch dessen Konklusion sei für x wahr. Diese Überlegung lässt sich in einer Folge von Schritten ausdrücken. Jeder Schritt ist entweder die Hypothese des zu beweisenden Satzes, Teil dieser Hypothese oder eine Aussage, die aus einer oder mehreren vorangehenden Aussagen folgt.

Mit »folgt« ist hier gemeint, dass wir unterstellen: Wenn es sich bei einer vorangehenden Aussage um die Hypothese eines Satzes handelt, dann ist die Konklusion dieses Satzes wahr und kann als Aussage unseres Beweises niedergeschrieben werden. Diese logische Regel wird häufig als **Modus ponens** bezeichnet, d. h. wenn wir wissen, dass H wahr ist, und wenn wir wissen, dass »wenn H , dann K « wahr ist, dann können wir daraus schließen, dass K wahr ist. Wir lassen zudem bestimmte logische Schritte zur Bildung einer Aussage zu, die aus einer oder mehreren vorangehenden Aussagen folgt. Wenn beispielsweise A und B zwei vorangegangene Aussagen sind, dann können wir daraus die Aussage » A und B « ableiten.

Tabelle 1.1 zeigt die Folge der Aussagen, die zum Beweis von Satz 1.2 erforderlich ist. Obwohl wir Sätze im Allgemeinen nicht in einer so stilisierten Form beweisen werden, erleichtert es diese Darstellung, Beweise als explizite Liste von Aussagen zu begreifen, die jeweils eine präzise Begründung haben. In Schritt (1) haben wir eine der gegebenen Behauptungen des Satzes wiederholt, nämlich dass x die Summe der Quadrate von vier ganzen Zahlen ist. Es ist in Beweisen häufig hilfreich, wenn wir Objekte benennen, auf die zwar Bezug genommen wird, die aber im Satz nicht benannt sind. Wir haben dies hier getan und den vier ganzen Zahlen die Namen a , b , c und d gegeben.

Aussage		Begründung
1.	$x = a^2 + b^2 + c^2 + d^2$	Gegeben
2.	$a \geq 1; b \geq 1; c \geq 1; d \geq 1$	Gegeben
3.	$a^2 \geq 1; b^2 \geq 1; c^2 \geq 1; d^2 \geq 1$	(2) und Gesetze der Arithmetik
4.	$x \geq 4$	(1), (3) und Gesetze der Arithmetik
5.	$2^x \geq x^2$	(4) und Satz 1.1

Tabelle 1.1: Formaler Beweis des Satzes 1.2

In Schritt (2) halten wir den anderen Teil der Hypothese des Satzes fest: dass die zu potenzierenden ganzen Zahlen größer oder gleich 1 sind. Diese Aussage repräsentiert technisch vier Aussagen, nämlich jeweils eine Aussage für jede der vier implizit gegebenen ganzen Zahlen. In Schritt (3) stellen wir dann fest, dass das Quadrat einer Zahl, die größer oder gleich 1 ist, auch größer oder gleich 1 ist. Wir begründen dies mit der Tatsache, dass Aussage (2) wahr ist, und den »Gesetzen der Arithmetik«. Das heißt, wir setzen voraus, dass der Leser einfache Aussagen zu Ungleichungen kennt und/oder beweisen kann, wie z. B. die Behauptung »wenn $y \geq 1$, dann $y^2 \geq 1$ «.

Schritt (4) setzt die Aussagen (1) und (3) ein. Die erste Aussage besagt, dass x die Summe von vier Quadraten ist, und Aussage (3) besagt, dass jedes dieser Quadrate größer oder gleich 1 ist. Wir schließen wieder anhand wohl bekannter Gesetze der Arithmetik, dass x größer oder gleich $1 + 1 + 1 + 1$ bzw. 4 ist.

Im letzten Schritt (5) verwenden wir Aussage (4), d. h. die Hypothese von Satz 1.1. Der Satz selbst stellt die Begründung für die Konklusion dar, da dessen Hypothese die vorangegangene Aussage ist. Da Aussage (5), die der Konklusion von Satz 1.1 entspricht, die Konklusion von Satz 1.2 bildet, wurde Satz 1.2 hiermit bewiesen. Wir sind von der Hypothese dieses Satzes ausgegangen, und es ist uns gelungen, per Deduktion seine Konklusion abzuleiten. ■

1.2.2 Reduktion auf Definitionen

In den vorigen beiden Sätzen wurden Begriffe verwendet, mit denen Sie vertraut sein sollten: z. B. ganze Zahlen, Addition und Multiplikation. In vielen anderen Sätzen, einschließlich vielen der Automatentheorie, haben die in der Behauptung verwendeten Begriffe Implikationen, die weniger offensichtlich sind. Folgendes Vorgehen erweist sich in vielen Beweisen als nützlich:

- Wenn Sie sich nicht sicher sind, wie Sie einen Beweis beginnen sollen, wandeln Sie alle in der Hypothese enthaltenen Begriffe in ihre Definitionen um.

Hier ist ein Beispiel für einen Satz, der einfach zu beweisen ist, nachdem seine Aussage in Grundbegriffen ausgedrückt wurde. In diesem Satz werden die folgenden Definitionen verwendet:

1. Eine Menge S sei endlich, wenn es eine ganze Zahl n gibt, sodass S genau n Elemente enthält. Wir schreiben $\|S\| = n$ wobei $\|S\|$ für die Anzahl der Elemente der Menge S steht. Ist die Menge S nicht endlich, dann sagen wir, S sei *unendlich*. Informell ausgedrückt ist eine unendliche Menge eine Menge, die mehr als eine durch ganze Zahlen auszudrückende Anzahl von Elementen enthält.
2. Wenn S und T Teilmengen einer Menge U sind, dann ist T die Komplementärmenge von S (in Bezug auf U), wenn $S \cup T = U$ und $S \cap T = \emptyset$. Das heißt, jedes Element von U ist entweder in S oder in T enthalten, aber nicht in S und in T , oder anders ausgedrückt: T besteht genau aus den Elementen von U , die nicht in S enthalten sind.

Satz 1.3 S sei eine endliche Teilmenge einer unendlichen Menge U . T sei die Komplementärmenge von S in Bezug auf U . Dann ist T unendlich.

Beweis ■ Dieser Satz besagt informell ausgedrückt Folgendes: Wenn es einen unendlichen Vorrat von etwas (U) gibt und man einen endlichen Teil (S) davon abzieht, dann bleibt ein unendlicher Vorrat übrig. Zuerst wollen wir die Fakten des Satzes, wie in Tabelle 1.2 dargestellt, umformulieren.

Ursprüngliche Aussage	Neue Aussage
S ist endlich.	Es gibt eine ganze Zahl n derart, dass $\ S\ = n$.
U ist unendlich.	Für keine ganze Zahl p gilt $\ U\ = p$.
T ist die Komplementärmenge von S .	$S \cup T = U$ und $S \cap T = \emptyset$.

Tabelle 1.2: Umformulierung der gegebenen Hypothesen des Satzes 1.3

Wir können noch immer keine weiteren Aussagen treffen und verwenden daher eine gängige Beweistechnik namens »Beweis durch Widerspruch«. Bei dieser Beweismethode, die in Abschnitt 1.3.3 näher erläutert wird, unterstellen wir, dass die Konklusion falsch ist. Unter Verwendung dieser Annahme und Teilen der Hypothese beweisen wir dann das Gegenteil einer der gegebenen Aussagen der Hypothese. Damit haben wir dann gezeigt, dass die Hypothese unmöglich in allen Teilen wahr und die Konklusion gleichzeitig falsch sein kann. Es bleibt lediglich die Möglichkeit übrig, dass die Konklusion wahr ist, wenn die Hypothese wahr ist. Dies bedeutet, dass der Satz wahr ist.

Im Fall von Satz 1.3 lautet das Gegenteil zur Konklusion » T ist endlich«. Angenommen, T sei endlich und die Aussage der Hypothese, die besagt, S ist endlich (also $\|S\| = n$ für eine ganze Zahl n), sei wahr. Wir können die Annahme, dass T endlich ist, in ähnlicher Weise umformulieren zu $\|T\| = m$ für eine ganze Zahl m .

Eine der gegebenen Aussagen besagt, dass $S \cup T = U$ und $S \cap T = \emptyset$. Das heißt, dass die Elemente von U genau den Elementen von S und T entsprechen. Folglich muss U genau $n + m$ Elemente enthalten. Da $n + m$ eine ganze Zahl ist, haben wir $\|U\| = n + m$ bewiesen, woraus folgt, dass U endlich ist. Genauer gesagt, haben wir bewiesen, dass die Anzahl der in U enthaltenen Elemente eine ganze Zahl ist, was der Definition von »endlich« entspricht. Die Aussage, dass U endlich ist, widerspricht jedoch der gegebenen Aussage, dass U unendlich ist. Folglich haben wir aus der Annahme, die Konklusion sei falsch, einen Widerspruch zu einer Hypothese abgeleitet, und nach dem Prinzip »Beweis durch Widerspruch« können wir damit folgern, dass der Satz wahr ist. ■

Beweise müssen nicht so wortreich sein. Nachdem wir die Gedankengänge, auf denen der Beweis beruht, dargestellt haben, wollen wir diesen Satz in einigen wenigen Zeilen erneut beweisen.

Beweis ■ (von Satz 1.3) Wir wissen, dass $S \cup T = U$ und dass S und T disjunkt sind, sodass $\|S\| + \|T\| = \|U\|$. Da S endlich ist, gilt $\|S\| = n$ für eine ganze Zahl n , und da U unendlich ist, gibt es keine ganze Zahl p derart, dass $\|U\| = p$. Angenommen, T sei endlich, sodass es eine ganze Zahl m gibt, für die gilt: $\|T\| = m$; dann ist $\|U\| = \|S\| + \|T\| = n + m$, was der gegebenen Behauptung widerspricht, dass es keine ganze Zahl p gibt, die gleich $\|U\|$ ist. ■

1.2.3 Andere Formen von Sätzen

Die »Wenn-dann«-Form von Sätzen ist in klassischen Gebieten der Mathematik am gebräuchlichsten. Es werden jedoch auch andere Behauptungen als Sätze bewiesen. In diesem Abschnitt untersuchen wir die gebräuchlichsten Formen von Aussagen und was in der Regel zu deren Beweis vonnöten ist.

Formen von »Wenn-dann«

Es gibt verschiedene Arten von Satzaussagen, die sich von der einfachen Aussageform »Wenn H , dann K « zwar unterscheiden, im Grunde genommen aber dasselbe aussagen: Wenn die Hypothese H für einen gegebenen Wert des/r Parameter/s wahr ist, dann ist die Konklusion K für denselben Wert wahr. Im Folgenden werden einige andere Möglichkeiten, »wenn H , dann K « auszudrücken, aufgeführt:

1. H impliziert K .
2. H nur dann, wenn K .
3. K , wenn H .
4. Wenn H gilt, folgt daraus K .

Es gibt zudem viele Varianten der Form (4), wie z. B. »wenn H gilt, dann gilt K «.

Beispiel 1.3 Die Behauptung von Satz 1.1 würde in diesen vier Aussageformen wie folgt erscheinen:

1. $x \geq 4$ impliziert $2^x \geq x^2$.
2. $x \geq 4$ nur dann, wenn $2^x \geq x^2$.
3. $2^x \geq x^2$, wenn $x \geq 4$.
4. Wenn $x \geq 4$ gilt, folgt daraus $2^x \geq x^2$.

Aussagen mit Quantoren

Viele Sätze beinhalten Aussagen, in denen der Allquantor (»für alle«) und der Existenzquantor (»es gibt«) oder Varianten dieser Quantoren, wie z. B. »für jedes« statt »für alle«, verwendet werden. Die Reihenfolge, in der diese Quantoren stehen, wirkt sich auf die Bedeutung der Aussage aus. Häufig ist es hilfreich, sich Aussagen mit mehreren Quantoren als Spiel zwischen zwei Spielern – dem Allquantor und dem Existenzquantor – vorzustellen, die abwechselnd Werte für die Parameter des Satzes festlegen. Der Allquantor muss alle möglichen Optionen berücksichtigen, und daher werden diese Optionen im Allgemeinen durch Variablen beschrieben und nicht durch Werte ersetzt. Der Existenzquantor muss dagegen lediglich einen Wert auswählen, der von den zuvor von den Spielern gewählten Werten abhängen kann. Die Reihenfolge, in der die Quantoren in der Aussage stehen, bestimmt, wer zuerst eine Auswahl treffen kann. Wenn der Spieler, der zuletzt eine Wahl treffen soll, stets einen zulässigen Wert findet, dann ist die Aussage wahr.

Betrachten Sie beispielsweise eine alternative Definition von »unendliche Menge«: Die Menge S ist unendlich, und zwar genau dann, wenn für alle ganzen Zahlen n gilt, dass es eine Teilmenge T von S gibt, die genau n Elemente enthält. Da hier »für alle« dem Quantor »es gibt« voransteht, müssen wir eine beliebige ganze Zahl n betrachten. Nun ist »es gibt« an der Reihe, eine Teilmenge von S auszuwählen, und kann dazu die Kenntnis der ganzen Zahl n nutzen. Wenn es sich bei S beispielsweise um die Menge der ganzen Zahlen handelt, dann könnte »es gibt« die Teilmenge $T = \{1, 2, \dots, n\}$ auswählen und somit unabhängig davon, welchen Wert n hat, eine wahre Aussage bilden. Dies ist der Beweis, dass die Menge der ganzen Zahlen unendlich ist.

Die folgende Aussage sieht wie die Definition von »unendlich« aus, aber sie ist inkorrekt, weil darin die Reihenfolge der Quantoren vertauscht wurde: »Es gibt eine Teilmenge T der Menge S derart, dass für alle n gilt, dass die Teilmenge T genau n Elemente enthält.« Wenn mit S eine Menge wie die Menge der ganzen Zahlen gegeben ist, kann hier der Spieler »es gibt« eine beliebige Menge T auswählen. Angenommen, für T wird $\{1, 2, 5\}$ gewählt. Der Spieler »für alle« muss für diese Auswahl zeigen, dass T für jedes mögliche n auch n Elemente enthält. Dies ist allerdings nicht möglich. Beispielsweise ist diese Aussage falsch für $n = 4$. Tatsächlich ist sie falsch für alle $n \neq 3$.

Zudem wird in der formalen Logik häufig der Operator \rightarrow an Stelle des Ausdrucks »wenn, dann« verwendet. Die Aussage »wenn H , dann K « kann daher in der mathematischen Fachliteratur auch in der Form $H \rightarrow K$ vorkommen. Wir werden diese Notation hier nicht verwenden.

Aussagen der Form »Genau dann, wenn«

Gelegentlich finden wir Aussagen der Form » A genau dann, wenn B «. Andere Varianten dieser Form sind » A ist äquivalent mit B « und » A dann – und nur dann –, wenn B «. Diese Aussage stellt eigentlich zwei Wenn-dann-Aussagen dar: »wenn A , dann B « und »wenn B , dann A «. Wir beweisen » A genau dann, wenn B «, indem wir diese beiden Behauptungen beweisen:

- 1.** Den *Wenn*-Teil: »Wenn B , dann A « und
- 2.** den *Nur-wenn*-Teil: »Wenn A , dann B «, der häufig in der äquivalenten Form » A nur dann, wenn B « ausgedrückt wird.

Es ist beliebig, in welcher Reihenfolge die Beweise erbracht werden. In vielen Sätzen ist ein Teil entschieden einfacher als der andere, und es ist üblich, den leichteren Teil zuerst zu beweisen und damit aus dem Weg zu räumen.

In der formalen Logik wird gelegentlich der Operator \leftrightarrow oder \equiv für Aussagen vom Typ »Genau dann, wenn ...«² verwendet. Das heißt, $A \equiv B$ und $A \leftrightarrow B$ bedeuten dasselbe wie »A genau dann, wenn B«.

Wie formal müssen Beweise sein?

Es ist nicht einfach, diese Frage zu beantworten. Beweise dienen im Grunde genommen nur dem Zweck, jemanden (und hierbei kann es sich um denjenigen handeln, der Ihre Hausarbeit benotet, oder um Sie selbst) von der Richtigkeit der Strategie zu überzeugen, die Sie in Ihrem Programm einsetzen. Wenn der Beweis überzeugend ist, dann ist er ausreichend. Wenn er den Leser des Beweises nicht überzeugen kann, dann wurde im Beweis zu viel weggelassen.

Ein Teil der Unsicherheit in Bezug auf Beweise geht auf die unterschiedlichen Kenntnisstände zurück, die die Leser des Beweises haben können. In Satz 1.2 haben wir daher unterstellt, dass Sie die arithmetischen Grundgesetze kennen und eine Aussage wie »wenn $y \geq 1$, dann $y^2 \geq 1$ « für wahr halten. Wären Ihnen die arithmetischen Gesetze nicht geläufig, dann müssten wir diese Aussage durch weitere Schritte in unserem deduktiven Beweis verifizieren.

Bestimmte Dinge sind in Beweisen jedoch erforderlich und ein Beweis wird inadäquat, wenn man sie weglässt. Jeder deduktive Beweis, in dem Aussagen verwendet werden, die nicht durch die gegebene Behauptung oder durch vorangegangene Aussagen gerechtfertigt sind, kann nicht adäquat sein. Der Beweis einer Aussage vom Typ »Genau dann, wenn ...« muss einen Beweis für den »Genau dann«-Teil und einen Beweis für den »Wenn«-Teil umfassen. Als weiteres Beispiel sei angeführt, dass induktive Beweise (die in Abschnitt 1.4 erörtert werden) Beweise der Basis und des Induktionsschritts erfordern.

Beim Beweis einer Aussage vom Typ »Genau dann, wenn ...« dürfen Sie nicht vergessen, dass beide Teile (also »genau dann« und »wenn«) bewiesen werden müssen. Gelegentlich ist es hilfreich, eine Aussage dieses Typs in eine Folge von Äquivalenzen aufzuspalten. Das heißt, um »A genau dann, wenn B« zu beweisen, können Sie zuerst »A genau dann, wenn C« und anschließend »C genau dann, wenn B« beweisen. Diese Methode funktioniert, solange Sie sich vor Augen halten, dass jeder Schritt in beiden Richtungen bewiesen werden muss. Wird ein Schritt lediglich in einer Richtung bewiesen, dann wird der gesamte Beweis dadurch ungültig.

Es folgt ein Beispiel für einen einfachen Beweis einer Aussage vom Typ »Genau dann, wenn ...«. In diesem Beweis werden die folgenden Notationen verwendet:

- 1.** $\lfloor x \rfloor$, die auf eine ganze Zahl abgerundete reelle Zahl x , ist die größte ganze Zahl kleiner oder gleich x .
- 2.** $\lceil x \rceil$, die auf eine ganze Zahl aufgerundete reelle Zahl x , ist die kleinste ganze Zahl größer oder gleich x .

² »gdw.«

Satz 1.4 x sei eine reelle Zahl. Dann gilt $\lfloor x \rfloor = \lceil x \rceil$ genau dann, wenn x eine ganze Zahl ist.

Beweis ■ (Genau-dann-Teil) In diesem Teil unterstellen wir $\lfloor x \rfloor = \lceil x \rceil$ und versuchen zu beweisen, dass x eine ganze Zahl ist. Mithilfe der Definitionen von Abrundung und Aufrundung stellen wir fest, dass $\lfloor x \rfloor \leq x$ und $\lceil x \rceil \geq x$. Gegeben ist jedoch $\lfloor x \rfloor = \lceil x \rceil$. Daher können wir in der ersten Ungleichung die Abrundung durch die Aufrundung ersetzen und zu dem Schluss kommen, dass $\lceil x \rceil \leq x$. Da sowohl $\lceil x \rceil \leq x$ als auch $\lceil x \rceil \geq x$ gilt, können wir auf Grund der Gesetze der Arithmetik schlussfolgern, dass $\lceil x \rceil = x$. Da $\lceil x \rceil$ stets eine ganze Zahl ist, muss auch x eine ganze Zahl sein.

(Wenn-Teil) Wir unterstellen nun, dass x eine ganze Zahl ist, und versuchen zu beweisen, dass $\lfloor x \rfloor = \lceil x \rceil$. Dieser Teil ist einfach. Gemäß den Definitionen von Abrundung und Aufrundung ist sowohl $\lfloor x \rfloor$ als auch $\lceil x \rceil$ gleich x , wenn x eine ganze Zahl ist, und folglich sind diese beiden Ausdrücke gleich. ■

1.2.4 Sätze, die keine Wenn-dann-Aussagen zu sein scheinen

Gelegentlich treffen wir auf einen Satz, der keine Hypothese zu haben scheint. Ein Beispiel hierfür ist folgende wohl bekannte Tatsache aus der Trigonometrie:

Satz 1.5 $\sin^2 \theta + \cos^2 \theta = 1$.

Diese Aussage besitzt tatsächlich eine Hypothese und diese Hypothese besteht aus allen Aussagen, die Sie kennen müssen, um diese Aussage interpretieren zu können. Beispielsweise umfasst die verborgene Hypothese hier die Aussage, dass θ ein Winkel ist und die Funktionen sinus und cosinus daher ihre übliche, für Winkel gültige Bedeutung haben. Ausgehend von diesen Definitionen und dem Satz des Pythagoras (in einem rechtwinkligen Dreieck ist das Quadrat der Hypotenuse gleich der Summe der Quadrate der beiden anderen Seiten) könnte man diesen Satz beweisen. Im Grunde genommen lautet die Wenn-dann-Form des Satzes eigentlich: »Wenn θ ein Winkel ist, dann $\sin^2 \theta + \cos^2 \theta = 1$.«

1.3 Weitere Formen von Beweisen

In diesem Abschnitt greifen wir verschiedene zusätzliche Themen auf, die mit der Bildung von Beweisen in Zusammenhang stehen:

1. Beweise in Bezug auf Mengen
2. Beweise durch Widerspruch
3. Beweise durch Gegenbeispiel

1.3.1 Beweise der Äquivalenz von Mengen

In der Automatentheorie müssen wir häufig einen Satz beweisen, der besagt, dass auf unterschiedliche Weise erstellte Mengen gleich sind. Häufig handelt es sich bei diesen Mengen um Mengen von Zeichenreihen, die als »**Sprachen**« bezeichnet werden, aber in diesem Abschnitt ist das Wesen der Mengen unwichtig. Wenn E und F zwei Ausdrücke sind, die Mengen repräsentieren, dann bedeutet die Aussage $E = F$, dass die beiden bezeichneten Mengen gleich sind. Genauer gesagt, jedes Element der durch E angegebenen Menge ist auch in der durch F angegebenen Menge enthalten, und jedes Element der durch F angegebenen Menge ist auch in der durch E angegebenen Menge enthalten.

Beispiel 1.4 Das Kommutativgesetz der Vereinigung von Mengen besagt, dass die Vereinigung der beiden Mengen R und S in beliebiger Reihenfolge erfolgen kann. Das heißt, $R \cup S = S \cup R$. In diesem Fall ist E der Ausdruck $R \cup S$ und F der Ausdruck $S \cup R$. Das Kommutativgesetz der Vereinigung von Mengen besagt, dass $E = F$. ■

Die Mengengleichheit $E = F$ kann auch als Genau-dann-wenn-Aussage formuliert werden: Ein Element x ist genau dann in E enthalten, wenn x in F enthalten ist. Infolgedessen skizzieren wir einen Beweis für jede Aussage, die die Gleichheit der beiden Mengen $E = F$ unterstellt. Dieser Beweis hat die Form jedes Genau-dann-wenn-Beweises:

1. Beweis, dass gilt, wenn x in E enthalten ist, dann ist x in F enthalten.
2. Beweis, dass gilt, wenn x in F enthalten ist, dann ist x in E enthalten.

Als Beispiel für dieses Beweisverfahren wollen wir das *Distributivgesetz der Vereinigung von Schnittmengen* beweisen:

Satz 1.6 $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

Beweis ■ Die beiden Mengenausdrücke lauten $E = R \cup (S \cap T)$ und

$$F = (R \cup S) \cap (R \cup T).$$

Wir werden die beiden Teile des Satzes nacheinander beweisen. Im »Wenn«-Teil unterstellen wir, dass x Element von E ist, und zeigen, dass x auch in F enthalten ist. In diesem Teil, der in Tabelle 1.3 zusammengefasst wird, werden die Definitionen von Vereinigung und Schnittmenge herangezogen, die wir als bekannt voraussetzen.

Wir müssen dann den »Genau-dann«-Teil des Satzes beweisen. Hier nehmen wir an, dass x Element von F ist, und zeigen, dass x in E enthalten ist. Die Beweisschritte sind in Tabelle 1.4 zusammengefasst. Da wir nun beide Teile der Genau-dann-wenn-Aussage bewiesen haben, gilt das Distributivgesetz von Vereinigungen von Schnittmengen als bewiesen. ■

Aussage	Begründung
1. x ist in $R \cup (S \cap T)$ enthalten.	gegeben
2. x ist in R oder x ist in $(S \cap T)$ enthalten.	(1) und Definition von Vereinigungsmenge
3. x ist in R oder x ist sowohl in S als auch T enthalten.	(2) und Definition von Schnittmenge
4. x ist in $R \cup S$ enthalten.	(3) und Definition von Vereinigungsmenge
5. x ist in $R \cup T$ enthalten.	(3) und Definition von Vereinigungsmenge
6. x ist in $(R \cup S) \cap (R \cup T)$ enthalten.	(4), (5) und Definition von Schnittmenge

Tabelle 1.3: Beweisschritte für den »Wenn«-Teil von Satz 1.6

Aussage	Begründung
1. x ist in $(R \cup S) \cap (R \cup T)$ enthalten.	gegeben
2. x ist in $R \cup S$ enthalten.	(1) und Definition von Schnittmenge
3. x ist in $R \cup T$ enthalten.	(1) und Definition von Schnittmenge
4. x ist in R oder x ist sowohl in S als auch T enthalten.	(2), (3) und Schlussfolgerung über Vereinigungsmengen
5. x ist in R oder x ist in $(S \cap T)$ enthalten.	(4) und Definition von Schnittmenge
6. x ist in $R \cup (S \cap T)$ enthalten.	(5) und Definition von Vereinigungsmenge

Tabelle 1.4: Beweisschritte für den »Genau-dann«-Teil von Satz 1.6

1.3.2 Die Umkehrung

Jede Wenn-dann-Aussage verfügt über eine äquivalente Form, die unter bestimmten Bedingungen einfacher zu beweisen ist. Die Umkehrung (oder Kontraposition) der Aussage »wenn H , dann K « und »wenn nicht K , dann nicht H «. Eine Aussage und ihre Umkehrung sind entweder beide wahr oder beide falsch, daher kann eine Aussage durch ihre Umkehrung bewiesen werden und umgekehrt.

Warum die Aussagen »wenn H , dann K « und »wenn nicht K , dann nicht H « logisch äquivalent sind, wird deutlich, wenn man feststellt, dass vier Fälle zu betrachten sind:

1. H und K sind beide wahr.
2. H ist wahr und K ist falsch.
3. K ist wahr und H ist falsch.
4. H und K sind beide falsch.

Es gibt nur eine Möglichkeit, die Wenn-dann-Aussage falsch werden zu lassen. Die Hypothese muss wahr und die Konklusion falsch sein, wie in Fall (2). In den anderen drei Fällen, einschließlich Fall (4), in dem die Konklusion falsch ist, ist die Wenn-dann-Aussage selbst wahr.

Überlegen Sie jetzt, in welchen Fällen die Umkehrung »wenn nicht K , dann nicht H « falsch ist. Damit diese Aussage falsch ist, muss ihre Hypothese (die »nicht K « lautet) wahr sein und ihre Konklusion (die »nicht H « lautet) muss falsch sein. Aber »nicht K « ist genau dann wahr, wenn K falsch ist, und »nicht H « ist genau dann falsch, wenn H wahr ist. Diese beiden Bedingungen repräsentieren wiederum Fall (2), womit gezeigt wurde, dass in jedem dieser vier Fälle die ursprüngliche Aussage und ihre Umkehrung entweder beide wahr oder beide falsch sind, dass sie also logisch äquivalent sind.

Genau-dann-wenn-Aussagen mit Mengen

Wie bereits erwähnt, sind Sätze, die Äquivalenzen von Mengen zum Inhalt haben, Genau-dann-wenn-Aussagen. Daher hätte Satz 1.6 auch wie folgt formuliert werden können: Ein Element x ist in $R \cup (S \cap T)$ enthalten genau dann, wenn x in $(R \cup S) \cap (R \cup T)$ enthalten ist.

Ein anderer gebräuchlicher Ausdruck einer Mengenäquivalenz beinhaltet die Wendung: »Alle und ausschließlich«. Satz 1.6 hätte beispielsweise auch so ausgedrückt werden können: »Alle Elemente von $R \cup (S \cap T)$ sind alle und ausschließlich Elemente von $(R \cup S) \cap (R \cup T)$.« Intuitiver: $A \equiv B$ gdw. $A \subseteq B \wedge B \subseteq A$: A äquivalent B genau-dann-wenn alle Elemente aus A liegen auch in B und alle Elemente aus B liegen auch in A .

Die Konversion

Die Begriffe »Umkehrung« (oder »Kontraposition«) und »Konversion« dürfen nicht miteinander verwechselt werden. Die *Konversion* einer Wenn-dann-Aussage ist die umgekehrte Implikation oder die »Gegenrichtung«; das heißt, die Konversion von »wenn H , dann K « lautet »wenn K , dann H «. Im Gegensatz zur Umkehrung, die mit der ursprünglichen Aussage logisch äquivalent ist, ist die Konversion *nicht* mit der ursprünglichen Aussage äquivalent. Bei den beiden Teilen eines Genau-dann-wenn-Beweises handelt es sich stets um eine Behauptung und deren Konversion.

Beispiel 1.5 Rufen Sie sich Satz 1.1 in Erinnerung, dessen Aussage lautete: »Wenn $x \geq 4$, dann $2^x \geq x^2$ «. Die Umkehrung dieser Aussage lautet: »Wenn nicht $2^x \geq x^2$, dann nicht $x \geq 4$ «. Weniger formal ausgedrückt und unter Verwendung der Tatsache, dass »nicht $a \geq b$ « dasselbe ist wie $a < b$, lautet die Umkehrung: »Wenn $2^x < x^2$, dann $x < 4$.« ■

Wenn wir einen Genau-dann-wenn-Satz beweisen müssen, dann eröffnet uns die Verwendung der Umkehrung in einem der Teile verschiedene Möglichkeiten. Angenommen, wir wollen die Mengenäquivalenz von $E = F$ beweisen. Statt zu beweisen, dass

»wenn x Element von E ist, dann ist x auch Element von F , und wenn x Element von F ist, dann ist x auch Element von E «, könnten wir eine Richtung umkehren. Eine äquivalente Beweisform ist:

- Wenn x Element von E ist, dann ist x auch Element von F , und wenn x nicht Element von E ist, dann ist x nicht Element von F .

Wir könnten E und F in der obigen Aussage auch vertauschen.

1.3.3 Beweis durch Widerspruch

Eine andere Möglichkeit, eine Aussage der Form »wenn H , dann K « zu beweisen, besteht im Beweis der Aussage:

- » H und nicht K impliziert Falschheit.«

Sie beginnen also mit der Annahme der Hypothese H und der Negation C der Konklusion K . Sie vervollständigen den Beweis, indem Sie zeigen, dass aus H und C etwas als falsch Bekanntes logisch folgt. Diese Form des Beweises wird **Beweis durch Widerspruch** genannt.

Beispiel 1.6 Erinnern Sie sich an Satz 1.3, in dem wir die Wenn-dann-Aussage mit der Hypothese $H = \text{»}U \text{ ist eine unendliche Menge, } S \text{ ist eine endliche Teilmenge von } U \text{ und } T \text{ ist die Komplementärmenge von } S \text{ in Bezug auf } U\text{«}$ und der Konklusion $K = \text{»}T \text{ ist unendlich«}$ bewiesen haben. Wir werden diesen Satz durch Widerspruch beweisen. Wir unterstellten »nicht K «; wir nahmen also an, dass T endlich ist.

Unser Beweis bestand darin, die Falschheit von H und nicht K abzuleiten. Wir haben zuerst gezeigt, dass auf Grund der Annahme, dass sowohl S als auch T endlich sind, auch U endlich sein muss. Da U aber gemäß der Hypothese H unendlich sein soll und eine Menge nicht gleichzeitig endlich und unendlich sein kann, haben wir die logische Aussage als »falsch« bewiesen. In Begriffen der Logik ausgedrückt, liegen eine Behauptung p (U ist endlich) und ihre Negation nicht p (U ist unendlich) vor. Wir nutzen dann die Tatsache, dass » p und nicht p « logisch äquivalent mit »falsch« ist. ■

Um zu verstehen, warum Beweise durch Widerspruch logisch korrekt sind, rufen Sie sich in Erinnerung, dass es vier Kombinationen von Wahrheitswerten für H und K gibt. Nur im zweiten Fall (H ist wahr und K ist falsch) ist die Aussage »wenn H , dann K « falsch. Indem gezeigt wurde, dass H und nicht K zu einer falschen Aussage führt, wurde gezeigt, dass Fall 2 nicht vorkommen kann. Folglich sind als Wahrheitswertkombinationen für H und K nur die drei Kombinationen möglich, bei denen »wenn H , dann K « wahr sind.

1.3.4 Gegenbeispiele

Im richtigen Leben müssen wir keine Sätze beweisen. Stattdessen werden wir mit etwas konfrontiert, das wahr zu sein scheint (z. B. eine Strategie zur Implementierung

eines Programms), und müssen entscheiden, ob dieser »Satz« wahr oder falsch ist. Um dieses Problem zu lösen, versuchen wir abwechselnd, den Satz zu beweisen, und falls dies nicht möglich ist, zu beweisen, dass dessen Behauptung falsch ist.

Bei Sätzen handelt es sich im Allgemeinen um Aussagen über eine unendliche Anzahl von Fällen, die möglicherweise alle Werte von Parametern eines Satzes sind. Nach streng mathematischer Konvention wird nur dann eine Aussage als »Satz« (oder Theorem) bezeichnet, wenn sie für eine unendliche Anzahl von Fällen gilt. Aussagen ohne Parameter oder Aussagen, die nur für eine endliche Anzahl von Werten ihrer Parameter gelten, werden **Beobachtungen** genannt. Um zu zeigen, dass ein angeblicher Satz kein Satz ist, genügt es zu zeigen, dass er in einem beliebigen Fall nicht gilt. Dies ist auf Programme übertragbar, da ein Programm im Allgemeinen als fehlerhaft gilt, wenn es mit einer Eingabe nicht korrekt funktioniert, mit der es korrekt arbeiten sollte.

Der Beweis, dass eine Aussage kein Satz ist, ist oft einfacher als der Beweis, dass eine Aussage ein Satz ist. Wenn S eine beliebige Aussage ist, dann ist die Aussage » S ist kein Satz« selbst eine Aussage ohne Parameter und kann daher als Beobachtung und nicht als Satz betrachtet werden. Es folgen zwei Beispiele. Das Erste zeigt eine Aussage, die offensichtlich kein Satz ist, und das Zweite eine Aussage, die ein Satz sein könnte und daher näher untersucht werden muss, bevor die Frage entschieden werden kann, ob es sich um einen Satz handelt.

Angeblicher Satz 1.7 Alle Primzahlen sind ungerade. (Formeller ausgedrückt: Wenn die ganze Zahl x eine Primzahl ist, dann ist x ungerade.)

Gegenbeispiel: Die ganze Zahl 2 ist eine Primzahl, aber 2 ist gerade. ■

Wir wollen nun einen »Satz« zur Modul-Arithmetik diskutieren. Eine grundlegende Definition muss hierzu zuerst eingeführt werden. Wenn a und b positive ganze Zahlen sind, dann ist $a \bmod b$ der Divisionsrest von a geteilt durch b , d. h. eine eindeutige ganze Zahl r im Bereich zwischen 0 und $b - 1$, und zwar derart, dass für eine bestimmte ganze Zahl q gilt: $a = qb + r$. Zum Beispiel: $8 \bmod 3 = 2$ und $9 \bmod 3 = 0$. Der erste angebliche Satz, dessen Falschheit wir beweisen werden, lautet:

Angeblicher Satz 1.8 Es gibt kein Paar von ganzen Zahlen a und b , derart dass

$$a \bmod b = b \bmod a.$$

Wenn man etwas mit Paaren von Objekten machen soll, wie hier a und b , lässt sich die Beziehung zwischen den beiden häufig vereinfachen, indem man die Symmetrie nutzt. In diesem Beispiel können wir uns auf den Fall konzentrieren, in dem $a < b$, da wir a und b vertauschen können, wenn $b < a$, und damit die gleiche Gleichung erhalten wie im angeblichen Satz 1.8. Wir dürfen allerdings nicht den dritten Fall ver-

gessen, in dem $a = b$ und der sich in unseren Beweisversuchen hier als der kritische erweist.

Angenommen $a < b$. Dann ist $a \bmod b = a$, weil in der Definition von $a \bmod b$ festgelegt ist, dass $q = 0$ und $r = a$. Das heißt, wenn $a < b$, gilt also $a = 0 * b + a$. Aber $b \bmod a < a$, weil eine beliebige Zahl $\bmod a$ stets einen Wert zwischen 0 und $a - 1$ hat. Daraus folgt, wenn $a < b$, ist $b \bmod a < a \bmod b$. Infolgedessen ist es unmöglich, dass $a \bmod b = b \bmod a$. Unter Verwendung des oben genannten Arguments der Symmetrie können wir zudem folgern, dass $a \bmod b \neq b \bmod a$, wenn $b < a$.

Betrachten wir jedoch den dritten Fall: $a = b$. Da für jede ganze Zahl x gilt, $x \bmod x = 0$, gibt es einen Fall, in dem gilt: $a \bmod b = b \bmod a$, wenn $a = b$. Damit verfügen wir über einen Gegenbeweis für den angeblichen Satz:

Gegenbeispiel: (des angeblichen Satzes 1.8) Sei $a = b = 2$. Dann gilt

$$a \bmod b = b \bmod a = 0.$$

Während der Suche nach einem Gegenbeispiel haben wir sogar die exakten Bedingungen entdeckt, unter denen der angebliche Satz gilt. Es folgt die korrekte Version des Satzes und dessen Beweis.

Satz 1.9 $a \bmod b = b \bmod a$ genau dann, wenn $a = b$.

Beweis ■ (Wenn-Teil) Angenommen $a = b$. Wie wir oben sehen konnten, gilt für alle ganzen Zahlen x , dass $x \bmod x = 0$. Folglich ist $a \bmod b = b \bmod a$, wenn $a = b$.

(Nur-wenn-Teil) Nun nehmen wir an, dass $a \bmod b = b \bmod a$. Die beste Technik ist in diesem Fall ein Beweis durch Widerspruch, und daher nehmen wir zudem die Negation der Konklusion an, d. h. angenommen, $a \neq b$. Da $a = b$ hiermit eliminiert wurde, müssen wir lediglich die Fälle $a < b$ und $b < a$ betrachten.

Wir haben bereits festgestellt, dass $a \bmod b = a$ und $b \bmod a < a$, wenn $a < b$. Aufgrund dieser Aussagen und der Hypothese $a \bmod b = b \bmod a$ können wir einen Widerspruch ableiten.

Gemäß der Symmetrie gilt, wenn $b < a$, dann $b \bmod a = b$ und $a \bmod b < b$. Wir leiten wieder einen Widerspruch zur Hypothese ab und folgern, dass auch der Genau-dann-Teil wahr ist. Damit haben wir nun beide Richtungen bewiesen und kommen zu dem Schluss, dass der Satz wahr ist. ■

1.4 Induktive Beweise

Es gibt eine spezielle Form von Beweisen, die so genannten »induktiven« Beweise (auch Induktionsbeweise genannt), die beim Umgang mit rekursiv definierten Objekten unabdingbar sind. Viele der bekanntesten induktiven Beweise beinhalten ganze Zahlen, aber in der Automatentheorie lernen wir auch induktive Beweise für

rekursiv definierte Konzepte wie Bäume und unterschiedliche Typen von Ausdrücken kennen, wie die regulären Ausdrücke, die in Abschnitt 1.1.2 kurz gestreift wurden. In diesem Abschnitt werden wir induktive Beweise zuerst anhand »einfacher« Induktion mit ganzen Zahlen vorstellen. Dann zeigen wir, wie »strukturelle« Induktionsbeweise für jedes beliebige rekursiv definierte Konzept geführt werden.

1.4.1 Induktive Beweise mit ganzen Zahlen

Angenommen, wir sollen die Aussage $S(n)$ für eine ganze Zahl n beweisen. Ein üblicher Ansatz besteht im Beweis von zwei Dingen:

1. Induktionsbeginn, wobei zu zeigen ist, dass $S(i)$ für eine bestimmte ganze Zahl i gilt. Gewöhnlich wird $i = 0$ oder $i = 1$ gewählt, aber es gibt Beispiele, in denen mit einem höheren Wert für i begonnen werden sollte, da die Aussage S möglicherweise für einige kleine ganze Zahlen falsch ist.

2. Induktionsschritt, wobei angenommen wird, dass $n \geq i$, wobei i den Wert der Induktionsbasis darstellt, und gezeigt wird, dass »wenn $S(n)$, dann $S(n + 1)$ «.

Diese beiden Teile sollten uns davon überzeugen, dass $S(n)$ für jede ganze Zahl n wahr ist, die größer oder gleich der Basiszahl i ist. Wir können wie folgt argumentieren: Angenommen $S(n)$ wäre für eine oder mehrere dieser ganzen Zahlen falsch. Dann müsste es einen kleinsten Wert von n (z. B. j) geben, für den gilt, dass $S(j)$ falsch ist und gleichzeitig gilt $j \geq i$. Die ganze Zahl j kann nicht gleich i sein, da wir im Basisteil beweisen, dass $S(i)$ wahr ist. Folglich muss j größer als i sein. Daraus folgt, dass $j - 1 \geq i$ und $S(j - 1)$ wahr ist.

Im Induktionsschritt haben wir jedoch bewiesen, dass $S(n)$ impliziert $S(n + 1)$, wenn $n \geq i$. Angenommen, es sei $n = j - 1$. Wir wissen aus dem Induktionsschritt, dass $S(j - 1)$ die Aussage $S(j)$ impliziert. Da wir auch $S(j - 1)$ kennen, können wir auf $S(j)$ schließen.

Wir haben die Negation dessen angenommen, was wir beweisen wollten: Das heißt, wir nahmen an, dass $S(j)$ für eine ganze Zahl $j \geq i$ falsch sei. In beiden Fällen haben wir einen Widerspruch abgeleitet, und damit liegt ein »Beweis durch Widerspruch« dafür vor, dass $S(n)$ für alle $n \geq i$ wahr ist.

Leider weist die obige Schlussfolgerung einen kleinen logischen Fehler auf. Unsere Annahme, dass wir das kleinste $j \geq i$ wählen können, für die $S(j)$ falsch ist, basiert auf unserem Glauben an das Prinzip der Induktion. Das heißt, die einzige Möglichkeit zu beweisen, dass wir eine solche ganze Zahl j finden können, besteht in einer Beweismethode, die im Grunde genommen einen induktiven Beweis darstellt. Der oben erörterte »Beweis« scheint jedoch vernünftig und entspricht unserem Realitätsverständnis. Wir verstehen ihn daher als integralen Bestandteil unseres logischen Schlussfolgerungssystems:

■ *Das Induktionsprinzip:* Wenn wir $S(i)$ beweisen und beweisen können, dass $S(n)$ für alle $n \geq i$ $S(n + 1)$ impliziert, dann können wir darauf schließen, dass $S(n)$ für alle $n \geq i$ gilt.

Die folgenden beiden Beispiele veranschaulichen den Einsatz des Induktionsprinzips zum Beweis von Sätzen, die für ganze Zahlen gelten.

Satz 1.10 Für alle $n \geq 0$:

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad (1.1)$$

Beweis ■ Der Beweis gliedert sich in zwei Teile: den Induktionsbeginn und den Induktionsschritt. Wir beweisen diese Teile nacheinander.

Induktionsbeginn: Für die Basis wählen wir $n = 0$. Es mag überraschen, dass der Satz selbst für $n = 0$ gilt, da die linke Seite der Gleichung (1.1) gleich $\sum_{i=1}^0$ ist, wenn $n = 0$. Es gibt allerdings eine allgemeine Regel, die besagt, dass eine Summe 0 ist, wenn die Obergrenze der Summe (in diesem Fall 0) kleiner als die Untergrenze (hier 1) ist, da die Summe dann mit null Termen gebildet wird. Das heißt: $\sum_{i=1}^0 i^2 = 0$.

Die rechte Seite der Gleichung (1.1) ist ebenfalls 0, da $0 \times (0 + 1) \times (2 \times 0 + 1) / 6 = 0$. Folglich ist die Gleichung (1.1) wahr, wenn $n = 0$.

Induktionsschritt: Wir nehmen nun an, dass $n \geq 0$. Wir müssen den Induktionsschritt beweisen, dass Gleichung (1.1) dieselbe Formel impliziert, wenn n durch $n + 1$ ersetzt wird. Diese Formel lautet:

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \quad (1.2)$$

Wir können die Gleichungen (1.1) und (1.2) vereinfachen, indem wir die Summen und die Produkte auf der rechten Seite erweitern. Die vereinfachten Gleichungen sehen so aus:

$$\sum_{i=1}^n i^2 = (2n^3 + 3n^2 + n) / 6 \quad (1.3)$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6) / 6 \quad (1.4)$$

Wir müssen (1.4) mithilfe von (1.3) beweisen, da diese Gleichungen im Induktionsprinzip den Aussagen $S(n+1)$ bzw. $S(n)$ entsprechen. Der »Trick« besteht darin, die Summe bis $n + 1$ auf der rechten Seite von (1.4) in eine Summe bis n plus den $(n + 1)$ -ten Term aufzuspalten. Auf diese Weise können wir die Summe bis n durch die linke Seite von (1.3) ersetzen und zeigen, dass (1.4) wahr ist. Diese Schritte lauten folgendermaßen:

$$\left(\sum_{i=1}^n i^2 \right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6) / 6 \quad (1.5)$$

$$(2n^3 + 3n^2 + n) / 6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6) / 6 \quad (1.6)$$

Zur endgültigen Verifizierung der Wahrheit von (1.6) ist nur einfache polynome Algebra auf der linken Seite erforderlich, um zu zeigen, dass die linke Seite mit der rechten identisch ist. ■

Beispiel 1.7 Im nächsten Beispiel beweisen wir Satz 1.1 aus Abschnitt 1.2.1. Dieser Satz besagt: Wenn $x \geq 4$, dann $2^x \geq x^2$. Wir haben einen informalen Beweis beschrieben, der auf der Vorstellung basierte, dass das Verhältnis $x^2/2^x$ schrumpft, wenn x über den Wert 4 hinaus wächst. Wir können diese Vorstellung präzisieren, wenn wir die Aussage $2^x \geq x^2$ durch eine vollständige Induktion über x beweisen und mit der Basis $x = 4$ beginnen. Beachten Sie, dass die Aussage für Werte $x < 4$ tatsächlich falsch ist.*

Induktionsbeginn: Wenn $x = 4$, dann ergibt sowohl 2^x als auch x^2 den Wert 16. Folglich gilt $2^x \geq x^2$.

Induktionsschritt: Angenommen, für ein beliebiges $x \geq 4$ gilt $2^x \geq x^2$. Mit dieser Aussage als Hypothese müssen wir beweisen, dass dieselbe Aussage mit $x + 1$ statt x wahr ist, d. h. $2^{[x+1]} \geq [x + 1]^2$. Dies entspricht den Aussagen $S(x)$ und $S(x + 1)$ im Induktionsprinzip. Die Tatsache, dass wir x statt n als Parameter verwenden, sollte nicht von Belang sein; x und n sind nur lokale Variablen.

Wie beim Satz 1.10 sollten wir $S(x + 1)$ umformulieren, sodass wir $S(x)$ einsetzen können. In diesem Fall können wir $2^{[x+1]}$ durch 2×2^x ersetzen. Da $S(x)$ besagt, dass $2^x \geq x^2$, können wir schließen, dass $2^{[x+1]} = 2 \times 2^x \geq 2x^2$.

Wir brauchen aber etwas anderes. Wir müssen zeigen, dass $2^{[x+1]} \geq (x + 1)^2$. Eine Möglichkeit, diese Aussage zu beweisen, besteht darin, $2^x \geq (x + 1)^2$ zu beweisen und mithilfe der Transitivität von \geq zu zeigen, dass $2^{[x+1]} \geq 2x^2 \geq (x + 1)^2$. In unserem Beweis für

$$2x^2 \geq (x + 1)^2 \quad (1.7)$$

können wir annehmen, dass $x \geq 4$. Wir beginnen mit der Vereinfachung von (1.7):

$$x^2 \geq 2x + 1 \quad (1.8)$$

Wir dividieren (1.8) durch x und erhalten damit:

$$x \geq 2 + \frac{1}{x} \quad (1.9)$$

Da $x \geq 4$, wissen wir dass $1/x \leq 1/4$. Folglich ist die linke Seite von (1.9) größer oder gleich 4 und die rechte Seite höchstens 2,25. Wir haben damit die Wahrheit von (1.9) bewiesen. Folglich sind auch die Gleichungen (1.8) und (1.7) wahr. Die Gleichung (1.7) ergibt wiederum $2x^2 \geq (x + 1)^2$ für $x \geq 4$ und ermöglicht den Beweis der Aussage $S(x + 1)$, die $2^{x+1} \geq [x + 1]^2$ lautete. ■

* Diese Stellungnahme ist nicht sauber; besser: Für $x := 3$ wäre die Aussage tatsächlich falsch. Dass für $0 \leq x \leq 2$ die Aussage wieder stimmt, ändert nichts an der minimalen Basis 4 =: x_0 , da die Aussage für alle $x \geq x_0$ gelten muss.

Ganze Zahlen als rekursiv definierte Konzepte

Wir erwähnten, dass induktive Beweise nützlich sind, wenn der Beweisgegenstand rekursiv definiert ist. Unsere ersten Beispiele waren allerdings Induktionen mit ganzen Zahlen, die wir normalerweise nicht für »rekursiv definiert« halten. Es gibt jedoch eine natürliche rekursive Definition einer nichtnegativen ganzen Zahl, und diese Definition entspricht in der Tat der Art und Weise, in der Induktionsbeweise mit ganze Zahlen erbracht werden: von den Objekten, die zuerst definiert wurden, zu den nachfolgend definierten Objekten.

Induktionsbeginn: 0 ist eine ganze Zahl (Ganzzahl, »Integer«).

Induktionsschritt: Wenn n eine Ganzzahl ist, dann ist es auch $n + 1$.

1.4.2 Allgemeinere Formen der Induktion mit ganzen Zahlen

Gelegentlich wird ein induktiver Beweis nur durch den Einsatz eines allgemeineren Schemas ermöglicht, als dem in Abschnitt 1.4.1 vorgeschlagenen, wo wir eine Aussage S für einen Basiswert bewiesen haben und dann bewiesen haben, dass »wenn $S(n)$, dann $S(n + 1)$ «. Zwei wichtige Verallgemeinerungen dieses Schemas sind:

- 1.** Wir können mit mehreren Basisfällen arbeiten. Das heißt, wir beweisen $S(i)$, $S(i + 1)$, ..., $S(j)$ für ein beliebiges $j > i$.
- 2.** Zum Beweis von $S(n + 1)$ können wir die Wahrheit aller Aussagen

$$S(i), S(i + 1), \dots, S(n)$$

statt lediglich $S(n)$ einsetzen. Überdies können wir nach dem Beweis der Basisfälle bis $S(j)$ annehmen, dass $n \geq j$ gilt statt einfach $n \geq i$.

Aus diesem Induktionsbeginn und dem Induktionsschritt ist die Schlussfolgerung zu ziehen, dass $S(n)$ wahr ist für alle $n \geq i$.

Beispiel 1.8 Das folgende Beispiel soll das Potenzial beider Prinzipien illustrieren. Die Aussage $S(n)$, die es zu beweisen gilt, lautet: Wenn $n \geq 8$, dann kann n als Summe eines Vielfachen von 3 und eines Vielfachen von 5 ausgedrückt werden. Beachten Sie, dass 7 nicht so dargestellt werden kann.

Induktionsbeginn: Als Basisfälle wurden $S(8)$, $S(9)$ und $S(10)$ gewählt. Die Beweise lauten $8 = 3 + 5$, $9 = 3 + 3 + 3$ und $10 = 5 + 5$.

Induktionsschritt: Angenommen, $n \geq 10$ und $S(8)$, $S(9)$, ..., $S(n)$ seien wahr. Wir müssen $S(n + 1)$ anhand dieser gegebenen Fakten beweisen. Unsere Strategie besteht darin, 3 von $n + 1$ zu subtrahieren, dann festzustellen, dass sich diese Zahl als Summe eines Vielfachen von 3 und eines Vielfachen von 5 ausdrücken lässt, und zu der Summe 3 addieren, um $n + 1$ angeben zu können.

Formaler ausgedrückt, stellen wir fest, dass $n - 2 \geq 8$, und daher können wir annehmen, dass $S(n - 2)$. Das heißt, $n - 2 = 3a + 5b$ für zwei ganze Zahlen a und b . Daraus folgt $n + 1 = 3 + 3a + 5b$, und somit lässt sich $n + 1$ als Summe von $3(a + 1)$ und $5b$ ausdrücken. Dies beweist, dass $S(n + 1)$ und beschließt den Induktionsschritt. ■

1.4.3 Strukturelle Induktion

In der Automatentheorie gibt es verschiedene rekursiv definierte Strukturen, über die Aussagen bewiesen werden müssen. Zu den wichtigsten Beispielen gehören die bekannten Bäume und Ausdrücke. Wie Induktionsbeweise verfügen rekursive Definitionen über einen Basisfall, in dem eine oder mehrere grundlegende Strukturen definiert werden, und einen Induktionsschritt, in dem komplexere Strukturen unter Verwendung zuvor definierter Strukturen definiert werden.

Beispiel 1.9 Es folgt die rekursive Definition eines Baums:

Induktionsbeginn: Ein einzelner Knoten ist ein Baum, und dieser Knoten ist die **Wurzel** des Baums.

Induktionsschritt: Wenn T_1, T_2, \dots, T_k Bäume sind, dann kann wie folgt ein neuer Baum gebildet werden:

1. Man beginnt mit einem neuen Knoten N , der die Wurzel des Baums darstellt.
2. Dann werden Kopien aller Bäume T_1, T_2, \dots, T_k hinzugefügt.
3. Anschließend fügt man Kanten vom Knoten N zu den Wurzeln der einzelnen Bäume T_1, T_2, \dots, T_k hinzu.

Abbildung 1.3 zeigt den induktiven Aufbau eines Baums mit der Wurzel N aus k kleineren Bäumen. ■

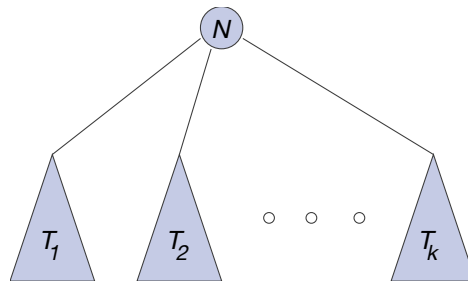


Abbildung 1.3: Induktiver Aufbau eines Baums

Beispiel 1.10 Es folgt eine weitere rekursive Definition. Diesmal definieren wir Ausdrücke mithilfe der arithmetischen Operatoren $+$ und $*$ sowie Zahlen und Variablen als Operanden.

Induktionsbeginn: Jede Zahl oder jeder Buchstabe (also eine Variable) ist ein Ausdruck.

Induktionsschritt: Wenn E und F Ausdrücke sind, dann seien auch $E + F$, $E * F$ und (E) Ausdrücke.

Beispielsweise sind gemäß der Induktionsbasis sowohl 2 als auch x Ausdrücke. Der Induktionsschritt besagt, dass $x + 2$, $(x + 2)$ und $2 * (x + 2)$ ebenfalls Ausdrücke sind. Beachten Sie, dass jeder dieser Ausdrücke von den zuvor definierten Ausdrücken abhängt. ■

Der der strukturellen Induktion zu Grunde liegende Gedankengang

Wir können informell erklären, warum die strukturelle Induktion eine gültige Beweismethode ist. Stellen Sie sich vor, wie durch aufeinander folgende rekursive Definitionen nacheinander festgelegt wird, dass bestimmte Strukturen X_1, X_2, \dots der Definition genügen. Die Grundelemente kommen zuerst, und die Tatsache, dass X_i in der Menge der definierten Strukturen enthalten ist, kann nur von der Mengenzugehörigkeit der definierten Menge von Strukturen abhängen, die X_i in der Liste vorangehen. So gesehen ist eine strukturelle Induktion nichts anderes als ein Induktionsschluss für die ganze Zahl n der Aussage $S(X_n)$. Dieser Induktionsschluss kann die allgemeine Form haben, die in Abschnitt 1.4.2 erörtert wird, und über mehrere Basisfälle und einen Induktionsschritt verfügen, in dem alle vorherigen Instanzen der Aussage zum Einsatz kommen. Wir dürfen allerdings nicht vergessen, dass dieser Gedankengang, wie in Abschnitt 1.4.1 erläutert, kein formaler Beweis ist und wir die Gültigkeit dieses Induktionsprinzips ebenso unterstellen müssen wie die des Induktionsprinzips jenes Abschnitts.

Wenn eine rekursive Definition vorliegt, können wir mithilfe der folgenden Beweismethode, die als *strukturelle Induktion* bezeichnet wird, Sätze über diese Definition beweisen. $S(X)$ sei eine Aussage über die Strukturen X , die durch eine bestimmte rekursive Definition erklärt werden.

- 1.** Als Basis wird $S(X)$ für die Basisstruktur(en) X bewiesen.
- 2.** Im Induktionsschritt wird eine Struktur X gewählt, die gemäß der rekursiven Definition aus Y_1, Y_2, \dots, Y_k gebildet wird. Wir nehmen an, die Aussagen $S(Y_1), S(Y_2), \dots, S(Y_k)$ seien wahr, und verwenden sie zum Beweis von $S(X)$.

Wir kommen zu dem Schluss, dass $S(X)$ für alle X wahr ist. Die nächsten beiden Sätze sind Beispiele für Fakten über Bäume und Ausdrücke, die sich beweisen lassen.

Satz 1.11 Jeder Baum besitzt genau einen Knoten mehr als Kanten.

Beweis ■ Die formale Aussage $S(T)$, die durch strukturelle Induktion bewiesen werden muss, lautet: »Wenn T ein Baum ist und n Knoten und e Kanten besitzt, dann gilt $n = e + 1$.«

Induktionsbeginn: Der Basisfall ist gegeben, wenn T aus nur einem Knoten besteht. Dann ist $n = 1$ und $e = 0$, und somit gilt die Relation $n = e + 1$.

Induktionsschritt: T sei ein Baum, der durch den Induktionsschritt der Definition aus dem Wurzelknoten N und k kleineren Bäumen T_1, T_2, \dots, T_k geformt wird. Wir können annehmen, dass die Aussagen $S(T_i)$ für $i = 1, 2, \dots, k$ gelten. Das heißt, wenn T_i n_i Knoten und e_i Kanten besitzt, dann gilt $n_i = e_i + 1$.

Bei den Knoten von T handelt es sich um den Knoten N sowie die Knoten der Bäume T_i . T besitzt folglich $1 + n_1 + n_2 + \dots + n_k$ Knoten. Die Kanten von T umfassen die k

Kanten, die wir explizit im Induktionsschritt der Definition hinzugefügt haben, plus die Kanten der Bäume T_i . T besitzt folglich

$$k + e_1 + e_2 + \dots + e_k \quad (1.10)$$

Kanten. Wenn wir in der Knotenanzahl von T n_i durch $e_i + 1$ ersetzen, dann ergibt sich, dass T

$$1 + [e_1 + 1] + [e_2 + 1] + \dots + [e_k + 1] \quad (1.11)$$

Knoten besitzt. (1.11) lässt sich unmittelbar wie folgt umformulieren:

$$k + 1 + e_1 + e_2 + \dots + e_k \quad (1.12)$$

Dieser Ausdruck ist genau um 1 größer als der Ausdruck in (1.10), der die Anzahl der Kanten von T bezeichnete. Folglich besitzt T genau eine um 1 höhere Anzahl von Knoten als Kanten. ■

Satz 1.12 Jeder Ausdruck enthält die gleiche Anzahl von linken und rechten Klammern.

Beweis ■ Formal beweisen wir die Aussage $S(G)$ für jeden Ausdruck G , der durch die rekursive Definition aus Beispiel 1.10 definiert wird: G enthält die gleiche Anzahl von linken und rechten Klammern.

Induktionsbeginn: Wenn G durch die Basis definiert wird, dann ist G eine Zahl oder Variable. Diese Ausdrücke enthalten 0 linke Klammern und 0 rechte Klammern. Folglich ist die Anzahl der Klammern gleich.

Induktionsschritt: Es gibt drei Regeln, nach denen der Ausdruck G gemäß dem Induktionsschritt der Definition gebildet worden sein kann:

1. $G = E + F$

2. $G = E * F$

3. $G = (E)$

Wir können annehmen, dass $S(E)$ und $S(F)$ wahr sind. Das heißt, E enthält dieselbe Anzahl von rechten und linken Klammern (sagen wir, jeweils n rechte und linke Klammern), und analog enthält F auch dieselbe Anzahl von rechten und linken Klammern (sagen wir, jeweils m rechte und linke Klammern). Wie folgt können wir dann die Anzahl der rechten und linken Klammern von G für jeden der drei Fälle berechnen:

1. Wenn $G = E + F$, dann enthält G $n+m$ linke und $n+m$ rechte Klammern, wobei jeweils n von E und m von F stammen.

- 2.** Wenn $G = E * F$, dann enthält G aus denselben Gründen wie im Fall (1) wieder jeweils $n+m$ linke und rechte Klammern.
- 3.** Wenn $G = (E)$, dann enthält G $n+1$ linke Klammern (eine linke Klammer ist explizit gegeben und die anderen n sind in E enthalten). Ebenso weist G auch $n+1$ rechte Klammern auf; eine ist explizit gegeben und die anderen sind in E enthalten.

In jedem der drei Fälle ergibt sich, dass in G dieselbe Anzahl von linken und rechten Klammern enthalten ist. Diese Beobachtung vervollständigt den Induktionsschritt und den Beweis. ■

1.4.4 Gegenseitige Induktion

Gelegentlich können wir nicht eine einzelne Aussage durch Induktion beweisen, sondern müssen eine Gruppe von Aussagen $S_1(n)$, $S_2(n)$, ..., $S_k(n)$ zusammen durch Induktion über n beweisen. In der Automatentheorie gibt es viele solcher Situationen. Mit Beispiel 1.11 geben wir ein Beispiel für eine häufig auftretende Situation, nämlich dass wir die Arbeitsweise eines Automaten erklären müssen, indem wir eine Gruppe von Aussagen beweisen, die jeweils einen Zustand beschreiben. Diese Aussagen besagen, mit welchen Eingabefolgen der Automat in die verschiedenen Zustände versetzt wird.

Streng genommen unterscheidet sich der Beweis einer Gruppe von Aussagen nicht vom Beweis der **Konjunktion** (logisches UND) aller Aussagen. Die Gruppe der Aussagen $S_1(n)$, $S_2(n)$, ..., $S_k(n)$ könnte beispielsweise durch die Einzelaussage $S_1(n)$ UND $S_2(n)$ UND ... UND $S_k(n)$ ersetzt werden. Gilt es allerdings, mehrere tatsächlich verschiedene Aussagen zu beweisen, ist es im Allgemeinen klarer, wenn man die Aussagen voneinander getrennt lässt und sie alle in eigenen Teilen der Basis und der Induktionsschritte beweist. Wir nennen diese Art von Beweis einen *gegenseitigen Induktionsbeweis*. Ein Beispiel soll die für einen gegenseitigen Induktionsbeweis notwendigen Schritte veranschaulichen.

Beispiel 1.11 Betrachten wir noch einmal den Ein-/Aus-Schalter, der in Beispiel 1.1 als Automat beschrieben wurde. Der Automat selbst wird in Abbildung 1.4 dargestellt. Da durch Drücken des Schalters der Zustand zwischen *Ein* und *Aus* hin- und hergeschaltet wird und der Schalter sich anfänglich im Zustand *Aus* befindet, erwarten wir, dass die folgenden Aussagen zusammengenommen die Arbeitsweise des Schalters erklären:

$S_1(n)$: Der Automat befindet sich nach n -maligem Drücken genau dann im Zustand *Aus*, wenn n gerade ist.

$S_2(n)$: Der Automat befindet sich nach n -maligem Drücken genau dann im Zustand *Ein*, wenn n ungerade ist.

Wir können unterstellen, dass S_1 S_2 impliziert und umgekehrt, da wir wissen, dass die Zahl n nicht gleichzeitig gerade und ungerade sein kann. Allerdings gilt für einen Automaten nicht immer, dass er sich stets in genau einem Zustand befindet. Zufällig

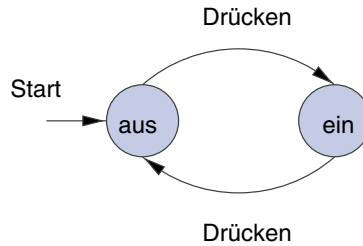


Abbildung 1.4: Nochmals der Automat aus Abbildung 1.1

befindet sich der in Abbildung 1.4 dargestellte Automat stets in genau einem Zustand, aber diese Tatsache muss im Rahmen der gegenseitigen Induktion bewiesen werden.

Wir stellen die Basis und die Induktionsschritte der Beweise für die Aussagen $S_1(n)$ und $S_2(n)$ nachfolgend dar. Die Beweise hängen von verschiedenen Fakten zu ungeraden und geraden ganzen Zahlen ab: Wenn wir 1 von einer geraden ganzen Zahl subtrahieren oder dazuaddieren, erhalten wir eine ungerade ganze Zahl, und wenn wir 1 von einer ungeraden ganzen Zahl 1 subtrahieren oder dazuaddieren, erhalten wir eine gerade ganze Zahl.

Induktionsbeginn: Als Basis wählen wir $n = 0$. Da zwei Aussagen vorliegen, die jeweils in beiden Richtungen zu beweisen sind (weil S_1 und S_2 Aussagen vom Typ »genau dann, wenn« sind), umfasst die Basis vier Fälle, und der Induktionsschritt umfasst ebenfalls vier Fälle.

- 1.** [S_1 ; Wenn]: Da 0 eine gerade ganze Zahl ist, müssen wir zeigen, dass sich der in Abbildung 1.4 gezeigte Automat nach 0-maligem Drücken im Zustand *Aus* befindet. Da dies der Anfangszustand ist, befindet sich der Automat nach 0-maligem Drücken in der Tat im Zustand *Aus*.
- 2.** [S_1 ; Nur-wenn]: Der Automat befindet sich nach 0-maligem Drücken im Zustand *Aus*, und daher müssen wir zeigen, dass 0 gerade ist. 0 ist aber gemäß der Definition von »gerade« eine gerade ganze Zahl, und daher muss nichts bewiesen werden.
- 3.** [S_2 ; Wenn]: Die Hypothese des Wenn-Teils von S_2 besteht darin, dass 0 eine ungerade ganze Zahl ist. Da diese Hypothese H falsch ist, ist jede Aussage der Form »wenn H , dann K « wahr, wie in Abschnitt 1.3.2 erörtert. Dieser Teil der Basis gilt daher.
- 4.** [S_2 ; Nur-wenn]: Die Hypothese, dass sich der Automat nach 0-maligem Drücken im Zustand *Ein* befindet, ist auch falsch, da der Automat nur durch mindestens einmaliges Drücken in den Zustand *Ein* versetzt werden kann. Da die Hypothese falsch ist, können wir auch hier darauf schließen, dass die Nur-wenn-Aussage wahr ist.

Induktionsschritt: Wir nehmen nun an, dass $S_1(n)$ und $S_2(n)$ wahr sind, und versuchen, $S_1(n+1)$ und $S_2(n+1)$ zu beweisen. Der Beweis gliedert sich wieder in vier Teile.

1. [$S_1(n+1)$; Wenn]: Die Hypothese dieses Teils besteht darin, dass $n+1$ gerade ist. Folglich ist n ungerade. Der Wenn-Teil der Aussage $S_2(n)$ besagt, dass sich der Automat nach n -maligem Drücken im Zustand *Ein* befindet. Der mit *Drücken* beschriftete Bogen von *Ein* nach *Aus* zeigt an, dass das $(n+1)$ -malige Drücken den Automaten in den Zustand *Aus* versetzt. Damit ist der Beweis für den Wenn-Teil von S_1 vollständig.
2. [$S_1(n+1)$; Nur-wenn]: Die Hypothese besteht darin, dass sich der Automat nach $(n+1)$ -maligem Drücken im Zustand *Aus* befindet. Wenn wir den Automaten in Abbildung 1.4. betrachten, wird deutlich, dass die einzige Möglichkeit, den Automaten mit wenigstens einer Aktion in den Zustand *Aus* zu versetzen, darin besteht, dass im Zustand *Ein* die Eingabe *Drücken* erfolgt. Wenn sich der Automat nach $(n+1)$ -maligem Drücken im Zustand *Aus* befindet, dann muss er sich nach n -maligem Drücken im Zustand *Ein* befunden haben. Wir können dann unter Verwendung des Nur-wenn-Teils von $S_2(n)$ darauf schließen, dass n ungerade ist. Folglich ist $(n+1)$ gerade. Dies ist die gewünschte Schlussfolgerung für den Nur-wenn-Teil von $S_1(n+1)$.
3. [$S_2(n+1)$; Wenn]: Dieser Teil entspricht im Grunde genommen Teil (1), wenn man die Rollen der Aussagen S_1 und S_2 sowie »gerade« und »ungerade« vertauscht. Der Leser sollte diesen Teil des Beweises selbst formulieren können.
4. [$S_1(n+1)$; Nur-wenn]: Dieser Teil entspricht im Grunde genommen Teil (2), wenn man die Rollen der Aussagen S_1 und S_2 sowie »gerade« und »ungerade« vertauscht. ■

Wir können Beispiel 1.11 das allgemeine Muster gegenseitiger Induktionen entnehmen:

- Jede Aussage muss im Induktionsbeginn und im Induktionsschritt getrennt bewiesen werden.
- Wenn es sich um Aussagen des Typs »genau dann, wenn« handelt, dann müssen sowohl im Induktionsbeginn als auch im Induktionsschritt beide Richtungen jeder Anweisung bewiesen werden.

1.5 Die zentralen Konzepte der Automatentheorie

In diesem Abschnitt stellen wir die wichtigsten Begriffsdefinitionen vor, die in der Automatentheorie von zentraler Bedeutung sind. Zu diesen Konzepten gehören »Alphabet« (eine Menge von Symbolen), »Zeichenreihe« (eine Liste aus Symbolen eines Alphabets) und »Sprache« (eine Menge von Zeichenreihen eines Alphabets).

1.5.1 Alphabete

Ein *Alphabet* ist eine endliche, nicht leere Menge von Symbolen. Gemäß Konvention wird ein Alphabet durch das Symbol Σ dargestellt. Zu den gängigen Alphabeten gehören:

1. $\Sigma = \{0, 1\}$, das **binäre Alphabet**
2. $\Sigma = \{a, b, \dots, z\}$, die Menge aller Kleinbuchstaben
3. Die Menge der ASCII-Zeichen oder die Menge aller druckbaren ASCII-Zeichen

1.5.2 Zeichenreihen

Eine **Zeichenreihe** (manchmal auch **Wort** oder **String** genannt) ist eine endliche Folge von Symbolen eines bestimmten Alphabets. Beispielsweise ist 01101 eine Zeichenreihe über dem binären Alphabet $\Sigma = \{0, 1\}$. Die Zeichenreihe 111 ist ein weiteres Beispiel für eine Zeichenreihe über diesem Alphabet.

Die leere Zeichenreihe

Die **leere Zeichenreihe** ist eine Zeichenreihe, die keine Symbole enthält. Diese Zeichenreihe wird durch das Symbol ε dargestellt und kann aus jedem beliebigen Alphabet stammen.

Länge einer Zeichenreihe

Es ist häufig hilfreich, **Zeichenreihen** nach ihrer **Länge** zu klassifizieren, d. h. der Anzahl der für Symbole verfügbaren Positionen. Beispielsweise hat die Zeichenreihe 01101 die Länge 5. Es ist üblich, die Länge einer Zeichenreihe als »Anzahl der Symbole« der Zeichenreihe zu definieren; umgangssprachlich ist diese Aussage akzeptabel, streng genommen ist sie aber nicht korrekt. So gibt es nur zwei Symbole, 0 und 1, in der Zeichenreihe 01101, aber fünf *Positionen* für Symbole, und daher hat sie die Länge 5. Sie können jedoch im Allgemeinen davon ausgehen, dass mit dem Ausdruck »die Anzahl von Symbolen« tatsächlich »die **Anzahl von Positionen**« gemeint ist.

Die Standard-Notation für die Länge einer Zeichenreihe w lautet $|w|$. So ist z. B. $|011| = 3$ und $|\varepsilon| = 0$.

Potenzen eines Alphabets

Wenn Σ ein Alphabet ist, können wir die Menge aller Zeichenreihen einer bestimmten Länge über Σ durch eine Potenznotation bezeichnen. Wir definieren Σ^k als die Menge der Zeichenreihen mit der Länge k , deren Symbole aus dem Alphabet Σ stammen.

Beispiel 1.12 Beachten Sie, dass $\Sigma^0 = \{\varepsilon\}$, ungeachtet dessen, welches Alphabet Σ bezeichnet. Das heißt, ε ist die einzige Zeichenreihe mit der Länge 0.

Wenn $\Sigma = \{0, 1\}$, dann $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$,

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

usw. Beachten Sie den Unterschied zwischen Σ und Σ^1 . Σ ist ein Alphabet mit den Symbolen 0 und 1 als Elementen. Σ^1 ist eine Menge von Zeichenreihen und enthält als Elemente die Zeichenreihen 0 und 1, die jeweils die Länge 1 haben. Wir werden hier nicht versuchen, diese beiden Mengen durch eine spezielle Notation zu unterscheiden, da aus dem Kontext hervorgeht, ob mit $\{0, 1\}$ oder ähnlichen Mengen Alphabete oder Mengen von Zeichenreihen gemeint sind. ■

Typkonvention für Symbole und Zeichenreihen

Im Allgemeinen werden wir für Symbole Kleinbuchstaben vom Beginn des Alphabets (oder Ziffern) und für Zeichenreihen Kleinbuchstaben vom Ende des Alphabets, typischerweise w, x, y, z verwenden. Sie sollten versuchen, sich diese Konvention einzuprägen, um besser erkennen zu können, von welchem Typ die jeweils diskutierten Elemente sind.

Die Menge aller Zeichenreihen über einem Alphabet Σ wird gemäß Konvention durch Σ^* bezeichnet, zum Beispiel $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$. Anders ausgedrückt:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$$

Gelegentlich ist es wünschenswert, die leere Zeichenreihe aus der Menge der Zeichenreihen auszuschließen. Die Menge der nicht leeren Zeichenreihen des Alphabets Σ wird mit Σ^+ angegeben. Dementsprechend haben wir die zwei Äquivalenzen

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

$$\blacksquare \Sigma^* = \Sigma^+ \cup \{\varepsilon\}$$

Konkatenation von Zeichenreihen

Angenommen, x und y seien Zeichenreihen. Dann steht xy für die **Konkatenation** von x und y , d. h. die Zeichenreihe, die sich durch die Aneinanderreihung von x und y ergibt. Genauer gesagt, wenn x eine aus i Symbolen bestehende Zeichenreihe $x = a_1a_2 \dots a_i$ und y eine aus j Symbolen bestehende Zeichenreihe $y = b_1b_2 \dots b_j$ ist, dann ist xy die Zeichenreihe der Länge $i+j$: $xy = a_1a_2 \dots a_ib_1b_2 \dots b_j$.

Beispiel 1.13 Seien $x = 01101$ und $y = 110$. Dann ist $xy = 01101110$ und $yx = 11001101$. Für eine beliebige Zeichenreihe w gilt die Gleichung $\varepsilon w = w\varepsilon = w$. Das heißt, ε ist die Identität für die Konkatenation, da die Konkatenation der leeren Zeichenreihe mit einer beliebigen Zeichenreihe die letztere Zeichenreihe ergibt (vergleichbar mit der Addition von 0, der Identität für die Addition, zu einer beliebigen Zahl x , die als Ergebnis x liefert). ■

1.5.3 Sprachen

Eine Menge von Zeichenreihen aus Σ^* , wobei Σ ein bestimmtes Alphabet darstellt, wird als **Sprache** bezeichnet. Wenn Σ ein Alphabet ist und $L \subseteq \Sigma^*$, dann ist L eine Sprache über dem Alphabet Σ . Beachten Sie, dass in den Zeichenreihen einer Sprache über Σ nicht alle Symbole aus Σ vorkommen müssen. Wenn also erklärt wurde, dass L eine Sprache über Σ ist, wissen wir, dass L gleichzeitig eine Sprache über jedem Alphabet ist, das eine Obermenge von Σ darstellt.

Die Wahl des Begriffs »Sprache« mag seltsam erscheinen. Die üblichen Sprachen können jedoch als Mengen von Zeichenreihen betrachtet werden. Als Beispiel lässt sich Englisch anführen, wobei die Sammlung zulässiger englischer Wörter eine Menge von Zeichenreihen über dem normalen lateinischen Alphabet ist. Ein anderes Beispiel ist C oder jede andere Programmiersprache, in der zulässige Programme eine Teilmenge der möglichen Zeichenreihen darstellen, die aus dem Alphabet der Sprache gebildet werden können. Dieses Alphabet ist eine Teilmenge des ASCII-Zeichensatzes. Das Alphabet verschiedener Programmiersprachen kann sich im Einzelnen leicht unterscheiden, aber im Allgemeinen umfasst es Groß- und Kleinbuchstaben, Ziffern, Satzzeichen und mathematische Symbole.

Es gibt allerdings zudem viele andere Sprachen, die beim Studium der Automaten in Erscheinung treten. Bei einigen handelt es sich um abstrakte Beispiele wie:

- 1.** Die Sprache aller Zeichenreihen, die aus n Nullen gefolgt von n Einsen bestehen, wobei $n \geq 0$: $\{\epsilon, 01, 0011, 000111, \dots\}$.
- 2.** Die Menge von Zeichenreihen aus Nullen und Einsen, die jeweils die gleiche Anzahl von Nullen und Einsen enthalten:
 $\{\epsilon, 01, 10, 0011, 0101, 1001, \dots\}$
- 3.** Die Menge der Binärzahlen, deren Wert eine Primzahl ist:
 $\{10, 11, 101, 111, 1011, \dots\}$
- 4.** Σ^* ist eine Sprache für jedes beliebige Alphabet Σ .
- 5.** \emptyset , die leere Sprache, ist eine Sprache über einem beliebigen Alphabet.
- 6.** $\{\epsilon\}$, die Sprache, die lediglich aus der leeren Zeichenreihe besteht, ist auch eine Sprache über einem beliebigen Alphabet. Beachten Sie, dass $\emptyset \neq \{\epsilon\}$. Die leere Sprache \emptyset enthält keine Zeichenreihen, $\{\epsilon\}$ enthält dagegen die leere Zeichenreihe.

Die einzige wichtige Einschränkung hinsichtlich der Bildung von Sprachen besteht in der Endlichkeit aller Alphabete. Sprachen können zwar eine unendliche Anzahl von Zeichenreihen enthalten, sie sind aber auf Zeichenreihen über einem festen, endlichen Alphabet beschränkt.

1.5.4 Probleme

In der Automatentheorie besteht ein *Problem* in der Frage, ob eine gegebene Zeichenreihe Element einer bestimmten Sprache ist. Wie wir sehen werden, stellt sich heraus, dass alles, was wir umgangssprachlich als »**Problem**« bezeichnen, als die Frage formuliert werden kann, ob eine bestimmte Zeichenreihe in einer gegebenen Sprache enthalten ist. Genauer gesagt, wenn Σ ein Alphabet ist und L eine Sprache über Σ , dann lautet das Problem L :

- Gegeben sei eine Zeichenreihe w in Σ^* . Entscheiden Sie, ob w in L enthalten ist oder nicht.

Beispiel 1.14 Das Problem des Testens, ob eine Zahl eine Primzahl ist, kann durch die Sprache L_p ausgedrückt werden, die aus allen binären Zeichenreihen besteht, deren Wert als Binärzahl eine Primzahl darstellt. Das heißt, wenn eine aus Nullen und Einsen bestehende Zeichenreihe gegeben ist, gilt es zu entscheiden, ob die Zeichenreihe die Binärdarstellung einer Primzahl ist. Bei einigen Zeichenreihen ist diese Entscheidung einfach. Beispielsweise kann 0011101 einfach aus dem Grund nicht die Repräsentation einer Primzahl sein, weil die Binärdarstellung jeder ganzen Zahl mit Ausnahme von 0 mit einer 1 beginnt. Es ist allerdings weniger offensichtlich, dass 11101 Element von L_p ist. Daher erfordert jede Lösung für dieses Problem eine beträchtliche Menge an Rechenressourcen irgendwelcher Art, wie beispielsweise Zeit und/oder Speicherplatz. ■

Mengenvorschrift als Möglichkeit zur Definition von Sprachen

Es ist allgemein üblich, eine Sprache mithilfe einer Mengenvorschrift zu beschreiben:

$\{w \mid \text{Aussage über } w\}$

Dieser Ausdruck wird wie folgt gelesen: »Die Menge der Wörter w , derart dass (Aussage über w , die auf der rechten Seite des senkrechten Strichs steht)«. Beispiele hierfür sind:

1. $\{w \mid w \text{ enthält eine gleiche Anzahl von Nullen und Einsen}\}.$
2. $\{w \mid w \text{ ist eine binäre ganze Zahl, die eine Primzahl ist}\}.$
3. $\{w \mid w \text{ ist ein syntaktisch fehlerfreies C-Programm}\}.$

Es ist auch üblich, w durch einen Ausdruck mit Parametern zu ersetzen und die Zeichenreihe der Sprache durch für die Parameter geltende Bedingungen zu beschreiben. Es folgen einige Beispiele; das erste Beispiel mit dem Parameter n und das zweite mit den Parametern i und j :

1. $\{0^n 1^n \mid n \geq 1\}.$ Dieser Ausdruck wird gelesen: »Die Menge von 0 hoch n 1 hoch n , derart dass n größer oder gleich 1 ist.« Diese Sprache besteht aus den Zeichenreihen $\{01, 0011, 000111, \dots\}$. Beachten Sie, dass wir ebenso wie bei einem Alphabet ein Symbol hoch n angeben können, um n Exemplare dieses Symbols darzustellen.
2. $\{0^i 1^j \mid 0 \leq i \leq j\}.$ Diese Sprache besteht aus Zeichenreihen, die einige (oder keine) Nullen gefolgt von mindestens ebenso vielen Einsen enthalten.

Ein potenziell unbefriedigender Aspekt unserer Definition des Begriffs »Problem« besteht darin, dass man sich Probleme gemeinhin nicht als Entscheidungsfragen (ist das Folgende wahr oder falsch?) vorstellt, sondern als Aufforderungen, eine Eingabe zu berechnen oder zu transformieren (wie lässt sich diese Aufgabe am besten lösen?). Die Aufgabe des Parsers eines C-Compilers lässt sich beispielsweise in unserem formalen Sinn als Problem beschreiben, bei dem eine ASCII-Zeichenreihe eingegeben wird und entschieden werden muss, ob diese Zeichenreihe Element von L_C – der Menge gültiger C-Programme – ist. Der Parser leistet allerdings mehr, als lediglich diese Frage zu entscheiden. Er erzeugt einen Parser-Baum, Einträge in einer Symboltabelle und möglicherweise anderes mehr. Der Compiler als Ganzes löst sogar das Problem, ein C-Programm in Objektcode für einen bestimmten Rechner umzuwandeln, was weit über die einfache Beantwortung der Frage der Gültigkeit des Programms hinausgeht.

Ist es eine Sprache oder ein Problem?

Mit den Begriffen »**Sprache**« und »**Problem**« ist eigentlich dasselbe gemeint. Welchem Begriff wir den Vorzug geben, hängt von unserem Standpunkt ab. Wenn wir uns nur mit Zeichenreihen an sich beschäftigen, z. B. in der Menge $\{0^n 1^n \mid n \geq 1\}$, dann tendieren wir dazu, uns eine Menge von Zeichenreihen als Sprache vorzustellen. In den letzten Kapiteln dieses Buches werden wir eher dazu neigen, Zeichenreihen eine »Semantik« zuzuordnen, uns Zeichenreihen z. B. als Beschreibung von Graphen, logischen Ausdrücken oder sogar ganzen Zahlen vorzustellen. In diesen Fällen, in denen der durch die Zeichenreihe repräsentierte Sachverhalt wichtiger als die Zeichenreihe selbst ist, wird eine Menge von Zeichenreihen eher als Problem verstanden.

Die Definition von »Problemen« als Sprachen hat sich über die Zeit hinweg jedoch als adäquate Methode bewährt, mit den wichtigen Fragen der Komplexitätstheorie umzugehen. In dieser Theorie geht es darum, Untergrenzen der Komplexität bestimmter Probleme zu beweisen. Besonders wichtig sind Techniken zum Beweis, dass bestimmte Probleme nicht innerhalb eines Zeitraums gelöst werden können, der nicht exponentiell mit der Größe der Eingabe wächst. In dieser Hinsicht ist die sprachbasierte Version bekannter Probleme ebenso schwierig wie die aufgabenorientierte Version.

Das heißt, wenn wir beweisen, dass schwer entscheidbar ist, ob eine gegebene Zeichenreihe zur Sprache L_X gehört, die alle in einer Programmiersprache X gültigen Zeichenreihen umfasst, dann wird es zweifellos ebenso schwierig sein, Programme der Sprache X in Objektcode zu übersetzen. Wenn es einfacher wäre, Code zu generieren, dann könnten wir den Übersetzer ausführen und schließen, dass es sich bei der Eingabe um ein gültiges Element von L_X handelt, wenn der Übersetzer Objektcode erzeugen kann. Da der letzte Schritt in der Bestimmung, ob Objektcode generiert wurde, nicht schwierig sein kann, können wir den schnellen Algorithmus zur Erzeugung von Objektcode einsetzen, um die Frage der Zugehörigkeit zu L_X effizient zu entscheiden. Dies widerspricht der Annahme, dass die Zugehörigkeit zu L_X schwierig zu bestimmen ist. Hiermit verfügen wir über einen Beweis durch Widerspruch für die Aussage:

»Wenn es schwierig ist, die Zugehörigkeit zu L_x bestimmen, dann ist es auch schwierig, in der Programmiersprache X geschriebene Programme zu kompilieren.«

Diese Technik, mit der die Schwierigkeit eines Problems gezeigt wird, indem mit seinem angenommenen effizienten Lösungsalgorithmus ein anderes, als schwierig bekanntes Problem effizient gelöst wird, wird als »Reduktion« des zweiten Problems auf das erste bezeichnet. Die Reduktion ist ein wichtiges Werkzeug beim Studium der Komplexität von Problemen, und sie wird durch unseren Ansatz stark erleichtert, Probleme als Fragen über die Zugehörigkeit zu einer Sprache zu betrachten statt als Fragen allgemeinerer Art.

ZUSAMMENFASSUNG VON KAPITEL 1

- **Endliche Automaten:** Endliche Automaten umfassen Zustände und Übergänge zwischen Zuständen, die in Reaktion auf Eingaben erfolgen. Sie sind beim Erstellen verschiedener Arten von Software nützlich, zu denen beispielsweise die lexikalische Analysekomponente von Compilern und Systeme zur Überprüfung der Fehlerfreiheit von Schaltkreisen oder Protokollen gehören.
- **Reguläre Ausdrücke:** Hierbei handelt es sich um eine strukturelle Notation zur Beschreibung der Muster, die durch einen endlichen Automaten repräsentiert werden können. Reguläre Ausdrücke werden in vielen gängigen Arten von Software verwendet, wie z. B. in Tools zur Suche von Mustern in Texten oder Datei-Namen.
- **Kontextfreie Grammatiken:** Diese sind wichtig zur Beschreibung der Struktur einer Programmiersprache und zugehöriger Mengen von Zeichenreihen. Sie werden in der Entwicklung der Parserkomponente von Compilern eingesetzt.
- **Turing-Maschinen:** Hierbei handelt es sich um Automaten, die die Leistungsfähigkeit echter Computer modellieren. Sie ermöglichen uns die Untersuchung der Frage, was von einem Computer geleistet werden kann. Sie ermöglichen es uns zudem, handhabbare Probleme – Probleme, die mit polynomialem Zeitaufwand gelöst werden können – von nicht handhabbaren zu unterscheiden.
- **Deduktive Beweise:** Bei dieser grundlegenden Beweismethode werden Aussagen aufgelistet, die entweder als wahr gegeben sind oder logisch aus vorangegangenen Aussagen folgen.
- **Beweis von Wenn-dann-Aussagen:** Viele Sätze werden in der Form »wenn (etwas), dann (etwas anderes)« formuliert. Die Aussage oder Aussagen, die »wenn« folgen, werden als Hypothese bezeichnet, und die »dann« nachstehenden Aussagen bilden die Konklusion. Deduktive Beweise von Wenn-dann-Aussagen beginnen mit der Hypothese und fahren mit Aussagen fort, die logisch aus der Hypothese oder vorangegangenen Aussagen folgen, bis die Konklusion als eine dieser Aussagen bewiesen wurde.