

LEHRBUCH

Andreas Müller

# Automaten und Sprachen: Theoretische Informatik für die Praxis

Mathematik, Anwendung, Intuition



Springer Vieweg

---

# Automaten und Sprachen: Theoretische Informatik für die Praxis

---

Andreas Müller

# Automaten und Sprachen: Theoretische Informatik für die Praxis

Mathematik, Anwendung, Intuition



Springer Vieweg

Andreas Müller  
OST Ostschweizer Fachhochschule  
Rapperswil, Schweiz

ISBN 978-3-662-70145-4                    ISBN 978-3-662-70146-1 (eBook)  
<https://doi.org/10.1007/978-3-662-70146-1>

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

© Der/die Herausgeber bzw. der/die Autor(en), exklusiv lizenziert an Springer-Verlag GmbH, DE, ein Teil von Springer Nature 2024

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jede Person benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des/der jeweiligen Zeicheninhaber\*in sind zu beachten.

Der Verlag, die Autor\*innen und die Herausgeber\*innen gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autor\*innen oder die Herausgeber\*innen übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planung/Lektorat: Iris Ruhmann  
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer-Verlag GmbH, DE und ist ein Teil von Springer Nature.  
Die Anschrift der Gesellschaft ist: Heidelberger Platz 3, 14197 Berlin, Germany

Wenn Sie dieses Produkt entsorgen, geben Sie das Papier bitte zum Recycling.

# Vorwort

Man könnte die Informatik etwas überspitzt als die computergestützte Verarbeitung von Zeichenketten bezeichnen. Sprachen im Sinne dieser Theorie sind Mengen von Zeichenketten. Die theoretische Informatik hebt einige für den Softwareingenieur interessante abstrakte Eigenschaften von Mengen von Zeichenketten hervor. Insbesondere versucht sie, die Eigenschaften eines zur Verarbeitung der Zeichenketten nötigen Algorithmus zu beschreiben und strebt damit eine Klassifikation der Sprachen an. Reguläre Sprachen (Kapitel 1) zum Beispiel zeichnen sich dadurch aus, dass sie von einem endlichen Automaten verarbeitet werden können. Sie können durch reguläre Ausdrücke (Kapitel 4) beschrieben werden, die dem Programmierer in Systembibliotheken jederzeit zur Verfügung stehen.

Die Konstruktion von Interpretern und Compilern beginnt oft mit einer formalen Definition der Sprache, die der Interpreter oder Compiler verstehen soll. Kontextfreie Grammatiken (Kapitel 5) haben sich für diesen Zweck als leistungsfähige Werkzeuge erwiesen. Die mathematische Theorie zeigt, dass sich zu jeder Grammatik ein nichtdeterministischer Stackautomat konstruieren lässt, der die gleiche Sprache akzeptiert. Der Praktiker wird damit aber nicht glücklich, denn für eine Software-implementation braucht er eine deterministische Lösung. Parsergeneratoren wie der in Kapitel 7 beschriebene Bison schlagen die Brücke zwischen Theorie und Praxis.

Erst die Turing-Maschine, deren Entwicklung im Kapitel 10 beginnt, kommt in ihrer Arbeitsweise einem modernen Computer nahe. Wir werden argumentieren, dass dieses Modell für die Berechnung genau genug ist, dass die daraus gezogenen Schlussfolgerungen tatsächlich auch auf Programme anwendbar sind. Gleichzeitig ist es einfach genug, dass wir allgemeine, mathematische Aussagen machen können. Zum Beispiel werden wir in Kapitel 11 Klassen von Aufgabenstellungen identifizieren, die mit dem Computer grundsätzlich nicht lösbar sind.

Die Komplexitätstheorie (Kapitel 12) befasst sich mit Fragen zur Laufzeit von Algorithmen, die sich unabhängig von der Art der eingesetzten Hardware beantworten lassen. Die Klassen P und NP ermöglichen Problemstellungen nach ihrer Skalierbarkeit zu unterscheiden. Besonders nützlich ist dies für die NP-vollständigen Probleme von Kapitel 13.



Karikatur des Autors  
gezeichnet von Ai Shimizu

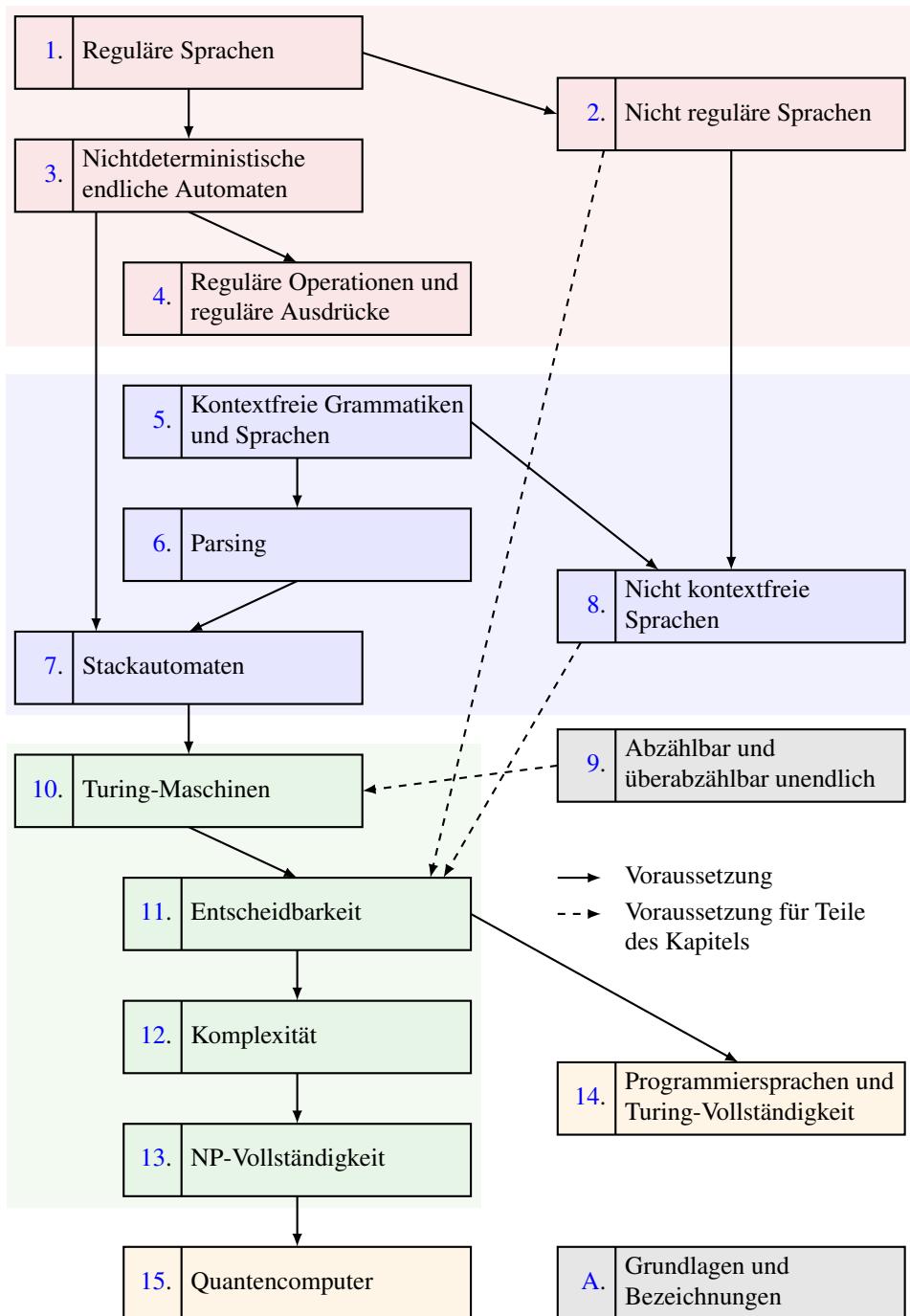


Abbildung 1: Abhängigkeiten der Kapitel des Buches.

Erkennt man sie rechtzeitig, kann man unausweichlichen Performanceproblemen frühzeitig aus dem Weg gehen.

Für den Entwickler ist wichtig, dass die höhere Programmiersprache, in der er seine Software schreibt, ihm vollen Zugang zu den Fähigkeiten des Computers gibt. Das Konzept der Turing-Vollständigkeit von Kapitel 14 stellt dies sicher.

Seit vielen Jahren verspricht uns die Forschung rund um Quantencomputer bald zu erwartende Durchbrüche, die die Informatik umwälzen könnten. Viele Ankündigungen und YouTube-Videos nehmen es im Interesse einer großen Anzahl Clicks mit dem Wahrheitsgehalt der Aussagen nicht so genau. Das Kapitel 15 soll den Leser in die Lage versetzen, das Thema etwas nüchterner zu betrachten.

Die mathematische Theorie gefällt sich auch darin, die Feinheiten der Definitionen und Sätze zu erforschen. Es entsteht zwangsläufig eine umfangreiche Nomenklatur, die oft nicht intuitiv ist und an deren Details sich der Anfänger nur schwer erinnern kann. Vor allem in der Komplexitätstheorie besteht die Gefahr, dass die arkane Terminologie den Gelegenheitsnutzer der Theorie abschreckt. Dieses Buch beschränkt sich bewusst darauf, die wichtigsten Begriffe herauszugreifen und die zentralen Aussagen so zu formulieren, dass sich für die Praxis nützliche, nachvollziehbare Schlussfolgerungen ergeben. Auch der Einstieg in die detaillierte Theorie wird dadurch erleichtert. Dem Leser, der nur einzelne Themen genauer studieren möchte, kann der Abhängigkeitsgraph von Abbildung 1 helfen, sein Studium zu optimieren.

Dieses Buch ist aus den Bemühungen des Autors entstanden, Informatikstudierenden an einer Fachhochschule die Erkenntnisse der theoretischen Informatik zugänglich zu machen. Diese mathematisch sehr attraktive Theorie stellt sich dem Anfänger oft sehr abstrakt und ohne unmittelbaren Nutzen dar. Mathematische Eleganz ist als Motivation auf dieser Stufe nicht genug. Dieses Buch bemüht sich daher, alle Themen mit unmittelbarem Anwendungsbezug zu entwickeln. In der Komplexitätstheorie bedeutet dies zum Beispiel die Betonung der Eins-zu-Eins-Reduktion auf einen gegebenen Katalog von NP-vollständigen Problemen. Dabei sind die Details des Nachweises, warum alle diese Probleme NP-vollständig sind, von untergeordneter Bedeutung. Im Mittelpunkt muss stehen, dass der Leser eine Intuition dafür entwickeln kann, wo er in seinen Softwareprojekten mit NP-vollständigen und damit nicht gut skalierenden Situationen rechnen muss. Die in [39] eingeführten polynomiellen Ausfüllrätsel unterstützen einen intuitiven Zugang zu den Fragestellungen der Komplexitätstheorie weiter.

Auch der bekannte Webcomic XKCD [37] nimmt gelegentlich Themen der theoretischen Informatik auf. Der Comic [36] (Abbildung 4.1 auf Seite 78) zum Beispiel beschreibt ein kompliziertes Szenario, in dem reguläre Ausdrücke den Ausschlag geben. Er suggeriert, dass es leichter wird, sich eine neue Fertigkeit anzueignen, wenn man eine Anwendung dafür im eigenen Erfahrungsbereich finden kann. Es entspricht den Intentionen dieses Buches, wenn sich der Leser zusätzlich zu den im Text angebotenen Beispielen nach weiteren Umsetzungen der Theorie in seiner eigenen Praxis umsieht.

Viele Anwendungen der Theorie sind mit Sourcecodebeispielen illustriert, die über schwarze QR-Codes gefunden werden können. Zum Beispiel wird gezeigt, wie der Scannergenerator Flex aus einem regulären Ausdruck zuerst einen nichtdeterministischen endlichen Automaten baut, den er anschließend in einen deterministischen endlichen Automaten verwand-



delt. Der Code kann von der Buchwebsite heruntergeladen werden. Er kann aber auch direkt auf der Website zum Buch mithilfe der dortigen Erläuterungen studiert werden. Die Website <https://autospr.ch> ist über den nebenstehenden QR-Code verlinkt.

Zu einzelnen Themen gibt es auch YouTube-Videos, die mit Erläuterungen auf der Buchwebsite eingebettet sind. Die blauen QR-Codes im Buch zeigen auf diese Videos.

Wie in jeder mathematischen Theorie kommt es auch bei der theoretischen Informatik auf die Präzision der Begriffe an. Die Definitionen fassen die Begriffe auf unzweideutige Weise, doch entsteht ein intuitives, praktisches Verständnis dafür meist erst mit dem Studium von Beispielen und vor allem auch mit der eigenen Auseinandersetzung damit. Dazu dienen die eingestreuten *Verständniskontrollen*, kleine Aufgaben, erkennbar an ihrem grünen QR-Code, mit dem sich die Lösung abrufen lässt. Sie sind allerdings kein Ersatz für die Übungsaufgaben, die am Ende jedes Kapitels zusammengestellt sind, und die der Vertiefung und Festigung des Gelernten dienen. Lösungen dazu sind ebenfalls auf der Website zum Buch abrufbar. Die formalen Definitionen müssen natürlich genauso eingeübt werden, dabei kann der Anki-Lernkartenstapel helfen, der auf der Website zum Buch verlinkt ist.

Das Buch setzt gewisse, meist allgemein bekannte Kenntnisse der Logik, der Mengenlehre und den Grundlagen der Graphentheorie voraus. Es bemüht sich, die allgemein übliche mathematische Notation zu verwenden, die Verwendung in der Literatur ist leider nicht ganz einheitlich. Anhang A fasst die Begriffe und Notationen zum Nachschlagen zusammen. Auch ein paar grundlegende Begriffe der Wahrscheinlichkeitsrechnung sowie die *O*-Notation, die für Laufzeitabschätzungen benötigt wird, finden sich dort.

Kollegen und Studierende haben das Buchprojekt unterstützt, indem sie Teile des Buches sorgfältig gelesen und viele Fehler gefunden haben. Der Dank des Autors für diese Arbeit gilt Marc Fisch, Laurin Heitzer, Lukas Krüdewagen, Roy Seitz und weiteren. Verbleibende Fehler sind selbstverständlich in der Verantwortung des Autors. Gefundene Fehler werden auf der Buchwebsite in einem Errata-PDF veröffentlicht. Leser sind eingeladen, ihre Fehlerfunde mit dem Erfassungsformular auf der Website dem Autor zu melden.

Ein großer Dank gilt auch der Zeichnerin 清水藍 (Ai Shimizu), die die oben abgebildete, wunderbare Karikatur des Autors wie auch die reizenden Charaktere des Orakels in den Kapiteln 3, 4, 7 und 12 und des Entscheiders und von Pinocchio in Kapitel 11 gezeichnet hat.

Bosco Gurin, im September 2024

Andreas Müller

# Inhaltsverzeichnis

<b>1 Reguläre Sprachen</b>	<b>1</b>
1.1 Alphabet und Sprache . . . . .	2
1.1.1 Alphabet . . . . .	2
1.1.2 Wörter . . . . .	3
1.1.3 Sprache . . . . .	4
1.1.4 Maschinen . . . . .	5
1.2 Deterministische endliche Automaten . . . . .	5
1.2.1 Zustandsdiagramm . . . . .	5
1.2.2 Formale Definition . . . . .	6
1.2.3 Automat für Wörter gerader Länge . . . . .	9
1.2.4 Automat für Ganzzahlen . . . . .	10
1.2.5 TCP Zustandsdiagramm . . . . .	11
1.3 Reguläre Sprachen . . . . .	11
1.3.1 Definition . . . . .	13
1.3.2 Endliche Sprachen sind regulär . . . . .	14
1.3.3 Durch drei teilbare Binärzahlen sind regulär . . . . .	15
1.4 Mengenoperationen — Produktautomat . . . . .	16
1.4.1 Produktzustände und Produktübergänge . . . . .	16
1.4.2 Produktautomat . . . . .	17
1.4.3 Mengenoperation für reguläre Sprachen . . . . .	18
1.5 Der Minimalautomat . . . . .	20
1.5.1 Automaten mit zu vielen Zuständen . . . . .	20
1.5.2 Zustände unterscheiden . . . . .	21
1.5.3 Reduktion auf eine minimale Zustandsmenge . . . . .	22
1.5.4 Automaten vergleichen . . . . .	26
1.6 Rekonstruktion des Automaten aus der Sprache . . . . .	27
1.6.1 Charakterisierung von Zuständen . . . . .	27
1.6.2 Der Myhill-Nerode-Automat . . . . .	28
1.6.3 Myhill-Nerode-Automat für Wörter gerader Länge . . . . .	31
1.6.4 Myhill-Nerode-Automat für durch drei teilbare Binärzahlen . . . . .	32
Übungsaufgaben . . . . .	35

---

<b>2 Nicht reguläre Sprachen</b>	<b>37</b>
2.1 Nicht reguläre Sprachen . . . . .	37
2.1.1 Limitierungen endlicher Automaten . . . . .	37
2.1.2 Die Sprache $\{\emptyset^n 1^n \mid n \geq 0\}$ . . . . .	38
2.2 Pumping-Lemma für reguläre Sprachen . . . . .	40
2.2.1 Wörter als Wege durch einen endlichen Automaten . . . . .	40
2.2.2 Pumpeigenschaft . . . . .	41
2.2.3 Das Pumping-Lemma für reguläre Sprachen . . . . .	42
2.2.4 Die Sprache $\{\emptyset^n 1^n \mid n \geq 0\}$ . . . . .	42
2.2.5 Die Sprache $\{w \in \{0, 1\}^* \mid  w _0 =  w _1\}$ . . . . .	44
2.3 Eine pumpbare, nicht reguläre Sprache . . . . .	44
Übungsaufgaben . . . . .	45
<b>3 Nichtdeterministische endliche Automaten</b>	<b>49</b>
3.1 Nichtdeterminismus . . . . .	49
3.1.1 Fehlende Übergänge . . . . .	50
3.1.2 Nicht eindeutige Übergänge . . . . .	51
3.1.3 Formale Definition . . . . .	53
3.2 Umwandlung in einen DEA . . . . .	56
3.2.1 Zustandsmengen als Zustände . . . . .	56
3.2.2 Der Thompson-NEA . . . . .	58
3.3 $\epsilon$ -Übergänge . . . . .	62
3.3.1 Übergänge ohne Zeichenverarbeitung . . . . .	62
3.3.2 Erreichbare Zustände . . . . .	64
3.3.3 Thompson-NEA für einen NEA $_{\epsilon}$ . . . . .	65
3.4 Probabilistische endliche Automaten . . . . .	67
3.4.1 Probabilistische Übergänge . . . . .	67
3.4.2 Definition eines probabilistischen endlichen Automaten . . . . .	68
3.4.3 Vergleich mit der NEA-Implementation . . . . .	68
Übungsaufgaben . . . . .	69
<b>4 Reguläre Operationen und reguläre Ausdrücke</b>	<b>71</b>
4.1 Reguläre Operationen . . . . .	71
4.1.1 Die Alternative . . . . .	72
4.1.2 Die Verkettung . . . . .	73
4.1.3 Die $*$ -Operation . . . . .	74
4.1.4 Reguläre Operationen und reguläre Sprachen . . . . .	75
4.1.5 Abgeleitete Operationen . . . . .	76
4.2 Reguläre Ausdrücke . . . . .	77
4.2.1 Zeichenketten zur Spezifikation von Sprachen . . . . .	79
4.2.2 Primitive Ausdrücke . . . . .	79
4.2.3 Reguläre Operationen . . . . .	80
4.2.4 Weitere Metazeichen für reguläre Ausdrücke . . . . .	82
4.3 Einen regulären Ausdruck in einen DEA umwandeln . . . . .	84
4.3.1 Konstruktion eines NEA . . . . .	84
4.3.2 Performance von Regex-Implementierungen . . . . .	86

4.4	Regulärer Ausdruck eines DEA . . . . .	91
4.4.1	Verallgemeinerter NEA . . . . .	91
4.4.2	1. Schritt: VNEA vorbereiten . . . . .	92
4.4.3	2. Schritt: Zwischenzustände entfernen . . . . .	93
4.4.4	Beispiel 1 . . . . .	94
4.4.5	Beispiel 2: Durch drei teilbare Binärzahlen . . . . .	95
4.5	Anwendungen . . . . .	97
4.5.1	Scanner-Generator <code>flex</code> . . . . .	97
4.5.2	Ragel, ein Zustandsmaschinencompiler . . . . .	98
4.5.3	Eingebettete lexikalische Analysatoren mit <code>re2c(1)</code> . . . . .	101
	Übungsaufgaben . . . . .	104
<b>5</b>	<b>Kontextfreie Grammatiken und Sprachen</b> . . . . .	<b>107</b>
5.1	Kontextfreie Grammatik und kontextfreie Sprache . . . . .	107
5.1.1	Warum reguläre Sprachen nicht reichen . . . . .	108
5.1.2	Klammerausdrücke erzeugen . . . . .	109
5.1.3	Formale Definitionen . . . . .	111
5.1.4	Grammatik für die Sprache $\emptyset^n 1^n$ . . . . .	112
5.1.5	Grammatik für die Sprache $1^n + 1^m = 1^{m+n}$ . . . . .	113
5.2	Syntaxbaum . . . . .	113
5.2.1	Konstruktion eines Syntaxbaumes . . . . .	114
5.2.2	Nicht eindeutige Syntaxbäume . . . . .	115
5.3	Auswertung . . . . .	117
5.3.1	Expression-Term-Factor-Grammatik . . . . .	117
5.3.2	Ausdrücke auswerten . . . . .	118
5.3.3	Kontrollstrukturen . . . . .	121
5.4	Reguläre Operationen . . . . .	124
5.4.1	Reguläre Operationen für Grammatiken . . . . .	124
5.4.2	Reguläre Sprachen sind kontextfrei . . . . .	126
5.5	Chomsky-Normalform . . . . .	127
5.5.1	Anforderungen an eine Normalform . . . . .	127
5.5.2	Umwandlung in Chomsky-Normalform . . . . .	129
5.5.3	Beispiel für die Umwandlung in Chomsky-Normalform . . . . .	131
	Übungsaufgaben . . . . .	133
<b>6</b>	<b>Parsing</b> . . . . .	<b>137</b>
6.1	Spezifikation von Programmiersprachen . . . . .	137
6.1.1	Geschichte . . . . .	137
6.1.2	Extended Backus-Naur-Form . . . . .	139
6.1.3	Beispiele von Grammatiken moderner Programmiersprachen . . . . .	139
6.1.4	Syntaxanalyse auf der Basis einer Grammatik . . . . .	143
6.2	Der Cocke-Younger-Kasami-Algorithmus . . . . .	143
6.2.1	Ideen . . . . .	143
6.2.2	Pseudocode . . . . .	144
6.3	Der CYK-Algorithmus in Tabellenform . . . . .	146
6.3.1	Ableitungstabelle . . . . .	147

6.3.2 Rekursion wird zu Iteration . . . . .	149
6.3.3 Laufzeit . . . . .	151
Übungsaufgaben . . . . .	152
<b>7 Stackautomaten</b>	<b>155</b>
7.1 Stackspeicher . . . . .	155
7.1.1 Stackoperationen . . . . .	155
7.1.2 Stackautomaten . . . . .	157
7.1.3 Die Sprachen $\emptyset^n 1^n$ und $\{w \mid  w _0 =  w _1\}$ . . . . .	158
7.1.4 Palindrome . . . . .	159
7.1.5 Klammerausdrücke . . . . .	160
7.2 Grammatik und Stackautomat . . . . .	161
7.2.1 Stack als Zwischenspeicher für die Regelanwendung . . . . .	161
7.2.2 Stackautomat zu einer Grammatik . . . . .	162
7.3 LR-Parser . . . . .	164
7.3.1 Warum Syntaxanalyse viel schwieriger ist . . . . .	164
7.3.2 Ein deterministischer $O(n)$ -Parser . . . . .	165
7.3.3 Parsergenerator Bison . . . . .	171
7.4 Grammatik eines Stackautomaten . . . . .	179
7.4.1 Regeln als Wege durch das Zustandsdiagramm . . . . .	180
7.4.2 Vorbereitung des Stackautomaten . . . . .	180
7.4.3 Ablesen von Regeln . . . . .	182
7.4.4 Beispiele . . . . .	184
Übungsaufgaben . . . . .	187
<b>8 Nicht kontextfreie Sprachen</b>	<b>189</b>
8.1 Pumping-Lemma . . . . .	189
8.1.1 Syntaxbäume . . . . .	189
8.1.2 Scherenschnitt in Syntaxbäumen . . . . .	193
8.1.3 Pumpeigenschaft von kontextfreien Sprachen . . . . .	195
8.2 Die Sprache $a^n b^n c^n$ . . . . .	196
Übungsaufgaben . . . . .	198
<b>9 Abzählbar und überabzählbar unendlich</b>	<b>201</b>
9.1 Eine unendliche Geschichte . . . . .	201
9.2 Abzählbar und überabzählbar unendliche Mengen . . . . .	204
9.2.1 Endlich und abzählbar unendlich . . . . .	205
9.2.2 Mengenoperationen . . . . .	207
9.2.3 Überabzählbar unendliche Mengen . . . . .	210
Übungsaufgaben . . . . .	213
<b>10 Turing-Maschinen</b>	<b>215</b>
10.1 Vom Stackautomaten zur Turing-Maschine . . . . .	215
10.1.1 Speicher . . . . .	216
10.1.2 Berechnung . . . . .	219
10.1.3 Zustandsdiagramm . . . . .	221

10.1.4	Formale Definition . . . . .	223
10.1.5	Beispiele von Turing-Maschinen . . . . .	224
10.1.6	Berechnungsgeschichte . . . . .	227
10.2	Varianten von Turing-Maschinen . . . . .	229
10.2.1	Verschiedene Bandalphabete . . . . .	229
10.2.2	Mehrspurmaschine . . . . .	232
10.2.3	Einseitig unendliches Band . . . . .	234
10.2.4	Zusätzliche Kopfbewegungen . . . . .	235
10.2.5	Mehrbandmaschine . . . . .	236
10.2.6	Turing-Maschinen und Programme . . . . .	241
10.3	Turing-erkennbare Sprachen . . . . .	241
10.3.1	Erkennbarkeit . . . . .	241
10.3.2	Aufzählbare Sprachen . . . . .	245
10.4	Die universelle Turing-Maschine . . . . .	247
10.4.1	Moderne Computer und Turing-Maschinen . . . . .	247
10.4.2	Ladbare Programme für Turing-Maschinen . . . . .	249
10.4.3	Universelle Turing-Maschine in Game of Life . . . . .	250
	Übungsaufgaben . . . . .	253
<b>11</b>	<b>Entscheidbarkeit</b> . . . . .	<b>257</b>
11.1	Algorithmen und Entscheidbarkeit . . . . .	257
11.1.1	Hilberts Vortrag am ICM 1900 . . . . .	257
11.1.2	Moderne Formulierung . . . . .	258
11.1.3	Entscheidbarkeit . . . . .	259
11.1.4	Berechenbarkeit . . . . .	262
11.2	Entscheidbare Probleme . . . . .	263
11.2.1	Entscheidbare Probleme für reguläre Sprachen . . . . .	263
11.2.2	Entscheidbare Probleme für kontextfreie Sprachen . . . . .	265
11.3	Das Akzeptanzproblem für Turing-Maschinen . . . . .	266
11.3.1	Das Lügner-Paradoxon . . . . .	266
11.3.2	Das Akzeptanzproblem $A_{TM}$ für Turing-Maschinen . . . . .	268
11.4	Reduktion . . . . .	269
11.4.1	Das Halteproblem . . . . .	270
11.4.2	Vergleich von Sprachen: Reduktion . . . . .	272
11.4.3	Implementationseigenschaften . . . . .	273
11.5	Spracheigenschaften und der Satz von Rice . . . . .	274
11.5.1	Spracheigenschaften . . . . .	274
11.5.2	Leerheitsproblem . . . . .	275
11.5.3	Regularitätsproblem . . . . .	276
11.5.4	Nichttriviale Eigenschaften . . . . .	278
11.5.5	Der Satz von Rice . . . . .	278
11.6	Nicht entscheidbare Probleme und die Berechnungsgeschichte . . . . .	280
11.6.1	Das Post-Korrespondenz-Problem . . . . .	281
11.6.2	Das Gleichheitsproblem für kontextfreie Sprachen . . . . .	284
	Übungsaufgaben . . . . .	287

<b>12 Komplexität</b>	<b>289</b>
12.1 Laufzeit von Turing-Maschinen . . . . .	290
12.1.1 Maßzahl für die Laufzeit . . . . .	290
12.1.2 Polynomielle Laufzeit klassischer Maschinen . . . . .	291
12.1.3 Polynomielle Reduktion . . . . .	292
12.1.4 Vertex-Coloring und Stundenplan . . . . .	293
12.2 Nichtdeterminismus . . . . .	296
12.2.1 Nichtdeterministische Turing-Maschinen . . . . .	297
12.2.2 Verifizierer . . . . .	301
12.2.3 Polynomielle Ausfüllrätsel . . . . .	304
12.3 Komplexitätsklassen P und NP . . . . .	306
12.3.1 Polynomielle Probleme . . . . .	306
12.3.2 Die Klasse NP . . . . .	307
12.3.3 Das P vs. NP-Problem . . . . .	308
Übungsaufgaben . . . . .	309
<b>13 NP-Vollständigkeit</b>	<b>313</b>
13.1 NP-vollständige Sprachen . . . . .	313
13.1.1 Die schwierigsten Probleme in der Klasse NP . . . . .	313
13.1.2 Formale Definition . . . . .	314
13.1.3 Alle NP-vollständigen Probleme sind äquivalent . . . . .	315
13.1.4 Der Umgang mit NP-Vollständigkeit in der Praxis . . . . .	316
13.2 Der Satz von Cook-Levin . . . . .	318
13.2.1 SAT: Erfüllbarkeit für logische Formeln . . . . .	318
13.2.2 Reduktion auf ein polynomielles Ausfüllrätsel . . . . .	319
13.2.3 Reduktion auf SAT . . . . .	322
13.2.4 3SAT ist NP-vollständig . . . . .	322
13.3 Weitere NP-vollständige Probleme . . . . .	324
13.3.1 3SAT und $k$ -CLIQUE . . . . .	324
13.3.2 Das Färbeproblem und das Stundenplanproblem . . . . .	326
13.3.3 SUBSET-SUM und Kryptographie . . . . .	329
13.3.4 CIRCUIT und MINESWEEPER-CONSISTENCY . . . . .	337
13.4 Ein Katalog von NP-vollständigen Problemen . . . . .	342
13.4.1 Der Reduktionsbaum von Karp . . . . .	343
13.4.2 BIP: Binary Integer Programming . . . . .	344
13.4.3 NP-vollständige Probleme zu Graph-Überdeckungen . . . . .	345
13.4.4 NP-vollständige Probleme über Familien von Mengen . . . . .	347
13.4.5 FEEDBACK-NODE-SET und FEEDBACK-ARC-SET . . . . .	352
13.4.6 NP-vollständige Probleme über Pfade in Graphen . . . . .	352
13.4.7 Probleme mit einer Aufteilung in zwei Teilmengen . . . . .	353
13.4.8 3D-MATCHING . . . . .	356
13.4.9 Kombinatorische Optimierungsprobleme . . . . .	357
Übungsaufgaben . . . . .	358

<b>14 Programmiersprachen und Turing-Vollständigkeit</b>	<b>361</b>
14.1 Turing-Vollständigkeit . . . . .	361
14.1.1 Programmiersprachen . . . . .	361
14.1.2 Turing-Vollständigkeit . . . . .	362
14.1.3 Moderne Programmiersprachen . . . . .	363
14.2 Die Sprache LOOP . . . . .	365
14.2.1 Grundelemente . . . . .	365
14.2.2 Die LOOP-Kontrollstruktur . . . . .	366
14.2.3 Das Halteproblem für LOOP . . . . .	369
14.2.4 Anwendungen . . . . .	370
14.3 Die Sprache WHILE . . . . .	373
14.3.1 Die WHILE-Schleife . . . . .	373
14.3.2 WHILE als Erweiterung von LOOP . . . . .	374
14.3.3 Turing-Vollständigkeit von WHILE . . . . .	374
14.3.4 Brainfuck . . . . .	375
14.3.5 Office . . . . .	377
14.4 Die Sprache GOTO . . . . .	377
14.4.1 Die GOTO-Sprache . . . . .	377
14.4.2 WHILE-Programm in GOTO übersetzen . . . . .	379
14.4.3 GOTO-Programm in WHILE übersetzen . . . . .	379
14.4.4 Turing-Maschinensimulator in GOTO . . . . .	380
Übungsaufgaben . . . . .	383
<b>15 Quantencomputer</b>	<b>387</b>
15.1 Analogcomputer . . . . .	387
15.1.1 Berechnung und natürliche Prozesse . . . . .	388
15.1.2 Differentialgleichung und Analogcomputer . . . . .	389
15.1.3 Quantenprozesse . . . . .	389
15.2 Schaltungen und Quantenschaltungen . . . . .	395
15.2.1 Universelle Quantencomputerbausteine . . . . .	396
15.2.2 Die Hadamard-Operation . . . . .	396
15.2.3 CNOT . . . . .	397
15.2.4 Das Toffoli-Gate . . . . .	398
15.2.5 Phasenverschiebung . . . . .	398
15.2.6 Der Approximationssatz . . . . .	399
15.2.7 Aufbau eines universellen Quantencomputers . . . . .	399
15.3 Komplexitätsklassen . . . . .	400
15.3.1 Probabilistische Algorithmen und BPP . . . . .	400
15.3.2 Die Klasse BQP . . . . .	406
<b>A Grundlagen und Bezeichnungen</b>	<b>409</b>
A.1 Logik . . . . .	409
A.1.1 Prädikate . . . . .	409
A.1.2 Logische Verknüpfungen . . . . .	410
A.1.3 Rechenregeln . . . . .	411
A.1.4 Normalformen . . . . .	411

A.2 Mengen . . . . .	411
A.2.1 Konstruktion von Mengen . . . . .	412
A.2.2 Mengenoperationen . . . . .	412
A.2.3 Kartesisches Produkt . . . . .	415
A.2.4 Abbildungen . . . . .	415
A.3 Graphen . . . . .	416
A.3.1 Ungerichtete Graphen . . . . .	416
A.3.2 Gerichtete Graphen . . . . .	417
A.3.3 Beschriftete Graphen . . . . .	417
A.4 Wahrscheinlichkeitsrechnung . . . . .	418
A.4.1 Ereignisse . . . . .	418
A.4.2 Wahrscheinlichkeit . . . . .	419
A.4.3 Bedingte Wahrscheinlichkeit und Unabhängigkeit . . . . .	419
A.4.4 Bernoulli-Experimente und Binomialverteilung . . . . .	420
A.5 <i>O</i> -Notation . . . . .	420
<b>Literatur</b>	<b>423</b>
<b>Index</b>	<b>427</b>



# Kapitel 1

## Reguläre Sprachen

In der modernen Informatik sind das Rechnen mit Zahlen und das Lösen mathematischer Probleme nur von nebенsächlicher Bedeutung. Dieser Aspekt der elektronischen Datenverarbeitung wird in Kursen zur numerischen Mathematik studiert. Nur Studiengänge zu rechnergestützten Wissenschaften oder der Mathematik, die sich mit der Lösung großer numerischer Probleme befassen, wie zum Beispiel der Strömungssimulation oder der numerischen Wetterprognose, unterrichten diesen Aspekt des Computereinsatzes. Der Normalstudiengang der Informatik enthält zwar meistens auch etwas Mathematik, aber kaum Numerik.

Die Informatik bildet Gegebenheiten der Realität in Computermodellen ab und ermöglicht, reale Prozesse nachzubilden. Zum Beispiel ermöglicht E-Mail, das Verfassen, Versenden, Empfangen, Verarbeiten und Archivieren von Briefen in einem Computersystem papierlos durchzuführen. Im Computer ist eine E-Mail eine Folge von Bytes, die einem bestimmten Format folgt. Das Format ermöglicht, verschiedene Eigenschaften wie den Adressaten, den Absender, das Thema, den Datentyp einer Meldung herauszulesen und die Verarbeitung damit zu steuern. Wird das Format nicht eingehalten, kann die Meldung nicht übermittelt oder zugestellt werden. Für die Verarbeitung von E-Mail ist also die grundlegende Voraussetzung, dass ein Computerprogramm das Format dieser Meldung erkennen und validieren kann.

Wenn ein Benutzer über einen Web-Gateway eine SMS senden will, dann muss das Gateway die SMS in das Format einer EMI-Message<sup>1</sup> bringen, welches vom European Telecommunication Standards Institute (ETSI) definiert worden ist. Abgesehen vom STX-Byte `0x02` am Anfang und dem ETX-Byte `0x03` am Ende besteht die Message aus ASCII-Zeichen und sieht ungefähr wie

`STX01/00045/0/30/66677789///1///68656C6C6F/CETX`

<sup>1</sup>Das External Machine Interface (EMI) ist eine Erweiterung des Universal Computer Protocol (UCP), welches verwendet wird, um mit einem short message service center (SMSC) für Mobiltelefone zu verbinden.

aus. Die genaue Bedeutung des Formats spielt für die aktuelle Diskussion keine Rolle. Wesentlich ist, dass alle beteiligten Kommunikationspartner in der Lage sein müssen, dieses Format zu erkennen, fehlerhafte Meldungen auszusortieren und Teile der Meldungen, wie den Adressaten der SMS, zu extrahieren.

Ein verteiltes System muss Anfragen zwischen verschiedenen Rechnern transportieren können. Dazu werden die Anfrageparameter in ein vorgängig vereinbartes Format gebracht, zum Beispiel als JSON-Objekte. Dieses wird dann in einen HTTP-Request eingepackt und an einen Server übermittelt. Der Server zerlegt die empfangene Message, extrahiert das JSON-Objekt und übergibt es einem Backend zur Verarbeitung.

Alle drei Beispiele haben illustriert, dass das Erkennen und Interpretieren von Datenblöcken eine zentrale Aufgabenstellung der Informatik geworden ist. In diesem Kapitel wird dies in den mathematischen Begriff der Sprache abstrahiert, der in Abschnitt 1.1 eingeführt wird. Die deterministischen endlichen Automaten, die in Abschnitt 1.2 definiert werden, sind das Werkzeug, mit dem die Aufgabe im einfachsten Fall der sogenannten regulären Sprachen gelöst werden kann. Endliche Automaten sind sehr effizient und die regulären Sprachen erstaunlich vielfältig. Zum Beispiel kann man zu einem endlichen Automaten immer einen Automaten minimaler Größe finden (Abschnitt 1.5), oder allein aus der Sprache den Automaten rekonstruieren (Abschnitt 1.6).

## 1.1 Alphabet und Sprache

Der Bau der ersten Computer war durch das Bedürfnis motiviert, komplexe mathematische Berechnungen auszuführen. Dazu sind zwar auch moderne Computer fähig, doch sind numerische Berechnungen für die meisten alltäglichen Computeranwendungen in PCs, Tablets oder Mobiltelefonen von zweitrangiger Bedeutung. Stattdessen wird hier die Verarbeitung von Textsegmenten oder ganz allgemein Datenblöcken mit vorgegebenem Format zentral. Netzwerkanwendungen nehmen Sequenzen von Zeichen entgegen, die z. B. als Ereignisse interpretiert werden und je nach Inhalt weitere Operationen auslösen. In diesem Sinne ist Informatik die Verarbeitung von Zeichenketten. Im Folgenden werden Zeichen und Zeichenketten mathematisch formalisiert. Damit wird die Grundlage für die späteren Aufgaben der Theorie geschaffen.

### 1.1.1 Alphabet

Über die Jahre sind in der Computertechnik viele verschiedene Alphabete definiert worden. In den Anfängen wurden Fernschreiber für die Kommunikation mit Computer verwendet. Diese Geräte können nur 5-Bit-Zeichen interpretieren. Ein spezielles Zeichen ist nötig, um zwischen Buchstaben und Ziffern umzuschalten. Abbildung 10.13 links in Kapitel 10 zeigt einen Lochstreifen mit fünf Datenspuren, der zur Aufzeichnung von Fernschreibermeldungen verwendet wurde.

Um diesen Einschränkungen auszuweichen, sind erweiterte Codierungen entwickelt worden. Von besonderer Bedeutung ist der American Standard Code for Information Interchange ASCII. Dieser Code verwendet 7 Bit, kann also 128 verschiedene Zeichen codieren. Viele davon dienen aber der Steuerung der Übertragung, nur die Zeichen mit Codes zwischen 32 und 126 codieren druckbare Zeichen. Dieser Zeichensatz reicht zwar aus,

um englischen Text darzustellen. Für andere Sprachen, die nicht das lateinische Alphabet verwenden oder Zeichen mit diakritischen Zeichen auszeichnen, müssen erweiterte Zeichenmengen definiert werden.

In der Mathematik, der Naturwissenschaft und im Ingenieurwesen wurden viele Zeichen erfunden, die für die Darstellung technischer Sachverhalte notwendig sind, aber weit über die Zeichen hinausgehen, die zur Wiedergabe von Text nötig sind. Viele slavische Sprachen verwenden das kyrillische Alphabet. Die asiatischen Sprachen, vor allem chinesisch und die japanische Kanji-Schrift, verwenden Tausende von Schriftzeichen. Moderne soziale Medien haben viele weitere Bildzeichen oder Emojis eingeführt, die als Abkürzungen nicht mehr wegzudenken sind. Für diese vielfältigen Anforderungen ist der Unicode-Standard geschaffen worden, der alle diese Zeichensätze in einem gemeinsamen Code zusammenfasst.

Aber auch sehr viel kleinere Zeichenmengen können in einer Anwendung zweckmäßig sein. Binärzahlen können mit den zwei Zeichen 0 und 1 codiert werden. Das Morse-Alphabet codiert alle Zeichen des lateinischen Alphabets mit nur zwei Zeichen, Punkt und Strich:

M	O	R	S	E	C	O	D	E
--	- - -	- - -	... .	.	- - -	- - -	- - -	.

Da allerdings die Länge der Pausen zwischen den Zeichen dazu dient, dem Empfänger die Wortgrenzen deutlich zu machen, könnte man die Pausen als ein weiteres Zeichen ansprechen.

Aus mathematischer Sicht ist also jede beliebige, endliche Menge als Alphabet geeignet. Wir verwenden daher die folgende Definition.

**Definition 1.1** (Alphabet). *Ein Alphabet ist eine nichtleere endliche Menge.*

Im folgenden Text werden die Zeichen eines Alphabets mit einer Schriftart mit fester Zeichenbreite dargestellt, nämlich a … z, A … Z, 0 … 9. Für Alphabete werden meist große griechische Buchstaben wie  $\Sigma = \{0, 1\}$  oder  $\Gamma = \{\emptyset, 1, \cup\}$  verwendet.

## 1.1.2 Wörter

Aus den Zeichen eines Alphabets können jetzt Zeichenketten geformt werden. Eine Zeichenkette  $w$  der Länge  $n$  über dem Alphabet  $\Sigma$  ist ein  $n$ -Tupel von Zeichen, also ein Element  $w \in \Sigma^n$ , wobei  $\Sigma^n$  das  $n$ -fache kartesische Produkt von  $\Sigma$  mit sich selbst ist<sup>2</sup>. Eine solche Zeichenkette heißt auch ein *Wort* der Länge  $n$ . Die Schreibweise einer Zeichenkette  $w$  als Tupel  $w = (A, u, t, o, S, p, r)$  ist sehr schwerfällig und nicht gut lesbar. Wörter werden daher einfach als Folgen von Zeichen  $w = \text{AutoSpr}$  geschrieben.

Da Zeichenketten beliebige Länge haben können, ist die Menge aller Wörter über dem Alphabet  $\Sigma$  die Vereinigung

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots \cup \Sigma^n \cup \dots = \bigcup_{k=0}^{\infty} \Sigma^k \quad (1.1)$$

<sup>2</sup>Für die Notation zum kartesischen Produkt siehe Abschnitt A.2 im Anhang.

der Wörter jeder Länge. Es gibt nur ein Wort der Länge 0, es enthält kein Zeichen. Als Tupel geschrieben hat es die Form () und heißt das *leere Wort*. Als Zeichenkette geschrieben würde man das leere Wort nicht sehen können, daher wird auch das Zeichen  $\varepsilon$  dafür verwendet. Im Programmcode ist die leere Zeichenkette meistens durch zwei unmittelbar aufeinanderfolgende Anführungszeichen "" erkennbar. Die Definition (1.1) von  $\Sigma^*$  als unendliche Vereinigung bedeutet nicht, dass Wörter unendlich lange sein können. Ein Wort  $w \in \Sigma^*$  ist Element einer der Mengen  $\Sigma^k$  und hat damit endliche Länge.

**Verständniskontrolle 1.1:** Sei  $\Sigma = \{\emptyset, 1\}$  das Alphabet, schreiben Sie  $\Sigma^*$  in aufzählender Form.



Die Länge eines Wortes  $w \in \Sigma^k$  ist  $k$ , sie wird auch als  $k = |w|$  geschrieben. Will man nur wissen, wie oft das Zeichen  $a \in \Sigma$  in einem Wort  $w \in \Sigma^*$  vorkommt, schreibt man dafür  $|w|_a$ .

**Verständniskontrolle 1.2:** Sei  $\Sigma = \{\emptyset, 1\}$ , schreiben Sie die Menge

$$L = \{w \in \Sigma^* \mid |w|_\emptyset = 2\}$$

in aufzählender Form.



### 1.1.3 Sprache

In der Verständniskontrolle 1.1 wurde eine Menge von Wörtern gebildet, eine Teilmenge der Menge  $\Sigma^*$  aller Wörter.

**Definition 1.2** (Sprache). *Eine Sprache  $L$  über dem Alphabet  $\Sigma$  ist eine Teilmenge von  $\Sigma^*$ .*

Die Menge

$$L = \{w \in \Sigma^* \mid |w|_\emptyset = 2\}, \quad \text{mit } \Sigma = \{\emptyset, 1\},$$

von Verständniskontrolle 1.1 ist also die Sprache der binären Zeichenketten, die genau zwei Nullen enthalten.

Um Wörter zu konstruieren, verwenden wir folgende Notationen: Eine Folge von  $k$  identischen Zeichen  $b$  wird als  $b^k$  geschrieben. Das Wort  $w = a^7c^9$  besteht aus 7 Zeichen  $a$  gefolgt von 9 Zeichen  $c$ . Diese Schreibweise für Wörter darf nicht mit der mathematischen Notation von Potenzen verwechselt werden. Zum Beispiel gelten die Beziehungen

$$a^3b^4 \neq b^4a^3, \quad |\emptyset^4| = |\emptyset\emptyset\emptyset\emptyset| = 4 \quad \text{und} \quad |1^0| = |\varepsilon| = 0,$$

die nach Lesart der Algebra unsinnig wären. Da die algebraische Notation in diesem Buch kaum je eine Rolle spielen wird, sind Exponenten fast immer als Zeichen- oder Wortwiederholungen zu lesen.

Von besonderer Bedeutung werden in den nachfolgenden Kapiteln die beiden Sprachen

$$L_1 = \{\emptyset^n 1^n \mid n \geq 0\} \quad \text{und} \quad L_2 = \{a^n b^n c^n \mid n \geq 0\}$$

sein. Die Sprache  $L_1$  besteht aus Wörtern aus einer Anzahl Nullen gefolgt von gleich vielen Einsen, sie wird in Kapitel 2 das Standardbeispiel einer nicht regulären Sprache sein. Ähnlich wird in Kapitel 8 die Sprache  $L_2$  als Standardbeispiel für eine nicht kontextfreie Sprache erkannt werden.

### 1.1.4 Maschinen

Die Informatik handelt von der maschinellen Verarbeitung von Zeichenketten. Ein in der Sprache Java geschriebenes Programm besteht aus dem Source-Code, der in einem UTF-8-codierten Textfile mit der Endung `java` bereitgestellt wird. Der Java-Compiler liest den Source-Code und wandelt ihn in Bytecode um. Jeder Fileinhalt, der vom Java-Compiler akzeptiert wird, kann als gültiger Java-Code betrachtet werden. Der Java-Compiler definiert also die Sprache

$$J = \{w \in \text{UTF-8}^* \mid \text{Der Java-Compiler akzeptiert } w.\},$$

die wir mit der Sprache Java identifizieren können.

Ganz ähnliche definiert der GNU-C-Compiler `gcc` die Sprache

$$C = \{w \in \text{ASCII}^* \mid \text{gcc akzeptiert } w.\},$$

die als der GNU-Dialekt von C bezeichnet wird.

Den Beispielen ist gemeinsam, dass ein Programm oder eine Maschine eine Zeichenkette liest und den Entscheid fällt, ob die Zeichenkette akzeptabel ist. Solche Programme oder Maschinen definieren also Sprachen. Die Eigenschaften von Programmen können somit studiert werden, indem die Eigenschaften der akzeptierten Sprachen untersucht werden.

In den folgenden Kapiteln sollen verschiedene Arten von Maschinen mit zunehmender Leistungsfähigkeit konstruiert werden. Es soll untersucht werden, wie mit der Komplexität der Maschine auch die Komplexität der akzeptierten Sprache größer wird und wie sich dies in Spracheigenschaften niederschlägt.

## 1.2 Deterministische endliche Automaten

Deterministische endliche Automaten sind das erste Beispiel eines Typs von Maschine, welche eine interessante und praktisch nützliche Klasse von Sprachen akzeptieren kann.

### 1.2.1 Zustandsdiagramm

Betreiber von Ski- und Sesselliftanlagen verwenden oft ein Drehkreuz wie in Abbildung 1.1 links, um zu kontrollieren, dass nur Besitzer eines Tickets die Anlage benutzen können. Das Drehkreuz hat zwei mögliche Zustände. Der Benutzer trifft das Drehkreuz normalerweise im verriegelten Zustand  $V$  an, der ihn daran hindert, die Anlage zu betreten. In den

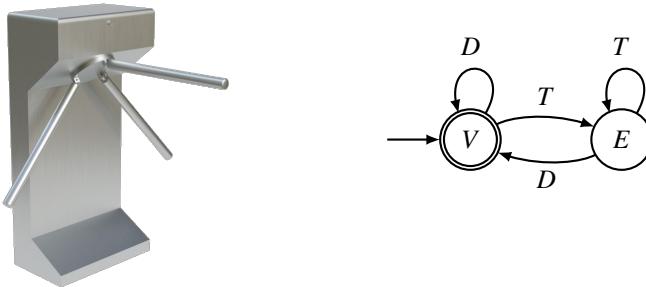


Abbildung 1.1: Drehkreuz zur Ticket-Kontrolle und Zustandsdiagramm zu dessen Modellierung

entriegelten Zustand  $E$  kann das Drehkreuz nur durch Präsentieren eines Tickets versetzt werden. Sobald der Benutzer das Drehkreuz dreht und in die Anlage eintritt, wird es automatisch wieder verriegelt. Software-Ingenieure verwenden ein Zustandsdiagramm wie in Abbildung 1.1 rechts, um ein solches Gerät zu modellieren.

Das Zustandsdiagramm drückt aus, wie sich die möglichen Handlungen auf den Zustand des Drehkreuzes auswirken können. Wenn das Drehkreuz im Zustand  $V$  verriegelt ist, kann der Benutzer zwar versuchen, das Drehkreuz zu drehen (Aktion  $D$ ), wird aber nicht erfolgreich sein, es wird im Zustand  $V$  verriegelt bleiben. Nur wenn er das Ticket präsentiert (Aktion  $T$ ), wird es entriegelt. An diesem Zustand ändert sich nichts, wenn er das Ticket ein weiteres Mal hinhält. Erst wenn er das Drehkreuz dreht, wird es wieder verriegelt.

Der Betreiber der Anlage hat eine klare Vorstellung davon, was eine akzeptable Transaktion ist. Zu Beginn des Tages erwartet er, dass das Drehkreuz im Zustand  $V$  initialisiert wird. Ebenso soll nach jeder Transaktion das Drehkreuz wieder verriegelt sein. Der Zustand  $V$  hat also eine besondere, mit dem doppelten Kreis hervorgehobene Bedeutung: Er signalisiert eine erfolgreiche Transaktion.

## 1.2.2 Formale Definition

Das Einführungsbeispiel hat illustriert, was alles für die Konstruktion eines deterministischen endlichen Automaten nötig ist. Die formale Definition lautet wie folgt.

**Definition 1.3** (Deterministischer endlicher Automat). *Ein deterministischer endlicher Automat (DEA) ist ein Quintupel  $A = (Q, \Sigma, \delta, q_0, F)$ . Die endliche Menge  $Q$  heißt die Menge der Zustände,  $\Sigma$  ist das Alphabet,  $q_0$  ist der Startzustand,  $F \subset Q$  die Menge der Akzeptierzustände. Die Funktion*

$$\delta: Q \times \Sigma \rightarrow Q : (q, a) \mapsto \delta(q, a)$$

*heißt die Übergangsfunktion.*

Die Forderung, dass die Übergangsfunktion  $\delta$  eine Funktion ist, bedeutet, dass es zu jedem Zustand  $q \in Q$  und zu jedem Zeichen  $a \in \Sigma$  genau einen möglichen Zielzustand

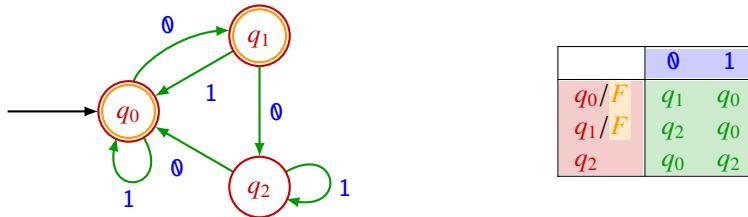


Abbildung 1.2: Zustandsdiagramm (links) und Tabellendarstellung (rechts) des deterministischen endlichen Automaten von Beispiel 1.4. Der Startzustand  $q_0$  wird im Zustandsdiagramm durch den einzigen Pfeil, der nicht von einem anderen Zustand herkommt, gekennzeichnet. In der Tabellendarstellung ist er immer der erste Zustand. Akzeptierzustände haben im Zustandsdiagramm einen doppelten Rand und in der Tabellendarstellung die Markierung /F.

$\delta(q, a) \in Q$  gibt. Dies ist die Rechtfertigung dafür, einen solchen Automaten *deterministisch* zu nennen.

*Beispiel 1.4.* Wir betrachten einen Automaten mit drei Zuständen

$$Q = \{q_0, q_1, q_2\}$$

und dem Alphabet

$$\Sigma = \{\emptyset, 1\}.$$

Startzustand soll der Zustand  $q_0$  sein und die Menge der Akzeptierzustände ist

$$F = \{q_0, q_1\}.$$

Für die vollständige Definition muss jetzt noch die Übergangsfunktion  $\delta: Q \times \Sigma \rightarrow Q$  definiert werden. Die Wertetabelle

	$\emptyset$	1
$q_0$	$q_1$	$q_0$
$q_1$	$q_2$	$q_0$
$q_2$	$q_0$	$q_2$

(1.2)

weist jedem Zustand  $q$  in der linken Spalte und jedem Zeichen  $a$  in der Kopfzeile den Zustand  $\delta(q, a)$  im grünen Bereich der Tabelle zu.  $\circ$

Im Gegensatz zum Einführungsbeispiel erlaubt die formale Definition mehr als einen Akzeptierzustand. Dies ist eine natürliche Verallgemeinerung, die notwendig ist, um aus einem Automaten  $A$  den im folgenden Satz 1.9 beschriebenen Automaten  $\bar{A}$  konstruieren zu können, der alle Wörter akzeptiert, die  $A$  nicht akzeptiert.

**Definition 1.5** (Komplementärer DEA). *Sei  $A = (Q, \Sigma, \delta, q_0, F)$  ein deterministischer endlicher Automat, dann hat der komplementäre deterministische endliche Automat  $\bar{A}$  als Akzeptierzustände die Menge  $Q \setminus F$ , also  $\bar{A} = (Q, \Sigma, \delta, q_0, Q \setminus F)$ .*

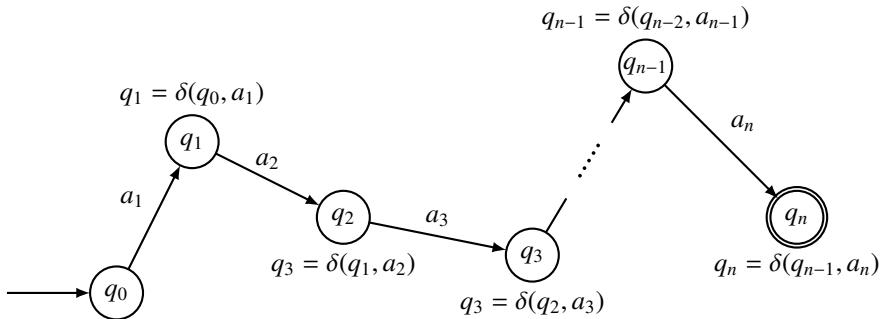


Abbildung 1.3: Das Wort  $w = a_1 a_2 \dots a_n$  definiert einen Pfad durch den endlichen Automaten, der beim Startzustand  $q_0$  beginnt. Die Übergangsfunktion berechnet den jeweils folgenden Zustand aus dem aktuellen Zustand und dem nächsten Zeichen des Wortes.

### Zustandsdiagramm

Die formale Definition ist nicht unbedingt besonders intuitiv oder leicht lesbar. Daher wird oft die Darstellung als Zustandsdiagramm verwendet. Der deterministische endliche Automat von Beispiel 1.4 ist in Abbildung 1.2 dargestellt. Zur leichten Identifikation der Elemente werden die gleichen Farben wie im Beispiel verwendet. Zustände werden als Kreise gezeichnet, Akzeptierzustände durch einen doppelten Kreis ausgezeichnet. Der Startzustand wird mit einem Pfeil bezeichnet, der nicht bei einem anderen Zustand beginnt. Die Übergangsfunktion  $\delta$  wird als mit den Alphabetzeichen beschriftete Pfeile symbolisiert. Bei jedem Zustand gibt es zu jedem Zeichen einen Pfeil. Dies ist, was *deterministisch* für einen endlichen Automaten bedeutet.

### Tabellendarstellung

Für die Implementation eines deterministischen endlichen Automaten ist die *Tabellendarstellung* (Abbildung 1.2 rechts) besonders gut geeignet. Sie entsteht aus der Wertetabelle (1.2) durch Ergänzung der fehlenden Information. In der Wertetabelle fehlt die Angabe des Startzustandes und der Akzeptierzustände. Der Startzustand  $q_0$  steht immer in der ersten Zeile. Die Akzeptierzustände werden durch die zusätzlichen Zeichen  $/F$  ausgezeichnet.

### Ein Wort akzeptieren

Ein deterministischer endlicher Automat kann dazu verwendet werden, eine Auswahl unter den Wörtern  $\Sigma^*$  zu treffen und damit eine Sprache zu definieren. Dazu muss zunächst erklärt werden, wie ein deterministischer endlicher Automat ein Wort verarbeitet.

Die Verarbeitung eines Wortes  $w = a_1 a_2 \dots a_n \in \Sigma^*$  durch den deterministischen endlichen Automaten  $A = (Q, \Sigma, \delta, q_0, F)$  beginnt mit dem endlichen Automaten im Startzustand. Für jedes Zeichen des Wortes werden jetzt nacheinander die Zustände ermittelt, die sich durch Anwendung der Übergangsfunktion ergeben. Im Zustandsdiagramm folgt man

dazu dem mit dem Zeichen beschrifteten Pfeil. Es entstehen so nacheinander die Zustände

$$\begin{aligned} q_0 &\mapsto \delta(q_0, a_1) \\ &\mapsto \delta(\delta(q_0, a_1), a_2), \\ &\mapsto \delta(\delta(\delta(q_0, a_1), a_2), a_3), \\ &\vdots \\ &\mapsto \delta(\dots \delta(\delta(\delta(a_0, a_1), a_2), a_3), \dots, a_n). \end{aligned}$$

Das Wort  $w$  definiert eine Folge von Zuständen, die einen Pfad durch den Automaten beschreiben, wie er in Abbildung 1.3 dargestellt ist.

**Definition 1.6** (Übergangsfunktion für Wörter). *Die Übergangsfunktion für Wörter ist die Abbildung*

$$\delta: Q \times \Sigma^* \rightarrow Q : (q, w) = (q, a_1 a_2 a_3 \dots a_n) \mapsto \delta(\dots, \delta(\delta(q, a_1), a_2), a_3), \dots, a_n).$$

Ein Wort wird akzeptiert, wenn am Ende der Pfeile, die durch die Zeichen eines Wortes definiert sind, ein Akzeptierzustand erreicht wird.

**Definition 1.7** (Ein Wort akzeptieren). *Ein Wort  $w \in \Sigma^*$  wird vom endlichen Automaten  $A = (Q, \Sigma, \delta, q_0, F)$  genau dann akzeptiert, wenn  $\delta(q_0, w) \in F$  ist. Die Sprache  $L(A) = \{w \in \Sigma^* \mid A \text{ akzeptiert } w\}$  heißt die von  $A$  akzeptierte Sprache.*

*Beispiel 1.8.* Das Wort **00101** wird mit dem endlichen Automaten von Beispiel 1.4 wie folgt verarbeitet. Ausgehend vom Startzustand  $q_1$  folgt man im Zustandsdiagramm von Abbildung 1.2 links mit jedem Zeichen des Wortes einem Pfeil und erreicht damit die Zustände

$$q_0 \xrightarrow{\emptyset} q_1 \xrightarrow{\emptyset} q_2 \xrightarrow{1} q_2 \xrightarrow{\emptyset} q_0 \xrightarrow{1} q_0.$$

Da der Zustand  $q_0$  ein Akzeptierzustand ist, wird **00101** von  $A$  akzeptiert und daher **00101**  $\in L(A)$ . Andererseits führt das Wort **1001** wegen

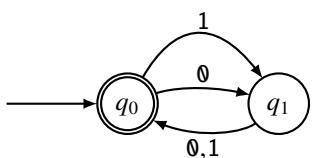
$$q_0 \xrightarrow{1} q_0 \xrightarrow{\emptyset} q_1 \xrightarrow{\emptyset} q_2 \xrightarrow{1} q_2$$

auf den Zustand  $q_2$ , der kein Akzeptierzustand ist,  $q_2 \notin F$ . Daher wird **1001** von  $A$  nicht akzeptiert, **1001**  $\notin L(A)$ .  $\circlearrowright$

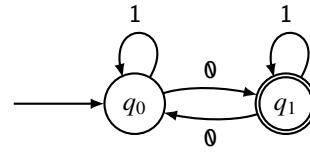
**Satz 1.9.** *Sei  $A = (Q, \Sigma, \delta, q_0, F)$  ein deterministischer endlicher Automat, dann akzeptiert der komplementäre Automat  $\bar{A} = (Q, \Sigma, \delta, q_0, Q \setminus F)$  genau die Wörter, die  $A$  nicht akzeptiert:  $L(\bar{A}) = \Sigma^* \setminus L(A)$ .*

### 1.2.3 Automat für Wörter gerader Länge

Ein Wort über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  hat gerade oder ungerade Länge. Wenn ein endlicher Automat ein Wort  $w \in \Sigma^*$  liest, ändert nach jedem Zeichen die Zahl der verarbeiteten Zeichen von gerade auf ungerade und umgekehrt. Um Wörter in Abhängigkeit vom Zweierrest der Länge zu klassifizieren, werden zwei Zustände benötigt. Der Zustand  $q_0$  steht



$$L(A) = \{w \in \Sigma^* \mid |w| \equiv 0 \pmod{2}\}$$



$$L(A) = \{w \in \Sigma^* \mid |w|_0 \equiv 1 \pmod{2}\}$$

Abbildung 1.4: Links: Deterministischer endlicher Automat, der Wörter gerader Länge akzeptiert. Zur Vereinfachung wurden die Pfeile von  $q_1$  nach  $q_0$  mit Beschriftungen  $\emptyset$  und 1 in einen Pfeil zusammengefasst. Rechts: Deterministischer endlicher Automat, der Wörter mit einer ungeraden Anzahl Nullen akzeptiert.

für eine gerade Anzahl von Zeichen, der Zustand  $q_1$  für eine ungerade Anzahl. Mit jedem Zeichen muss der Automat den Zustand wechseln. Dies führt auf das Zustandsdiagramm in Abbildung 1.4 links.

Im Zustandsdiagramm in Abbildung 1.4 rechts ändern nur die Nullen den Zustand, die Einsen haben also keinen Einfluss darauf, ob ein Wort akzeptiert wird. Die akzeptierte Sprache ist in diesem Fall die Sprache der Wörter, die eine ungerade Zahl von Nullen enthält.

*Verständniskontrolle 1.3:* Konstruieren Sie einen endlichen Automaten über dem Alphabet  $\Sigma = \{\emptyset, 1\}$ , der Zeichenketten akzeptiert, in denen Einsen (1) immer allein stehen, also zum Beispiel

1, 010, 0100, 0100001



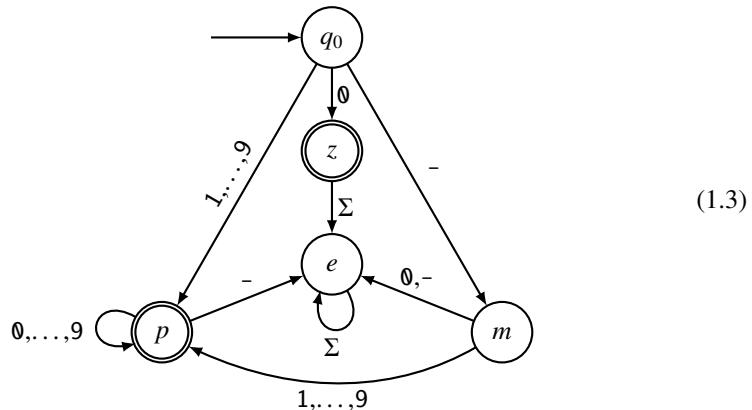
aber nicht

11, 0110, 0100011.

## 1.2.4 Automat für Ganzzahlen

Endliche Automaten können in Anwendungen die Plausibilisierung von Benutzereingaben übernehmen. Als Beispiel suchen wir einen deterministischen endlichen Automaten, der Ganzzahlen ohne führende Nullen mit optionalem negativem Vorzeichen akzeptiert. Das Alphabet dieses Automaten ist  $\Sigma = \{\emptyset, \dots, 9, -\}$ , ein positives Vorzeichen ist nicht zugelassen. Die Darstellung einer von Null verschiedenen Zahl hat keine führende Nullen, wenn sie nach dem nur bei negativen Zahlen vorhandenen negativen Vorzeichen nicht mit dem Zeichen  $\emptyset$  beginnt. Für die Zahl 0 ist dies natürlich nicht möglich. Auch die mathematisch korrekte Schreibweise  $-0$  für die Zahl 0 ist nicht zugelassen.

## Der Automat



ist eine mögliche Lösung dieser Aufgabe. Das leere Wort ist keine korrekte Ganzzahl, der Startzustand  $q_0$  ist daher kein Akzeptierzustand. Eine positive Ganzzahl muss mit einer von  $\emptyset$  verschiedenen Ziffer beginnen, dies wird durch den Akzeptierzustand  $p$  wiedergegeben. Weitere Ziffern dürfen folgen. Der Zustand  $m$  wird mit einem führenden Minuszeichen erreicht, aber erst wenn noch mindestens eine von Null verschiedene Ziffer und beliebige viele weitere Ziffern folgen, entsteht eine akzeptable negative Zahl. Für eine alleinstehende Null steht der Zustand  $z$ . Folgen weitere Zeichen, geht der Automat in den Zustand  $e$ , aus dem es keinen Weg zurück gibt. Der Zustand  $e$  hat also die Funktion eines Fehlerzustands, der immer ein nicht akzeptierbares Wort anzeigt.

## 1.2.5 TCP Zustandsdiagramm

Deterministische endliche Automaten sind auch ein beliebtes Hilfsmittel, um Kommunikationsprotokolle zu spezifizieren. Das Transmission Control Protocol (TCP) [23] definiert aufbauend auf dem Internet Protocol (IP) [22] ein verbindungsbasieretes Protokoll. Die Kommunikationspartner müssen über den aktuellen Zustand der Verbindung informiert sein. In Abbildung 1.5 wird das Zustandsdiagramm aus RFC 793 [23] mit der gleichen Farbcodierung für die Elemente eines deterministischen endlichen Automaten wie in Beispiel 1.4 codiert. Die roten Zustände sind die *socket states*, die man sich zum Beispiel mit dem Befehl `netstat` auf einem Computer anzeigen lassen kann (Abbildung 1.6).

## 1.3 Reguläre Sprachen

Ein deterministischer endlicher Automat akzeptiert eine Sprache. Da die Fähigkeiten eines deterministischen endlichen Automaten ziemlich beschränkt sind, ergibt sich eine besonders übersichtliche Klasse von Sprachen.

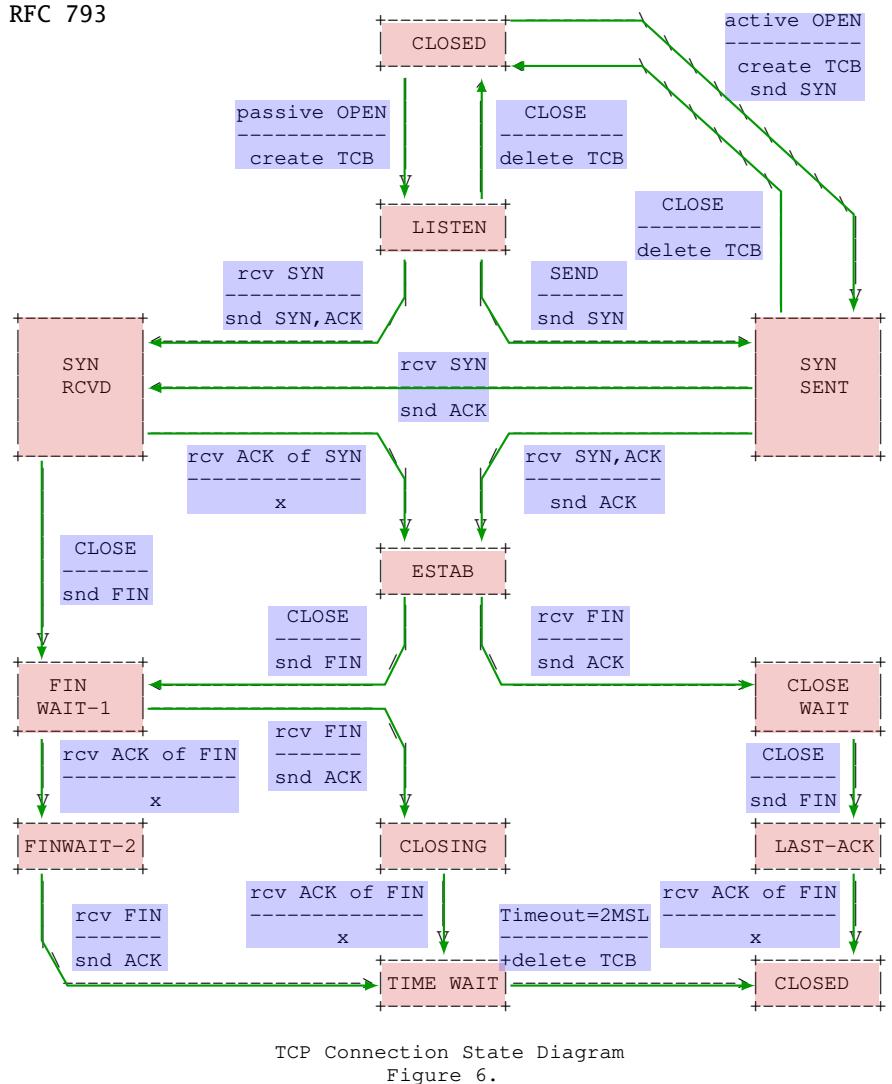


Abbildung 1.5: Zustandsdiagramm des Internet-Protokolls TCP aus dem Standarddokument RFC 793 [23].

```
afm@telescope:~$ netstat -n --tcp -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 0.0.0.0:22              0.0.0.0.*             LISTEN
tcp      0      0 127.0.0.1:631            0.0.0.0.*             LISTEN
tcp      0      0 0.0.0.0:43293            0.0.0.0.*             LISTEN
tcp      0      0 0.0.0.0:38207            0.0.0.0.*             LISTEN
tcp      0      0 0.0.0.0:10080            0.0.0.0.*             LISTEN
tcp      0      0 0.0.0.0:111              0.0.0.0.*             LISTEN
tcp      0      208 192.168.199.10:22       192.168.199.122:62778 ESTABLISHED
tcp      1      0 192.168.199.10:49386       192.168.199.2:631      CLOSE_WAIT
tcp      0      0 192.168.199.10:842       192.168.199.2:2049      ESTABLISHED
tcp6     0      0 ::10000                ::.*                  LISTEN
tcp6     0      0 ::22                   ::.*                  LISTEN
tcp6     0      0 ::1:631                ::.*                  LISTEN
tcp6     0      0 ::33453                ::.*                  LISTEN
tcp6     0      0 ::43981                ::.*                  LISTEN
tcp6     0      0 ::111                 ::.*                  LISTEN
```

Abbildung 1.6: Der Output des `netstat`-Befehls zeigt den TCP Socketzustand für alle aktiven TCP-Sockets auf dem System an. Die Sockets im Zustand LISTEN sind bereit, Verbindungen entgegenzunehmen. Die beiden Sockets im Zustand ESTABLISHED gehören zu Verbindungen zu anderen Maschinen. Die Verbindung im Zustand CLOSE\_WAIT wurde kürzlich geschlossen.

### 1.3.1 Definition

In Abschnitt 1.1.4 wurde illustriert, wie Maschinen eine Sprache definieren können. Deterministische endliche Automaten definieren die sogenannten regulären Sprachen.

**Definition 1.10** (reguläre Sprache). *Eine Sprache  $L$  heißt regulär, wenn es einen deterministischen endlichen Automaten  $A$  gibt derart, dass  $L(A) = L$ .*

Um zu zeigen, dass eine Sprache regulär ist, muss also im Allgemeinen ein deterministischer endlicher Automat konstruiert werden, welcher die Sprache akzeptiert<sup>3</sup>.

---

Verständniskontrolle 1.4: Sei  $\Sigma = \{0, 1\}$ . Ist die Sprache

$$L = \{w \in \Sigma^* \mid \text{Das Wort } 11 \text{ kommt nicht als Teilwort in } w \text{ vor.}\}$$

regulär?

---



<sup>3</sup>Es ist tatsächlich möglich, reguläre Sprachen auch an der Anzahl der Wörter jeder Länge zu erkennen. Hat eine Sprache  $L$  für jedes  $k \in \mathbb{N}$  genau  $a_k$  Wörter der Länge  $k$ , dann bildet man die sogenannte erzeugende Funktion  $f_L(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_k z^k + \dots$ . Die analytische Kombinatorik [13] zeigt, dass die Sprache  $L$  genau dann regulär ist, wenn die erzeugende Funktion  $f_L(z)$  eine rationale Funktion von  $z$  ist.

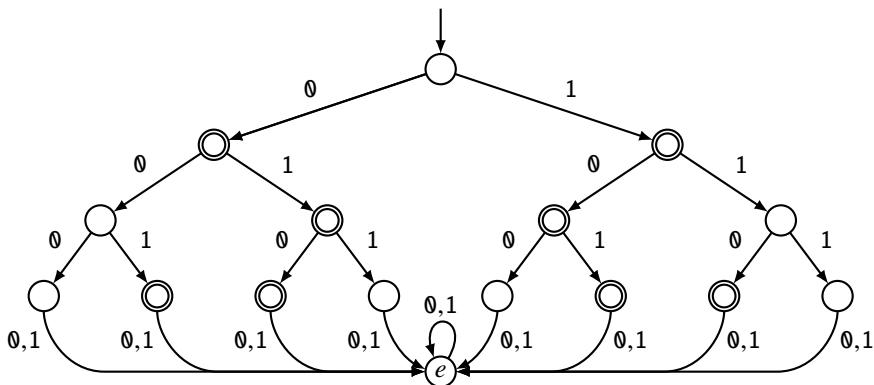


Abbildung 1.7: Deterministischer endlicher Automat, der die Sprache  $L$  von (1.4) akzeptiert.

### 1.3.2 Endliche Sprachen sind regulär

Als Motivationsbeispiel betrachten wir die endliche Sprache

$$L = \{0, 1, 01, 10, 101, 010, 001, 110\} \quad (1.4)$$

über dem Alphabet  $\Sigma = \{0, 1\}$ . Der deterministische endliche Automat von Abbildung 1.7 hat einen eigenen Zustand für jedes Wort der Länge  $\leq 3$ . Die Übergangsfunktion von einem Wort zum nächsten ist durch das Anhängen des Zeichens an das Wort gegeben. Da die Sprache  $L$  nur Wörter mit höchstens 3 Zeichen enthält, reicht für die längeren Wörter ein einziger Zustand  $e$  aus.

Die Konstruktion lässt sich natürlich verallgemeinern und führt auf den folgenden Satz.

**Satz 1.11.** *Endliche Sprachen sind regulär.*

*Beweis.* Sei  $n = \max_{w \in L} |w|$  die Länge des längsten Wortes in der Sprache  $L$ . Als Zustandsmenge verwenden wir die Menge

$$Q = \{w \in \Sigma^* \mid |w| \leq n\} \cup \{e\}$$

der Wörter mit Länge  $\leq n$  und den zusätzlichen Zustand  $e$  für alle längeren Wörter. Sie enthält  $|Q| = |\Sigma|^{n+1}$  Zustände. Startzustand ist das leere Wort  $\varepsilon \in Q$ . Als Menge der Akzeptierzustände verwenden wir  $F = L \subset Q$ . Die Übergangsfunktion ist definiert durch

$$\delta: Q \times \Sigma \rightarrow Q : (q, a) \mapsto \delta(q, a) = \begin{cases} qa & \text{falls } q \text{ ein Wort mit Länge } |q| < n \text{ ist} \\ e & \text{falls } q \text{ ein Wort der Länge } |q| = n \text{ ist} \\ e & \text{falls } q = e. \end{cases}$$

Dabei ist mit  $qa$  das Wort gemeint, welches aus  $q$  durch Anhängen des Zeichens  $a$  entsteht. Das 5-Tupel  $(Q, \Sigma, \delta, \varepsilon, L)$  ist ein endlicher Automat, der die Sprache  $L$  akzeptiert.  $\square$

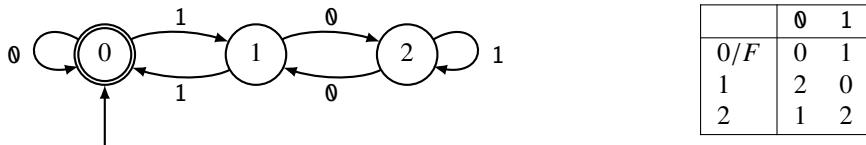


Abbildung 1.8: Deterministischer endlicher Automat, der die durch drei teilbaren Binärzahlen akzeptiert. Links das Zustandsdiagramm, rechts die Tabellendarstellung.

### 1.3.3 Durch drei teilbare Binärzahlen sind regulär

Man finde einen deterministischen endlichen Automaten, der durch drei teilbare Binärzahlen akzeptiert. Wenn dies gelingt, beweist es, dass die Sprache

$$L = \{w \in \{0, 1\}^* \mid w \text{ ist eine durch drei teilbare Binärzahl.}\}$$

regulär ist. Eine Binärzahl kann die drei möglichen Dreierreste 0, 1 oder 2 haben. Es ist daher anzunehmen, dass der Automat mit den drei Zuständen  $Q = \{0, 1, 2\}$  für die Dreierreste auskommen sollte. Wir nehmen an, dass das leere Wort dem Zahlenwert 0 entspricht und dass daher der Zustand 0 der Startzustand des Automaten ist. Den Wert einer Binärzahl zeigen wir durch eine tiefgestellte 2 an, also z. B.  $1011_2 = 22$ .

Der endliche Automat liest ein Wort von links nach rechts, es ist also zu bestimmen, wie sich der Rest ändert, wenn dem Wort ein weiteres Zeichen angehängt wird. Sei also  $w$  ein Wort mit Rest  $r$ , wir müssen ausrechnen, welchen Rest die Wörter  $w0$  und  $w1$  haben. In einem Zahlensystem mit Basis  $b$  entspricht das Anhängen einer Null immer der Multiplikation mit der Basis. Das Wort  $w0$  hat daher den Rest

$$w_2 \equiv r \pmod{2} \quad \Rightarrow \quad w0_2 = 2 \cdot w_2 \equiv 2r \pmod{2}.$$

Das Wort  $w1$  entsteht aus  $w0$  durch Addition von 1, also

$$w1_2 = w0_2 + 1 \quad \Rightarrow \quad w1_2 \equiv 2r + 1 \pmod{2}.$$

Damit kann man jetzt das Zustandsdiagramm von Abbildung 1.8 links bzw. die Tabellendarstellung in Abbildung 1.8 rechts aufbauen.

Durch Wahl eines anderen Akzeptanzzustandes können natürlich auch die Sprachen  $L_1$  der Binärzahlen mit Dreierrest 1 und  $L_2$  der Binärzahlen mit Dreierrest 2 konstruiert werden. Ebenso lassen sich für jede Basis  $b$ , nicht nur für das Binärsystem, und für jeden Divisor  $d$ , nicht nur für den Divisor 3, deterministische endliche Automaten konstruieren, die genau die Zahlen in der Basis  $b$  mit vorgegebenem Rest  $r \pmod{d}$  akzeptieren. Ein solcher Automat braucht  $d$  Zustände, die den möglichen Resten  $0, 1, \dots, d-1$  entsprechen. Hängt man an ein Wort  $w$  mit Rest  $r$  ein Zeichen  $z$  an, dann wird der Rest

$$w_b \equiv r \pmod{d} \quad \Rightarrow \quad wz_b = b \cdot w_b + z_b \equiv b \cdot r + z_b \pmod{d}.$$

Die Sprache

$$L_{r,b,d} = \{w \in \{0, \dots, b-1\}^* \mid w_b \text{ hat Rest } r \text{ bei Teilung durch } d.\}$$

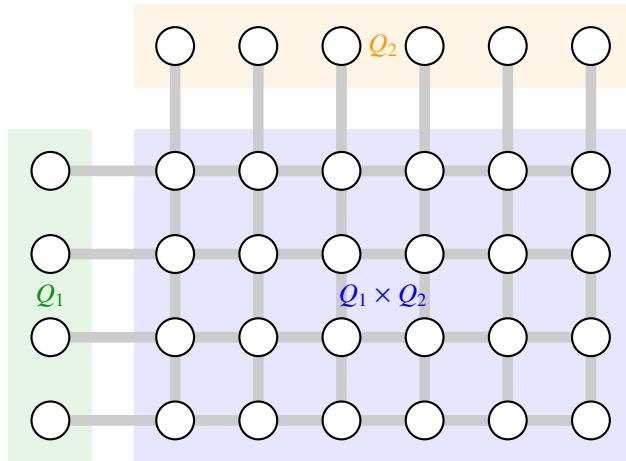


Abbildung 1.9: Zustandsmenge  $Q_1 \times Q_2$  für einen endlichen Automaten, der die Zustandsänderungen in beiden Automaten  $A_1$  und  $A_2$  nachvollziehen kann.

der Darstellungen von Zahlen in der Basis  $b$  mit Rest  $r$  bei Teilung durch  $d$  ist regulär. Die Zustandsdiagramme dieser Automaten sehen recht kunstvoll aus. Man kann sie in [40] finden.

**Verständniskontrolle 1.5:** Sei  $\Sigma = \{0, 1, 2\}$ . Finden Sie einen deterministischen endlichen Automaten, der die Sprache

$$L = \{w \in \Sigma^* \mid w \text{ ist eine ungerade Zahl im Dreiersystem.}\}$$

akzeptiert.



## 1.4 Mengenoperationen — Produktautomat

Da Sprachen nur Mengen von Wörtern sind, darf man auch fragen, ob die Vereinigungsmenge, die Schnittmenge oder die Differenzmenge von zwei regulären Sprachen wieder regulär ist. In diesem Abschnitt werden daher zwei reguläre Sprachen  $L_1$  und  $L_2$  über dem gleichen Alphabet  $\Sigma$  betrachtet, die von den deterministischen endlichen Automaten

$$\begin{aligned} A_1 &= (Q_1, \Sigma, \delta_1, q_{01}, F_1) \\ \text{und} \quad A_2 &= (Q_2, \Sigma, \delta_2, q_{02}, F_2) \end{aligned}$$

akzeptiert werden.

### 1.4.1 Produktzustände und Produktübergänge

Ein deterministischer Automat, der  $L_1 \cup L_2$ ,  $L_1 \cap L_2$  oder  $L_1 \setminus L_2$  akzeptiert, muss Zustände haben, mit denen er alle Zustandsänderungen nachvollziehen kann, die in den Automaten

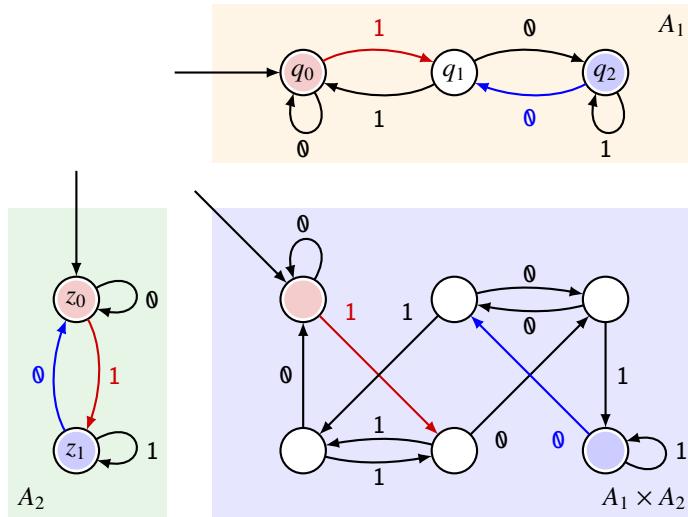


Abbildung 1.10: Die Übergangsfunktion  $\delta = \delta_1 \times \delta_2$  ist definiert durch  $\delta(q, z, a) = (\delta_1(q, a), \delta_2(z, a))$ . Speziell hervorgehoben ist der blaue Übergang  $\delta(q_2, z_1, 0) = (q_1, z_0)$  und der rote Übergang  $\delta(q_0, z_0, 1) = (q_1, z_1)$ .

$A_1$  und  $A_2$  während der Verarbeitung eines Wortes stattfinden. In den Paaren

$$(q_1, q_2) \in Q_1 \times Q_2$$

kann die Information codiert werden, in welchem Zustand jeder der Automaten  $A_1$  und  $A_2$  sich gerade befindet (Abbildung 1.9).

Die Verarbeitung eines Zeichens  $a \in \Sigma$  ändert den Zustand im Automaten  $A_1$  von  $q_1$  in  $\delta_1(q_1, a)$ , im Automaten  $A_2$  von  $q_2$  in  $\delta_2(q_2, a)$ . Die Übergangsfunktion  $\delta = \delta_1 \times \delta_2$  muss auf der Zustandsmenge  $Q_1 \times Q_2$  definiert werden. Sie muss aus einem Zeichen  $a \in \Sigma$  und aus den Zuständen  $(q_1, q_2) \in Q_1 \times Q_2$  den Zustand  $(\delta_1(q_1, a), \delta_2(q_2, a)) \in Q_1 \times Q_2$  machen. Daher ist

$$\delta = \delta_1 \times \delta_2: Q_1 \times Q_2 \times \Sigma \rightarrow Q_1 \times Q_2 : ((q_1, q_2), a) \mapsto (\delta_1(q_1, a), \delta_2(q_2, a)).$$

Sie heißt die *Produktübergangsfunktion*.

In Abbildung 1.10 sind zwei Übergänge der Produktübergangsfunktion dargestellt. Die Komponenten des blauen Produktzustandes  $(q_2, z_1)$  werden vom Zeichen  $0$  auf  $q_1$  in  $A_1$  bzw.  $z_0$  in  $A_2$  abgebildet, der Bildzustand ist daher  $\delta(q_2, z_1, 0) = (q_1, z_0)$ . Die Komponenten des roten Produktzustandes  $(q_0, z_0)$  werden vom Zeichen  $1$  auf die Zustände  $q_1$  bzw.  $z_1$  abgebildet, der Bildzustand ist daher  $\delta(q_0, z_0, 1) = (q_1, z_1)$ .

## 1.4.2 Produktautomat

Aus den beiden Automaten  $A_1$  und  $A_2$  lässt sich ein neuer Automat konstruieren.

**Definition 1.12** (Produktautomat). Sind  $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$  und  $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$  deterministische endliche Automaten über dem Alphabet  $\Sigma$ , dann ist

$$A = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{01}, q_{02}), F)$$

ein deterministischer endlicher Automat mit Akzeptierzuständen  $F \subset Q_1 \times Q_2$ . Er heißt der Produktautomat mit Akzeptierzuständen  $F$ .

### 1.4.3 Mengenoperation für reguläre Sprachen

Für die Mengenoperationen von zwei regulären Sprachen lassen sich mithilfe des Produktautomaten sofort deterministische endliche Automaten dadurch angeben, dass man die Menge  $F$  der Akzeptierzustände festlegt.

**Satz 1.13** (Mengenoperationen für reguläre Sprachen). Seien  $A_i = (Q_i, \Sigma, \delta_i, q_{0i}, F_i)$ ,  $i = 1, 2$ , deterministische endliche Automaten, die die Sprachen  $L_i = L(A_i)$  akzeptieren. Dann gilt

- a) Der Produktautomat mit den Akzeptierzuständen  $F = F_1 \times F_2$  akzeptiert die Schnittmenge  $L_1 \cap L_2$ .
- b) Der Produktautomat mit den Akzeptierzuständen  $F = F_1 \times Q_2 \cup Q_1 \times F_2$  akzeptiert die Vereinigungsmenge  $L_1 \cup L_2$ .
- c) Der Produktautomat mit den Akzeptierzuständen  $F = F_1 \times (Q_2 \setminus F_2)$  akzeptiert die Differenzmenge  $L_1 \setminus L_2$ .
- d) Der Produktautomat mit den Akzeptierzuständen  $F = F_1 \times (Q_2 \setminus F_2) \cup (Q_1 \setminus F_1) \times F_2$  akzeptiert die symmetrische Differenz  $L_1 \Delta L_2 = L_1 \setminus L_2 \cup L_2 \setminus L_1$ .

Die Vereinigungsmenge, Schnittmenge, Differenzmenge und symmetrische Differenz von regulären Sprachen sind regulär.

*Beweis.* Damit ein Wort  $w \in L(A_1) \cap L(A_2)$  der Schnittmenge akzeptiert wird, muss der Automat  $A_1$  wie auch der Automat  $A_2$  nach der Verarbeitung des Wortes in einem Akzeptierzustand sein, als Menge der Akzeptierzustände für den Produktautomaten muss daher die Menge  $F = F_1 \times F_2$  gewählt werden, wie in Abbildung 1.11 a) dargestellt.

Nach dem gleichen Muster muss für die Vereinigungsmenge als Akzeptierzustandsmenge  $F_1 \times Q_2 \cup Q_1 \times F_2$  gewählt werden. Für die Differenzmenge ist es  $F = F_1 \times (Q_2 \setminus F_2)$  und für die symmetrische Differenz ist es

$$F = (F_1 \times Q_2) \Delta (Q_1 \times F_2) = F_1 \times (Q_2 \setminus F_2) \cup (Q_1 \setminus F_1) \times F_2.$$

□

Man beachte, dass die Menge der Akzeptierzustände für eine Operation  $\Delta$  durch

$$F = (F_1 \times Q_2) \Delta (Q_1 \times F_2) \quad (1.5)$$

gegeben ist, außer für die Schnittmenge, wo  $F = F_1 \times F_2$  ist. Es ist aber

$$(F_1 \times Q_2) \cap (Q_1 \times F_2) = F_1 \times F_2,$$

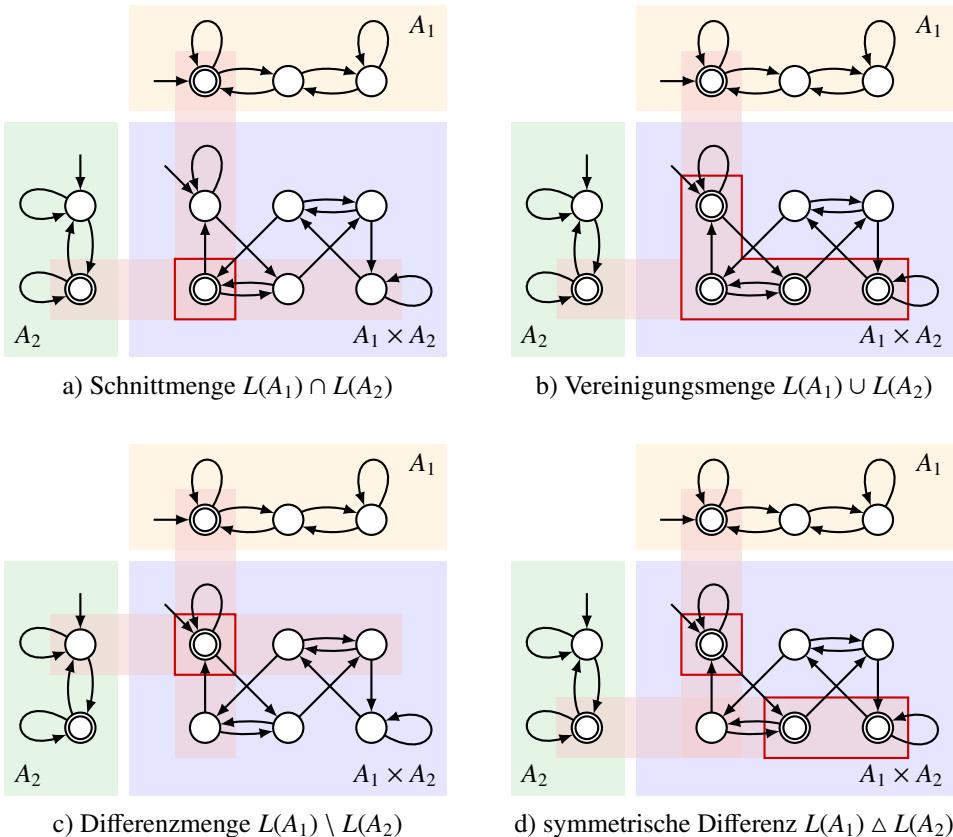
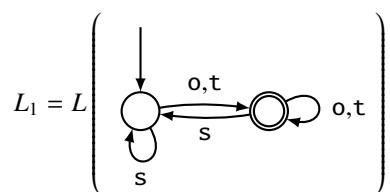


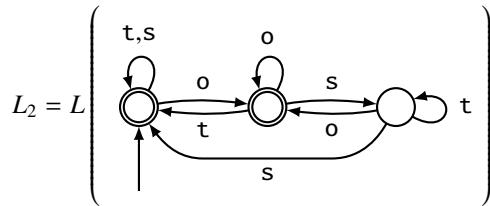
Abbildung 1.11: Akzeptierzustände (rot) für den Produktautomaten, damit in a) die Vereinigungsmenge  $L(A_1) \cup L(A_2)$ , in b) die Schnittmenge  $L(A_1) \cap L(A_2)$ , in c) die Differenzmenge  $L(A_1) \setminus L(A_2)$  und in d) die symmetrische Differenz  $L(A_1) \Delta L(A_2)$  akzeptiert wird.

die Regel (1.5) gilt also in allen Fällen.

*Verständniskontrolle 1.6:* Finden Sie einen DEA für die Differenzmenge  $L = L_1 \setminus L_2$  der Sprachen



und



über dem Alphabet  $\Sigma = \{o, s, t\}$ .

---

## 1.5 Der Minimalautomat

Die Definition eines deterministischen endlichen Automaten verlangt nicht, dass jeder Zustand auch tatsächlich erreichbar ist. Es ist also möglich, dass ein Automat viel mehr Zustände enthält, als wirklich benötigt werden. Zwei Automaten können völlig verschiedene Zustandsmengen haben und trotzdem die gleiche Sprache akzeptieren. Es wird ein Algorithmus benötigt, der deterministische endliche Automaten vergleichen kann.

Objekte vergleichen ist in der Informatik in den wenigsten Fällen eine triviale Aufgabe. Schon beim Vergleich von UTF-8-Zeichenketten muss zum Beispiel berücksichtigt werden, dass viele Unicode-Zeichen mehrere verschiedene Darstellungen haben. Die japanische Hiragana-Schrift hat besonders viele diakritische Zeichen. Das Zeichen  $\hat{\wedge}$  wird *he* gelesen, aber mit den Handakuten genannten zwei kleinen Strichen wird  $\hat{\wedge}$  als *be* gelesen. Dieses Zeichen kann in Unicode jedoch auf zwei Arten erzeugt werden. Einerseits gibt es für  $\hat{\wedge}$  das Unicode-Zeichen U+3079, es kann aber auch als Kombinationszeichen aus einem Handakuten U+3099 und dem Zeichen U+3078  $\hat{\wedge}$  erzeugt werden. Beim Vergleich von Zeichenketten muss diese Zweideutigkeit berücksichtigt werden.

Schon der Vergleich von ASCII-Zeichenketten wird mehrdeutig, wenn Groß- und Kleinschreibung nicht unterschieden werden. Eine beliebte Lösung ist dann, alle Zeichenketten vor dem Vergleich zum Beispiel groß zu schreiben. Durch so eine Standarddarstellung wird der Vergleich einfacher. Der im Folgenden konstruierte Minimalautomat wird als Standardform eines Automaten geeignet sein, um Automaten zu vergleichen.

### 1.5.1 Automaten mit zu vielen Zuständen

Die Sprache  $L = \{w \in \Sigma^* \mid |w| \geq 1\}$  über dem Alphabet  $\Sigma = \{0, 1\}$  kann mit nur zwei Zuständen  $q_0$  und  $q_1$  akzeptiert werden. Der Zustand  $q_1$  wird mit dem ersten Zeichen erreicht, für alle weiteren Zeichen bleibt der Automat im Zustand  $q_1$ . Abbildung 1.12 links zeigt einen ganz anderen deterministischen endlichen Automaten, der ebenfalls die Sprache  $L$  akzeptiert, aber eine viel größere Zahl von Zuständen hat.

Offenbar sind alle Akzeptierzustände des großen Automaten gleichwertig und können zusammengelegt werden. Dies bedeutet aber nicht, dass die Akzeptierzustände eines Automaten automatisch zusammengelegt werden können. Die beiden Akzeptierzustände des Automaten für Ganzzahlen von Abschnitt 1.2.4 haben völlig verschiedene Bedeutung, was man zum Beispiel daran sehen kann, dass alle Übergänge ausgehend vom Zustand  $z$  in

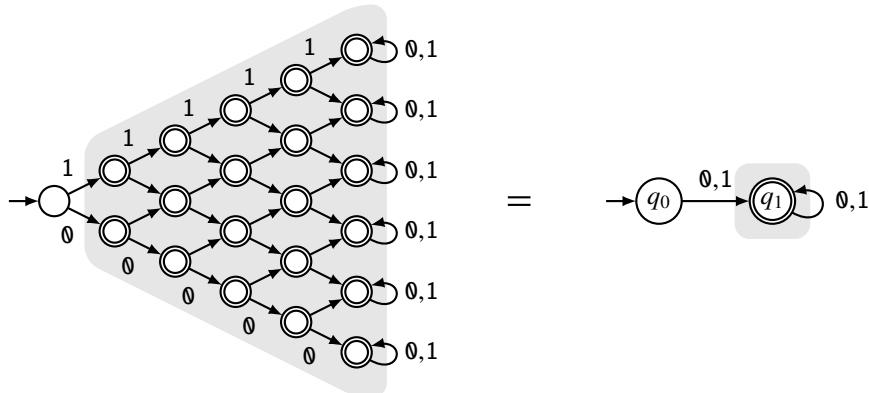


Abbildung 1.12: Deterministischer endlicher Automat mit viel zu vielen Zuständen und Reduktion auf einen Automaten minimaler Größe.

den Fehlerzustand  $e$  führen, während vom Zustand  $p$  aus viele weitere akzeptable Wörter gebildet werden können. Gesucht ist also eine zuverlässige Methode, mit der entschieden werden kann, ob zwei Zustände unterscheidbar sind.

### 1.5.2 Zustände unterscheiden

Wir betrachten einen deterministischen endlichen Automaten  $A$  und zwei Zustände  $q, q' \in Q$ . Der Zustand  $q'$  ist redundant, wenn sich an den vom Automaten akzeptierten Wörtern nichts ändert, wenn alle Übergänge, die in  $q'$  enden, nach  $q$  umgeleitet werden, wenn man also die Übergangsfunktion umdefiniert in

$$\delta': Q \times \Sigma \rightarrow Q : (z, a) \mapsto \begin{cases} q & \text{falls } \delta(z, a) = q' \\ \delta(z, a) & \text{sonst} \end{cases} \quad (1.6)$$

(Abbildung 1.13). Ein Wort  $w$  wird vom Automaten  $A$  akzeptiert, wenn die Übergänge auf einen Akzeptierzustand führen. Wenn auf dem Weg dorthin der Zustand  $q'$  angetroffen wird, lässt sich das Wort  $w$  in zwei Teilwörter  $w = w_1 w_2$  aufteilen, wobei das Wort  $w_1$  in den Zustand  $q'$  führt. Das Wort  $w_2$  führt vom Zustand  $q'$  zu einem Akzeptierzustand. Damit sich durch den Sprung von  $q'$  und  $q$  an den akzeptierten Wörtern nichts ändert, muss das Wort  $w_2$  auch von  $q$  aus zu einem Akzeptierzustand führen. Die Menge der Wörter, die von  $q'$  zu einem Akzeptierzustand führen, und die Menge der Wörter, die von  $q$  zu einem Akzeptierzustand führen, muss also gleich sein.

**Definition 1.14.** Ist  $A = (Q, \Sigma, \delta, q_0, F)$  ein deterministischer endlicher Automat und  $q \in Q$  ein Zustand, dann ist

$$L(q) = \{w \in \Sigma^* \mid \delta(q, w) \in F\}$$

die Menge der Wörter, die von  $q$  zu einem Akzeptierzustand führen.

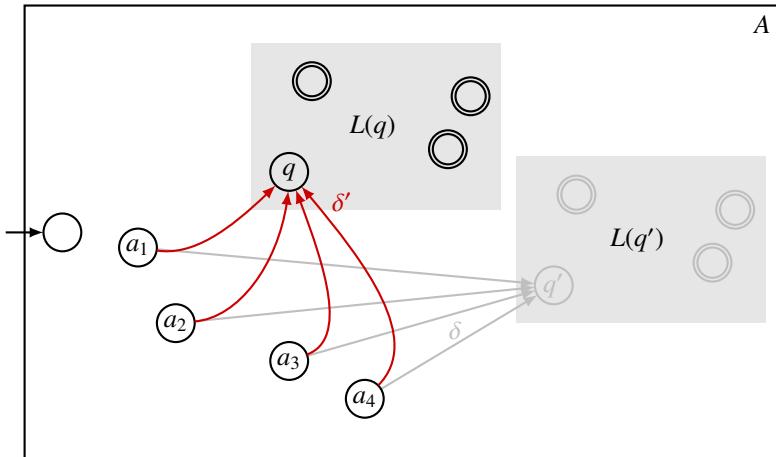


Abbildung 1.13: Umleitung der Übergänge, die im Zustand  $q'$  enden, in den Zustand  $q$ . Die Umleitung ändert die akzeptierte Sprache nicht, wenn  $L(q) = L(q')$  ist.

Die Menge  $L(q)$  kann man auch als die akzeptierte Sprache verstehen, wenn  $q$  als Startzustand des Automaten verwendet würde. Das Kriterium für zusammenlegbare Zustände lässt sich mit den Mengen  $L(q)$  wie folgt formulieren.

**Satz 1.15.** Ist  $A = (Q, \Sigma, \delta, q_0, F)$  ein deterministischer endlicher Automat und sind  $q$  und  $q'$  Zustände von  $A$  mit der Eigenschaft  $L(q) = L(q')$ , dann ist der Automat  $A' = (Q \setminus \{q'\}, \Sigma, \delta', q_0, F \setminus \{q'\})$  mit der Übergangsfunktion  $\delta'$  von (1.6) ein deterministischer endlicher Automat, der die gleiche Sprache  $L(A) = L(A')$  akzeptiert, den Zustand  $q'$  aber nicht verwendet.

### 1.5.3 Reduktion auf eine minimale Zustandsmenge

Ob zwei Zustände  $q$  und  $q'$  in einen Zustand zusammengelegt werden können, hängt nach Satz 1.15 davon ab, ob  $L(q) = L(q')$ . Die Zustände können nicht zusammengelegt werden, wenn es ein Wort  $w \in L(q) \setminus L(q')$  oder  $w \in L(q') \setminus L(q)$  gibt. Wir suchen nach einem Algorithmus, der herausfinden kann, ob  $L(q) = L(q')$  ist.

#### Akzeptierzustände

Ganz offensichtlich verschieden sind die Mengen  $L(q)$  und  $L(q')$ , wenn  $q$  ein Akzeptierzustand ist und  $q'$  nicht oder umgekehrt. Wenn  $q \in F$  ist, ist  $\varepsilon \in L(q)$ , wenn  $q' \notin F$  ist  $\varepsilon \notin L(q')$ . Die Zustände von  $F$  lassen sich also nicht mit Zuständen von  $Q \setminus F$  zusammenlegen.

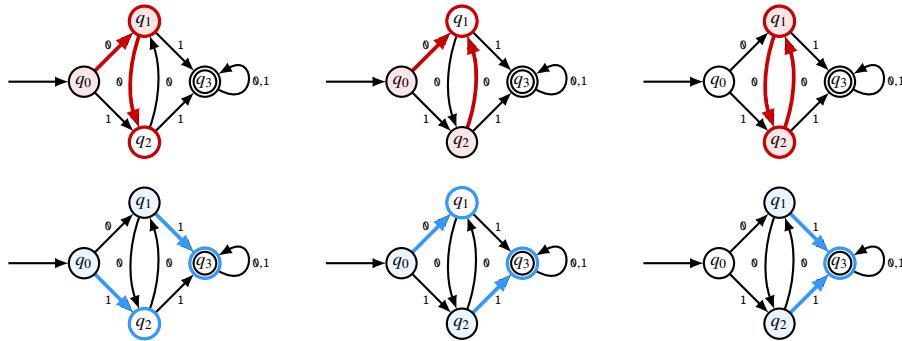
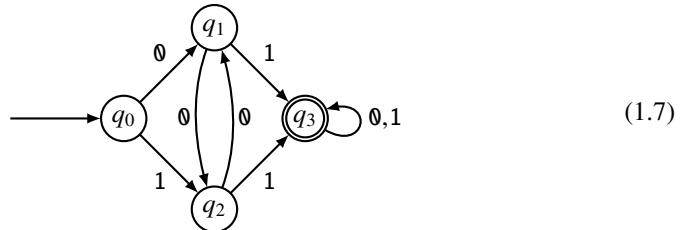


Abbildung 1.14: Übergänge von Paaren von Zuständen mit einem einzelnen Zeichen. In der oberen Zeile die Übergänge mit dem Zeichen 1 in rot, unten die Übergänge mit dem Zeichen 0 in blau. Die Ausgangszustände sind farbig hinterlegt, die Zielzustände mit einem farbigen Kreis markiert. Wenn ein Paar  $(q, q')$  in ein Paar von Akzeptier- und Nichtakzeptierzuständen übergeht, dann sind  $q$  und  $q'$  nicht zusammenlegbar. Dies ist bei den beiden Übergängen links und Mitte in der unteren Zeile der Fall.

*Beispiel 1.16.* Im endlichen Automaten



sind die Paare

$$q_0 \not\equiv q_3, \quad q_1 \not\equiv q_3 \quad \text{und} \quad q_2 \not\equiv q_3$$

von Zuständen nicht zusammenlegbar.  $\circ$

### Längere Wörter

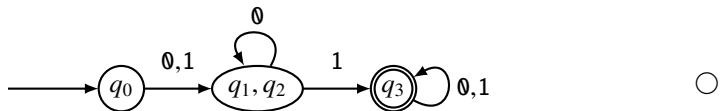
Die Mengen  $L(q)$  und  $L(q')$  können sich auch in einem längeren Wort als dem leeren Wort unterscheiden. Nehmen wir an, dass  $w$  ein Wort der Länge 1 in  $L(q) \setminus L(q')$  ist. Das Zeichen  $w$  überführt die Zustände  $q$  und  $q'$  in die Zustände  $q_1 = \delta(q, w)$  und  $q'_1 = \delta(q', w)$ . Da  $w \in L(q)$  ist, muss  $q_1$  ein Akzeptierzustand sein, während  $q'_1$  kein Akzeptierzustand sein darf. Die Situation tritt also genau dann ein, wenn das Paar  $(q, q')$  nach einem Übergang mit einem einzelnen Zeichen in ein Paar  $(\delta(q, w), \delta(q', w))$  von Zuständen übergeht, von denen einer ein Akzeptierzustand ist und der andere nicht.

*Beispiel 1.17.* Für den endlichen Automat (1.7) haben wir in Beispiel 1.16 bereits drei Paare von Zuständen identifiziert, die nicht zusammenlegbar sind. Es bleiben noch die Paare  $(z_0, z_1)$ ,  $(z_0, z_2)$  und  $(z_1, z_2)$  zu untersuchen. In Abbildung 1.14 sind alle Übergänge von

Zustandspaaren mit verschiedenen Zeichen dargestellt. Die Übergänge der oberen Zeile enden jeweils in einem Paar von Zuständen, über die wir noch nicht wissen, ob sie möglicherweise zusammengelegt werden können. In der unteren Zeile wird das Paar  $(q_0, q_1)$  auf  $(q_2, q_3)$  abgebildet, in dem die Zustände nicht zusammengelegt werden können. Ebenso wird das Paar  $(q_0, q_2)$  auf  $(q_1, q_3)$  abgebildet, welche ebenfalls nicht zusammenlegbar sind. Das letzte Paar  $(q_1, q_2)$  wird auf  $(q_3, q_3)$  abgebildet. Es bleibt also nur noch das Paar  $(q_1, q_2)$  welches möglicherweise zusammengelegt werden kann.  $\circ$

Noch längere Wörter können auf ähnliche Art erhalten werden. Nehmen wir an, es gibt ein Wort  $w = aw_1$  der Länge  $n$  in  $L(q) \setminus L(q')$ . Dann ist das Wort  $w_1 \in L(\delta(q, a)) \setminus L(\delta(q', a))$ . Wenn man also bereits alle Paare von Zuständen bestimmt hat, die sich durch Wörter der Länge  $n - 1$  unterscheiden lassen, dann kann man durch nur einen Übergang mit nur einem Zeichen auch herausfinden, ob sich  $q$  und  $q'$  mit einem Wort der Länge  $n$  unterscheiden lassen.

*Beispiel 1.18.* Für den endlichen Automaten (1.7) bleibt nur noch das Paar  $(q_1, q_2)$  auf Zusammenlegbarkeit hin zu untersuchen. Die Übergänge mit den Zeichen  $\emptyset$  und 1 führen auf die Paare  $(q_1, q_2)$  bzw.  $(q_3, q_3)$ , es gibt also keine Möglichkeit auf ein Paar zu kommen, welches nicht zusammenlegbar ist. Somit müssen  $q_1$  und  $q_2$  zusammenlegbar sein. Der minimierte Automat ist daher



**Definition 1.19** (Minimalautomat). *Der Minimalautomat zu einem Automaten A ist der Automat, der entsteht, wenn man alle zusammenlegbaren Zustände zusammenlegt.*

### Tabelle

Der eben skizzierte Algorithmus lässt sich mithilfe einer Tabelle besonders leicht durchführen (siehe Beispiel 1.20). Jedes Feld der Tabelle entspricht einem Paar  $(q, q')$  von Zuständen. In ein Feld wird ein Kreuz eingetragen, sobald ein Übergang gefunden wurde, der zeigt, dass die beiden Zustände nicht äquivalent sind. Dazu wird wie folgt vorgegangen.

1. Die Paare aus einem Akzeptier- und einem Nichtakzeptierzustand sind sicher nicht äquivalent. In alle Felder  $F \times (Q \setminus F)$  und  $(Q \setminus F) \times F$  wird ein Kreuz eingetragen (rote Kreuze in Beispiel 1.20).
2. Jedes Feld  $(q, q')$ , in dem noch kein Kreuz eingetragen ist, wird jetzt für alle Zeichen  $a \in \Sigma$  überprüft, ob auf dem Feld  $(\delta(q, a), \delta(q', a))$  bereits ein Kreuz eingetragen ist. Wenn ja, kann auch im Feld  $(q, q')$  ein Kreuz eingetragen werden. Andernfalls bleibt das Feld leer (blaue Kreuze in Beispiel 1.20).
3. Schritt 2 wird so lange wiederholt, bis sich bei einem Durchgang durch alle noch leeren Felder nichts mehr ändert.

Felder der Tabelle, die nach dem Algorithmus immer noch leer sind, lassen sich nicht unterscheiden. Es konnte rekursiv kein Wort gefunden werden, mit dem sich die Zustände unterscheiden lassen. Die beiden Zustände können daher zusammengelegt werden.

*Beispiel 1.20.* Für den endlichen Automat (1.7) ergeben sich der Reihe nach die Tabellen

	$q_0$	$q_1$	$q_2$	$q_3$
$q_0$	$\equiv$		$\times$	
$q_1$		$\equiv$	$\times$	
$q_2$			$\equiv$	$\times$
$q_3$	$\times$	$\times$	$\times$	$\equiv$

	$q_0$	$q_1$	$q_2$	$q_3$
$q_0$	$\equiv$	$\times$	$\times$	$\times$
$q_1$	$\times$	$\equiv$		$\times$
$q_2$	$\times$		$\equiv$	$\times$
$q_3$	$\times$	$\times$	$\times$	$\equiv$

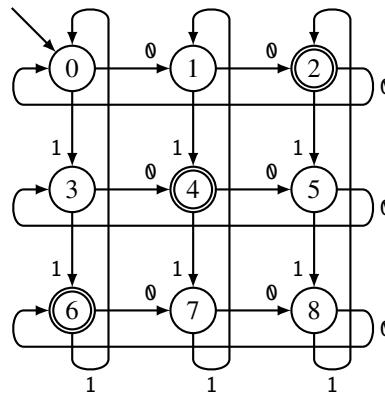
	$q_0$	$q_1$	$q_2$	$q_3$
$q_0$	$\equiv$	$\times$	$\times$	$\times$
$q_1$	$\times$	$\equiv$	$\equiv$	$\times$
$q_2$	$\times$	$\equiv$	$\equiv$	$\times$
$q_3$	$\times$	$\times$	$\times$	$\equiv$

Sie zeigen, dass die beiden Zustände  $q_1$  und  $q_2$  wie bereits bekannt zusammenlegbar sind (grüne  $\equiv$ -Zeichen in Beispiel 1.20).  $\circ$

*Verständniskontrolle 1.7:* Finden Sie den Minimalautomaten zum deterministischen endlichen Automaten mit dem Zustandsdiagramm



[autospr.ch/v/1.7.pdf](https://autospr.ch/v/1.7.pdf)



## Rechenaufwand

Der Algorithmus arbeitet im Schritt 2 alle  $O(n^2)$  Felder der Tabelle ab, wobei jedes Mal die gleiche Arbeit notwendig ist<sup>4</sup>. In jedem Durchgang durch die Tabelle kommt mindestens ein neues Kreuz hinzu, im schlimmsten Fall muss die Tabelle also  $O(n^2)$  mal durchgearbeitet werden. Somit lassen sich alle zusammenlegbaren Zustandspaare mit Aufwand  $O(n^4)$  bestimmen.

<sup>4</sup>Die Notation  $O(n)$  für die Laufzeitgrößenordnung wird in Anhang A.5 aufgefrischt

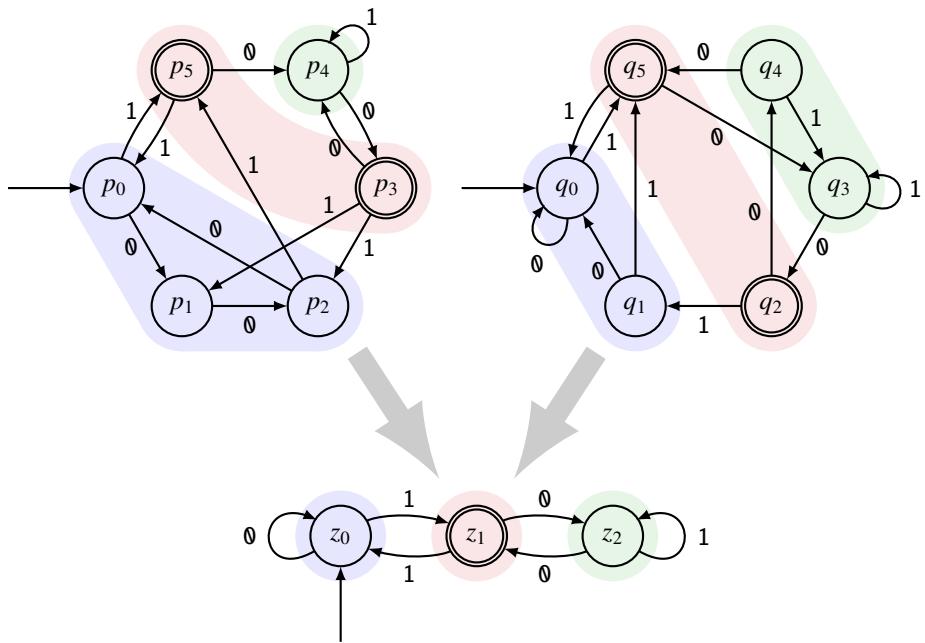


Abbildung 1.15: Zwei deterministische endliche Automaten, die durch Reduktion auf den minimalen Automaten verglichen werden sollen. Der minimale Automat in der Mitte unten ist für beide Automaten gleich, die beiden Automaten akzeptieren also die gleiche Sprache.

### 1.5.4 Automaten vergleichen

Der in Satz 1.15 konstruierte minimale Automat kann auch dazu verwendet werden, zwei Automaten bzw. die davon akzeptierten Sprachen zu vergleichen. Die beiden Automaten in Abbildung 1.12 akzeptieren beide die Sprache  $L = \{w \in \Sigma^* \mid |w| \geq 1\}$ , bestehend aus Wörtern der Länge  $|w| \geq 1$ . Tatsächlich ist der rechte Automat in Abbildung 1.12 der zum linken Automaten gehörige minimale Automat.

Als weiteres Beispiel betrachten wir die beiden Automaten von Abbildung 1.15. Auf den ersten Blick scheinen die Automaten nicht viel miteinander zu tun zu haben. Genauere Betrachtung zeigt, dass die mit gleicher Farbe hinterlegten Zustände jeweils zusammengelegt werden können, wobei der bekannte Automat für Binärzahlen mit Rest 1 entsteht. In Abbildung 1.16 sind die beiden Tabellen mit der Durchführung des Minimalalgorithmus für die Automaten von Abbildung 1.12 dargestellt.

	$p_0$	$p_1$	$p_2$	$p_3$	$p_4$	$p_5$		$q_0$	$q_1$	$q_2$	$q_3$	$q_4$	$q_5$	
$p_0$	$\equiv$	■	■	①	②	①		$q_0$	$\equiv$	■	①	②	②	①
$p_1$	■	$\equiv$	■	①	②	①		$q_1$	■	$\equiv$	①	②	②	①
$p_2$	■	■	$\equiv$	①	②	①		$q_2$	①	①	$\equiv$	①	①	■
$p_3$	①	①	①	$\equiv$	①	■		$q_3$	②	②	①	$\equiv$	■	①
$p_4$	②	②	②	①	$\equiv$	①		$q_4$	②	②	①	■	$\equiv$	①
$p_5$	①	①	①	■	①	$\equiv$		$q_5$	①	①	■	①	①	$\equiv$

Abbildung 1.16: Tabellen für die Durchführung des Minimal-Algorithmus für die beiden deterministischen endlichen Automaten von Abbildung 1.15. Mit ① werden Paare von Zuständen aus Akzeptier- und Nichtakzeptierzuständen bezeichnet. Die mit ② markierten Paare gehen nach einem Übergang in ein Paar aus Akzeptier- und Nichtakzeptierzustand über.

## 1.6 Rekonstruktion des Automaten aus der Sprache

Kann man allein aus der Sprache, also der Menge  $L$  der Wörter, den deterministischen Automaten rekonstruieren? Die Charakterisierung verschiedener Zustände in der Konstruktion des minimalen Automaten bildet die Basis des Satzes von Myhill-Nerode, der eine positive Antwort gibt.

### 1.6.1 Charakterisierung von Zuständen

In der Konstruktion des minimalen Automaten wurde festgestellt, dass sich Zustände  $q$  und  $q'$  nicht unterscheiden lassen, wenn die Mengen  $L(q)$  und  $L(q')$  gleich sind. Ein Wort  $w \in \Sigma^*$  überführt einen Automaten vom Startzustand in einen Zustand. Dieser ist zwar nicht bekannt, wenn der Automat nicht bekannt ist, aber wir können das Wort verwenden, um den Zustand zu beschriften. Die Wörter von  $\Sigma^*$  werden damit zu den Zuständen.

Die Menge  $\Sigma^*$  ist unendlich, sie kann also sicher nicht die Zustandsmenge eines endlichen Automaten sein. Vielmehr müssen sie sich in endlich viele Zustände zusammenfassen lassen. Das Kriterium dafür, ob sie zusammenlegbar sind, ist wieder die Frage, ob die Mengen der Wörter, die ausgehend von diesen Zuständen akzeptiert werden können, übereinstimmen. Dies sind jetzt die Mengen

$$L(w) = \{u \in \Sigma^* \mid wu \in L\},$$

die aus Zeichenketten  $u$  bestehen, die angehängt an  $w$  ein Wort der Sprache ergeben. Zwei Wörter  $w$  und  $w'$  führen also in den gleichen Zustand, wenn die Mengen  $L(w)$  und  $L(w')$  übereinstimmen. Die nötigen Zustände lassen sich also allein durch die Analyse der Menge  $L$  ermitteln.

## 1.6.2 Der Myhill-Nerode-Automat

In diesem Abschnitt sollen die Ideen, die auf den Minimalautomaten geführt haben, noch einen Schritt weiterentwickelt werden, so dass sich ein allgemeiner Algorithmus zur Bestimmung des Minimalautomaten jeder beliebigen Sprache allein aus der Sprache, d. h. aus der Menge  $L$ , ergibt.

### Ein unendlicher Automat für jede beliebige Sprache

Für eine beliebige Sprache  $L$  ist es sehr einfach, einen Automaten mit unendlich vielen Zuständen zu konstruieren, der genau die Wörter der Sprache akzeptiert. Dazu verwenden wir als Menge der Zustände die Menge aller Wörter  $Q = \Sigma^*$  und als Übergangsfunktion die Abbildung

$$\delta(w, a) = wa,$$

welche das Zeichen  $a$  an das Wort  $w$  anhängt. Als Startzustand müssen wir das leere Wort  $q_0 = \varepsilon$  verwenden. Die Akzeptanzzustände sind genau die Wörter, die in der Sprache sind, also  $F = L$ .

Die Menge der Wörter  $\Sigma^*$  können wir als Baum visualisieren. Für den Fall  $\Sigma = \{\emptyset, 1\}$  handelt es sich um einen binären Baum, wie er in Abbildung 1.17 dargestellt ist. Die Übergänge folgen für das Zeichen  $\emptyset$  den Pfeilen nach rechts unten und für das Zeichen 1 den Pfeilen nach rechts oben.

Für die Sprache

$$L = \{w \in \Sigma^* \mid |w| \geq 2 \wedge |w|_1 \equiv 1 \pmod{2}\}$$

sind die Wörter der Sprache in Abbildung 1.17 durch rote Dreiecke hervorgehoben. Die ersten roten Dreiecke tauchen in der dritten Spalte auf, weil die Zustände dort für Wörter mit Länge  $\geq 2$  stehen. Ein Pfad zu einem roten Dreieck enthält immer eine ungerade Anzahl von Pfeilen, die nach rechts oben zeigen, dadurch wird die Bedingung  $|w|_1 \equiv 1 \pmod{2}$  realisiert.

### Reduktion der Zustandsmenge

Der so konstruierte Automat  $(Q, \Sigma, \delta, q_0, F)$  ist ganz offensichtlich deterministisch und er akzeptiert auch genau die Sprache  $L$ , aber er ist nicht endlich. Im Gegenteil, die Zustandsmenge ist unendlich. Wir müssen die Zustandsmenge also noch auf die kleinstmögliche Zustandsmenge reduzieren.

Die Mengen  $L(w)$  beschreiben die Wörter, die der Automat akzeptiert, der als Startzustand das Wort  $w$  verwendet. In Abbildung 1.17 sind zwei solche Mengen als orange bzw. grüne Dreiecke hervorgehoben. Die Dreiecke gleicher Farbe haben bis auf die durch die Darstellung bedingte Skalierung den gleichen Inhalt. Die Mengen  $L(w)$  für die Spitzen dieser Dreiecke sind also gleich, die entsprechenden Zustände müssen zusammengelegt werden. Es verbleiben dann die vier Zustände des Automaten in Abbildung 1.18. Die grüne Menge  $L(w) = L(\emptyset)$  wird auf den grünen Zustand abgebildet, dasselbe gilt für die orange Menge  $L(w) = L(\emptyset 1)$ .

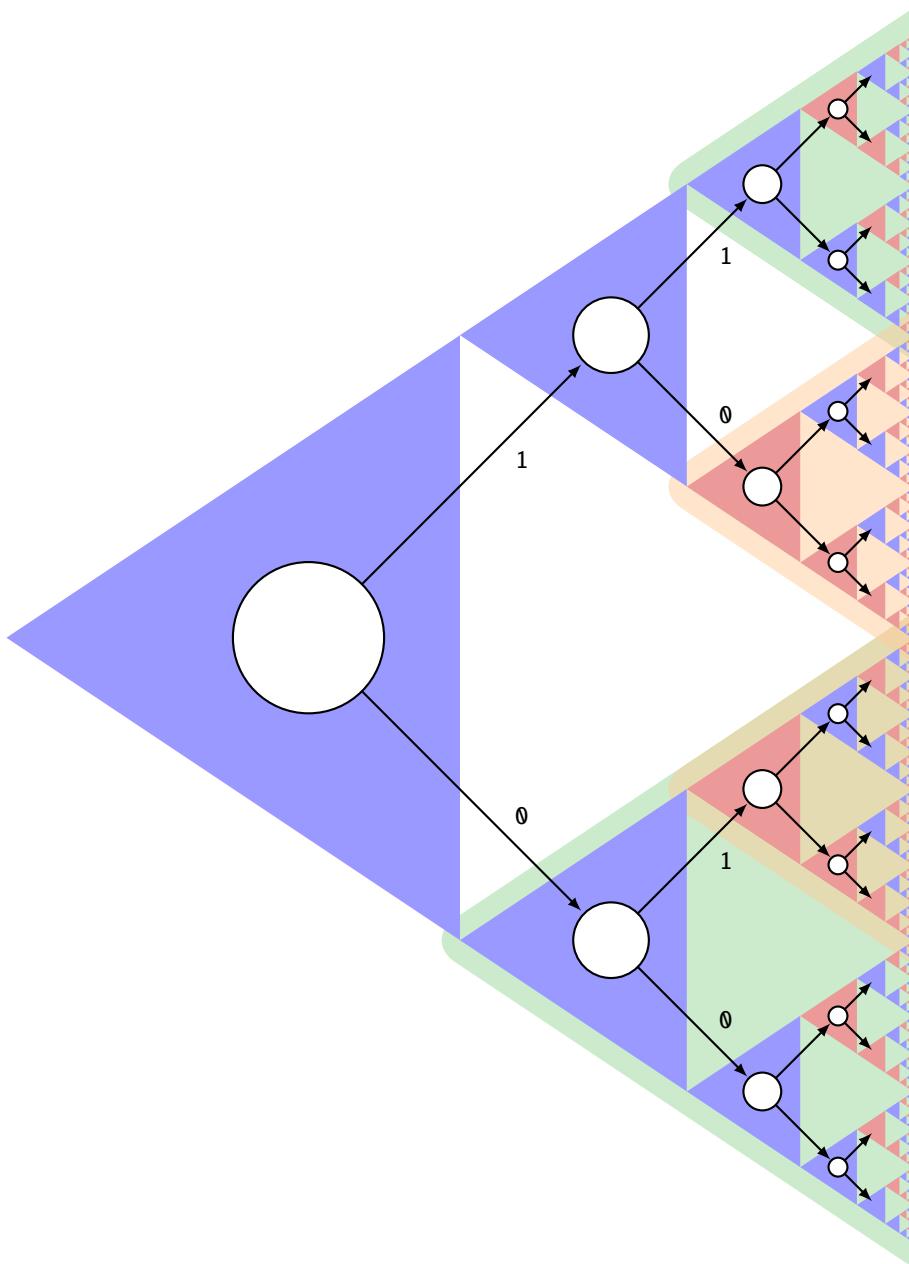


Abbildung 1.17: Herleitung des endlichen Automaten für die Sprache  $L = \{w \in \Sigma^* \mid |w| \geq 2 \wedge |w|_1 \equiv 1 \pmod{2}\}$  mithilfe des Myhill-Nerode-Automaten. Einige der Mengen  $L(w)$  sind als orange bzw. grüne Dreieck hervorgehoben. Die gleichfarbigen Mengen  $L(w)$  können zusammengelegt werden und ergeben den endlichen Automaten von Abbildung 1.18 mit vier Zuständen.

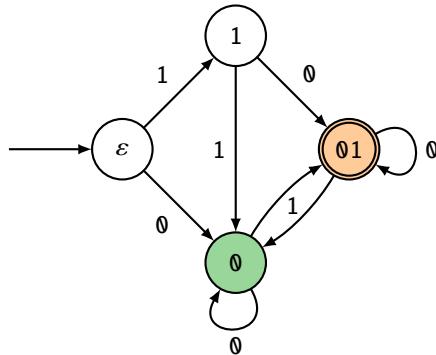


Abbildung 1.18: Endlicher Automat abgeleitet aus den Mengen  $L(w)$  von Abbildung 1.17. Die grünen und orangen Dreiecke, die für verschiedene  $L(w)$  in Abbildung 1.17 standen, sind als Zustände mit den gleichen Farben dargestellt.

### Der Satz von Myhill-Nerode

Die vorangegangenen Überlegungen zur Konstruktion eines Automaten, der die Sprache  $L$  akzeptiert, lassen sich im folgenden Satz zusammenfassen.

**Satz 1.21** (Myhill-Nerode). *Sei  $L$  eine Sprache über dem Alphabet  $\Sigma$ . Dann akzeptiert der deterministische Automat*

$$A = (Q = \Sigma^*, \Sigma, \delta, q_0 = \varepsilon, F = L)$$

mit der Übergangsfunktion

$$\delta: Q \times \Sigma = \Sigma^* \times \Sigma \rightarrow \Sigma^*: (w, a) \mapsto \delta(w, a) = wa$$

genau die Sprache  $L$ . Der minimierte Automat ist

$$A' = (Q', \Sigma, \delta', L, F).$$

Er hat als Zustände die Mengen  $L(w)$ , also  $Q' = \{L(w) \mid w \in \Sigma^*\}$ , die Übergangsfunktion  $\delta'(L(w), a) = L(wa)$  und als Menge der Akzeptierzustände jene Mengen  $L(w)$ , die das leere Wort enthalten.

Die Sprache  $L$  ist genau dann regulär, wenn die Menge  $Q'$  endlich ist.

Der Satz gibt eine Anleitung, wie der deterministische endliche Automat gefunden werden kann, der eine Sprache  $L$  akzeptiert.

1. Bestimme die Mengen  $L(w)$ , sie bilden die Zustände des Automaten. Dies kann zum Beispiel mithilfe einer Tabelle durchgeführt werden, wie in Tabelle 1.1 für das Beispiel vorgeführt. Der Startzustand ist die Menge  $L(\varepsilon) = L$ .
2. Bestimme die Übergangsfunktion  $\delta$  aus  $\delta(w, a) = L(wa)$ .
3. Bestimme die Akzeptierzustände als diejenigen  $L(w)$ , welche  $\varepsilon$  enthalten.

Wenn sich im ersten Schritt herausstellt, dass die Menge  $Q'$  nicht endlich ist, dann kann die Sprache nicht regulär sein. Siehe Abschnitt 2.1.2 für eine Anwendung dieser Beobachtung.

$w$	$L(w)$	Zustand
$\varepsilon$	$L = \{01, 10, 001, 010, 100, 111, \dots\}$	$\varepsilon$
$0$	$\{1, 10, 001, 010, 100, \dots\}$	$0$
$1$	$\{0, 00, 11, 000, 011, 110, \dots\}$	$1$
$00$	$\{1, 10, 001, 010, 100, \dots\}$	$0$
$01$	$\{0, 11, 011, 101, 110, \dots\}$	$01$
$10$	$\{0, 11, 011, 101, 110, \dots\}$	$01$
$11$	$\{1, 10, 001, 010, 100, \dots\}$	$0$
$000$	$\{1, 10, 001, 010, 100, \dots\}$	$0$
$001$	$\{0, 11, 011, 101, 110, \dots\}$	$01$
$010$	$\{0, 11, 011, 101, 110, \dots\}$	$01$
$011$	$\{1, 10, 001, 010, 100, \dots\}$	$0$
$100$	$\{0, 11, 011, 101, 110, \dots\}$	$01$
$101$	$\{1, 10, 001, 010, 100, \dots\}$	$0$
$110$	$\{1, 10, 001, 010, 100, \dots\}$	$0$
$111$	$\{0, 11, 011, 101, 110, \dots\}$	$01$
...	...	...

Tabelle 1.1: Tabelle der Mengen  $L(w)$  für den Automaten von Abbildung 1.18.

### 1.6.3 Myhill-Nerode-Automat für Wörter gerader Länge

Über dem Alphabet  $\Sigma = \{0, 1\}$  betrachten wir die reguläre Sprache

$$L = \{w \in \Sigma^* \mid |w| \equiv 0 \pmod{2}\}$$

von Wörtern gerader Länge und versuchen, einen minimalen endlichen Automaten zu finden, der  $L$  akzeptiert.

#### Die Zustandsmenge

Nach Satz 1.21 müssen als Zustände die Mengen  $L(w)$  verwendet werden. Sie bestehen aus Wörtern  $u \in \Sigma^*$  derart, dass  $wu \in L$  ist, also gerade Länge bekommt. Offenbar bedeutet dies, dass für Wörter  $w$  mit gerader Länge genau die Wörter  $u$  mit gerader Länge in  $L(w)$  sind, während für  $w$  mit ungerader Länge die  $u$  mit ungerader Länge in  $L(w)$  sind. Der Automat verwendet also die beiden Zustände

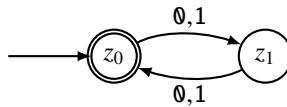
$$\begin{aligned} z_0 &= L = \{u \in \Sigma^* \mid |u| \equiv 0 \pmod{2}\} \\ z_1 &= \{u \in \Sigma^* \mid |u| \equiv 1 \pmod{2}\}. \end{aligned}$$

#### Die Übergangsfunktion

Durch Hinzufügen eines Zeichens ändert sich der Zweierrest von  $w$ , jeder Übergang führt also von  $z_0$  nach  $z_1$  bzw. von  $z_1$  nach  $z_0$ :

$$\delta(z_0, a) = z_1 \quad \text{und} \quad \delta(z_1, a) = z_0$$

für alle  $a \in \Sigma$ . Damit lässt sich auch das Zustandsdiagramm



zeichnen.

*Verständniskontrolle 1.8:* Finden Sie die Zustände eines endlichen Automaten für die Sprache

$$L = \{0^n 1 0 \mid n > 0\}$$

über dem Alphabet  $\Sigma = \{0, 1\}$  mit der Methode von Myhill-Nerode.



## 1.6.4 Myhill-Nerode-Automat für durch drei teilbare Binärzahlen

Wir wissen bereits, dass die Sprache

$$L = \{w \in \Sigma^* \mid w \equiv 0 \pmod{3}\}$$

der durch drei teilbaren Binärzahlen über dem Alphabet  $\Sigma = \{0, 1\}$  regulär ist. Die Automatenkonstruktion nach Myhill-Nerode von Satz 1.21 müsste diesen Automaten rekonstruieren.

### Die Zustandsmenge

Die Zustandsmenge besteht aus den verschiedenen Mengen

$$L(w) = \{u \in \Sigma^* \mid wu \in L\}.$$

Sie können durch Aufzählung aller Wörter bestimmt werden, wie dies in der Tabelle 1.2 durchgeführt ist. Die Tabelle zeigt, dass der Automat drei Zustände braucht. Die blaue Menge  $L(\varepsilon) = L = z_0$  ist der Startzustand.

### Die Übergangsfunktion $\delta$

Aus der Tabelle 1.2 lässt sich auch bereits die Übergangsfunktion ablesen. Die ersten drei Zeilen der Tabelle zeigen zum Beispiel, dass

$$\delta(z_0, \emptyset) = z_0 \quad \text{und} \quad \delta(z_0, 1) = z_1$$

ist. Aus den Zeilen für  $w = 1$  und  $w = 10$  folgt, dass  $\delta(z_1, \emptyset) = z_2$  ist. Auf diese Weise findet man die Wertetabelle

$\delta$	$\emptyset$	1
$z_0$	$z_0$	$z_1$
$z_1$	$z_2$	$z_0$
$z_2$	$z_1$	$z_2$

für die Übergangsfunktion, die mit der Tabelle von Abbildung 1.8 übereinstimmt. Damit ist es jetzt auch leicht, das Zustandsdiagramm zu zeichnen: Es ergibt sich natürlich wieder das bekannte Diagramm von Abbildung 1.8.

$w$	$L(w)$	Zustand
$\epsilon$	$\{\epsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, \dots\}$	$z_0$
$0$	$\{\epsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, \dots\}$	$z_0$
$1$	$\{1, 10, 001, 100, 111, 0010, 0101, 1000, 1011, 1110, \dots\}$	$z_1$
$00$	$\{\epsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, \dots\}$	$z_0$
$01$	$\{1, 10, 001, 100, 111, 0010, 0101, 1000, 1011, 1110, \dots\}$	$z_1$
$10$	$\{01, 010, 101, 0001, 0100, 0111, 1010, 1101, \dots\}$	$z_2$
$11$	$\{\epsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, \dots\}$	$z_0$
$000$	$\{\epsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, \dots\}$	$z_0$
$001$	$\{1, 10, 001, 100, 111, 0010, 0101, 1000, 1011, 1110, \dots\}$	$z_1$
$010$	$\{01, 010, 101, 0001, 0100, 0111, 1010, 1101, \dots\}$	$z_2$
$011$	$\{\epsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, \dots\}$	$z_0$
$100$	$\{1, 10, 001, 100, 111, 0010, 0101, 1000, 1011, 1110, \dots\}$	$z_1$
$101$	$\{01, 010, 101, 0001, 0100, 0111, 1010, 1101, \dots\}$	$z_2$
$110$	$\{\epsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1001, \dots\}$	$z_0$
$111$	$\{1, 10, 001, 100, 111, 0010, 0101, 1000, 1011, 1110, \dots\}$	$z_1$
...	...	...
$w_2 \equiv 0 \pmod{3}$	$\{u \in \Sigma^* \mid u_2 \equiv 0 \pmod{3}\}$	$z_0$
$w_2 \equiv 1 \pmod{3}$	$\left\{ u \in \Sigma^* \mid \begin{array}{l}  u  \text{ gerade} \Rightarrow u_2 \equiv 2 \pmod{3} \\  u  \text{ ungerade} \Rightarrow u_2 \equiv 1 \pmod{3} \end{array} \right\}$	$z_1$
$w_2 \equiv 2 \pmod{3}$	$\left\{ u \in \Sigma^* \mid \begin{array}{l}  u  \text{ gerade} \Rightarrow u_2 \equiv 1 \pmod{3} \\  u  \text{ ungerade} \Rightarrow u_2 \equiv 2 \pmod{3} \end{array} \right\}$	$z_2$

Tabelle 1.2: Tabelle der Mengen  $L(w)$  für die Sprache der durch drei teilbaren Binärzahlen  $L = \{w \in \Sigma^* \mid w \equiv 0 \pmod{3}\}$ . Der besseren Übersichtlichkeit halber sind die Mengen farbig codiert. Es zeigt sich, dass nur drei Mengen vorkommen, dies sind die drei Zustände des Myhill-Nerode Automaten der Sprache. Im unteren Teil der Tabelle ist die algebraische Charakterisierung der Mengen  $L(w)$  in Abhängigkeit vom Rest von  $w$  bei Teilung durch 3 dargestellt.

### Algebraische Bestimmung der Zustandsmenge

Weiter oben wurden die Mengen  $L(w)$  einfach aufgelistet. In diesem Abschnitt sollen sie etwas systematischer algebraisch bestimmt werden.

Es ist also zu untersuchen, welchen Rest die Verkettung von  $wu$  hat. Seien  $w, u \in \Sigma^*$  zwei binäre Wörter. Die Verkettung der Wörter bedeutet als arithmetische Operation, dass die erste Binärzahl mit  $2^{|u|}$  multipliziert wird, bevor die zweite addiert wird, also

$$wu_2 = w_2 \cdot 2^{|u|} + u_2. \quad (1.8)$$

Da nur der Rest modulo 3 interessiert, müssen wir den Rest von  $2^k$  für alle  $k$  bestimmen. Wegen  $2 \cdot 1 = 2$  und  $2 \cdot 2 \equiv 1 \pmod{3}$  sind die Reste

$k$	0	1	2	3	4	5	6	7	...
$2^k$	1	2	4	8	16	32	64	128	...
Rest von $2^k$	1	2	1	2	1	2	1	2	...

Dreierrest von $w$	$ w' $ gerade	$ w' $ ungerade
0	$w'_2 \equiv r \pmod{3}$	$w'_2 \equiv r \pmod{3}$
1	$w'_2 \equiv r + 2 \pmod{3}$	$w'_2 \equiv r + 1 \pmod{3}$
2	$w'_2 \equiv r + 1 \pmod{3}$	$w'_2 \equiv r + 2 \pmod{3}$

Tabelle 1.3: Bedingungen an die Wörter  $w' \in L(w)$  für verschiedene Sprachen  $L_r$  von (1.9).

Die Menge  $L(w)$  besteht also aus den Wörtern  $u$

$$L(w) = \{u \in \Sigma^* \mid w_2 \cdot 2^{|u|} + u_2 \equiv 0 \pmod{3}\},$$

ist also durch den Rest von  $w$  bei Teilung durch 3 vollständig bestimmt. Da es nur drei mögliche Reste gibt, erhalten wir die folgenden drei Mengen:

**Fall  $w_2 \equiv 0 \pmod{3}$ :** Der Rest von  $wu$  ist wegen (1.8) genau der Rest von  $u$ :  $wu_2 \equiv 0 \cdot 2^{|u|} + u_2 \equiv u_2 \pmod{3}$ . Somit ist

$$L(w) = \{u \in \Sigma^* \mid u_2 \equiv 0 \pmod{3}\} = L$$

in diesem Fall.

**Fall  $w_2 \equiv 1 \pmod{3}$ :** Der Rest von  $wu$  hängt zusätzlich davon ab, ob die Länge von  $u$  gerade oder ungerade ist:

$$\begin{aligned} |u| \text{ gerade} &\Rightarrow (1 + u_2 \equiv 0 \pmod{3} \Leftrightarrow u_2 \equiv 2 \pmod{3}) \\ |u| \text{ ungerade} &\Rightarrow (2 + u_2 \equiv 0 \pmod{3} \Leftrightarrow u_2 \equiv 1 \pmod{3}). \end{aligned}$$

**Fall  $w \equiv 2 \pmod{3}$ :** Analog zum Fall  $w \equiv 1 \pmod{3}$  erhält man:

$$\begin{aligned} |u| \text{ gerade} &\Rightarrow (2 + u_2 \equiv 0 \pmod{3} \Leftrightarrow u_2 \equiv 1 \pmod{3}) \\ |u| \text{ ungerade} &\Rightarrow (1 + u_2 \equiv 0 \pmod{3} \Leftrightarrow u_2 \equiv 2 \pmod{3}). \end{aligned}$$

### Andere Reste

Die Analyse der Mengen  $L(w)$  im vorangegangenen Abschnitt war allgemein genug, dass sich jetzt auf die gleiche Weise auch die endlichen Automaten zu den Sprachen

$$L_r = \{w \in \Sigma^* \mid w_2 \equiv r \pmod{3}\} \tag{1.9}$$

bestimmen lassen, wobei  $L_0$  der bereits untersuchte Fall der durch 3 teilbaren Binärzahlen ist.

Für die Mengen  $L(w)$  folgt wieder

$$L(w) = \{u \in \Sigma^* \mid w_2 \cdot 2^{|u|} + u_2 \equiv r \pmod{3}\}.$$

Wieder legt der Dreierrest von  $w$  vollständig fest, welche Wörter  $u$  in  $L(w)$  sind. Und wieder ergeben sich unterschiedliche Bedingungen je nach Länge von  $u$ , die in der Tabelle 1.3 zusammengestellt sind.

## Übungsaufgaben

**1.1.** Wie viele Wörter enthalten die folgenden Sprachen über dem Alphabet  $\Sigma = \{\emptyset, 1\}$ ?

- a)  $L = \{w \in \Sigma^* \mid |w| \leq 5\}$
- b)  $L = \{w \in \Sigma^* \mid |w| \leq n\}$
- c)  $L = \{w \in \Sigma^* \mid |w|_0 \leq 2 \wedge |w|_1 \leq 3\}$
- d)  $L = \{w \in \Sigma^* \mid |w|_0 \leq 1291\}$

**1.2.** In einem endlichen Automaten

$$A = (\{q_1, q_2, q_3, q_4, q_5\}, \{u, d\}, \delta, q_3, \{q_3\})$$

ist die Funktion  $\delta$  durch die folgende Tabelle definiert:

	u	d
$q_1$	$q_1$	$q_2$
$q_2$	$q_1$	$q_3$
$q_3$	$q_2$	$q_4$
$q_4$	$q_3$	$q_5$
$q_5$	$q_4$	$q_5$

- a) Zeichnen Sie das Zustandsdiagramm.
- b) Akzeptiert der Automat das leere Wort  $\varepsilon$ ?
- c) Akzeptiert der Automat ein Wort der Länge 3?
- d) Bestimmen Sie alle Wörter der Länge  $\leq 4$ , die der Automat akzeptiert.
- e) Wie viele verschiedene Wörter akzeptiert der Automat?

**1.3.** Finden Sie einen deterministischen endlichen Automaten, der die Sprache

$$L = \{w = \{\emptyset, 1\}^* \mid |w|_0 \equiv |w|_1 \pmod{3}\}$$

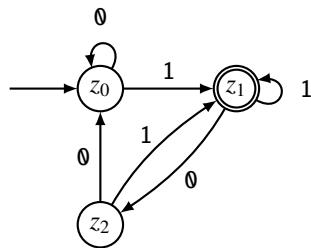
akzeptiert, bestehend aus Wörtern, deren Anzahl von Nullen und Einsen den gleichen Rest bei Teilung durch drei haben.

**1.4.** Konstruieren Sie zu jeder der folgenden Sprachen einen endlichen Automaten, der genau diese Sprache akzeptiert. Die Sprachen verwenden die Alphabete  $\Sigma_1 = \{a, b\}$  und  $\Sigma_2 = \{\emptyset, 1\}$ .

- a)  $L = \{w \in \Sigma_1^* \mid w \text{ beginnt mit } a \text{ und enthält höchstens ein } b.\}$
- b)  $L = \{w \in \Sigma_1^* \mid w \text{ enthält eine gerade Anzahl as und mindestens zwei bs.}\}$

- c)  $L = \{w \in \Sigma_1^* \mid w \text{ hat gerade Länge und eine ungerade Anzahl bs.}\}$
- d)  $L = \{w \in \Sigma_1^* \mid w \text{ enthält nicht genau zwei as.}\}$
- e)  $L = \{w \in \Sigma_2^* \mid \text{An jeder ungeraden Position von } w \text{ steht ein 1.}\}$
- f)  $L = \{w \in \Sigma_2^* \mid |w| \leq 5\}$
- g)  $L = \emptyset \subset \Sigma_2^*$

**1.5.** Untersuchen Sie, welche der Zustände im Automaten



äquivalent sind, und finden Sie den minimalen Automaten.

**1.6.** Betrachten Sie den folgenden, in Tabellenform gegebenen Automaten über dem Alphabet  $\Sigma = \{\emptyset, 1\}$ :

	∅	1
$z$	$e$	$z_1$
$z_0/F$	$z_0$	$z_1$
$z_1$	$z_2$	$z_0$
$z_2$	$z_1$	$z_3$
$z_3$	$z_1$	$z_2$
$e$	$e$	$e$

Zeichnen Sie das Zustandsdiagramm des Automaten. Konstruieren Sie den zugehörigen minimalen Automaten.

Lösungen: <https://autospr.ch/uebungen/AutoSpr-101.pdf>



## Kapitel 2

# Nicht reguläre Sprachen

In Kapitel 1 wurden endliche Automaten erfolgreich für verschiedene Aufgaben eingesetzt. Die Vielfalt von Algorithmen deutet an, dass sich dahinter ein nützliches Werkzeug verbirgt. Dies werden die regulären Ausdrücke in Kapitel 4 noch verdeutlichen. Jedes Werkzeug führt nur dann zu guten Resultaten, wenn es der Aufgabe angemessen ist. Der Anwendungsbereich der deterministischen endlichen Automaten sind die regulären Sprachen. Es ist also wichtig, erkennen zu können, ob eine Sprache regulär ist. Der Myhill-Nerode-Automat von Abschnitt 1.6 kann auch die Information liefern, dass die Sprache  $\{\varnothing^n 1^n \mid n \geq 0\}$  nicht regulär ist. Seine Anwendung ist allerdings schwerfällig. Das Pumping-Lemma von Abschnitt 2.2 nutzt eine einfache Eigenschaft regulärer Sprachen, um nicht reguläre Sprachen sehr viel leichter zu erkennen.

### 2.1 Nicht reguläre Sprachen

Ein endlicher Automat ist eine derart simplistische Art von Maschine, dass man nicht erwarten kann, dass alle Sprachen damit akzeptiert werden können. Es ist daher auch keine Überraschung, dass es nicht reguläre Sprachen gibt. In diesem Abschnitt soll zunächst heuristisch argumentiert werden, von welcher Art von Sprachen man nicht erwarten kann, dass sie regulär sind. Für die Sprache  $L = \{\varnothing^n 1^n \mid n \geq 0\}$  wird anschließend gezeigt, dass diese Heuristik tatsächlich richtig ist und dass es für  $L$  keinen endlichen Automaten geben kann, der  $L$  akzeptiert.

#### 2.1.1 Limitierungen endlicher Automaten

Ein endlicher Automat verfügt nur über endlich viele verschiedene Zustände. Es ist daher für endliche Automaten unmöglich, Wörter aufgrund eines Kriteriums zu unterscheiden, welches mehr verschiedene Ausprägungen hat als die Anzahl vorhandener Zustände des Automaten. Zum Beispiel reichen  $N$  Zustände nicht, um mehr als  $N$  verschiedene Wörter

zu unterscheiden. Es gibt  $N + 1$  Wörter  $\emptyset^k$  mit  $k = 0, \dots, N$ . Ein endlicher Automat mit  $N$  Zuständen kann mindestens zwei davon nicht unterscheiden. Es gibt zwei solche Wörter, nach deren Verarbeitung der Automat im gleichen Zustand ist, obwohl die beiden Wörter verschieden sind.

Für die Sprache

$$L_1 = \{\emptyset^n 1^n \mid n \geq 0\}$$

ist es in diesem Lichte unwahrscheinlich, dass sie von einem endlichen Automaten akzeptiert werden kann. Ein solcher müsste in der Lage sein, die Anzahl  $n$  der Nullen eines Wortes  $w = \emptyset^n 1^n$  zu registrieren und dann nachzuzählen, dass die Anzahl der Einsen damit übereinstimmt. Das gilt auch für die Sprache

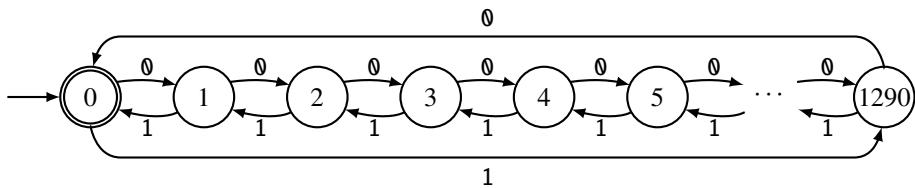
$$L_2 = \{w \in \{\emptyset, 1\}^* \mid |w|_{\emptyset} = |w|_1\},$$

deren Wörter gleich viele Nullen wie Einsen enthalten müssen. Bei der Verarbeitung eines Wortes mit  $n$  Nullen muss diese Anzahl irgendwo gespeichert werden, damit auch die Einsen abgezählt werden können.

Andere Sprachen, wie zum Beispiel die Sprache

$$L_3 = \{w \in \{\emptyset, 1\}^* \mid |w|_{\emptyset} \equiv |w|_1 \pmod{1291}\},$$

könnten bei oberflächlicher Betrachtung auch den Eindruck erwecken, dass zum Akzeptieren von Wörtern die Anzahl der Nullen und Einsen gezählt werden muss. Dies trifft aber nicht zu. Es ist nur nötig, den Rest der Anzahl von Nullen und Einsen modulo 1291 zu ermitteln. Dazu ist ein endlicher Automat sehr wohl in der Lage, denn es gibt nur 1291 verschiedene Reste, die durch die 1291 verschiedenen Zustände des Automaten



abgebildet werden können. Die Zustände zeigen den Rest der Differenz  $|w|_{\emptyset} - |w|_1$  modulo 1291. Genau die Wörter, bei denen  $|w|_{\emptyset}$  und  $|w|_1$  den gleichen Rest haben, werden von dem Automaten akzeptiert.

## 2.1.2 Die Sprache $\{\emptyset^n 1^n \mid n \geq 0\}$

Nach den heuristischen Überlegungen des vorangegangenen Abschnittes dürfte die Sprache

$$L_1 = \{\emptyset^n 1^n \in \{\emptyset, 1\}^* \mid n \geq 0\}$$

über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  nicht regulär sein. Der Satz 1.21 von Myhill-Nerode ermöglicht, dies zu beweisen. Dazu bauen wir die Tabelle 2.1 der Mengen  $L(w)$  für die Wörter  $w \in \Sigma^*$  auf. Die Anfangsstücke  $\emptyset^l$  müssen für alle  $l \in \mathbb{N}$  unterschieden werden können. Sie

$w$	$L(w)$	Zustand
$\epsilon$	$\{01, 0011, \dots, 0^k 1^k \mid k \geq 0\}$	$q_0$
$0$	$\{1, 011, \dots, 0^k 1^{k+1} \mid k \geq 0\}$	$q_1$
$1$	$\emptyset$	$e$
$00$	$\{11, 0111, \dots, 0^k 1^{k+2} \mid k \geq 0\}$	$q_2$
$01$	$\{\epsilon\}$	$p_0$
$10$	$\emptyset$	$e$
$11$	$\emptyset$	$e$
$000$	$\{111, 01111, \dots, 0^k 1^{k+3} \mid k \geq 0\}$	$q_3$
$001$	$\{1\}$	$p_1$
$010$	$\emptyset$	$e$
$011$	$\emptyset$	$e$
$100$	$\emptyset$	$e$
$101$	$\emptyset$	$e$
$110$	$\emptyset$	$e$
$111$	$\emptyset$	$e$
$\vdots$	$\vdots$	$\vdots$
$0001$	$\{11\}$	$p_2$
$\vdots$	$\vdots$	$\vdots$
$0^l$	$\{0^k 1^{k+l} \mid k \geq 0\}$	$q_l$
$\vdots$	$\vdots$	$\vdots$
$0^{k+1} 1$	$\{1^k\}$	$p_k$
$\vdots$	$\vdots$	$\vdots$

Tabelle 2.1: Tabelle der  $L(w)$  für die Sprache  $L_1 = \{0^n 1^n \mid n \geq 0\}$  nach dem Satz 1.21 von Myhill-Nerode zur Bestimmung der Zustände eines endlichen Automaten, der die Sprache  $L_1$  akzeptiert. Die Zustände  $q_k$  und  $p_k$ ,  $k \in \mathbb{N}$ , bilden unendliche Familien von verschiedenen Zuständen. Die Sprache  $L_1$  kann daher nicht von einem deterministischen endlichen Automaten akzeptiert werden und ist daher nicht regulär.

codieren die Anzahl der Nullen, die später durch die gleiche Anzahl Einsen aufgewogen werden müssen. Die Mengen

$$q_l = L(0^l) = \{0^k 1^{k+l} \mid k \geq 0\}$$

sind alle verschieden. Damit haben wir eine unendliche Menge von Zuständen  $q_0, q_1, \dots, q_l, \dots$  gefunden. Insbesondere reichen endlich viele Zustände nicht aus. Es kann daher keinen endlichen Automaten geben, der die Sprache  $L_1$  akzeptiert.

**Satz 2.1.** *Die Sprache  $\{0^n 1^n \mid n \geq 0\}$  ist nicht regulär.*

## 2.2 Pumping-Lemma für reguläre Sprachen

Der Satz 1.21 von Myhill-Nerode war ausreichend, um zu zeigen, dass die Sprache  $\{\emptyset^n 1^n \mid n \geq 0\}$  nicht regulär ist. Die Bestimmung der Mengen  $L(w)$  ist aber eher mühsam und die Identifikation einer unendlichen Menge von nötigen Zuständen nicht selbstverständlich. Daher wäre ein Kriterium nützlich, welches in der Anwendung einfacher ist und die Schlussfolgerung, dass eine Sprache nicht regulär ist, erleichtert.

### 2.2.1 Wörter als Wege durch einen endlichen Automaten

Ein deterministischer endlicher Automat  $A$  verarbeitet ein Wort  $w = a_1 \dots a_n$ , indem ausgehend vom Startzustand des Automaten zu jedem Zeichen  $a_k$  des Wortes mit der Übergangsfunktion  $\delta$  aus dem aktuellen Zustand  $q$  ein neuer Zustand  $\delta(q, a_k)$  berechnet wird. Die Folge

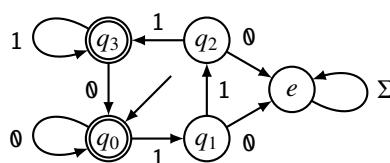
$$q_0, \quad \delta(q_0, a_1), \quad \delta(\delta(q_0, a_1), a_2), \quad \dots, \quad e = \underbrace{\delta(\dots, \delta(\delta(q_0, a_1), a_2), \dots, a_n)}_{=n}$$

von Zuständen ist durch das Wort vollständig bestimmt. Sie beschreibt einen Pfad durch den endlichen Automaten, wie er in Abbildung 1.3 dargestellt ist. Für ein Wort  $w \in L(A)$  der akzeptierten Sprache endet der Pfad in einem Akzeptierzustand des Automaten.

Hat der Automat  $|Q| = N$  Zustände, dann muss der Pfad spätestens nach dem  $N$ -ten Zeichen auf einem Zustand eintreffen, der bereits im Pfad vorkommt. Es gibt also einen Zustand  $q$ , der der erste Selbstkreuzungspunkt des zum Wort  $w$  gehörigen Pfades ist (Abbildung 2.1). Dieser Zustand teilt das Wort in drei Teile  $w = \textcolor{red}{x} \textcolor{blue}{y} \textcolor{violet}{z}$  ein. Der erste Teil, mit  $\textcolor{red}{x}$  bezeichnet, überführt den Automaten vom Startzustand in den Zustand  $q = \delta(q_0, \textcolor{red}{x})$ . Der zweite, mit  $\textcolor{blue}{y}$  bezeichnet, macht aus dem Zustand  $q$  wieder den Zustand  $q$ , also  $q = \delta(q, \textcolor{blue}{y})$ . Der dritte, mit  $\textcolor{violet}{z}$  bezeichnet, führt von  $q$  bis zum Ende des Pfades im Zustand  $e = \delta(q, \textcolor{violet}{z})$ .

Aus der Konstruktion lassen sich auch unmittelbar zwei Eigenschaften der Teile ablesen. Die ersten beiden Teile zusammen haben höchstens die Länge  $N$ , also  $|\textcolor{red}{x} \textcolor{blue}{y}| \leq N$ . Weiter kann der Teil  $\textcolor{blue}{y}$  nicht leer sein, also  $|\textcolor{blue}{y}| > 0$ .

**Verständniskontrolle 2.1:** Die Sprache  $L$ , bestehend aus Wörtern über dem Alphabet  $\Sigma = \{0, 1\}$ , die Einsen immer mindestens als Dreiergruppen enthalten, wird vom deterministischen endlichen Automaten



akzeptiert. Verwenden Sie den Automaten, um die folgenden Wörter wie eben beschrieben in die Teile  $\textcolor{red}{x}$ ,  $\textcolor{blue}{y}$  und  $\textcolor{violet}{z}$  zu zerlegen.

- a)  $w = 111110000$

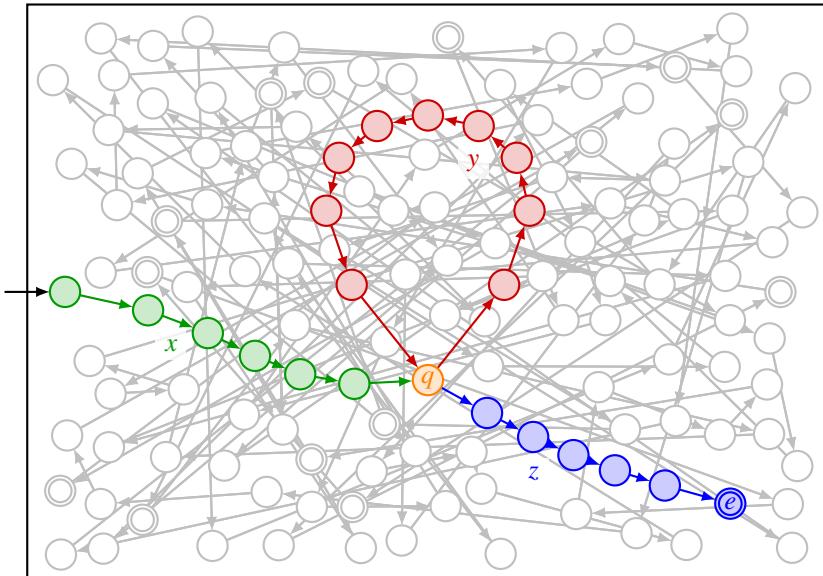


Abbildung 2.1: Pfad durch einen deterministischen endlichen Automaten bestimmt durch ein akzeptiertes Wort. Der erste Selbstkreuzungspunkt  $q$  des Pfades unterteilt das Wort  $w = \text{xyz}$  in drei Teile  $x$  vom Startzustand bis zu  $q$ ,  $y$  von  $q$  bis wieder zurück zu  $q$  und  $z$  von  $q$  bis zum Akzeptierzustand.

b)  $w = 011111000$

c)  $w = 1110$

---

## 2.2.2 Pumpeigenschaft

Ist  $w$  ein genügend langes Wort der Sprache  $L(A)$ , dann lässt es sich wie in Abschnitt 2.2.1 besprochen in die drei Teile  $w = \text{xyz}$  zerlegen. Der zugehörige Pfad im Automaten  $A$  endet in einem Akzeptierzustand. Da das rote Teilstück  $y$  vom Zustand  $q$  wieder zurück zum Zustand  $q$  führt, kann es auch mehrmals durchlaufen werden. Auch das  $k$ -fach durchlaufene Wort  $y^k$  führt von  $q$  wieder zurück nach  $q$ . Daher führt auch  $\text{xy}^k\text{z}$  vom Startzustand zum Akzeptierzustand  $e$ . Somit ist auch  $\text{xy}^k\text{z} \in L(A)$ . Man kann das rote Teilstück sogar ganz weglassen, dann ist  $k = 0$  und  $\text{xy}^0\text{z} = \text{xz} \in L(A)$ . Die genügend langen Wörter der Sprache  $L(A)$  haben daher die folgende Pumpeigenschaft.

**Definition 2.2** (Pumpeigenschaft regulärer Sprachen). Eine Sprache  $L$  hat die Pumpeigenschaft für die Länge  $N$ , wenn sich jedes Wort  $w \in L$  mit Länge  $|w| \geq N$  in drei Teile  $w = \text{xyz}$  mit den Eigenschaften  $|\text{xy}| \leq N$  und  $|\text{y}| > 0$  zerlegen lässt, so dass die gepumpten Wörter  $\text{xy}^k\text{z} \in L$  für alle  $k \in \mathbb{N}$  sind.

### 2.2.3 Das Pumping-Lemma für reguläre Sprachen

Zu einer regulären Sprache  $L$  gibt es nach Definition einen endlichen Automaten  $A$ , der sie akzeptiert. Es ist also  $L = L(A)$ . Da es sich um einen deterministischen endlichen Automaten handelt, ist die Zahl  $N$  der Zustände endlich. Nach Abschnitt 2.2.1 lässt sich auf dem zum Wort  $w$  gehörigen Pfad ein Zustand finden, der mehr als einmal überquert wird. Er hat nach 2.2.2 die Pumpeigenschaft für die Länge  $N$  zur Folge. Es gilt daher der folgende Satz.

**Satz 2.3** (Pumping-Lemma für reguläre Sprachen). *Ist die Sprache  $L$  regulär, dann gibt es eine Zahl  $N > 0$ , die Pumping-Length, mit der Eigenschaft, dass Wörter  $w \in L$  mit  $|w| \geq N$  in drei Teile  $w = xyz$  aufgeteilt werden können mit den Eigenschaften*

1.  $|y| > 0$
2.  $|xy| \leq N$
3.  $xy^kz \in L$  für alle  $k \in \mathbb{N}$ .

Man beachte, dass der Satz nur die Implikation “regulär  $\Rightarrow$  pumpbar” behauptet. Die Umkehrung gilt im Allgemeinen nicht. In Abschnitt 2.3 wird ein Gegenbeispiel gezeigt, eine Sprache, die die Pumpeigenschaft hat, aber trotzdem nicht regulär ist.

### 2.2.4 Die Sprache $\{\emptyset^n 1^n \mid n \geq 0\}$

Wir wissen bereits aus Satz 2.1 mit Sicherheit, dass die Sprache  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  nicht regulär ist. Wir können jetzt aber auch das Pumping-Lemma 2.3 für den Beweis verwenden.

*Beweis.* Wir führen den Beweis, indem wir die Annahme,  $L$  sei regulär, zu einem Widerspruch führen.

1. Annahme:  $L$  ist regulär.
2. Nach dem Pumping-Lemma 2.3 gibt es die Pumping-Length  $N \in \mathbb{N}$ .
3. Wir wählen das Wort  $w = \emptyset^N 1^N$  und versuchen zu zeigen, dass es die Pumpeigenschaft nicht hat. Wir können es graphisch als

$$w = \begin{array}{cccccc|cccc} & & & & & & N & & & 2N \\ & 0 & 0 & \emptyset & 0 & 0 & | & 1 & 1 & 1 & 1 \end{array}$$

darstellen.

4. Nach dem Pumping-Lemma kann das Wort in drei Teile  $xyz$  aufgeteilt werden, wobei  $|y| > 0$  und  $|xy| \leq N$ :

$$w = \begin{array}{c|c|c|c|c|c|c|c} x & 0 & y & 0 & 0 & 0 & 1 & z & 1 & 1 & 1 \\ \hline & & & & & & & & & & \end{array} \begin{array}{c} N \\ | \\ 2N \end{array}$$

Insbesondere besteht der Teil  $y$  ausschließlich aus Nullen.

5. Beim Pumpen wird die Anzahl der Kopien des Teils  $y$  verändert:

$$xy^2z = \boxed{\begin{array}{c} N + |y| \\ \textcolor{red}{x} \textcolor{green}{\emptyset} \quad \textcolor{red}{\emptyset} \textcolor{red}{y} \textcolor{red}{\emptyset} \quad \textcolor{red}{\emptyset} \textcolor{red}{y} \quad \textcolor{blue}{\emptyset} \quad \textcolor{blue}{\emptyset} \quad 1 \textcolor{blue}{z} \quad 1 \quad 1 \quad 1 \end{array}} \qquad 2N + |y|$$

Dadurch ändert sich die Anzahl der Nullen, während die Zahl der Einsen gleich bleibt.

6. Die gepumpten Wörter  $xy^kz \notin L$  sind für  $k \neq 1$  nicht mehr in der Sprache  $L$ , im Widerspruch zur Aussage des Pumping-Lemmas. Der Widerspruch zeigt, dass die Annahme,  $L$  sei regulär, nicht wahr sein kann.  $\square$

Im Beweis hätte man auch den Fall  $k = 0$  des Pumping-Lemmas verwenden können. In diesem Fall wird die Anzahl der Nullen reduziert, die Anzahl der Einsen bleibt aber gleich. Das Wort  $xy^0z = xz$  ist wieder nicht mehr in der Sprache, die Pumpeigenschaft ist verletzt. Da der Teil  $y$  entfernt wurde, spricht man manchmal auch davon, dass er “abgepumpt” worden ist.

Das Pumping-Lemma stellt fest, dass in einer regulären Sprache Wörter genügend großer Länge aufgepumpt werden können. Um zu widerlegen, dass eine Sprache regulär ist, muss man daher nur ein genügend langes Wort finden, welches bei jeder denkbaren Unterteilung in drei Teile gemäß den Regeln des Pumping-Lemmas nicht aufgepumpt werden kann, ohne dass mindestens eines der gepumpten Wörter nicht mehr in der Sprache liegt. Daher hat es im Schritt 3 des Beweises gereicht, ein einzelnes Wort zu konstruieren, welches aber genügend lang sein musste. Im Schritt 4 mussten dann alle möglichen Unterteilungen betrachtet werden.

Es ist nicht zulässig, im Schritt 4 zusätzliche Annahmen über den Inhalt der drei Teile zu machen. Es dürfen nur die Informationen verwendet werden, die sich aus dem Pumping-Lemma und aus der Konstruktion des Wortes ergeben. Zum Beispiel wurde das Wort  $w = \emptyset^N 1^N$  im Beweis so gewählt, dass aus  $|xy| \leq N$  folgt, dass  $y$  aus lauter Nullen besteht. Die Wahl  $w = \emptyset^{N/2+1} 1^{N/2+1}$  hätte auch ein genügend langes Wort ergeben, aber es wäre nicht mehr möglich gewesen, zu schließen, dass der Teil  $y$  in einer zulässigen Unterteilung aus lauter Nullen besteht.

Nach diesem Muster lässt sich die Nichtregularität vieler weiterer Beispiele von Sprachen zeigen.

**Verständniskontrolle 2.2:** Verwenden Sie das Pumping Lemma, um zu zeigen, dass die Sprache

$$L = \{\emptyset^n 1^m \mid 0 < n < m\}$$



nicht regulär ist. Was ändert sich an Ihrem Beweis, wenn nicht nur  $n < m$  gefordert wird, sondern  $n + 1 < m$ ?

### 2.2.5 Die Sprache $\{w \in \{0, 1\}^* \mid |w|_0 = |w|_1\}$

Die Sprache

$$L = \{w \in \{0, 1\}^* \mid |w|_0 = |w|_1\}$$

besteht aus Wörtern, die gleich viele Nullen wie Einsen enthalten. Um ein Wort als zugehörig zu  $L$  zu erkennen, müsste eine Maschine in der Lage sein, die beliebig vielen möglichen Nullen zu zählen und mit der Anzahl der Einsen zu vergleichen. Dazu ist ein endlicher Automat nicht fähig. Dieses heuristische Argument suggeriert, dass  $L$  nicht regulär ist.

Für einen strengen Beweis können wir wieder das Pumping-Lemma bemühen. Dazu ist zu zeigen, dass es ein genügend langes Wort gibt, welches bei jeder beliebigen zulässigen Unterteilung nicht gepumpt werden kann. Da die Sprache  $\{0^n 1^n \mid n \geq 0\}$  in  $L$  enthalten ist, können wir das gleiche Wort  $w = 0^N 1^N$  verwenden, welches sich schon im Beweis in Abschnitt 2.2.4 die Pumpeigenschaft nicht hatte. Der Beweis für die Nichtregularität von  $\{0^n 1^n \mid n \geq 0\}$  ist daher wörtlich auch für die Nichtregularität von  $L$  anwendbar.

## 2.3 Eine pumpbare, nicht reguläre Sprache

Die Tatsache, dass genügend lange Wörter einer regulären Sprache die Pumpeigenschaft haben, heißt nicht, dass eine Sprache mit dieser Eigenschaft automatisch regulär ist. Der folgende Satz gibt ein Gegenbeispiel, also eine Sprache, die die Pumpeigenschaft hat, aber trotzdem nicht regulär ist.

**Satz 2.4.** *Die nicht reguläre Sprache*

$$L = \{a^i b^j c^k \mid i = 0 \vee j = k\}$$

hat die Pumpeigenschaft für die Länge  $N = 1$ .

*Beweis.* Zwei Dingen müssen gezeigt werden:

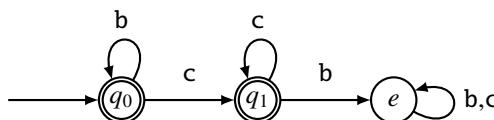
1. Die Sprache  $L$  ist nicht regulär.
2. Jedes Wort mit mindestens einem Zeichen ist aufpumpbar.

Zunächst beachten wir, dass sich die Sprache als die disjunkte Vereinigung

$$L = \underbrace{\{b^j c^k \mid j, k \geq 0\}}_{= L_1} \cup \underbrace{\{a^i b^k c^k \mid i > 0 \wedge k \geq 0\}}_{= L_2}$$

schreiben lässt.  $L_2$  besteht aus den Wörtern von  $L$  die mit a beginnen,  $L_1$  aus allen anderen. Es ist also  $L_2 = L \setminus L_1$ .

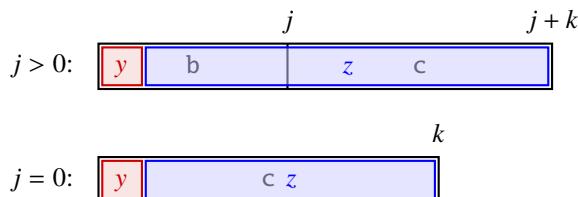
1. Die Sprache  $L_1$  besteht aus den Wörtern, die vom endlichen Automaten



akzeptiert werden und ist daher regulär.

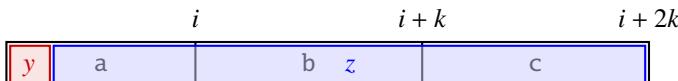
Für die Sprache  $L_2$  betrachten wir die Mengen  $L(ab^j) = \{b^l c^{j+l} \mid l \geq 0\}$ . Sie sind alle verschieden, sie bilden eine unendliche Menge von Zuständen des Myhill-Nerode-Automaten. Mit dem Satz 1.21 von Myhill-Nerode folgt, dass die Sprache  $L_2$  nicht regulär ist. Wäre  $L$  regulär, dann müsste auch  $L \setminus L_1 = L_2$  regulär sein, was aber nicht der Fall ist. Somit ist  $L$  tatsächlich nicht regulär.

2. Wir zeigen jetzt, dass jedes Wort mit mindestens einem Zeichen aufgepumpt werden kann. Die Wörter  $b^j c^k$  der Teilsprache  $L_1$  bestehen nur aus  $b$  und  $c$  und können wegen  $N = 1$  nur mit dem ersten Zeichen als  $y$  wie in



aufgeteilt werden. Sie beginnen also mit einem  $b$  oder  $c$ . Da es in  $L_1$  auf die Anzahl dieser Zeichen nicht ankommt, kann das erste Zeichen  $y$  beliebig aufgepumpt werden. Das gepumpte Wort wird immer noch in der Sprache  $L_1$  sein. Somit hat  $L_1$  die Pumpeigenschaft.

Die Wörter  $a^i b^k c^k$  der Sprache  $L_2$  beginnen mit einem  $a$ . Auch hier muss die Aufteilung



das erste Zeichen, ein  $a$ , als  $y$  verwenden. Pumpst man das erste Zeichen auf, ändert sich nichts daran, dass  $b$  und  $c$  in gleicher Zahl vorkommen. Das aufgepumpte Wort ist also weiterhin in  $L_2$ . Hat das Wort nur ein einziges  $a$  und pumpst man dieses ab, liegt es nicht mehr in  $L_2$ , aber in  $L_1$ . Für Wörter in  $L_1$  kommt es nicht auf die Anzahl der  $b$  und  $c$  an. In jedem Fall bleibt das Wort beim Pumpen in  $L$ .  $\square$

Obwohl die Pumpeigenschaft vorhanden ist, ist die Sprache  $L$  von Satz 2.4 nicht regulär. Regularität einer Sprache kann man also niemals aus der Pumpeigenschaft schließen. Dazu muss man einen endlichen Automaten oder gleichwertig einen regulären Ausdruck (siehe Kapitel 4) vorlegen.

## Übungsaufgaben

**2.1.** Zeigen Sie: Das Komplement einer nicht regulären Sprache ist nicht regulär.

**2.2.** Welche der folgenden Aussagen ist wahr bzw. falsch:

- a) Jede Teilmenge einer regulären Sprache ist regulär.
- b) Jede Obermenge einer regulären Sprache ist regulär.

- c) Die Vereinigung zweier nicht regulärer Sprachen ist nicht regulär.
- d) Die Schnittmenge zweier nicht regulärer Sprachen ist nicht regulär.

**2.3.** Betrachten Sie über dem Alphabet  $\Sigma = \{\emptyset, 1, \_\}$  die Sprache  $L$ , deren Wörter aus zwei durch genau ein Leerzeichen getrennten Binärzahlen bestehen, wobei die zweite größer sein muss als die erste. Zur Sprache gehören also zum Beispiel

$$10000\_\underline{10001}, \quad 0\_\underline{10001}$$

aber nicht

$$10000\_\underline{10000}, \quad 10000\_\_\underline{10001}, \quad 10001\_\underline{0}.$$

Ist  $L$  regulär?

**2.4.** Sei  $\Sigma = \{a, b, \dots, z\}$  das Alphabet bestehend aus allen Kleinbuchstaben. Die Sprache  $L = \{w \in \Sigma^* \mid \text{es gibt zwei verschiedene Buchstaben } a, b \in \Sigma \text{ mit } |w|_a = |w|_b > 1\}$

besteht aus Wörtern, die mindestens zwei Buchstaben mehr als einmal und in gleicher Anzahl enthalten. Die Wörter

$$\text{essen,} \quad \text{rapperswil,} \quad \text{seenachtfest}$$

sind in  $L$ , nicht aber

$$\text{trinken,} \quad \text{pfaeffikon,} \quad \text{montag.}$$

Ist die Sprache  $L$  regulär?

**2.5.** Ist die Sprache

$$L = \{w \in \Sigma^* \mid |w|_{\emptyset} \geq 2^{|w|_1}\}$$

über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  regulär?

**2.6.** In dieser Aufgabe geht es um die Frage, ob man binäre Ganzzahlen mit Hilfe eines endlichen Automaten auf Gleichheit testen kann. Sei  $\Sigma = \{\emptyset, 1, =\}$ . Ist die Sprache

$$L = \{w=w \mid w \in \{\emptyset, 1\}^*\} \tag{2.1}$$

regulär?

*Hinweis.* Bitte beachten Sie die beiden unterschiedlichen Bedeutungen von  $=$  bzw.  $\equiv$  in Formel (2.1). Das eine ist das mathematische Gleichheitszeichen, das andere ein Symbol im Alphabet  $\Sigma$ . Die beiden Zeichen sind zwar in zwei verschiedenen Schriftarten gesetzt, die normalerweise Zeichen von Formelsymbolen zu unterscheiden gestatten, aber im Falle des Gleichheitszeichens sind die Unterschiede sehr klein.

Die Gleichung (2.1) besagt, dass die Sprache  $L$  aus Zeichenketten von Nullen, Einsen und Gleichheitszeichen besteht, die wie folgt strukturiert sind. Sie beginnen mit einer Folge von Nullen und Einsen, dann kommt ein Gleichheitszeichen. Anschließend wird die Folge von Nullen und Einsen nochmals wiederholt.

**2.7.** Ist die Sprache

$$L = \{w \in \{1\}^* \mid |w| \text{ ist prim.}\}$$

regulär?

**2.8.** Sei  $\Sigma = \{0, 1\}$  und

$$L = \{w \in \Sigma^* \mid w \text{ ist die Binärdarstellung einer Primzahl.}\}$$

Ist  $L$  regulär?

**2.9.** Ist die Sprache

$$L = \{1^n + 1^m = 1^{n+m} \mid n, m \geq 0\}$$

über dem Alphabet  $\Sigma = \{1\}$  regulär?

Lösungen: <https://autospr.ch/uebungen/AutoSpr-102.pdf>



# Kapitel 3

## Nichtdeterministische endliche Automaten

Die in Kapitel 1 konstruierten deterministischen endlichen Automaten werden manchmal etwas unübersichtlich, da sie für jedes mögliche Zeichen in jedem Zustand einen Übergang vorsehen müssen. Die in Abschnitt 3.1 eingeführten nichtdeterministischen endlichen Automaten lösen das Problem.

Manchmal ist bei der Verarbeitung einer Zeichenkette nicht klar, wie die Verarbeitung weitergeführt werden soll. Zum Beispiel beginnen Zeilen in Logfiles oft mit einem Zeitstempel und weiteren allgemeinen Informationen, bis nach und nach die relevante Information gefunden wird. Meist werden solche Logfiles ja für menschliche Leser und nicht für die maschinelle Verarbeitung formatiert. Ein deterministischer endlicher Automat müsste also bei der Verarbeitung bereits “wissen”, was später auf der Zeile steht. Im Moment der Verarbeitung ist unbestimmt, welcher Übergang der passende ist. Auch dieses Problem kann ein nichtdeterministischer endlicher Automat lösen.

Nichtdeterministische endliche Automaten führen aber auf neue Schwierigkeiten. Die Simulation eines nichtdeterministischen endlichen Automaten erfordert lange Laufzeiten. Glücklicherweise lässt sich jeder nichtdeterministische endliche Automat mit dem Algorithmus von Abschnitt 3.2 auf einen deterministischen endlichen Automaten zurückführen. Mit einem probabilistischen endlichen Automaten (Abschnitt 3.4) kann man statt definitiver Aussagen, ob ein Wort zu einer Sprache gehört, auch die Wahrscheinlichkeit definieren, mit der der Automat ein Wort als zu einer Sprache zugehörig erkennt. Durch Wiederholung der Verarbeitung kann man beliebig zuverlässige Antworten erhalten.

### 3.1 Nichtdeterminismus

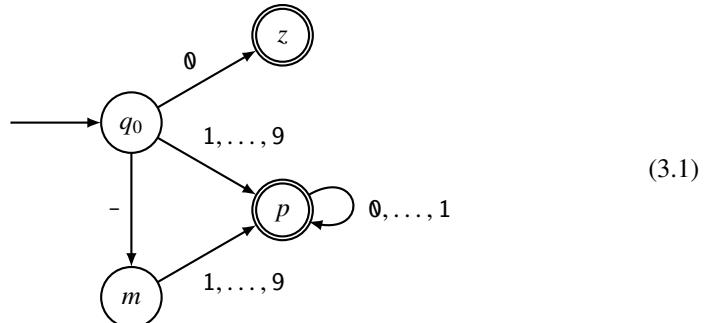
Der erste Entwurf eines Programms ist oft nicht ganz vollständig. Einige Grenzfälle sind vielleicht noch nicht implementiert. In diesen Fällen verhält sich das Programm fehlerhaft,

möglicherweise ist das Verhalten sogar undefiniert. In anderen Fällen könnte zusätzliche Information benötigt werden, die ein anderer Algorithmus liefern könnte, der aber auch noch nicht implementiert ist. In so einem Fall könnte ein Wert zufällig in der Hoffnung gewählt werden, dass der Algorithmus mit etwas Glück damit funktioniert. Oder der Entwickler könnte den Benutzer oder eine AI-Schnittstelle konsultieren, um die Informationslücke zu füllen. Wie auch immer das Problem gelöst wird, die Lösung ist nicht Teil der aktuell zu programmierenden Verarbeitung. Das aktuelle Programm ist nicht in der Lage den weiteren Gang der Rechnung vollständig zu bestimmen. Man spricht von einem nichtdeterministischen Programm.

Nichtdeterminismus deklariert also einen Teil der Problemlösung als von außen kommandiert und untersucht, ob das verbleibende Problem gelöst werden kann. Im Falle endlicher Automaten wird sich Nichtdeterminismus immer als eliminierbar herausstellen.

### 3.1.1 Fehlende Übergänge

Der deterministische endliche Automat (1.3), der die Sprache  $L$  der Ganzzahlen erkennt, verwendet den Zustand  $e$  und alle Übergänge, die dorthin führen, nur dazu, mögliche Fehler zu erkennen. Erreicht die Verarbeitung eines Wortes den Zustand  $e$ , dann bleibt der Automat in diesem Zustand und es gibt keine Möglichkeit mehr, einen Akzeptierzustand zu erreichen. Der Zustand  $e$  kann als “Fehlerzustand” betrachtet werden, der signalisiert, dass das verarbeitete Wort einen “Fehler” enthält und daher nicht in der Sprache der Ganzzahlen enthalten sein kann. Statt des einen Zustands  $e$  könnten auch mehrere Zustände verwendet werden, mit denen sich die verschiedenen Arten von Fehlern unterscheiden ließen. Alle diese Zustände ändern allerdings nichts daran, ob ein Wort akzeptiert wird, sie geben nur zusätzliche Information über den Grund. Ist man nur an der Frage interessiert, ob ein Wort akzeptiert wird, könnte man den Automaten auch durch den viel einfacheren Automaten



ersetzen, in dem  $e$  und alle Übergänge dorthin weggelassen sind. In der Tabellendarstellung

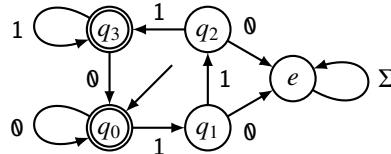
äußern sich die fehlenden Übergänge durch leere Felder in der Tabelle (rot). Der Wert der Übergangsfunktion ausgehend vom Zustand  $z$ , von wo aus es keine Übergänge gibt, ist die leere Menge  $\delta(z, a) = \emptyset$ .

Jede Ganzzahl in  $L$  führt den Automaten (3.1) auf einen Akzeptierzustand, jedes andre Wort kann nicht verarbeitet werden. Das Wort  $0123$  überführt den Automaten mit dem ersten Zeichen  $0$  in den Zustand  $z$ , wo es keinen Übergang für das zweite Zeichen  $1$  gibt. Ähnlich führt das Wort  $-0$  erst zum Zustand  $m$ , dort gibt es aber keinen Übergang für das folgende Zeichen  $0$ .

Wir lernen aus diesem Beispiel Folgendes. Ein akzeptiertes Wort führt vom Startzustand zu einem Akzeptierzustand. Zustände und Übergänge, die nicht zu einem Akzeptierzustand führen können, werden nie erreicht. Lässt man sie weg, bleibt ein Automat, der immer noch alle Wörter der Sprache akzeptieren kann.

Trifft die Verarbeitung eines Wortes auf einen fehlenden Übergang kann die Verarbeitung abgebrochen und das Wort als nicht zur Sprache gehörig erkannt werden. Einen solchen auf das notwendigste reduzierten endlichen Automaten nennen wir einen *nichtdeterministischen endlichen Automaten* (NEA).

**Verständniskontrolle 3.1:** Reduzieren Sie den DEA



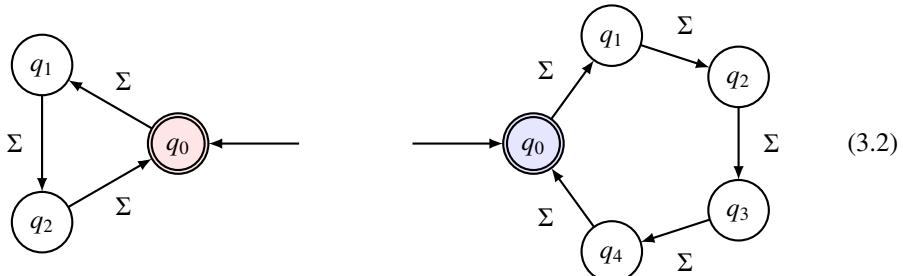
von Verständniskontrolle 2.1, der Wörter akzeptiert, in denen Einsen immer mindestens als Dreiergruppen auftreten, auf einen NEA.

### 3.1.2 Nicht eindeutige Übergänge

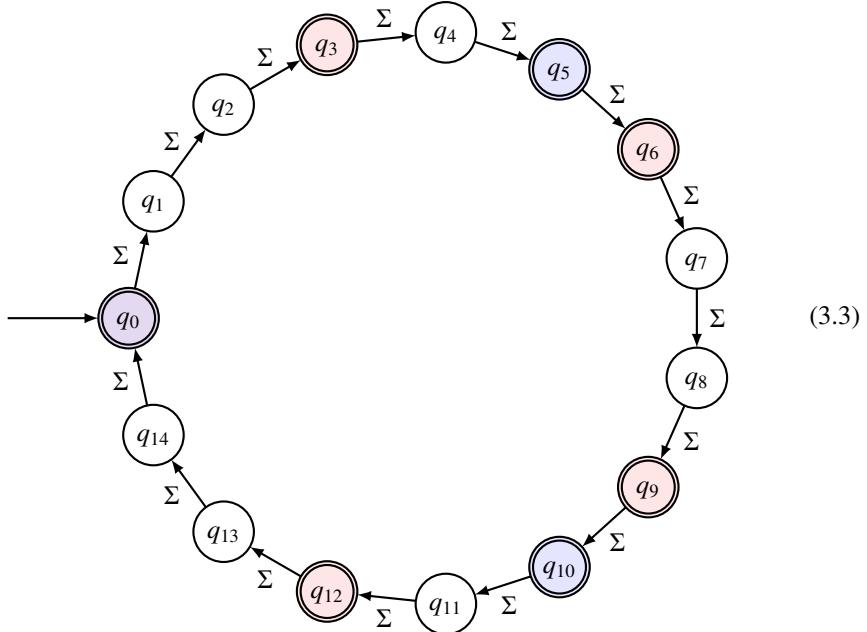
Die Sprache

$$\begin{aligned} L &= \{w \in \Sigma^* \mid |w| \text{ ist durch 3 oder durch 5 teilbar.}\} \\ &= \{w \in \Sigma^* \mid |w| \equiv 0 \pmod{3}\} \cup \{w \in \Sigma^* \mid |w| \equiv 0 \pmod{5}\} \end{aligned}$$

ist als Vereinigung regulärer Sprache wieder regulär und wird daher von einem deterministischen endlichen Automaten akzeptiert. Aus den beiden Automaten



die Wörter durch drei (links) oder durch fünf teilbarer (rechts) Länge akzeptieren, kann man mithilfe der Konstruktion des Produktautomaten den Automaten

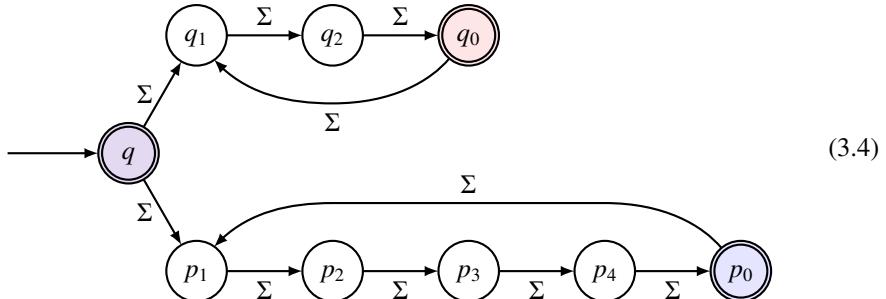


für die Sprache  $L$  finden. Da genau die Zahlen mit Fünfzehnerrest 0, 3, 5, 6, 9, 10 und 12 durch drei oder durch fünf teilbar sind, kann man  $L$  auch schreiben als

$$L = \{w \in \Sigma^* \mid |w| \text{ hat Fünfzehnerrest } 0, 3, 5, 6, 9, 10 \text{ und } 12.\},$$

was den Automaten (3.3) ebenfalls rechtfertigt.

Wäre schon vor der Verarbeitung des Wortes bekannt, ob es durch drei oder durch fünf teilbare Länge hat, dann könnte direkt einer der Automaten (3.2) verwendet werden. Man könnte also den Automaten



mit einer Verzweigung verwenden. Bei der Verarbeitung des ersten Zeichens des Wortes stehen zwei mögliche Übergänge  $q \rightarrow q_1$  oder  $q \rightarrow p_1$  zur Verfügung. Außer für Wörter mit durch 15 teilbarer Länge, für die beide Teilautomaten funktionieren, kann das Wort

nur akzeptiert werden, wenn für den ersten Schritt der “richtige” Übergang gewählt wird. Da die Länge des Wortes bei der Verarbeitung des ersten Zeichens noch nicht bekannt ist, ist auch nicht bekannt, welcher Übergang der “Richtige” ist.

Der zweideutige Übergang des Automaten (3.4) äußert sich in der Tabellendarstellung des Automaten dadurch, dass die zwei möglichen Zielzustände  $q_1$  und  $p_1$  in das gleiche Feld in der Tabelle eingetragen werden müssen. Der Wert der Übergangsfunktion ist also nicht ein einzelner Zustand, sondern eine Menge von Zuständen, im vorliegenden Fall  $\delta(q, a) = \{q_1, p_1\}$ .

---

**Verständniskontrolle 3.2:** Über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  sei die Sprache

$$L = \{w \in \Sigma^* \mid \text{Die letzten zwei Zeichen von } w \text{ sind Nullen.}\}$$

gegeben. Ein endlicher Automat verarbeitet ein Wort von  $L$  von links nach rechts, er “weiß” also nie, ob gerade die zweitletzte  $\emptyset$  verarbeitet wird, oder ob es sich um eine gewöhnliche Null handelt. Zeichnen Sie ein Zustandsdiagramm für  $L$ , welches diese Zweideutigkeit ausdrückt.



Ein endlicher Automat kann möglicherweise einfacher formuliert werden, indem alternative Gründe dafür, ein Wort zu akzeptieren, wie im Beispiel die Teilbarkeit der Länge durch 3 oder 5, durch alternative Pfade im Automaten, im Beispiel (3.4) die Pfade nach oben oder unten, ausgedrückt werden. Nicht alle Pfade führen zu einem Akzeptierzustand, einzelne Pfade mögen sogar auf fehlende Übergänge führen, aber für ein Wort der Sprache gibt es immer mindestens einen Pfad, der zu einem Akzeptierzustand führt.

### 3.1.3 Formale Definition

Die Beispiele der vorangegangenen Abschnitte haben Ansätze illustriert, wie der Begriff eines deterministischen endlichen Automaten erweitert werden kann. Das Einzige, was sich ändert, ist die Spezifikation der Übergangsfunktion. Es wird nicht mehr verlangt, dass es in jedem Zustand für jedes Zeichen einen Übergang gibt. Der Zielzustand  $\delta(q, a)$  darf sogar undefiniert sein. Für Zustände, in denen es mehr als einen möglichen Übergang gibt, muss  $\delta(q, a)$  diese Vielfalt der Möglichkeiten wiedergeben. Dies wird möglich, wenn  $\delta(q, a)$  als Teilmenge  $\delta(q, a) \subset Q$  der Zustandsmenge betrachtet wird.

**Definition 3.1** (nichtdeterministischer endlicher Automat). *Ein nichtdeterministischer endlicher Automat, abgekürzt NEA, ist ein 5-Tupel  $A = (Q, \Sigma, \delta, q_0, F)$ . Die endliche Menge  $Q$  heißt die Menge der Zustände. Die Teilmenge  $F \subset Q$  enthält die Akzeptierzustände,  $q_0$  heißt der Startzustand. Die Funktion*

$$\delta: Q \times \Sigma \rightarrow P(Q) : (q, a) \mapsto \delta(q, a) \subset Q$$

heißt die (nichtdeterministische) Übergangsfunktion.

#### Ein Wort akzeptieren

Die Verarbeitung eines Wortes  $w \in \Sigma^*$  durch einen NEA muss entscheiden, ob das Wort vom Automaten akzeptiert wird. Eine erfolgreiche Verarbeitung des Wortes  $w = a_1 a_2 \dots a_n$

```

boolean akzeptiert(Zustand q, String w) {
    if |w| = 0 {
        if q ∈ F
            return true;
        else
            return false;
    } else {
        a1 := w[0] // nächstes Zeichen in Σ
        w' = w.substr(1); // Rest des Wortes
        foreach q' ∈ δ(q, a1) {
            if akzeptiert(q', w')
                return true;
        }
        return false;
    }
}

```

Abbildung 3.1: Pseudocode für eine rekursive Implementation eines nichtdeterministischen endlichen Automaten. Es handelt sich um einen Backtracking-Algorithmus.

mit Länge  $n = |w|$  ist eine Folge von Zuständen

$$q_0 \xrightarrow{a_1} q_{i_1} \xrightarrow{a_2} q_{i_2} \xrightarrow{a_3} q_{i_3} \xrightarrow{a_4} \dots \xrightarrow{a_{n-2}} q_{i_{n-2}} \xrightarrow{a_{n-1}} q_{i_{n-1}} \xrightarrow{a_n} q_{i_n} \quad (3.5)$$

Jeder Zustand  $q_{i_k}$  muss vom vorangegangenen Zustand  $q_{i_{k-1}}$  aus mit der Übergangsfunktion  $\delta$  erreichbar sein. Dazu muss  $q_{i_k}$  in der Menge der möglichen Zielzustände  $\delta(q_{i_{k-1}}, a_k)$  liegen, es muss also  $q_{i_k} \in \delta(q_{i_{k-1}}, a_k)$  sein für alle  $k = 1, \dots, n$ . Akzeptiert wird das Wort, wenn  $q_{i_n}$  ein Akzeptierzustand ist.

Es ist möglich, dass alle denkbaren Folgen (3.5) von Zuständen in einem Zustand enden, der kein Akzeptierzustand ist. Wenn dies geschieht, wird das Wort nicht akzeptiert. Es ist sogar möglich, dass es gar keine solche Folge gibt. Ist nämlich die Menge  $\delta(q_{i_{k-1}}, a_k)$  die leere Menge, dann bricht die Folge dort ab, weil es keinen möglichen Folgezustand mehr gibt.

### Implementation eines NEA

Wird das Wort  $w = a_1 a_2 \dots a_n \in \Sigma^*$  vom nichtdeterministischen Automaten  $A$  akzeptiert? Da schon der erste Übergang  $\delta(q_0, a_1)$  eine Teilmenge von Zuständen sein kann, muss als Zielzustand des ersten Übergangs ein Zustand in  $\delta(q_0, a_1)$  gewählt werden, von dem ausgehend das verbleibende Wort  $w' = a_2 \dots a_n$  zu einem Akzeptierzustand führt. Dies eröffnet die Möglichkeit einer rekursiven Implementation. Dazu wird eine Funktion `akzeptiert(Zustand q, String w)` mit booleschen Rückgabewerten definiert, welche genau dann `true` zurück gibt, wenn es ausgehend vom Zustand  $q$  einen Pfad durch den Automaten gibt, der auf einen Akzeptierzustand führt.

Für das leere Wort  $w = \varepsilon$  kann `akzeptiert`( $q, \varepsilon$ ) nur dann den Wert `true` zurückgeben, wenn der Zustand  $q \in F$  ein Akzeptierzustand ist.

Ein längeres Wort  $w = a_1 a_2 \dots a_n$  wird in das erste Zeichen  $a = a_1$  und den Rest  $w' = a_2 \dots a_n$  zerlegt. Für einen der möglichen Zustände  $q' \in \delta(q, a_1)$  muss das Wort  $w'$  einen von  $q'$  ausgehenden Weg durch den Automaten ergeben, der in einem Akzeptierzustand endet. Dies wiederum kann der Funktionsaufruf `akzeptiert`( $q', w'$ ) testen. Dieser Algorithmus wird vom Pseudocode in Abbildung 3.1 realisiert. Es handelt sich um einen Backtracking-Algorithmus.

## Laufzeit

Die Laufzeit des Algorithmus kann wie folgt abgeschätzt werden. Der Algorithmus terminiert für ein Wort der Länge  $n$  spätestens beim  $n$ -ten rekursiven Aufruf. In jedem Aufruf, bei dem das zweite Argument nicht das leere Wort ist, müssen in der `foreach`-Schleife so viele Möglichkeiten getestet werden, wie die Menge  $\delta(q, a_1)$  Elemente hat. Diese Anzahl ist beschränkt durch  $|\delta(q, a_1)| \leq |Q|$ . Die Gesamtzahl der getesteten Pfade ist somit beschränkt durch  $O(|Q|^{n-1})$ . Der Aufwand wächst also im schlimmsten Fall exponentiell mit der Wortlänge. Im Gegensatz dazu ist die Laufzeit, die ein DEA zur Verarbeitung eines Wortes braucht immer  $O(n)$ .

## Verifikation

Sei  $w = a_1 a_2 \dots a_n \in \Sigma^*$  ein Wort, welches vom nichtdeterministischen endlichen Automaten  $A$  akzeptiert wird. Dies bedeutet, dass es eine Folge von Zuständen wie in (3.5) gibt, die in einem Akzeptierzustand  $q_{i_n} \in F$  endet. Es kann aufwendig sein, einen solchen Pfad zu finden, aber es ist sehr einfach, zu kontrollieren, ob ein vorgeschlagener Pfad  $(q_0, q_{i_1}, q_{i_2}, \dots, q_{i_n})$  tatsächlich eine korrekte Verarbeitung des Wortes  $w$  ist. Zur Vereinfachung der Notation schreiben wir  $i_0 = 0$ . Es muss nur überprüft werden, ob der Folgezustand  $q_{i_k}$  vom Ausgangszustand  $q_{i_{k-1}}$  aus mit einem Zeichen  $a_k$  erreichbar ist, d. h. es muss

$$q_{i_k} \in \delta(q_{i_{k-1}}, a_k) \quad \forall k = 1, \dots, n, \quad (3.6)$$

gelten. Außerdem muss der letzte Zustand der Folge ein Akzeptierzustand sein, also  $q_{i_n} \in F$ .

Dieses Phänomen, dass das Finden einer Lösung viel schwieriger ist als die Verifikation einer vorgeschlagenen Lösung, wird uns auch in späteren nichtdeterministischen Konstruktionen wie den Stackautomaten in Kapitel 7 und den nichtdeterministischen Turing-Maschinen in Kapitel 12 begegnen. Es manifestiert sich auch in der Tatsache, dass die Simulation eines NEA exponentiell langsamer als ein DEA sein kann.

## Das Orakel

Schon im Beispiel (3.4) wurde darauf hingewiesen, dass es ganz einfach wäre, ein Wort zu akzeptieren, wenn bei jedem nicht eindeutig bestimmten Übergang bereits bekannt wäre, welcher Übergang gewählt werden muss. Man bräuchte ein Orakel, welches man bei jeder Unbestimmtheit danach fragen kann, welcher Weg gewählt werden muss. Für das Beispiel in Abbildung 3.2 braucht man die Hilfe des Orakels beim ersten Übergang vom Zustand

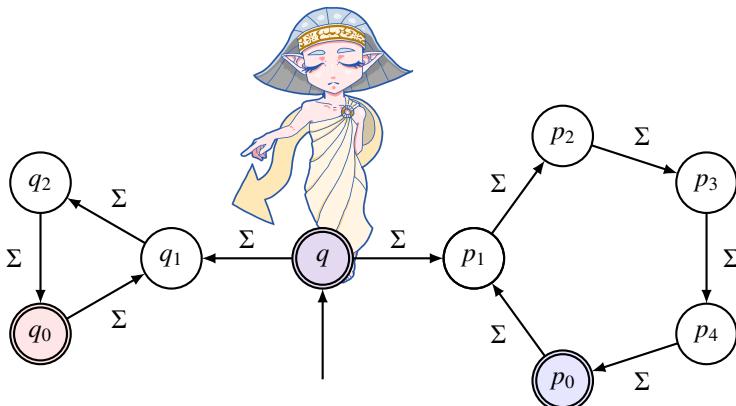


Abbildung 3.2: Ein Orakel kann behilflich sein, die korrekte Folge von Zuständen zu finden, mit denen das Wort akzeptiert werden kann. Bei jedem nicht eindeutig bestimmten Übergang wird das Orakel nach dem zu wählenden Weg gefragt. Führt der Weg trotzdem nicht zu einem Akzeptierzustand, gehört das Wort nicht zur akzeptierten Sprache.

$q$  aus. Man kann sich das Orakel als “gute Fee” vorstellen, welches immer den richtigen Weg weist, wenn es tatsächlich einen Weg zu einem Akzeptierzustand gibt.

Da das Orakel immer eine Antwort gibt, muss man selbst nachprüfen, ob die Antwort des Orakels tatsächlich zu einem Akzeptierzustand führt. Dies ist aber, wie im vorangegangenen Abschnitt beschrieben, immer möglich.

## 3.2 Umwandlung in einen DEA

In Abschnitt 3.1.3 wurde ein Algorithmus beschrieben, der im schlimmsten Fall exponentielle Laufzeit  $O(|Q|^{|w|-1})$  braucht, um zu entscheiden, ob ein Wort von einem nichtdeterministischen endlichen Automaten akzeptiert werden kann. Damit geht die besonders attraktive Eigenschaft der deterministischen endlichen Automaten verloren, dass sie die Zugehörigkeit eines Wortes zu einer regulären Sprache in der Zeit  $O(|w|)$  entscheiden können. In diesem Abschnitt wird gezeigt, wie jeder nichtdeterministische endliche Automat mit einer Konstruktion, die der Thompson-NEA genannt wird, in einen deterministischen endlichen Automaten umgewandelt werden kann. Er zeigt, dass Nichtdeterminismus bei endlichen Automaten eliminierbar ist.

### 3.2.1 Zustandsmengen als Zustände

Sei  $A = (Q, \Sigma, \delta, F, q_0)$  ein nichtdeterministischer endlicher Automat und  $w = a_1 a_2 \dots a_n \in \Sigma^*$  ein Wort, von welchem untersucht werden soll, ob es von  $A$  akzeptiert wird. Gemäß dem in Abbildung 3.1 beschriebenen Algorithmus muss bei jedem nicht eindeutig bestimmten Übergang untersucht werden, welcher der möglichen Folgezustände schließlich zu einem Akzeptierzustand führen wird. Für das erste Zeichen ist die Menge der möglichen Folgezustände gleich der Menge  $\delta(q_0, a_1)$ . Nach der Verarbeitung des zweiten Zeichens könnte

der Automat in jedem Zustand der Vereinigung

$$\bigcup_{q' \in \delta(q_0, a_1)} \delta(q', a_2)$$

aller Zustände sein, die von den Zuständen in  $\delta(q_0, a_1)$  aus erreichbar sind.

Die Menge der Zustände, in denen der Automat sein könnte, wird also mit jedem Zeichen komplizierter. Es ist daher nötig, die Übergangsfunktion etwas zu erweitern. Es muss möglich sein, statt eines Zustands eine Menge von Zuständen und statt eines Zeichens ein Wort als Argumente anzugeben.

### Übergangsfunktion für Zustandsmengen

Im ersten Schritt soll die Übergangsfunktion  $\delta: Q \times \Sigma \rightarrow P(Q)$  erweitert werden auf eine Funktion, die als erstes Argument eine Teilmenge  $R \subset Q$  von Zuständen hat.

**Definition 3.2** (Übergangsfunktion für Zustandsmengen). *Sei  $R \subset Q$  eine beliebige Teilmenge der Zustandsmenge, dann sei*

$$\delta: P(Q) \times \Sigma : (R, a) \mapsto \delta(R, a) = \bigcup_{q \in R} \delta(q, a)$$

*die Menge aller mit dem Zeichen  $a$  von Zuständen in  $R$  aus erreichbaren Zustände.*

Wir bezeichnen die Funktion wieder mit  $\delta$ , obwohl sie einen anderen Definitionsbereich hat. Da aber für einelementige Zustandsmengen  $\{q\}$

$$\delta(\{q\}, a) = \delta(q, a)$$

gilt, kann man die neue Übergangsfunktion als eine Erweiterung der ursprünglichen betrachten.

### Übergangsfunktion für Wörter

Im zweiten Schritt soll die Übergangsfunktion iteriert werden, so dass als zweites Argument ein Wort  $w \in \Sigma^*$  verwendet werden kann.

**Definition 3.3** (Übergangsfunktion für Wörter). *Für einen endlichen Automaten (deterministisch oder nicht) ist*

$$\delta(R, w) = \delta(R, a_1 a_2 \dots a_n) = \begin{cases} \delta(R, a_1) & \text{für } n = 1 \\ \bigcup_{q \in \delta(R, a_1 \dots a_{n-1})} \delta(q, a_n) & \text{für } n > 1. \end{cases}$$

*Für einelementige Zustandsmengen  $\{q\}$  schreiben wir auch*

$$\delta(q, w) = \delta(\{q\}, w)$$

*für jedes Wort  $w \in \Sigma^*$ .*

Die Menge  $\delta(R, w)$  besteht also aus allen Zuständen, die mit dem Wort  $w$  von Zuständen von  $R$  aus erreichbar sind.

### Akzeptierte Wörter eines Automaten

Die Menge  $\delta(q, w)$  enthält alle Zustände, in denen der Automat nach der Verarbeitung des Wortes  $w$  ausgehend vom Zustand  $q$  aus sein könnte. Ein Wort  $w \in \Sigma^*$  wird akzeptiert, wenn das Wort vom Startzustand  $q_0$  aus zu einem Akzeptierzustand führen kann, wenn also  $\delta(q_0, w)$  einen Akzeptierzustand enthält. Die von  $A$  akzeptierte Sprache ist daher die Menge

$$L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \cap F \neq \emptyset\} \quad (3.7)$$

der Wörter, für die  $\delta(q_0, w)$  die Menge der Akzeptierzustände schneidet.

### Der Zustandsmengen-Automat

Mit den eben eingeführten Bezeichnungen lässt sich jetzt ein neuer Automat

$$A' = (P(Q), \Sigma, \delta, \{q_0\}, \mathcal{F})$$

konstruieren, wobei

$$\mathcal{F} = \{R \subset Q \mid R \cap F \neq \emptyset\} \quad (3.8)$$

die Menge aller Teilmengen von  $Q$  ist, die mindestens einen Akzeptierzustand enthalten. Wegen (3.7) akzeptiert  $A'$  die gleiche Sprache wie  $A$ :  $L(A) = L(A')$ .

**Satz 3.4** (DEA in einen NEA umwandeln). *Zu einem nichtdeterministischen endlichen Automaten  $A = (Q, \Sigma, \delta, q_0, F)$  ist  $A' = (P(Q), \Sigma, \delta, \{q_0\}, \mathcal{F})$  mit  $\mathcal{F}$  wie in (3.8) definiert ein deterministischer endlicher Automat, der die gleiche Sprache akzeptiert.*

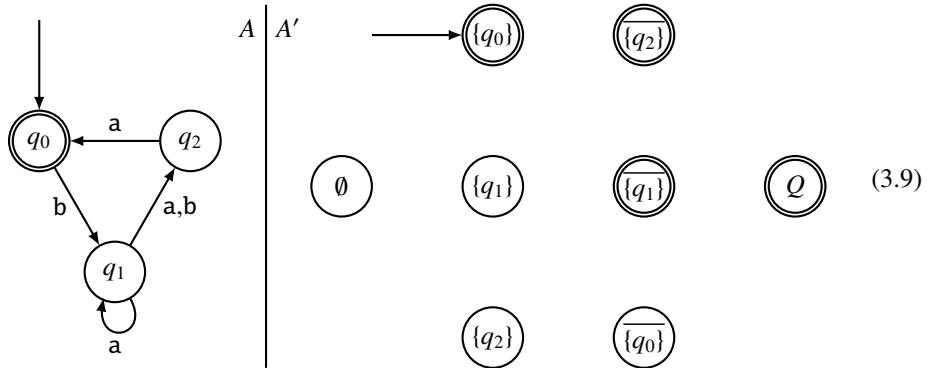
Die Potenzmenge  $P(Q)$  der Zustandsmenge hat  $|P(Q)| = 2^{|Q|}$  Elemente. Selbst für kleine Zustandsmengen ist die Menge der Zustände von  $A'$  sehr groß. Für eine große Anzahl von Zuständen in  $Q$  kann  $P(Q)$  so groß sein, dass eine Implementation die Zustandsmenge  $P(Q)$  niemals aufzählen kann. In NEAs, wie sie in der Praxis tatsächlich auftreten, wird jedoch nur ein kleiner Teil der Mengen in  $P(Q)$  tatsächlich erreicht, was das Problem etwas entschärft.

### 3.2.2 Der Thompson-NEA

Die Definition eines deterministischen endlichen Automaten gemäß Satz 3.4 erklärt nicht, wie die tatsächlich benötigte Menge der erreichbaren Zustände von  $A'$  in  $P(Q)$  bestimmt werden kann. Der Thompson-NEA zeigt einen Weg auf, wie diese Menge bestimmt und zusammen mit der Übergangsfunktion effizient implementiert werden kann. Die Konstruktion geht zurück auf Ken Thompson, der sie im Rahmen seiner Implementation von regulären Ausdrücken (Kapitel 4) im QED-Editor Ende der sechziger Jahre entwickelt hat.

Für die Konstruktion des Thompson-NEA sei  $A = (Q, \Sigma, \delta, q_0, F)$  ein nichtdeterministischer endlicher Automat. Als Beispiel zur Illustration der entwickelten Theorie verwenden

wir den Automaten mit dem Zustandsdiagramm



über dem Alphabet  $\Sigma = \{a, b\}$ . Die Zustände, Akzeptanzzustände und der Startzustand des zugehörigen Automaten  $A'$  von Satz 3.4 ist in (3.9) rechts dargestellt. Die dritte Spalte enthält die zweielementigen Mengen, also zum Beispiel  $\{q_2\} = \{q_0, q_1\}$ .

### Zustandsmengen als Mengen von Markierungen

Zu Beginn der Verarbeitung eines Wortes  $w$  ist der Automat  $A$  im Zustand  $q_0$ , was dem Zustand  $\{q_0\}$  des Automaten  $A'$  entspricht. Im Laufe der Verarbeitung werden weitere Zustandsmengen erreicht. Eine solche Zustandsmenge  $R \subset Q$  kann man dadurch darstellen, dass man die in  $R$  vertretenen Zustände markiert. Abbildung 3.3 zeigt alle Zustände des Automaten  $A'$  für das Beispiel (3.9).

### Die Übergangsfunktion

Die Übergangsfunktion des Automaten kann jetzt für jeden Ausgangszustand  $R \subset Q$  einfach dadurch ermittelt werden, dass die Menge  $\delta(R, a)$  bestimmt wird. Für den Startzustand  $\{q_0\}$  sind die Übergänge

$$\begin{aligned}\delta: (\{q_0\}, a) &\mapsto \emptyset \\ \delta: (\{q_0\}, b) &\mapsto \{q_1\},\end{aligned}$$

die in den gleichen Farben im Zustandsdiagramm (3.10) eingetragen sind. Ausgehend von  $\{q_1\}$  findet man analog

$$\begin{aligned}\delta: (\{q_1\}, a) &\mapsto \{q_1, q_2\} \\ \delta: (\{q_1\}, b) &\mapsto \{q_2\}.\end{aligned}$$

Etwas mehr Aufwand erfordern die mehrelementigen Zustandsmengen. Für  $\overline{\{q_0\}} = \{q_1, q_2\}$  muss man überlegen, was mit beiden Zuständen passiert:

$$\text{a-Übergang : } \left\{ \begin{array}{l} \delta: (\{q_1\}, a) \mapsto \{q_1, q_2\} \\ \delta: (\{q_2\}, a) \mapsto \{q_0\} \end{array} \right\} \Rightarrow \delta: (\{q_1, q_2\}, a) \mapsto \{q_0, q_1, q_2\}$$

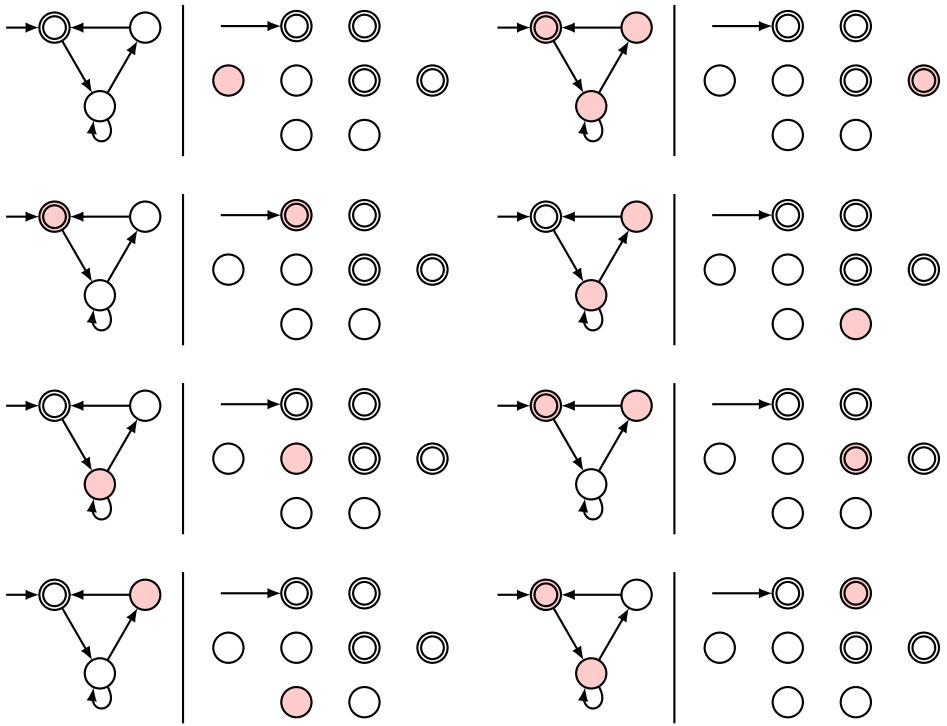


Abbildung 3.3: Zustandsmengen des Automaten  $A'$  als Menge von Markierungen der Zustände des Automaten  $A$  am Beispiel der Automaten von (3.9).

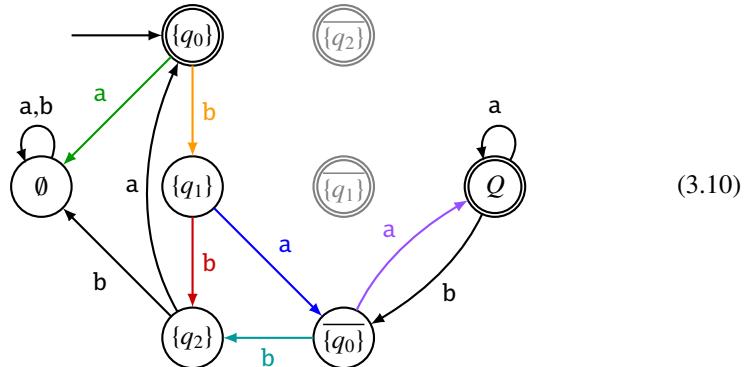


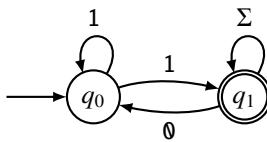
Abbildung 3.4: Thompson-NEA zum nichtdeterministischen Automaten (3.9). Die Zustände sind Mengen von Zuständen und die Übergangsfunktion von  $R \subset Q$  aus ist jeweils die Vereinigung der von den Zuständen in  $R$  erreichbaren Zustände.

$$\text{b-Übergang : } \left\{ \begin{array}{l} \delta : (\{q_1\}, b) \mapsto \{q_2\} \\ \delta : (\{q_2\}, b) \mapsto \emptyset \end{array} \right\} \quad \Rightarrow \quad \delta : (\{q_1, q_2\}, b) \mapsto \{q_2\}.$$

Der Zielzustand ist die Vereinigung der von einzelnen Zuständen  $q_1$  und  $q_2$  aus erreichbaren Zustände.

Durch wiederholte Anwendung dieser Vorgehensweise erreichen wir schließlich das vollständige Zustandsdiagramm (3.10) für den endlichen Automaten  $A'$ . Diese Konstruktion eines DEA aus einem NEA heißt der *Thompson-NEA*. Man beachte, dass die Zustände, die nicht erreicht wurden, grau dargestellt sind und dass die von dort ausgehenden Übergänge nicht ermittelt wurden.

**Verständniskontrolle 3.3:** Verwenden Sie den Thompson-NEA, um den folgenden NEA über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  in einen DEA umzuwandeln:



Zeichnen Sie auch die nicht erreichbaren Zustände und Übergänge.

## Implementation

Es ist nicht nötig, die möglicherweise sehr große Menge der erreichbaren Zustände des Automaten  $A'$  explizit zu konstruieren. Diese Art der Implementation würde zwar den geringsten Rechenaufwand für die Verarbeitung eines einzelnen Zeichens versprechen, dies wird aber erkauft durch einen großen Speicherbedarf für die Zustandsmenge und die Übergangsfunktion.

Stattdessen kann man die Zustände von  $A'$  als mit den Zuständen  $Q$  indizierte Vektoren von booleschen Werten realisieren. Die Menge  $R = \{q_1, q_3, q_4\}$  wird zum Beispiel durch den Vektor

$q_0$	$q_1$	$q_2$	$q_3$	$q_4$	$q_5$
$v[q_0] = \text{false}$	$v[q_1] = \text{true}$	$v[q_2] = \text{false}$	$v[q_3] = \text{true}$	$v[q_4] = \text{true}$	$v[q_5] = \text{false}$

dargestellt. Der Startzustand ist der Vektor, der nur für  $q_0$  den Wert `true` hat. Die Berechnung des Zielzustandes der Übergangsfunktion ausgehend von einem solchen Vektor

$v$  und für das Zeichen  $a$  läuft nach dem folgenden Pseudocode ab:

```

result[q'] = false  $\forall q' \in Q$ ;
foreach  $q \in Q$  {
    if ( $v[q]$ ) {
         $v' = \delta(q, a)$  ;
        foreach  $q' \in Q$  {
            if  $v'[q']$  {
                result[q'] = true;
            }
        }
    }
}
}

```

Zuerst wird der Resultatvektor *result* auf `false` initialisiert. Anschließend wird  $\delta(q, a)$  für jeden Zustand  $q$  berechnet, der in  $v$  den Wert `true` hat. Für alle  $q' \in \delta(q, a)$  wird dann der Resultatvektor auf `true` gesetzt. Dadurch wird die Vereinigung aller  $\delta(q, a)$  für  $q \in Q$  bestimmt. Dies ist zwar deutlich aufwendiger, aber der Aufwand ist immer noch beschränkt durch  $|Q|^2$ .

Der auf diese Art konstruierte deterministische endliche Automat hat wieder Laufzeit  $O(|w|)$  für die Verarbeitung eines Wortes  $w$ . Dies ist immer noch eine Verbesserung gegenüber der im schlimmsten Fall exponentiellen Laufzeit  $O(|Q|^{|w|-1})$  eines nichtdeterministischen endlichen Automaten.

### 3.3 $\varepsilon$ -Übergänge

Bis jetzt waren Zustandsübergänge in einem endlichen Automaten immer an die Verarbeitung eines Zeichens gebunden. Dies ist jedoch nicht unbedingt nötig und ermöglicht, die Beschreibung einer Sprache weiter zu vereinfachen.

#### 3.3.1 Übergänge ohne Zeichenverarbeitung

Für die Sprache

$$L = \{w \in \Sigma^* \mid |w| \text{ ist durch 3 oder 5 teilbar.}\}$$

wurde der Automat (3.4) gefunden. Mit dem ersten Zeichen erfolgt ein nichtdeterministischer Übergang in einen der Teilautomaten, die Wörter mit durch 3 bzw. durch 5 teilbarer Länge akzeptieren. Der zusammengesetzte Automat (3.4) ist immer noch etwas kompliziert, weil das für den ersten Übergang “verbrauchte” Zeichen berücksichtigt werden muss. Daher verzweigt der erste Übergang in die Zustände  $q_1$  und  $p_1$ , statt der Startzustände  $q_0$  und  $p_0$  der ursprünglichen Teilautomaten von (3.2). Die Konstruktion könnte vereinfacht werden, wenn ein Übergang auch ohne die Verarbeitung eines Zeichens möglich wäre.

**Definition 3.5** ( $\varepsilon$ -erweitertes Alphabet). *Sei  $\Sigma$  ein Alphabet, welches das Symbol  $\varepsilon$  nicht enthält. Das um  $\varepsilon$  erweiterte Alphabet ist  $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ .*

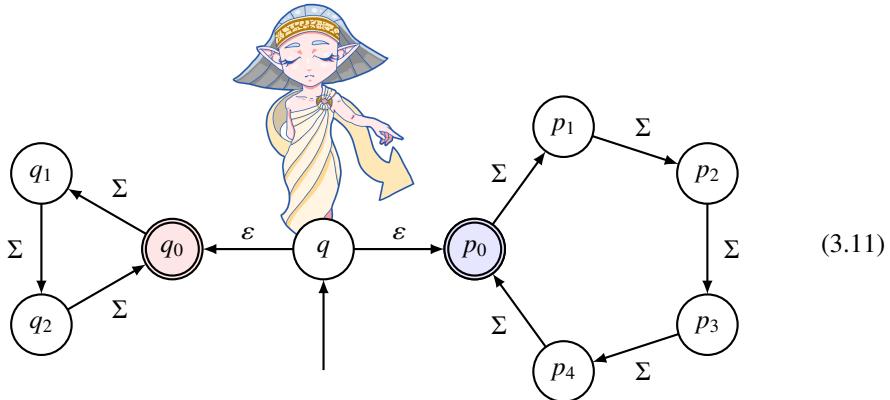
Das Symbol  $\varepsilon$  für das leere Wort erlaubt nun, eine Übergangsfunktion zu konstruieren, welche auch Übergänge ohne ein Alphabetzeichen zulässt.

**Definition 3.6** (NEA mit  $\varepsilon$ -Übergängen). Ein nichtdeterministischer endlicher Automat mit  $\varepsilon$ -Übergängen (NEA $_{\varepsilon}$ ) ist ein 5-Tupel  $A = (Q, \Sigma, \delta, q_0, F)$  mit einer Übergangsfunktion, die auf  $Q \times \Sigma_{\varepsilon}$  definiert ist:

$$\delta: Q \times \Sigma_{\varepsilon} \rightarrow P(Q) : (q, a) \mapsto \delta(q, a).$$

Ein  $\varepsilon$ -Übergang ist ein Übergang von  $q \in Q$  zu einem Zustand in  $\delta(q, \varepsilon)$ . Im Zustandsdiagramm werden solche Übergangspfeile mit  $\varepsilon$  beschriftet.

Bei der Verarbeitung eines Wortes  $w$  können  $\varepsilon$ -Übergänge jederzeit verwendet werden, ohne dass dazu ein Zeichen des Wortes verarbeitet werden muss. Damit kann der Automat (3.4) zu dem etwas einfacheren Automaten



umgebaut werden. Beim Zustand  $q$  liegt wieder Nichtdeterminismus vor. Für die Wahl des richtigen  $\varepsilon$ -Übergangs ist das Orakel zu konsultieren. Der Automat (3.11) konnte konstruiert werden, ohne dass die beiden Teilautomaten modifiziert werden mussten.

Das Akzeptieren eines Wortes und die akzeptierte Sprache können wie für einen gewöhnlichen NEA definiert werden. Da zusätzliche Übergänge zur Verfügung stehen, wird der Akzeptierzustand im Allgemeinen nach einer größeren Anzahl von Übergängen erreicht, als das Wort lang ist.

**Definition 3.7** (Wort akzeptieren in einem NEA $_{\varepsilon}$ ). Ein Wort  $w = a_1 a_2 \dots a_n \in \Sigma^*$  wird von einem NEA $_{\varepsilon}$   $A = (Q, \Sigma, \delta, q_0, F)$  akzeptiert, wenn es eine Folge  $q_0, q_1, \dots, q_m$  mit  $m \geq n$  von Zuständen gibt, derart, dass jeder Übergang  $q_{k-1} \rightarrow q_k$  ein Übergang mit einem Zeichen  $a_i$  oder ein  $\varepsilon$ -Übergang ist. Die Zeichenübergänge verwenden der Reihe nach die Zeichen  $a_1, \dots, a_n$ .

**Definition 3.8** (Akzeptierte Sprache eines NEA $_{\varepsilon}$ ). Ist  $A$  ein NEA $_{\varepsilon}$ , dann ist  $L(A) \subset \Sigma^*$  die Menge der Wörter, die von  $A$  akzeptiert werden.

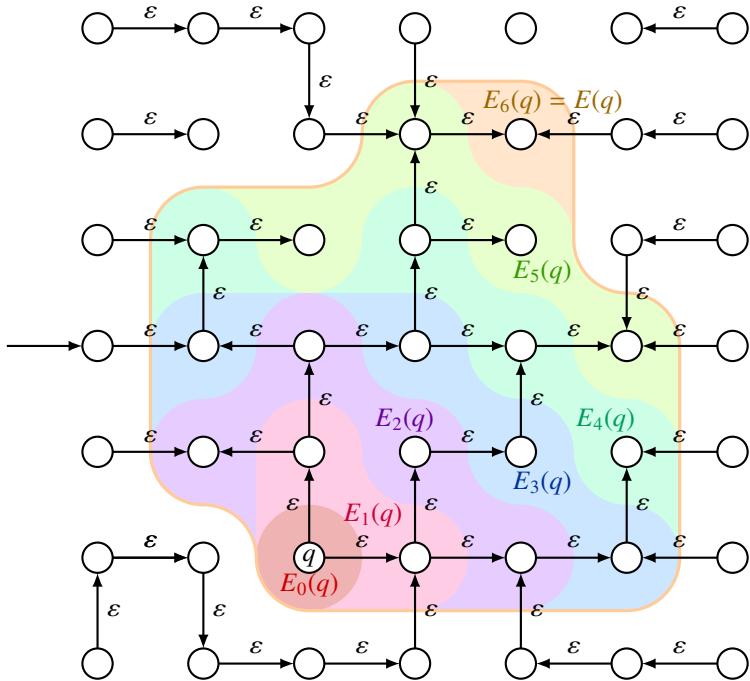


Abbildung 3.5:  $\epsilon$ -erreichbare Zustände in einem endlichen Automaten mit  $\epsilon$ -Übergängen. Nur die  $\epsilon$ -Übergänge sind dargestellt. Die geschachtelten Mengen  $E_0(q) \subset E_1(q) \subset \dots \subset E_k(q) = E(q)$  werden ab einem bestimmten Index nicht mehr größer.

### 3.3.2 Erreichbare Zustände

In Abschnitt 3.1.3 wurde ein Algorithmus beschrieben, der entscheiden kann, ob ein Wort von einem NEA akzeptiert wird. Dazu wurde für jedes Zeichen die Menge der Zustände bestimmt, in denen der Automat nach der Verarbeitung dieses Zeichens sein kann. Dazu musste zunächst die Menge  $\delta(R, a)$  der Zustände ermittelt werden, die von den Zuständen in  $R$  aus durch Verarbeitung des Zeichens  $a$  erreicht werden können. In einem  $\text{NEA}_\epsilon$  stehen weitere Übergänge zur Verfügung, die die Menge der erreichbaren Zustände vergrößern können.

**Definition 3.9** (In  $k$   $\epsilon$ -Übergängen erreichbare Zustände). Sei  $A = (Q, \Sigma, \delta, q_0, F)$  ein  $\text{NEA}_\epsilon$ . Wir definieren die Menge  $E_k$  der in  $k$   $\epsilon$ -Übergängen erreichbaren Zustände wie folgt (siehe auch Abbildung 3.5).

1. Ist  $q$  ein Zustand, dann sei  $E_0(q) = \{q\}$ . Ist  $R \subset Q$  eine Menge von Zuständen, dann ist  $E_0(R) = R$ .
2.  $E_1(q) = \delta(q, \epsilon)$  und  $E_1(R) = R \cup \delta(R, \epsilon) = E_0(R) \cup \delta(E_0(R), \epsilon)$ .
3. Die Mengen  $E_k(q)$  und  $E_k(R)$  sind rekursiv durch

$$E_k(q) = E_1(E_{k-1}(q)) = E_{k-1}(q) \cup \delta(E_{k-1}(q), \epsilon)$$

bzw.

$$E_k(R) = E_1(E_{k-1}(R)) = E_{k-1}(R) \cup \delta(E_{k-1}(R), \varepsilon)$$

definiert.

Die Mengen  $E_k(R)$  umfasst alle Zustände, die in höchstens  $k$  Schritten ausschließlich mit  $\varepsilon$ -Übergängen erreicht werden können (Abbildung 3.5). Insbesondere werden die Mengen  $E_k(R)$  mit zunehmendem  $k$  immer größer, es gilt also

$$\begin{aligned} \{q\} &\subset E_1(q) \subset E_2(q) \subset \dots \subset E_{k-1}(q) \subset E_k(q) \subset \dots \subset Q \\ R &\subset E_1(R) \subset E_2(R) \subset \dots \subset E_{k-1}(R) \subset E_k(R) \subset \dots \subset Q. \end{aligned}$$

Da eine Folge von endlichen Mengen, die alle in einer gemeinsamen endlichen Obermenge enthalten sind, nicht beliebig groß werden kann, ändert sich die Mengen  $E_k(R)$  ab einem gewissen  $k$  nicht mehr. Diese maximale Menge von Zuständen wird  $E(R)$  bezeichnet.

**Definition 3.10** ( $\varepsilon$ -erreichbare Zustände). *Die Menge aller von einem Zustand  $q$  oder einer Zustandsmenge  $R \subset Q$  mit  $\varepsilon$ -Übergängen erreichbaren Zustände ist*

$$E(q) = \bigcup_{k=0}^{\infty} E_k(q) \quad \text{und} \quad E(R) = \bigcup_{k=0}^{\infty} E_k(R).$$

Die Menge der durch Verarbeitung eines Zeichens erreichbaren Zustände wird durch die Anwesenheit von  $\varepsilon$ -Übergängen ebenfalls vergrößert. Wir definieren die Übergangsfunktion für ein Wort neu, um diese zusätzlich möglichen  $\varepsilon$ -Übergänge zu berücksichtigen.

**Definition 3.11.** *Sei  $A = (Q, \Sigma, \delta, q_0, F)$  ein NEA $_{\varepsilon}$ ,  $R \subset Q$  eine Menge von Zuständen und  $w = a_1 a_2 \dots a_n \in \Sigma^*$  ein Wort. Dann sind die Mengen  $R_k$  definiert durch*

$$R_0 = E(R), \quad R_1 = E(\delta(R_0, a_1)), \quad R_2 = E(\delta(R_1, a_2)), \quad \dots, \quad R_n = E(\delta(R_{n-1}, a_n)) \tag{3.12}$$

und die Übergangsfunktion für das Wort  $w$  durch  $\delta(R, w) = R_n$ .

Die roten Blöcke in (3.12) zeigen, dass man die bei der Verarbeitung des Wortes  $w$  erreichbaren Zustände dadurch erhält, dass man nach jedem Übergang mit einem Zeichen  $a_k$  zusätzlich alle mit  $\varepsilon$ -Übergängen erreichbaren Zustände hinzufügt. Das Wort wird akzeptiert, wenn  $\delta(E(q_0), w)$  einen Akzeptierzustand enthält.

### 3.3.3 Thompson-NEA für einen NEA $_{\varepsilon}$

Die Konstruktion des Thompson-NEA ist auch auf einen NEA $_{\varepsilon}$  anwendbar. Es sind nur zwei kleine Anpassungen nötig, die die Definition 3.11 suggeriert.

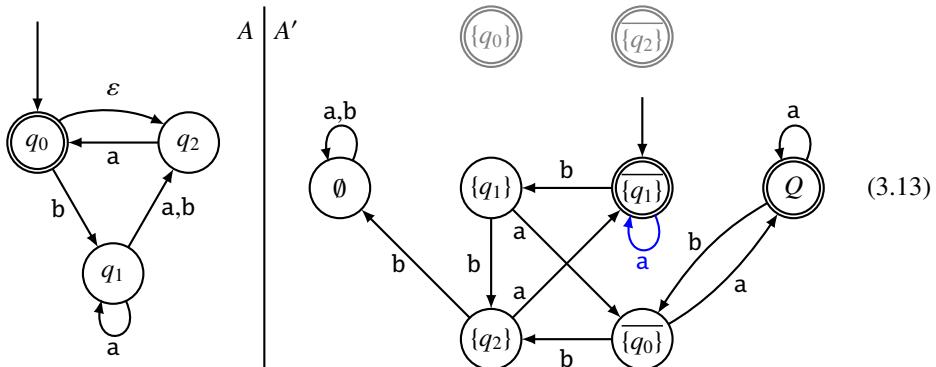
1. Der Startzustand ist nicht mehr die Menge  $\{q_0\}$ , sondern die Menge aller Zustände  $E(q_0)$ , die ohne Verarbeitung eines Zeichens von  $q_0$  aus erreichbar sind.
2. Den nach der Verarbeitung eines Zeichens mit den Übergängen für das Zeichen  $a$  erreichten Zuständen  $\delta(R, a)$  müssen außerdem alle Zustände hinzugefügt werden, die durch  $\varepsilon$ -Übergänge erreicht werden können, dies ergibt  $E(\delta(R, a))$ .

Als Beispiel betrachten wir die Variante (3.13) des Automaten (3.9), die zusätzlich einen  $\varepsilon$ -Übergang von  $q_0$  nach  $q_2$  enthält. Wann immer der Zustand  $q_0$  erreicht wird, kann ohne Verarbeitung eines weiteren Zeichens auch der Zustand  $q_2$  erreicht werden.

Wir führen die Transformation für den  $\text{NEA}_\varepsilon$  (3.13) durch. Der Startzustand für den Automaten  $A'$  besteht nach Punkt 1 aus allen Zuständen  $E(q_0)$ , die von  $q_0$  aus mit  $\varepsilon$ -Übergängen erreichbar sind, also die Menge  $\{q_0, q_2\} = \overline{\{q_1\}}$ . Die Übergänge in  $A'$  aus diesem Zustand sind

$$\begin{aligned}\delta : (\{q_0, q_2\}, a) &\mapsto E(\{q_2\}) = \{q_0, q_2\} \\ \delta : (\{q_0, q_2\}, b) &\mapsto E(\{q_1\}) = \{q_1\}.\end{aligned}$$

Bei der Verarbeitung des Zeichens  $a$  wird vom Zustand  $q_2$  aus der Zustand  $q_0$  erreicht, von dem aus mit dem  $\varepsilon$ -Übergang wieder  $q_2$  erreicht werden kann. Durch wiederholte Anwendung dieser Konstruktion ergibt sich das Zustandsdiagramm

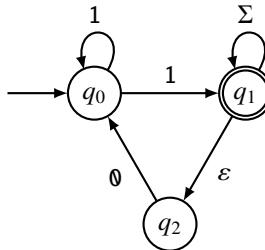


Als Beispiel für den Unterschied, den der  $\varepsilon$ -Übergang in (3.13) macht, sei das erste Zeichen eines Wortes betrachtet. Im ursprünglichen Automaten (3.9) gibt es keinen Übergang für das Zeichen  $a$  ausgehend vom Startzustand  $q_0$ . Im neuen Automaten (3.13) ermöglicht der  $\varepsilon$ -Übergang, ein Zeichen  $a$  am Wortanfang mittels des Umweges über  $q_2$  zu verarbeiten. Dies äußert sich auch im Zustandsdiagramm des zugehörigen DEA. Im Automaten  $A'$  in (3.9) führt der  $a$ -Übergang aus dem Startzustand direkt nach  $\emptyset$ , in dem der Automat verbleiben wird. Ein mit  $a$  beginnendes Wort kann nicht akzeptiert werden. Der Automat  $A'$  in (3.13) dagegen hat einen  $a$ -Übergang vom Startzustand  $\{q_1\}$  zurück zum Zustand  $\{q_1\}$  (blau). Für jedes von  $A'$  akzeptierte Wort  $w$  ist auch  $aw$  ein akzeptiertes Wort.

Das Beispiel illustriert, dass die Konstruktion des Thompson-NEA auch für einen  $\text{NEA}_\varepsilon$  funktioniert. Damit ist der folgende Satz gezeigt.

**Satz 3.12.** Zu jedem  $\text{NEA}_\varepsilon A$  gibt es einen  $\text{DEA} A'$ , der die gleiche Sprache akzeptiert. Die akzeptierte Sprache  $L(A)$  eines  $\text{NEA}_\varepsilon$  ist regulär.

*Verständniskontrolle 3.4:* Verwenden Sie den Thompson-NEA um den folgenden NEA<sub>ε</sub> über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  in einen DEA umzuwandeln.

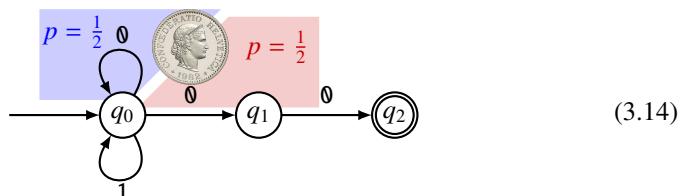


## 3.4 Probabilistische endliche Automaten

Nichtdeterminismus verlangt nach einer externen Informationsquelle, die den Nichtdeterminismus auflösen kann. Statt eines schwer fassbaren Orakels könnte man auch den zu wählenden Übergang erwürfeln.

### 3.4.1 Probabilistische Übergänge

Wir betrachten das Beispiel des Automaten



welcher in der Verständniskontrolle 3.2 für die Sprache

$$L = \{w \in \{\emptyset, 1\}^* \mid \text{die letzten zwei Zeichen von } w \text{ sind Nullen.}\}$$

gefunden worden ist. Der  $\emptyset$ -Übergang beim Zustand  $q_0$  ist zweideutig. In Ermangelung eines Orakels wird eine Münze geworfen, die mit Wahrscheinlichkeit<sup>1</sup>  $p = \frac{1}{2}$  den einen oder anderen Weg wählen wird.

Bei der Verarbeitung des Wortes **100** wird die Münze bei der ersten  $\emptyset$  zum ersten Mal geworfen. Wenn dabei der Übergang zu  $q_1$  gewählt wird, wird der nächste Übergang deterministisch sein und das Wort wird akzeptiert werden. Falls aber der Übergang nach  $q_0$  gewählt wird, kann das Wort nicht akzeptiert werden. Die Wahrscheinlichkeit, dass das Wort **100** akzeptiert wird ist somit  $\frac{1}{2}$ .

<sup>1</sup>Die verwendete Notation zur Wahrscheinlichkeitsrechnung ist in Abschnitt A.4 zusammengefasst.

Sei jetzt  $w$  ein Wort, welches mit zwei Nullen endet, aber davor  $k$  weitere Nullen hat. Die Verarbeitung führt nur dann in den Akzeptierzustand, wenn bei den ersten  $k$  Nullen immer der Übergang nach  $q_0$  gewählt wird. Bei der zweitletzten Null muss dann der Übergang nach  $q_1$  gewählt werden, damit das Wort akzeptiert werden kann. Ein akzeptables Wort wird also mit Wahrscheinlichkeit  $p = 2^{-k-1}$  akzeptiert.

Die Wahrscheinlichkeit, dass ein akzeptables Wort zufällig nicht akzeptiert wird, ist  $1-p$ . Diese Wahrscheinlichkeit ist zwar sehr nahe bei 1, aber durch  $n$ -malige Wiederholung wird die Wahrscheinlichkeit, dass es in allen  $n$  Wiederholungen nie akzeptiert wird  $p^n$ . Durch Vergrößern von  $n$  kann diese Wahrscheinlichkeit beliebig klein gemacht werden.

### 3.4.2 Definition eines probabilistischen endlichen Automaten

Das Beispiel (3.14) hat gezeigt, wie man das Orakel in einem Automaten durch ein Zufallsexperiment ersetzen kann. Das lässt sich ganz allgemein formalisieren.

**Definition 3.13.** Ein probabilistischer endlicher Automat (PEA) ist ein 5-Tupel

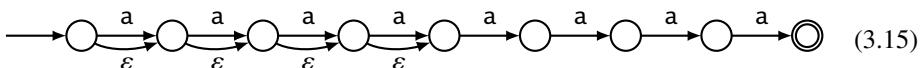
$$A = (Q, \Sigma, \delta, q_0, F)$$

mit Zuständen  $Q$ , Alphabet  $\Sigma$ , Startzustand  $q_0$  und den Akzeptierzuständen  $F \subset Q$ . Die Übergangsfunktion ist eine Zufallsvariable mit Werten in der Menge der Zustände.

Der Wert  $\delta(q, a)$  der Übergangsfunktion ist also nicht eindeutig bestimmt, sondern nimmt die möglichen Zustände  $q' \in Q$  mit der Wahrscheinlichkeit  $P(\delta(q, a) = q') = q'$  an. Ein deterministischer endlicher Automat kann als probabilistischer Automat angesehen werden, bei dem die Werte der Übergangsfunktion immer sicher sind.

### 3.4.3 Vergleich mit der NEA-Implementation

Ob ein Wort  $w$  von einem nichtdeterministischen endlichen Automaten akzeptiert wird, kann durch Durchprobieren aller möglichen Übergänge ermittelt werden. Der Automat



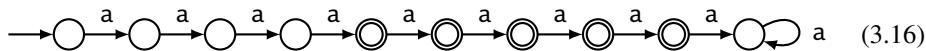
akzeptiert Wörter bestehend aus vier bis acht Zeichen  $a$ . Es gibt nur eine Möglichkeit, das Wort  $a^4$  zu akzeptieren: Es müssen die vier  $\epsilon$ -Übergänge gewählt werden.

Um den nichtdeterministischen endlichen Automaten (3.15) zu simulieren, müssen alle möglichen Übergänge durchprobiert werden. In jedem der ersten vier Zustände sind dies zwei mögliche Übergänge. Je nach Reihenfolge der Versuche kommt die erfolgreiche Wahl erst ganz am Schluss, es ist also mit langer Laufzeit für das Akzeptieren des Wortes zu rechnen.

Gibt man den verschiedenen möglichen Übergängen eine positive Wahrscheinlichkeit, entsteht ein probabilistischer Automat. Dass ein Wort  $w$  akzeptiert wird, ist jetzt ein Bernoulli-Experiment mit einer gegebenen Wahrscheinlichkeit  $p$ . Die Wahrscheinlichkeit, dass das Wort  $w$  nicht akzeptiert wird, ist  $1 - p$  und die Wahrscheinlichkeit, dass es in keiner von  $n$  Wiederholungen akzeptiert wird, ist  $(1 - p)^n$ . Durch Erhöhung der Anzahl der

Wiederholungen kann man also die Wahrscheinlichkeit dafür, dass ein akzeptables Wort nicht als solches erkannt wird, beliebig klein gemacht werden.

Verwandelt man den Automaten (3.15) mit dem Thompson-NEA in einen DEA, entsteht der deterministische Automat



Dieser ist in der Lage, ein Wort  $w$  in Laufzeit  $O(|w|)$  daraufhin zu prüfen, ob es akzeptiert werden kann.

## Übungsaufgaben

**3.1.** Sei  $\Sigma$  ein Alphabet und  $a \in \Sigma$ . Ist die Sprache

$$L = \{w \in \Sigma^* \mid w \text{ enthält an der drittletzten Stelle das Zeichen } a.\}$$

regulär? Falls ja, konstruieren Sie einen nichtdeterministischen endlichen Automaten, der die Sprache akzeptiert.

**3.2.** Konstruieren Sie für jede der folgenden Sprachen über dem Alphabet  $\Sigma = \{0, 1\}$  einen NEA. Reduzieren Sie diesen für die Teilaufgaben a)–e) auf einen minimalen DEA.

- a)  $L_1 = \{w \in \Sigma^* \mid w \text{ enthält mindestens zwei } 1 \text{ nacheinander.}\}$
- b)  $L_2 = \{w \in \Sigma^* \mid w \text{ ist eine durch drei teilbare Binärzahl.}\}$
- c)  $L_3 = L_1 \cap L_2$
- d)  $L_4 = L_1 \setminus L_2$
- e)  $L_5 = L_1 \cup L_2$

Lösungen: <https://autospr.ch/uebungen/AutoSpr-103.pdf>



## Kapitel 4

# Reguläre Operationen und reguläre Ausdrücke

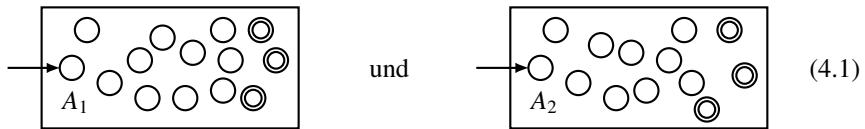
Reguläre Sprachen werden von einem deterministischen endlichen Automaten akzeptiert. Ein solcher ist zwar einfach als Programm zu realisieren, aber dies ist kaum eine praktikabler Weg für die alltägliche Anwendung der Theorie der regulären Sprachen. Dazu wird eine einfache Spezifikationssprache für reguläre Sprachen benötigt. Reguläre Ausdrücke stellen genau dies bereit. Sie basieren auf den regulären Operationen, die in Abschnitt 4.1 dargestellt werden. Eine grundlegende Form regulärer Ausdrücke wird in Abschnitt 4.2 eingeführt. In Abschnitt 4.3 wird gezeigt, wie jeder reguläre Ausdruck in einen deterministischen Automaten umgewandelt werden kann, womit auch die Implementation einer Regex-Engine skizziert ist. Abschnitt 4.4 zeigt dann, dass umgekehrt jede reguläre Sprache mit einem regulären Ausdruck spezifiziert werden kann.

### 4.1 Reguläre Operationen

Der Produktautomat von Abschnitt 1.4 hat ermöglicht, für beliebige reguläre Sprachen  $L_1$  und  $L_2$  deterministische endliche Automaten für die Sprachen  $L_1 \cup L_2$ ,  $L_1 \cap L_2$  und  $L_1 \setminus L_2$  zu konstruieren, was auch zeigt, dass die Mengenoperationen wieder reguläre Sprachen erzeugen. Die Mengenoperationen sind aber nicht die natürlichen Operationen mit Wörtern. In der Praxis gehört die Verkettung von Wörtern zu den wichtigsten Operationen. In den bisher untersuchten Konstruktionen kam sie aber nirgends zur Sprache. In diesem Abschnitt konstruieren wir daher einen zweckmäßigeren Satz von Operationen, der die Mengenoperationen ersetzt und nicht nur die Regularität von Sprachen erhält, sondern, wie wir in Abschnitt 4.4 zeigen werden, alle regulären Sprachen aus einfachen Bausteinen zu erzeugen gestattet.

Im Folgenden seien  $L_1$  und  $L_2$  reguläre Sprachen über einem gemeinsamen Alphabet  $\Sigma$ , die von den Automaten  $A_1$  und  $A_2$  akzeptiert werden. Wir betrachten die Automaten im

Wesentlichen als Black Box und stellen sie symbolisch mit ihren vereinfachten Zustandsdiagrammen



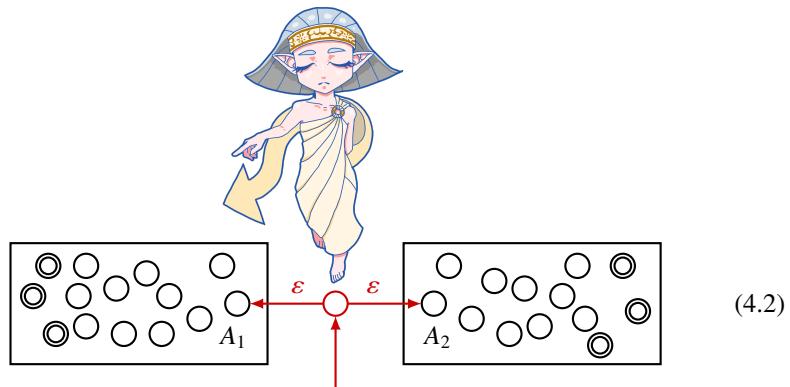
dar. Die Komponenten von  $A_1$  und  $A_2$  bezeichnen wir wenn nötig mit

$$A_k = (Q_k, \Sigma, \delta_k, q_{0k}, F_k),$$

mit  $k = 1, 2$ . Die akzeptierten Sprachen sind  $L_k = L(A_k)$ . Wir nehmen zudem an, dass die Zustandsmengen  $Q_1$  und  $Q_2$  disjunkt sind.

### 4.1.1 Die Alternative

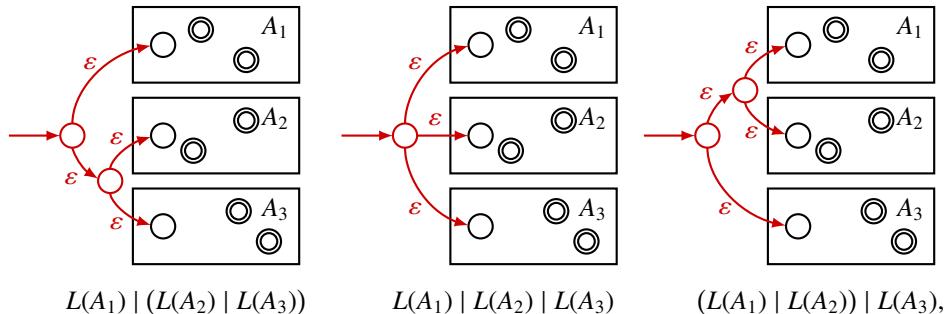
Die Vereinigungsmenge  $L_1 \cup L_2$  enthält alle Wörter, die von  $A_1$  oder von  $A_2$  akzeptiert werden. Wüsste man von Anfang an, ob ein Wort zu  $L_1$  bzw.  $L_2$  gehört, könnte man auch von Anfang an den richtigen Automaten wählen, um zu prüfen, ob das Wort akzeptiert werden kann. Dies ist ein Anwendungsfall für nichtdeterministische  $\varepsilon$ -Übergänge. Der nichtdeterministische Automat



akzeptiert genau die Wörter der Vereinigung  $L_1 \cup L_2$ . Die Vereinigungsmenge  $L_1 \cup L_2$  heißt auch die *Alternative* von  $L_1$  und  $L_2$  und wird mit  $L_1 \mid L_2$  bezeichnet.

Die Alternative ist assoziativ, da die Mengenoperation der Vereinigung assoziativ ist. Auch die Konstruktion eines Automaten für die Alternative ist assoziativ, denn die drei

## Automaten



die durch Anwendung der Automatenkonstruktion für die Alternative in unterschiedlicher Reihenfolge entstanden sind, akzeptieren alle die Vereinigungsmenge  $L(A_1) \cup L(A_2) \cup L(A_3)$ , sie sind also gleichbedeutend.

### 4.1.2 Die Verkettung

In der Einleitung haben wir auf die praktische Bedeutung der Verkettung von Wörtern hingewiesen. Sie gibt Anlass zum Operator der Verkettung der Sprachen  $L_1$  und  $L_2$  und ist wie folgt definiert.

**Definition 4.1** (Verkettung). *Seien  $L_1$  und  $L_2$  Sprachen. Die Verkettung  $L_1L_2$  der Sprachen ist die Menge*

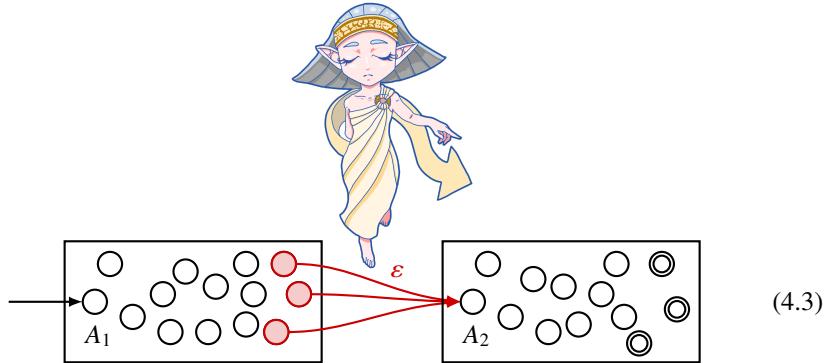
$$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}.$$

*Sie besteht aus allen Verkettungen von Wörtern von  $L_1$  bzw.  $L_2$ .*

Um zu testen, ob ein Wort  $w \in \Sigma^*$  in  $L_1L_2$  ist, muss es in zwei Teilwörter  $w_1 \in L_1$  und  $w_2 \in L_2$  zerlegt werden, welche von den Automaten  $A_1$  bzw.  $A_2$  akzeptiert werden. Das Teilwort  $w_1$  führt also im Automaten  $A_1$  vom Startzustand zu einem Akzeptierzustand. Wo genau das Wort  $w_1$  aufhört, ist nicht deterministisch bestimmbar. Anschließend muss der Rest des Wortes im Automaten  $A_2$  weiterverarbeitet werden und auch dort zu einem Akzeptierzustand führen.

Die eben beschriebene Vorgehensweise kann man auch mit  $\epsilon$ -Übergängen von den Akzeptierzuständen von  $A_1$  zum Startzustand von  $A_2$  realisieren. Die Akzeptierzustände von  $A_1$  müssen dazu zu gewöhnlichen Zuständen abgewertet werden, da sonst die Wörter von  $L_1$  allein akzeptiert würden, obwohl sie nicht notwendigerweise in  $L_1L_2$  liegen. Damit

ist der nichtdeterministische Automat



gefunden, der genau  $L_1 L_2$  akzeptiert. Man beachte, dass die  $\varepsilon$ -Übergänge nichtdeterministisch sind. Wann immer man bei einem Akzeptierzustand von  $A_1$  ankommt, muss das Orakel konsultiert werden, um zu erfahren, ob jetzt der richtige Zeitpunkt ist, in den Automaten  $A_2$  zu wechseln.

Die Verkettung von drei Sprachen  $L_1$ ,  $L_2$  und  $L_3$  ist eine assoziative Operation, denn die Verkettung von Wörtern  $w_i \in L_i$  zu einem Wort

$$w = w_1 w_2 w_3 = (w_1 w_2) w_3 = w_1 (w_2 w_3) \quad (4.4)$$

hängt nicht davon, in welcher Reihenfolge die Verkettungen vorgenommen werden. Die Klammern in (4.4) sind nicht als Zeichen im Alphabet der Sprache zu interpretieren, sondern spezifizieren hier die Reihenfolge der Verkettungsoperationen.

### 4.1.3 Die \*-Operation

Durch Iteration der Verkettungsoperation können nacheinander auch die Mengen

$$L_1, \quad L_1 L_1 = L_1^2, \quad L_1 L_1 L_1 = L_1^3, \quad \dots, \quad L_1^k, \quad L_1^{k+1} = L_1^k L_1, \quad \dots$$

von mehrfachen Verkettungen gebildet werden. Der Vollständigkeit halber definieren wir auch noch  $L_1^0 = \{\varepsilon\}$ .

**Definition 4.2** (\*-Operation). *Sei  $L_1$  eine beliebige Sprache, dann ist die Vereinigung aller Sprachen  $L_1^k$  die Sprache*

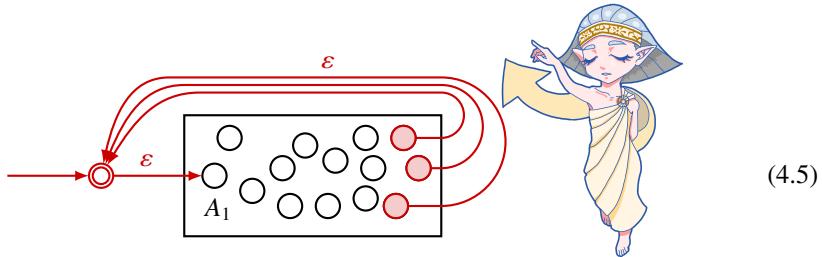
$$L_1^* = \bigcup_{k=0}^{\infty} L_1^k.$$

*Sie heißt die \*-Operation für die Sprache  $L_1$ .*

Die Automatenkonstruktion (4.3) für die Verkettung ist nicht auf die \*-Operation anwendbar. Zwar kann jede Sprache  $L_1^k$  von einem Automaten akzeptiert werden, der durch Verkettungen von  $k$  Kopien des Automaten  $A_1$  entsteht. Aber die \*-Operation würde erfordern, unendlich viele Kopien von  $A_1$  zu verketten, was nicht mehr einen endlichen Automaten ergäbe.

Ein Wort  $w \in L_1^k$  setzt sich zusammen aus Teilwörtern  $w_1, \dots, w_k \in L_1$ , die zusammen  $w = w_1 \dots w_k$  ergeben. Das Wort  $w_1$  führt vom Startzustand von  $A_1$  zu einem Akzeptierzustand. Mit dem nächsten Wort muss jetzt wieder beim Startzustand begonnen werden, das nächste Teilwort führt dann wieder zu einem Akzeptierzustand. Ein endlicher Automat für  $L_1^*$  entsteht also, indem man von den Akzeptierzuständen mit  $\varepsilon$ -Übergängen zum Startzustand zurückführt.

Der Automat für  $L_1^*$  muss aber auch das leere Wort  $\varepsilon$  akzeptieren, was nur möglich ist, wenn der Startzustand ein Akzeptierzustand ist. Wir dürfen aber nicht einfach den Startzustand von  $A_1$  in einen Akzeptierzustand verwandeln, denn dadurch würden auch alle Wörter akzeptabel, die innerhalb  $A_1$  zum Startzustand zurückführen. Die korrekte Konstruktion (4.5) fügt daher einen neuen Startzustand hinzu, der auch ein Akzeptierzustand ist. Von dort führt ein  $\varepsilon$ -Übergang in den Startzustand von  $A_1$ . Von den Akzeptierzuständen von  $A_1$  führen  $\varepsilon$ -Übergänge zurück zum neuen Startzustand. Der nichtdeterministische Automat



akzeptiert also genau die Wörter von  $L_1^*$ . Die  $\varepsilon$ -Übergänge sind wieder nichtdeterministisch: Wann immer man in einem Akzeptierzustand von  $A_1$  vorbeikommt, ist das Orakel zu konsultieren, um herauszufinden, ob an dieser Stelle ein neues Wort  $w_i$  beginnt.

**Verständniskontrolle 4.1:** Über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  sind die Sprachen

$$L_1 = L\left(\begin{array}{c} \downarrow \\ \textcircled{0} \xrightarrow{1} \textcircled{1} \end{array}\right), \quad L_2 = L\left(\begin{array}{c} \downarrow \\ \textcircled{0} \xrightarrow{1} \textcircled{1} \end{array}\right), \quad L_3 = L\left(\begin{array}{c} \downarrow \\ \textcircled{0} \xrightarrow{1} \textcircled{0} \xrightarrow{1} \textcircled{1} \end{array}\right), \quad L_4 = L\left(\begin{array}{c} \downarrow \\ \textcircled{0} \xrightarrow{1} \textcircled{0} \xrightarrow{1} \textcircled{1} \end{array}\right)$$



gegeben. Zeichnen Sie das Zustandsdiagramm für die Sprache  $L = L_1(L_2 \mid L_3)^*L_4$ .

#### 4.1.4 Reguläre Operationen und reguläre Sprachen

Für alle drei Operationen, die Alternative, die Verkettung und die  $*$ -Operation, waren Automatenkonstruktionen möglich, die aus Automaten für die Teilsprachen einen nichtdeterministischen Automaten für die neue Sprache konstruiert haben. Die neuen Sprachen sind also immer auch regulär, wenn die Teilsprachen bereits regulär waren. Dies rechtfertigt die folgende Definition.

**Definition 4.3** (reguläre Operationen). *Die Alternative, die Verkettung und die \*-Operation heißen die regulären Operationen.*

Wir halten das Hauptresultat über die regulären Operationen wie folgt fest.

**Satz 4.4** (Reguläre Operationen von regulären Sprachen). *Seien  $L_1$  und  $L_2$  reguläre Sprachen, dann ist die Alternative  $L_1|L_2 = L_1 \cup L_2$ , die Verkettung  $L_1L_2$  und die \*-Operation  $L_1^*$  regulär.*

### 4.1.5 Abgeleitete Operationen

Aus den drei grundlegenden regulären Operationen lassen sich viele Varianten ableiten.

#### Option

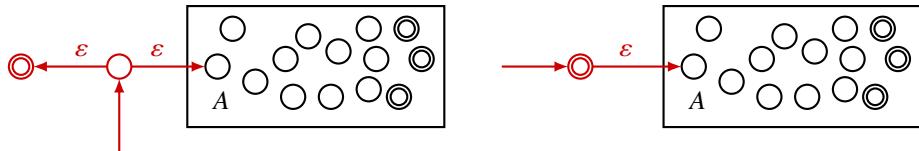
In vielen Anwendungen möchte man ausdrücken können, dass ein Teil eines Wortes vorhanden sein kann, aber nicht vorhanden sein muss, dass er also optional ist.

**Definition 4.5** (Option). *Ist  $L$  eine Sprache, dann ist*

$$L^? = L \mid \{\epsilon\}$$

*die Sprache, in der ein Wort von  $L$  optional ist.*

Ist  $L$  regulär, dann können zum Akzeptieren der Sprache  $L^?$  mindestens die zwei verschiedenen Automatenkonstruktionen



verwendet werden. Die linke Konstruktion ist die Alternative zwischen dem leeren Wort und der Sprache  $L$ . Die rechte ergibt sich daraus durch Zusammenlegen der beiden roten Zustände, denn der  $\epsilon$ -Übergang macht den Startzustand für alle praktischen Zwecke zu einem Akzeptierzustand.

#### +-Operation

Die \*-Operation erlaubt beliebige Verkettungen von Wörtern, dazu gehört insbesondere auch das leere Wort. Es gilt also immer  $\epsilon \in L^*$  selbst dann, wenn die Sprache  $L$  das leere Wort nicht enthält. Wenn dies nicht gewünscht wird, kann die +-Operation verwendet werden.

**Definition 4.6** (+-Operation). *Ist  $L$  eine Sprache, dann besteht die Sprache*

$$L^+ = LL^* = L^*L = \bigcup_{k=1}^{\infty} L^k \quad (4.6)$$

*aus Verkettungen von mindestens einem Wort von  $L$ . Dies ist die +-Operation.*

Für die  $*$ -Operation  $L^*$  wurde bereits ein Automatenkonstruktion angegeben. Die  $+$ -Operation kann gemäß (4.6) auf zwei Arten durch eine Automatenkonstruktion realisiert werden: Einerseits als Verkettung des Automaten  $A$  mit der  $*$ -Konstruktion für  $A$ , oder andererseits als Verkettung in umgekehrter Reihenfolge.

### Beschränkte Iteration

Die  $*$ -Operation und die  $+$ -Operation konstruieren unbeschränkte Verkettungen von Wörtern der Sprache  $L$ . Die Unbeschränktheit machte die Konstruktion (4.5) eines Automaten notwendig. Aus praktischer Sicht ist es wünschenswert, auch Verkettungen einer beliebigen beschränkten Anzahl von Wörtern von  $L$  zu einer neuen Sprache zusammenfassen zu können.

**Definition 4.7** (beschränkte Iteration). *Ist  $L$  eine reguläre Sprache, dann ist*

$$L^{\{n,m\}} = L^n \cup L^{n+1} \cup \dots \cup L^m = \bigcup_{k=n}^m L^k$$

*die Sprache der Wörter, die durch Verkettung von zwischen  $n$  und  $m$  Wörtern von  $L$  entstehen. Lässt man eine der Schranken weg, entstehen die Sprachen*

$$L^{\{n,\}} = \bigcup_{k=n}^{\infty} L^k \quad \text{und} \quad L^{\{,\}m} = \bigcup_{k=0}^m L^k.$$

Die  $*$ -Operation und die  $+$ -Operation sind Spezialfälle dieser allgemeineren Iterationskonstruktion, es gilt

$$L^* = L^{\{0,\}} \quad \text{und} \quad L^+ = L^{\{1,\}}.$$

Ebenso kann die Option als  $L^? = L^{\{0,1\}}$  ausgedrückt werden.

## 4.2 Reguläre Ausdrücke

Jedem Computeranwender sind Wildcards wie `*.txt` für die Beschreibung einer großen Zahl von passenden Filenamen bekannt. Benutzer von Datenbanken haben möglicherweise SQL-LIKE-Patterns kennengelernt, mit denen man in einer Tabellenspalte nach Wörtern suchen kann, die einem bestimmten Muster folgen. Diese Arten von Musterbeschreibung sind jedoch ziemlich primitiv. Natürlich besteht die Gefahr, dass komplexere Muster auch deutlich mehr Rechenaufwand benötigen, um eine Übereinstimmung festzustellen. Es stellt sich also das Problem, eine Musterbeschreibung zu konzipieren, die eine möglichst große Klassen von Sprachen spezifizieren kann, aber trotzdem effizient implementiert werden kann.

Die regulären Sprachen können von deterministischen endlichen Automaten erkannt werden, sie sind also immer in einer Zeit auswertbar, die proportional zur Wortlänge ist. Sie sind unter Mengenoperationen abgeschlossen. Weiter ermöglichen die regulären Operationen, mit einer intuitiven und leicht lesbaren Notation aus regulären Sprachen neue



Abbildung 4.1: XKCD-Webcomic #208 vom 10. Januar 2007 zu regulären Ausdrücken [36]. Die Skriptsprache Perl, auf die im zweitletzten Bild angespielt wird, hat mit ihrer besonders reichhaltigen Syntax für reguläre Ausdrücke deren Anwendung in den Neunzigerjahren nachhaltig geprägt.  
© 2007 xkcd.com

reguläre Sprachen zu konstruieren. Hier liegt also eine sehr große Klasse von Sprachen vor, für die es eine effiziente Implementation gibt. Was jedoch noch fehlt, ist eine einfache maschinenlesbare Notation, die für Suchfunktionen in Editoren, Textverarbeitungswerkzeugen und Datenbanken eingesetzt werden kann.

In diesem Abschnitt soll die Notation zu einer vollwertigen Spezifikation von regulären Sprachen ausgebaut werden, die sich in der Praxis für die Mustersuche in Text bewährt hat. XKCD hat über diese sogenannten *regulären Ausdrücke* den Comic von Abbildung 4.1 gemacht. Dank der bisher entwickelten Theorie können reguläre Ausdrücke sehr effizient mit endlichen Automaten implementiert werden (Abschnitt 4.3). Der Beweis, dass sich damit sogar jede beliebige reguläre Sprache spezifizieren lässt, muss allerdings bis Abschnitt 4.4 warten.

### 4.2.1 Zeichenketten zur Spezifikation von Sprachen

Die in diesem Abschnitt entwickelten regulären Ausdrücke sind Zeichenketten über einem vorgegebenen Alphabet, welche reguläre Sprachen beschreiben sollen. Ein *regulärer Ausdruck*  $r$  ist also eine prägnante Notation für eine reguläre Sprache, die wir mit  $L(r)$  bezeichnen werden. Die Schreibweise soll so gestaltet sein, dass man sie intuitiv für die Textsuche verwenden kann. Sucht man zum Beispiel nach dem Wort “Ausdruck”, ist man sich gewohnt, in ein Suchfeld die Zeichenkette Ausdruck einzugeben. Man möchte also für die Suche nach einem Wort das Wort selbst als regulären Ausdruck  $r = \text{Ausdruck}$  verwenden können.

### 4.2.2 Primitive Ausdrücke

Eine Suche nach dem Wort `muster` kann aber im Licht der regulären Operationen bereits als eine Verkettung betrachtet werden. Als Erstes wird daher eine Notation für einzelne Zeichen benötigt. Wir betrachten die Zeichenkette  $r = m$  als primitiven *regulären Ausdruck*, der genau die Sprache

$$L(r) = L(m) = \{m\}$$

spezifiziert.

Die meisten Musterspezifikationsnotationen ermöglichen die Suche nach einem beliebigen Zeichen auszudrücken. SQL-LIKE-Patterns zum Beispiel verwenden dafür den Unterstrich `_`, die Filenamensmuster das Fragezeichen `?`. Reguläre Ausdrücke verwenden dafür den Punkt. Der reguläre Ausdruck  $r = .$  steht also für die Sprache

$$L(r) = L(.) = \Sigma.$$

In vielen Anwendungen möchte man auch die Sprache spezifizieren können, die nur das leere Wort enthält. Das leere Wort ist eine Zeichenkette, die keine Zeichen enthält. Wir haben hierfür die Schreibweise  $\epsilon$  verwendet, die natürlich für die Verwendung auf einer Computertastatur nicht geeignet ist. In Programmcode würde man eine leere Zeichenkette typischerweise als zwei unmittelbar aufeinander folgende Anführungszeichen `""` schreiben. Das Eingabefeld einer Benutzerschnittstelle kann man einfach leer lassen,

um ein leeres Wort zu spezifizieren. Wir betrachten daher die leere Zeichenkette  $r = \varepsilon$  als regulären Ausdruck, der genau die Sprache

$$L(r) = L(\varepsilon) = L("") = \{\varepsilon\}$$

spezifiziert.

### 4.2.3 Reguläre Operationen

Die regulären Operationen konstruieren aus einfachen regulären Sprachen neue reguläre Sprachen. Damit diese Operationen auch beim Aufbau von regulären Ausdrücken genutzt werden können, erweitern wir die für Sprachen bereits eingeführte Notation der regulären Operationen auf reguläre Ausdrücke.

**Alternative:** Seien  $r_1$  und  $r_2$  reguläre Ausdrücke mit zugehörigen Sprachen  $L(r_1)$  und  $L(r_2)$ . Als regulären Ausdruck, der die Alternative  $L(r_1) \mid L(r_2)$  akzeptiert, verwenden wir die Verkettung der beiden Ausdrücke mithilfe des Zeichens  $|$ :

$$L(r_1 \mid r_2) = L(r_1) \mid L(r_2) = L(r_1) \cup L(r_2).$$

Der reguläre Ausdruck  $r = \emptyset \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$  akzeptiert also die Sprache

$$L(r) = L(\emptyset \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) = \{\emptyset, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

von Wörtern, die nur genau eine Ziffer lang sind.

**Verkettung:** Die Suche nach dem Muster **muster** ist eigentlich eine Spezifikation der Sprache, die durch Verkettung der Sprachen  $L(m)$ ,  $L(u)$ ,  $L(s)$ ,  $L(t)$ ,  $L(e)$  und  $L(r)$  entsteht. Dies möchte man natürlich einfach als den regulären Ausdruck  $r = \text{muster}$  schreiben können. Für zwei beliebige reguläre Ausdrücke  $r_1$  und  $r_2$  ist daher die naheliegendste Schreibweise für den regulären Ausdruck, der die durch Verkettung von  $L(r_1)$  und  $L(r_2)$  entstehende Sprache akzeptiert, die Verkettung der regulären Ausdrücke:

$$r = r_1 r_2 \quad \Rightarrow \quad L(r) = L(r_1 r_2) = L(r_1) L(r_2).$$

Für das **muster**-Beispiel gilt also

$$L(\text{muster}) = L(m)L(u)L(s)L(t)L(e)L(r) = \{\text{muster}\}.$$

**\*-Operation:** Die  $*$ -Operation  $L^*$  konstruiert beliebige Verkettungen von Wörtern einer Sprache  $L$ . Der hochgestellte Stern eignet sich jedoch nicht als maschinenlesbare Notation. Zu einem regulären Ausdruck  $r$  mit akzeptierter Sprache  $L(r)$  schreiben wir daher die  $*$ -Operation mit einem gewöhnlichen Stern in der Form

$$L(r^*) = L(r)^*.$$

Zum Beispiel bedeutet der reguläre Ausdruck  $\cdot^*$  eine beliebige Verkettung von beliebigen Zeichen, es ist also

$$L(\cdot^*) = L(\cdot)^* = \Sigma^*.$$

*Beispiel 4.8.* Man finde einen regulären Ausdruck für die Sprache

$$L = \{\emptyset^m 1^n \mid m \geq 0 \wedge n \geq 0\}.$$

Die Wörter der Sprache bestehen aus beliebig vielen Nullen,  $\emptyset^*$ , gefolgt von beliebig vielen Einsen,  $1^*$ , also ist der gesuchte reguläre Ausdruck  $r = \emptyset^* 1^*$ .  $\circ$

*Beispiel 4.9.* Man finden einen regulären Ausdruck für Wörter über dem Alphabet  $\Sigma = \{a, b, c\}$ , in dem die Buchstaben in alphabetischer Reihenfolge vorkommen und die mit mindestens einem a beginnen.

Nach dem initialen a kommen die Buchstaben a beliebig oft vor,  $a^*$ , b beliebig oft vor,  $b^*$ , und c beliebig oft vor,  $c^*$ . Alles verkettet ergibt den regulären Ausdruck  $r = aa^*b^*c^*$ .  $\circ$

*Verständniskontrolle 4.2:* Finden Sie reguläre Ausdrücke für die folgenden Sprachen:



autospr.ch/v/4.2.pdf

- a)  $L = \{w \in \{0, 1\}^* \mid |w|_0 = 1\}$
- b)  $L = \{w \in \{0, 1\}^* \mid |w|_0 = 1 \vee |w|_1 = 1\}$
- c)  $L = \{w \in \{0, 1\}^* \mid w \text{ endet mit zwei Nullen.}\}$

## Akzeptieren

Man sagt, ein regulärer Ausdruck  $r$  akzeptiert ein Wort  $w \in \Sigma^*$ , wenn  $w \in L(r)$  ist. Ein solches Wort passt auf den regulären Ausdruck  $r$ , oder englisch  $w$  matches  $r$ .

## Maskierungszeichen, “Escaping”

Mit der Definition der regulären Operationen haben die Zeichen  $|$  und  $*$  in regulären Ausdrücken eine spezielle Bedeutung erhalten. Dies war natürlich auch schon mit dem Zeichen  $.$ , welches ein beliebiges Zeichen spezifiziert, eingetreten. Diese Zeichen stehen also nicht mehr direkt zur Verfügung, um Wörter einer Sprache mit regulären Ausdrücken zu beschreiben. Wie soll man die Zeichenkette  $*.*$  als regulären Ausdruck formulieren, wenn beide vorkommenden Zeichen eine spezielle Bedeutung haben?

Ein Maskierungszeichen dient dazu, die spezielle Bedeutung eines Zeichens aufzuheben. Die meisten Dialekte von regulären Ausdrücken verwenden den Backslash  $\backslash$  als Maskierungszeichen. Die Zeichenkette  $*.*$  erhält man also mithilfe des regulären Ausdrucks  $\backslash*\backslash.\backslash*$ . Der Ausdruck  $\backslash|\backslash.*\backslash$  spezifiziert also eine beliebige Folge von Punkten zwischen zwei vertikalen Strichen.

## Gruppierung

Auf welchen Teil eines regulären Ausdrucks bezieht sich der Stern der  $*$ -Operation genau? Für einfache Ausdrücke wie  $.^*$  stellt sich diese Frage nicht, doch wie spezifiziert man eine Verkettung von beliebig vielen Silben bla? Der naheliegende Ausdruck  $\text{bla}^*$  beschreibt

$$L(\text{bla}^*) = L(\text{b})L(\text{l})L(\text{a})^*$$

und damit die zwei Zeichen bl gefolgt von beliebig vielen a. Um anzugeben, dass sich der Stern auf die ganze Silbe bezieht, braucht man Klammern als Gruppierungszeichen. Der Ausdruck  $(\text{bla})^*$  bezeichnet eine Verkettung von beliebig vielen Silben bla.

Die Klammern haben damit ebenfalls eine besondere Bedeutung erhalten und müssen gegebenenfalls maskiert werden. Die Suche nach einem Klammerausdruck in einem Text muss also mit  $\backslash(\cdot\cdot\cdot)\backslash$  erfolgen.

Bei vielen Regex-Engines haben die Klammern noch eine zusätzliche Bedeutung. Vor allem in Programmierbibliotheken ermöglichen sie auch den Zugriff auf den Teil einer gefundenen Zeichenkette, der auf den in den Klammern eingeschlossenen regulären Ausdruck passt. Zum Beispiel zeichnet die **rot** hervorgehobene erste Klammer im regulären Ausdruck  $r = (\text{0}|1)^*\backslash.\text{ }(\text{0}|1)^*$  den Vorkommateil einer binären Bruchzahl aus.

### 4.2.4 Weitere Metazeichen für reguläre Ausdrücke

In Abschnitt 4.1.5 wurden abgeleitete reguläre Operationen eingeführt, die jetzt natürlich ebenfalls auf reguläre Ausdrücke ausgeweitet werden können.

#### Option

Ein Fragezeichen als Exponent einer Sprache war als *Option*  $L^? = L \mid \{\epsilon\}$  eingeführt worden. Für reguläre Ausdrücke schreiben wir daher

$$L(r)^? = L(r?) = L(r|\epsilon) = L(r|) :$$

Die letzte Schreibweise mag etwas ungewohnt sein, da man das leere Wort neben dem Vertikalstrich “nicht sieht”. Man beachte auch, dass es wie bei der  $*$ -Operation nötig sein kann, mit Klammern klarzustellen, worauf sich das ?-Zeichen bezieht.

#### Die $+$ -Operation

Die  $+$ -Operation war als Variante der  $*$ -Operation eingeführt worden, in der das leere Wort ausgeschlossen war. Für reguläre Ausdrücke wird mit

$$L(r)^+ = L(r+)$$

eine entsprechende Variante definiert. Auch hier sind gegebenenfalls Klammern nötig.

## Beschränkte Iteration

Die *beschränkte Iteration* war eine weitere Erweiterung der Stern-Operation, die die +-Operation und die Option als Spezialfälle enthielt. Auch sie kann mit

$$L(r)^{\{n,m\}} = L(r\{n,m\}),$$

gegebenenfalls mit zusätzlichen Klammern um den Ausdruck  $r$ , auf reguläre Ausdrücke ausgedehnt werden.

## Zeichenklassen

Eine besonders häufige Form der Alternative sind Zeichenklassen. Zum Beispiel besteht eine Dezimalzahl aus einer Verkettung von Ziffern 0–9, die man natürlich also

$$(0|1|2|3|4|5|6|7|8|9)^+$$

schreiben kann. Ein gewöhnliches Wort enthält keine Ziffern und ist daher von der Form

$$(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o||q|r|s|t|u|v|w|x|y|z)^+$$

Beide Schreibweisen sind sehr schwerfällig und würden die Verwendung regulärer Ausdrück zum Beispiel als Suchmuster in interaktiven Anwendungen unnötig kompliziert machen. Daher stellen Regex-Engines oft die Möglichkeit zur Verfügung, solche Zeichenmengen einfacher zu definieren. Eckige Klammern können wie die Mengenklammern eine Liste von akzeptablen Zeichen enthalten. Die Vokale und Konsonanten sind zum Beispiel

$$\begin{aligned} [aieu] &= a|i|u|e|o \\ [bcd...uvwxyz] &= b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z. \end{aligned}$$

Da solche Teilbereiche eines Alphabets ebenfalls häufig vorkommen, verstehen Regex-Engines auch die Notation  $[a-z]$  für die Menge aller Zeichen zwischen a und z. Damit bekommt das Zeichen - in einer Zeichenklasse eine spezielle Bedeutung, die nötigenfalls mit \ aufgehoben werden kann oder noch einfacher dadurch, dass - am Anfang aufgelistet wird. Die Zeichenklasse  $[-+*/]$  bezeichnet also die arithmetischen Operatoren, das - ist hier wörtlich zu verstehen, nicht als Angabe eines Bereiches.

Will man alle Zeichen außer einer vorgegebenen Menge spezifizieren, steht dafür das Sonderzeichen ^ zur Verfügung. Die nicht alphabetischen Zeichen sind also durch  $[^a-zA-Z]$  spezifiziert. Das Zeichen ^ hat diese Bedeutung nur am Anfang einer Zeichenklasse.

*Beispiel 4.10.* Man finde einen regulären Ausdruck für großgeschriebene Wörter.

Solche Wörter beginnen mit einem Großbuchstaben,  $[A-Z]$ , gefolgt von beliebig vielen Kleinbuchstaben,  $[a-z]^*$ . Verkettet ergibt sich der reguläre Ausdruck  $r = [A-Z][a-z]^*$ .



*Beispiel 4.11.* Man finde einen regulären Ausdruck für IPv4-Adressen.

IPv4-Adressen bestehen aus vier Dezimalzahlen, die durch Punkte getrennt sind. Die Zahlen können nicht größer als 255 sein, dies lässt sich aber mit einem regulären Ausdruck nur sehr schwerfällig Ausdrücken. Wir beschränken uns daher auf den einfachen Ausdruck  $[0-9]\{1,3\}$ , der zwar zum Beispiel die zulässige Darstellung **0001** ausschließt, aber für viele praktische Zwecke ausreichen wird. Verkettung mit Punkten ergibt den Ausdruck  $r = [0-9]\{0,3\}\cdot[0-9]\{0,3\}\cdot[0-9]\{0,3\}\cdot[0-9]\{0,3\}$ .  $\circlearrowright$

Viele Regex-Engines definieren eine große Zahl weiterer Zeichenklassen, für die sie auch vielfältige erweiterte Notationen bereitstellen. Man konsultiere dazu am besten die Dokumentation der verwendeten Engine.

### Verankerungszeichen

Bei der Anwendung für eine Suchfunktion in einem Editor formuliert man einen regulären Ausdruck  $r$ , der nur eine Teilzeichenkette des durchsuchten Textes beschreibt. Meistens arbeitet die Suche zeilenweise, die Teilzeichenkette könnte also am Anfang, am Ende oder irgendwo in der Mitte stehen, oder auch die ganze Zeile umfassen. Diese vier Fälle kann man unterscheiden, indem man verlangt, dass die ganze Zeile auf einen der drei Ausdrücke

$$\cdot^*r, \quad r\cdot^*, \quad \cdot^*r\cdot^* \quad \text{bzw.} \quad r \quad (4.7)$$

passt. Diese korrekte Schreibweise ist allerdings etwas schwerfällig. Daher stellen Regex-Engines oft die *Verankerungszeichen*  $\wedge$  und  $\$$  zur Verfügung. Das Zeichen  $\wedge$  zu Beginn eines regulären Ausdruck bedeutet, dass die gesuchte Zeichenkette am Zeilenanfang stehen muss.  $\$$  bedeutet, dass sie am Ende der Zeile stehen muss. Die Ausdrücke (4.7) können daher als

$$r\$, \quad \wedge r, \quad r \quad \text{bzw.} \quad \wedge r\$$$

eingegeben werden.

## 4.3 Einen regulären Ausdruck in einen DEA umwandeln

Reguläre Ausdrücke können erst zu einem praktisch anwendbaren Werkzeug werden, wenn man eine effiziente Software-Implementation dafür bauen kann. Wenn für jeden regulären Ausdruck ein deterministischer endlicher Automat konstruiert werden kann, der die gleiche Sprache akzeptiert, dann kann eine Implementation des Automaten in Software dazu verwendet werden, Wörter zu akzeptieren, die vom regulären Ausdruck akzeptiert werden.

### 4.3.1 Konstruktion eines NEA

Reguläre Ausdrücke bestehen aus einzelnen Zeichen, Zeichenklassen und regulären Operationen. Für die regulären Operationen ist bereits bekannt, wie sie durch Operationen auf endlichen Automaten realisiert werden können.

Eine Zeichenklasse ist eine Alternative, zum Beispiel definieren reguläre Ausdrücke in Perl die Zeichenklasse

$$\backslash d = [\text{0-9}] = \text{0} | \text{1} | \text{2} | \text{3} | \text{4} | \text{5} | \text{6} | \text{7} | \text{8} | \text{9}$$

bestehend aus Ziffern. Mithilfe der Konstruktion für die Alternative lässt sich aus Automaten für die einzelnen Zeichen  $\text{0}, \dots, \text{1}$  ein nichtdeterministischer endlicher Automat für die Zeichenklasse erzeugen.

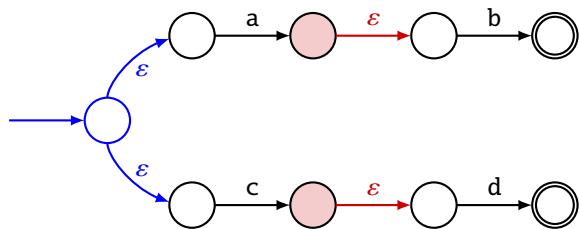
Es bleiben daher nur noch Automaten für einzelne Zeichen, für beliebige Zeichen und für das leere Wort zu konstruieren. Dafür können die Automaten

$$L_a = \{a\} = L\left(\underbrace{\xrightarrow{\text{a}} \textcircled{\text{O}}}_{=: A_a}\right), \quad L_\cdot = \Sigma = L\left(\underbrace{\xrightarrow{\Sigma} \textcircled{\text{O}}}_{=: A_\cdot}\right) \quad \text{und} \quad L_\epsilon = \{\epsilon\} = L\left(\underbrace{\xrightarrow{\epsilon} \textcircled{\text{O}}}_{=: A_\epsilon}\right). \quad (4.8)$$

verwendet werden. Damit folgt der folgende Satz.

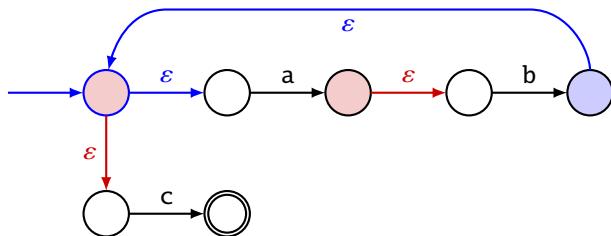
**Satz 4.12.** Zu jedem regulären Ausdruck  $r$  gibt es einen endlichen Automaten  $A$ , der die gleiche Sprache  $L(r) = L(A)$  akzeptiert. Der Automat entsteht aus den Automaten  $A_a$ ,  $a \in \Sigma$ ,  $A_\cdot$  und  $A_\epsilon$  von (4.8) durch Anwendung der regulären Operationen.

**Beispiel 4.13.** Gesucht ist ein nichtdeterministischer Automat für den regulären Ausdruck  $ab | cd$ . Im Zustandsdiagramm



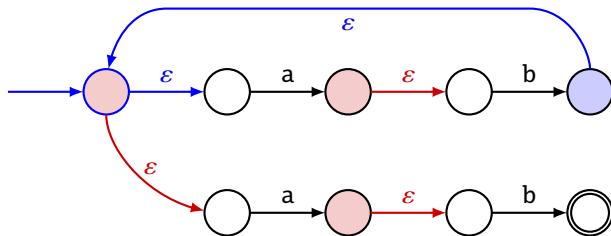
sind die Konstruktionen für die regulären Operationen farblich hervorgehoben. Zuerst wird rot die Verkettung der Einzeichenautomaten  $A_a$  und  $A_b$  bzw.  $A_c$  und  $A_d$  gebildet. Die ausgefüllten Zustände sind ehemalige Akzeptierzustände, die im Laufe des Umwandlungsprozesses zu gewöhnlichen Zuständen geworden sein. Anschließend wird blau die Alternative dazwischen konstruiert.  $\circ$

**Beispiel 4.14.** Gesucht ist ein nichtdeterministischer Automat für den regulären Ausdruck  $(ab)^*c$ . Wieder sind die Verkettungen rot und die  $*$ -Operation blau hervorgehoben. Zuerst wird die Verkettungen der Einzeichenautomaten  $A_a$  und  $A_b$  gebildet (rot). Die  $*$ -Operation ist blau. Das Resultat wird mit dem Einzeichenautomaten  $A_c$  verkettet. Es gibt sich der Automat



Die ausgefüllten Zustände sind ehemalige Akzeptierzustände, die im Laufe des Umwandlungsprozesses zu gewöhnlichen Zuständen geworden sein.  $\circ$

*Beispiel 4.15.* Gesucht ist ein nichtdeterministischer endlicher Automat für den regulären Ausdruck  $(ab)^*$ . Als Implementation wird  $(ab)^*ab$  gewählt. Für den Automaten



wurde erst die Verkettung  $ab$  gebildet. Darauf wurde die blau hervorgehobene \*-Operation angewendet, die schließlich mit einer weiteren Verkettung von  $a$  und  $b$  verkettet wurde.  $\circ$

*Verständniskontrolle 4.3:* Zeichnen Sie das Zustandsdiagramm eines endlichen Automaten, der genau die Wörter akzeptiert, die auf den regulären Ausdruck

$$(\emptyset((ab)^*|cd))^*$$



passen.

### 4.3.2 Performance von Regex-Implementationen

In diesem Abschnitt soll untersucht werden, wie verschiedene Programmiersprachen reguläre Ausdrücke implementieren.

#### DEA- oder NEA-Implementation

Der Satz 4.12 garantiert, dass sich zu jedem regulären Ausdruck ein endlicher Automat finden lässt, mit dem genau die Wörter akzeptiert werden können, die der reguläre Ausdruck akzeptiert. Der so konstruierte endliche Automat ist aber im Allgemeinen ein nichtdeterministischer Automat mit  $\epsilon$ -Übergängen. Durch Anwendung der Sätze 3.4 und 3.12 kann daraus ein deterministischer Automat gemacht werden. Beide Arten von Automaten sind

gleichermaßen als Basis für eine Implementation geeignet. Der Unterschied äußert sich in der Rechenzeit, die für die Entscheidung benötigt wird, ob ein Wort  $w$  akzeptiert werden kann. Ein deterministischer endlicher Automat oder der Thompson-NEA eines nichtdeterministischen endlichen Automaten gestatten, dies in einer Zeit zu entscheiden, die wie  $O(|w|)$  von der Wortlänge abhängt.

Ein nichtdeterministischer Automat kann mit dem Algorithmus von Abbildung 3.1 implementiert werden, dessen Laufzeit im schlimmsten Fall exponentiell wie  $O(|Q|^{|w|-1})$  von der Wortlänge abhängt. Die Laufzeit kann also in einem Maß anwachsen, welches für interaktive Anwendungen ungeeignet ist.

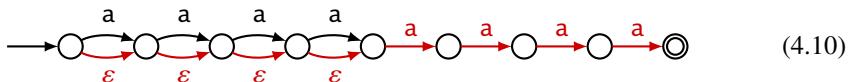
### Implementationsdiagnose mit Laufzeitmessungen

Die exponentielle Abhängigkeit der Laufzeit von der Wortlänge ist mit einem geeigneten regulären Ausdruck experimentell leicht feststellbar. Wir verwenden den regulären Ausdruck

$$r_n = \underbrace{a?a?a? \dots a?}_{n} \underbrace{aaa \dots a}_{n}, \quad (4.9)$$

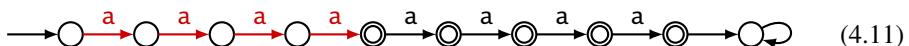
der Wörter aus lauter Zeichen  $a$  akzeptiert, sofern ihre Länge zwischen  $n$  und  $2n$  liegt. Ein deterministischer endlicher Automat akzeptiert das Wort  $a^n$  in einer Zeit, die wie  $O(n)$  von der Wortlänge abhängt. Dies ist die erwartete Performance einer optimalen Implementierung.

Der aus dem regulären Ausdruck  $r$  konstruierte und durch Weglassen von ein paar  $\varepsilon$ -Übergängen vereinfachte NEA $_{\varepsilon}$  ist



(dargestellt ist hier und in (4.11) der Fall  $n = 4$ ). Um das Wort  $a^n$  zu akzeptieren, muss jeweils der rot eingezeichnete  $\varepsilon$ -Übergang genommen werden, über die nachfolgenden rot markierten  $a$ -Übergänge wird der Akzeptanzzustand erreicht. In einer Implementation, die alle Übergänge nacheinander durchprobiert, ist dies die letzte von  $2^n$  zu prüfenden Kombinationen. Die Laufzeit ist daher  $O(2^n)$ .

Wandelt man den NEA $_{\varepsilon}$  (4.10) nach Satz 3.12 in einen DEA um, erhält man



Er akzeptiert das Wort  $a^n$  auf dem rot eingezeichneten Pfad durch den Automaten in einer Zeit, die wie  $O(n)$  von der Wortlänge abhängt.

Die in den folgenden Abschnitten präsentierten Laufzeitmessungen wurden mit dem verlinkten Code auf einem MacBook Pro mit Apple M1 Pro CPU mit 3.2 GHz durchgeführt. Außer im Fall der RE2-Bibliothek, die mit Homebrew [21] installiert wurde, wurden die Programme und Bibliotheken von MacOS 14.5 und der Compiler von Xcode 15.4 verwendet.



[autospqr.ch/c/1](http://autospqr.ch/c/1)

## Die C-Standardbibliothek

Reguläre Ausdrücke fanden schon sehr früh Eingang in viele Unix-Programme wie `lex` (siehe auch Abschnitt 4.5.1), `sed`, `awk` und `expr` oder die gebräuchlichen Editoren. Viele dieser Werkzeuge verwenden eigene Variationen der Regex-Syntax. Erst später entstand die `regex`-Bibliothek, so dass neue Anwendungen eine gemeinsame Implementation mit einheitlicher Syntax nutzen konnten. Die Handbuchseite `regex(3)` beschreibt im Wesentlichen drei Funktionen. Mit der Funktion

```
int regcomp(regex_t *restrict preg, const char *restrict pattern,
           int cflags);
```

wird der reguläre Ausdruck im Argument `pattern` in eine opake interne Darstellung eines endlichen Automaten im ersten Argument vom Typ `regex_t` der Funktion kompiliert. Um eine Zeichenkette mit dem regulären Ausdruck zu vergleichen, wird sie als zweites Argument der Funktion

```
int regexec(const regex_t *restrict preg, const char *restrict string,
            size_t nmatch, regmatch_t pmatch, int eflags);
```

übergeben. Diese Funktion folgt den Übergängen des endlichen Automaten mit den Zeichen des Eingabewortes `string` und gibt als Rückgabewert den Wert 0, wenn das Wort vom regulären Ausdruck akzeptiert wird. Die von der Bibliothek implementierte Syntax regulärer Ausdrücke erlaubt mit runden Klammern auch Teilzeichenketten zu markieren, auf die im Falle einer Übereinstimmung zugegriffen werden kann. Die verbleibenden Argumente der Funktion `regexec` dienen dazu, Start- und Endpunkte dieser Teilzeichenketten zurückzugeben.

Die dritte Funktion `regfree` dient nur dazu, den von der internen Darstellung verwendeten Speicher freizugeben.

Um herauszufinden, ob die `regex`-Bibliothek eine DEA-Implementation von (4.10) verwendet, müssen erst der reguläre Ausdruck `re` und das zu testende Wort `s` erzeugt werden, dann kann mit dem Code

```
regex_t a;
regcomp(&a, re, REG_NOSUB);
int     match = regexec(&a, s, 0, NULL, 0);
regfree(a);
```

das Wort `s` mit dem Automaten verarbeitet werden. Die Messresultate sind in Abbildung 4.2 zusammengestellt. Die Laufzeit in der Abbildung rot dargestellt, passt gut zu einer Logarithmusfunktion, die das rote Band in der Abbildung begrenzt.

## Die C++-Standardbibliothek

Die C++-Standardbibliothek verfügt über eine eigene Regex-Klasse. Eine Instanz der Klasse `std::regex` wird mit dem regulären Ausdruck als Konstruktorgargument instantiiert, dabei wird auch eine interne Darstellung eines endlichen Automaten erzeugt. Ob der reguläre Ausdruck ein Wort akzeptiert, wird mit der Funktion `std::regex_match`

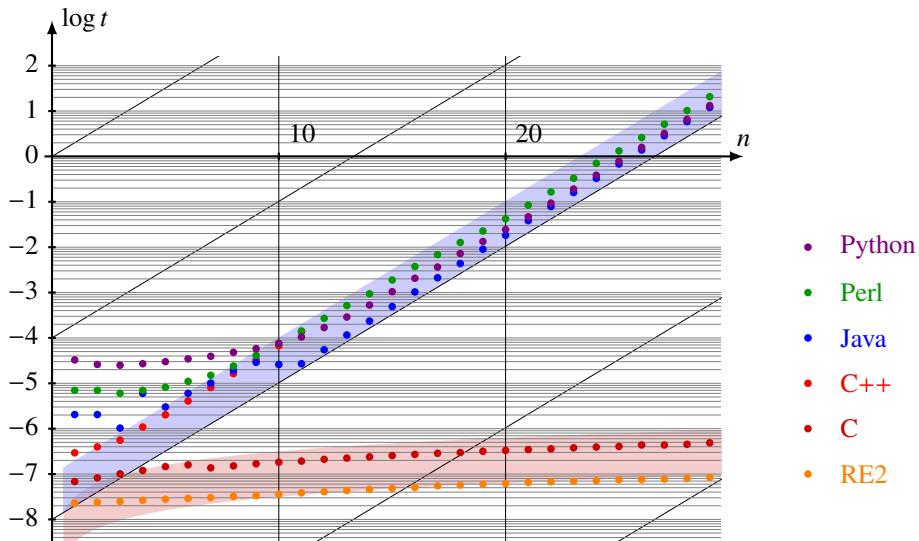


Abbildung 4.2: Laufzeit für verschiedene Regex-Implementationen. Dargestellt ist die Laufzeit in Sekunden für die Verarbeitung der Zeichenkette  $a^n$  mit dem regulären Ausdruck  $r_n$  von (4.9) in Abhängigkeit von  $n$ . Die Implementationen, die einen NEA verwenden, sind erkennbar am exponentiellen Anstieg der Laufzeit, der sich in der logarithmischen Skala in einer Geraden ungefähr mit Steigung  $\log 2$  (schräge Linien) äußert. Alle NEA-Implementation haben für großes  $n$  ähnliches Laufzeitwachstum im blauen Band. Die Laufzeit der DEA-Implementationen (C und RE2) wächst linear, in der logarithmischen Darstellung wird dies durch eine Logarithmus-Kurve wiedergegeben (rotes Band am unteren Rand).

mit dem Wort als erstem und dem Regex-Objekt als zweitem Argument getestet. Die Messresultate sind in Abbildung 4.2 hellrot dargestellt. Die Implementation auf MacOS funktioniert nur bis  $n = 10$ , für größere reguläre Ausdrücke wirft der Konstruktor eine Exception, die anzeigt, dass eine voreingestellte interne Limitierung überschritten sei. Die vorhandenen Daten deuten jedoch an, dass die Laufzeit exponentiell anwächst, dass die Implementation also einen NEA verwendet.

## Die RE2-Bibliothek

Als Google-Projekt wurde die RE2-Bibliothek veröffentlicht, die ein C++-Interface zur Verfügung stellt. Ihr API ist zwar inkompatibel zu `std::regex`, folgt aber einem ähnlichen Muster. Eine Instanz der RE2-Klasse wird mit einem regulären Ausdruck als Konstruktorargument instanziert. Der Test, ob ein Wort akzeptiert wird, erfolgt nicht über eine Methode sondern über die Funktion `RE2::FullMatch` mit der Zeichenkette als erstem und dem RE2-Objekt als zweitem Argument. Die gemessenen Laufzeiten in Abbildung 4.2 zeigen, dass RE2 eine DEA-Implementation verwendet, die fast eine Größenordnung schneller ist als die Implementation der C-Bibliothek.

## Java

Java stellt zwei APIs für die Anwendung von regulären Ausdrücken zur Verfügung. Die `java.lang.String`-Klasse definiert eine Methode `matches` mit einem regulären Ausdruck als Zeichenkette als Argument und einem `boolean` Rückgabewert, der anzeigt, ob der reguläre Ausdruck die Zeichenkette akzeptiert. Dieses API erlaubt offenbar nicht, die Erstellung des endlichen Automaten von der Ausführung zu trennen und eignet sich daher nicht so gut für die Abschätzung der Leistung.

Das Package `java.util.regex` stellt ein erweitertes API zur Verfügung, welches die Erzeugung des endlichen Automaten separiert. Der Code

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
...
public boolean matches(String string, String regex) {
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(string);
    return m.matches()
}
```

instanziert zunächst ein Pattern-Objekt `p`, welches den endlichen Automaten enthält. Im zweiten Schritt wird dann eine Zeichenkette verarbeitet und ermittelt, ob der reguläre Ausdruck das Wort akzeptiert.

Die Messresultate in Abbildung 4.2 zeigen, dass Java eine NEA-basierte Implementation verwendet. Diese Limitierung des `java.util.regex`-Package ist gut bekannt und wurde auch im JDK Enhancement Proposal [7] angesprochen. Besonderer Kritikpunkt war die schlecht vorhersagbare Performance. Leider hat sich daraus nichts weiter entwickelt.

## Perl

Die Skriptsprache Perl wurde von Larry Wall 1987 primär als Werkzeug zur Erzeugung von Reports entwickelt. Gemäß der Handbuchseite `perl(1)` ist der Name das Akronym von *Pathologically Eclectic Rubbish Lister*. Für das Pattern-Matching mit regulären Ausdrücken steht in Perl die besonders kompakte Notation

```
$string =~ m/$regex/
```

zur Verfügung.

Die Regex-Engine von Perl basierte ursprünglich auf einer Bibliothek von Henry Spencer, die dann aber stark erweitert wurden. Im Perl regulären Ausdruck  $(\emptyset^*) 1 \backslash g1$  bedeutet  $\backslash g1$ , dass an dieser Stelle die gleiche Zeichenkette folgen muss, die schon vom Klammerausdruck  $(\emptyset^*)$  akzeptiert wurde ( $\backslash g1$  bezieht sich auf das “grouping” mit der ersten Klammer). Der Perl reguläre Ausdruck akzeptiert also Wörter, die nach dem einzelnen 1 genau die gleiche Zeichenkette enthalten wie vor dem 1. Somit besteht die akzeptierte Sprache aus Wörtern der Form  $\emptyset^n 1 \emptyset^n$ . Diese Sprache ist aber nicht regulär, wie man mit dem Pumping-Lemma 2.3 von Kapitel 2 leicht zeigen kann. Ein regulärer Ausdruck in

Perl kann also Sprachen beschreiben, die nicht regulär sind und für die man auch nicht erwarten kann, dass ein deterministischer endlicher Automat zur Implementation verwendet werden könnte.

Die Messresultate in Abbildung 4.2 zeigen, dass auch Perl einen nichtdeterministischen Ansatz für die Implementation seiner Regex-Engine verwendet.

## Python

Die Python-Standardbibliothek ermöglicht die Verwendung von regulären Ausdrücken über das `re`-Modul. Die Funktion

```
re.match(regex, word)
```

entscheidet, ob der reguläre Ausdruck `regex` das Wort `word` akzeptiert. Die Laufzeit ist exponentiell, wie die Messresultate in Abbildung 4.2 zeigen. Grund dafür ist wie im Fall von Perl die Erweiterungen der Syntax von regulären Ausdrücken, die nicht immer eine Implementation mit einem DEA erlauben.

Neben dem `re`-Modul gibt es ein neueres, nur in Python3 unterstütztes Modul `regex`, welches wo möglich eine DEA-Implementation verwendet und daher Laufzeit  $O(n)$  hat.

## 4.4 Regulärer Ausdruck eines DEA

Ein regulärer Ausdruck spezifiziert eine reguläre Sprache, ein endlicher Automat dafür kann sofort aus dem regulären Ausdruck abgeleitet werden. Offen bleibt die Möglichkeit, dass es reguläre Sprachen gibt, die nicht mit regulären Ausdrücken beschrieben werden können. Ziel dieses Abschnitts ist zu zeigen, dass dies nicht geschieht. Jede reguläre Sprache kann mit einem regulären Ausdruck spezifiziert werden. Wir zeigen dies, indem wir einen endlichen Automaten in einen regulären Ausdruck mit gleicher akzeptierter Sprache verwandeln.

### 4.4.1 Verallgemeinerter NEA

Die Pfeile in einem  $\text{NEA}_\epsilon$  sind mit einzelnen Zeichen aus dem Alphabet oder mit den Symbolen  $\epsilon$  angeschrieben. Beides sind reguläre Ausdrücke. Diese Beobachtung legt nahe, dass das Konzept eines  $\text{NEA}_\epsilon$  erweitert werden kann, indem man reguläre Ausdrücke als Beschriftung für die Übergänge zulässt.

**Definition 4.16.** Ein verallgemeinerter nichtdeterministischer endlicher Automat (VNEA) ist ein endlicher Automat  $A = (Q, \Sigma, \delta, q_0, F)$ , wobei die Übergänge mit regulären Ausdrücken angeschrieben sind. Ein mit dem regulären Ausdruck  $r$  beschrifteter Übergang vom Zustand  $q$  nach  $q'$  kann mit dem Wort  $w$  genommen werden, wenn der reguläre Ausdruck das Wort  $w \in L(r)$  akzeptiert.

Zu jedem regulären Ausdruck  $r$  gibt es den VNEA



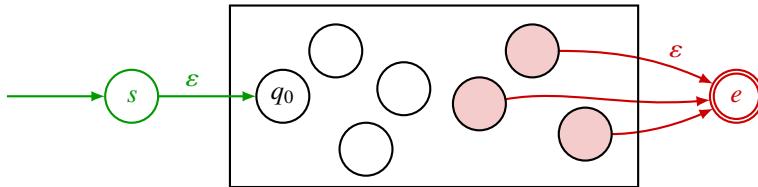


Abbildung 4.3: Vorbereitung eines VNEA  $A$  für den Reduktionsprozess auf einen Automaten der Form (4.12). Ein neuer Startzustand  $s$  und ein vereinheitlichter Akzeptierzustand  $e$  sorgen dafür, dass alle drei Bedingungen von Abschnitt 4.4.2 erfüllt sind.

Der Übergang vom Startzustand  $q_0$  zum Akzeptierzustand  $q_a$  ist nur mit einem Wort  $w \in \Sigma$  möglich, welches vom regulären Ausdruck  $r$  akzeptiert wird. Die akzeptierte Sprache ist  $L(r)$ . Auch kann jeder Übergang in einem VNEA immer durch die aus dem regulären Ausdruck konstruierten Teilautomaten ersetzt werden. Wenn es gelingt, einen beliebigen endlichen Automaten  $A$  in einen VNEA der Form (4.12) zu verwandeln, ohne die akzeptierte Sprache zu verändern, dann ist mit  $r$  ein regulärer Ausdruck gefunden, der ebenfalls die gleiche Sprache  $L(r) = L(A)$  akzeptiert. Es folgt dann der folgende Satz.

**Satz 4.17** (Regulärer Ausdruck eines endlichen Automaten). *Ist  $A$  ein endlicher Automat, dann gibt es einen regulären Ausdruck  $r$ , der die gleiche Sprache  $L(r) = L(A)$  akzeptiert wie  $A$ .*

Die nachfolgenden Abschnitte konstruieren den VNEA in zwei Schritten. Im ersten Schritt wird der Automat in eine Form gebracht, von der man tatsächlich erwarten kann, dass sie in die Form (4.12) gebracht werden kann. Im zweiten Schritt werden dann einer nach dem anderen alle Zustände außer dem Start- und dem einzigen verbleibenden Akzeptierzustand entfernt.

#### 4.4.2 1. Schritt: VNEA vorbereiten

Der Automat (4.12) hat drei Eigenschaften, die bei einem beliebigen endlichen Automaten, ob DEA, NEA oder VNEA, nicht vorausgesetzt werden dürfen:

1. Es gibt keine Übergänge, die zum Startzustand führen.
2. Es gibt nur einen einzigen Akzeptierzustand.
3. Es gibt keine Übergänge aus dem Akzeptierzustand.

Die Eigenschaft 1 kann immer erreicht werden, indem man dem Automaten einen Startzustand  $s$  und einen  $\epsilon$ -Übergang von  $s$  zum ursprünglichen Startzustand hinzufügt (grüne Modifikationen in Abbildung 4.3).

Die Eigenschaft 2 kann erreicht werden, indem ein neuer Akzeptierzustand  $e$  und  $\epsilon$ -Übergänge von allen Akzeptierzuständen des ursprünglichen Automaten nach  $e$  hinzugefügt werden (rote Modifikation in Abbildung 4.3). Damit ist dann auch gleich die Eigenschaft 3 erfüllt.

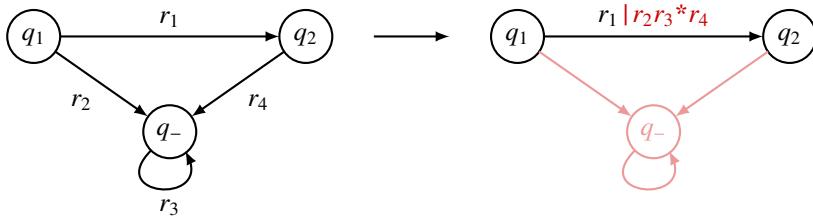


Abbildung 4.4: Entfernung des Zwischenzustandes  $q_-$ . Für alle Paare von Zuständen  $q_1$  und  $q_2$  müssen Übergänge, die über den Zwischenzustand  $q_-$  von  $q_1$  nach  $q_2$  führen, durch eine rot dargestellte Alternative zum direkten Pfad von  $q_1$  nach  $q_2$  hinzugefügt werden.

**Verständniskontrolle 4.4:** Konstruieren Sie einen deterministischen endlichen Automaten, der Wörter gerader Länge über dem Alphabet  $\Sigma = \{0, 1\}$  akzeptiert, und bereiten Sie ihn auf die Umwandlung in einen regulären Ausdruck vor.



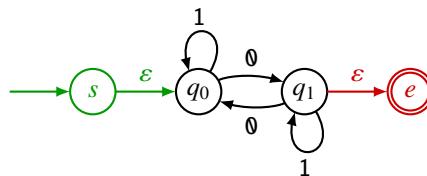
#### 4.4.3 2. Schritt: Zwischenzustände entfernen

Nach den Vorbereitungen von Schritt 1 sind jetzt alle Zustände außer  $s$  und  $e$  zu entfernen. Bei der Entfernung eines Zustands  $q$  gehen auch alle Wege verloren, die über  $q$  als Zwischenzustand führen. Damit sich die akzeptierte Sprache des Automaten nicht ändert, müssen diese Wege durch den Automaten als Alternativen zu den verbleibenden Übergängen hinzugefügt werden.

In Abbildung 4.4 wird der Zustand  $q_-$  entfernt. Es gibt einen Pfad von  $q_1$  nach  $q_2$  über den Zwischenzustand  $q_-$ , der mit einem Wort genommen werden kann, welches vom regulären Ausdruck  $r_2r_3^*r_4$  akzeptiert wird. Daher muss dem direkten Weg von  $q_1$  nach  $q_2$  mit Beschriftung  $r_1$  der reguläre Ausdruck für den Pfad über  $q_-$  als Alternative hinzugefügt werden. Wenn es noch keinen direkten Pfad zwischen  $q_1$  und  $q_2$  gibt, entsteht ein neuer Übergang von  $q_1$  nach  $q_2$  mit dem regulären Ausdruck  $r_2r_3^*r_4$ .

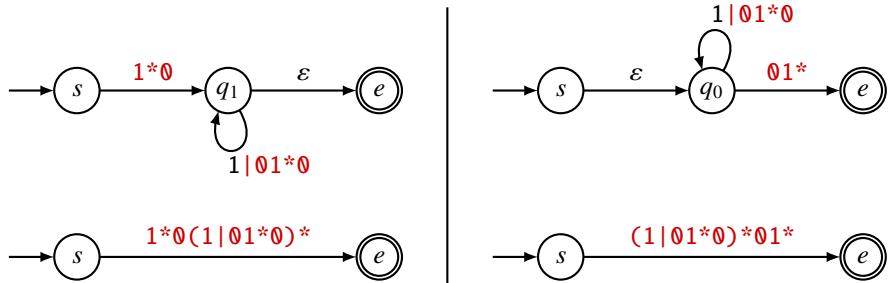
Man beachte, dass der entstehende reguläre Ausdruck von der Reihenfolge abhängt, in der die Zwischenzustände entfernt werden, wie das folgende Beispiel zeigt.

**Beispiel 4.18.** Die Sprache  $L = \{w \in \{0, 1\}^* \mid |w|_0 \text{ ist ungerade.}\}$  ist regulär, sie wird vom Automaten



akzeptiert, in dem bereits der erste Schritt der Reduktion auf die Form (4.12) durchgeführt ist. Jetzt werden die Zwischenzustände entfernt und zwar links in der Reihenfolge  $q_0, q_1$

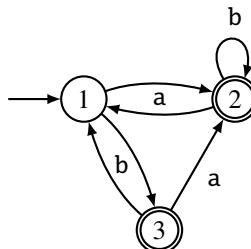
und rechts in der umgekehrten Reihenfolge:



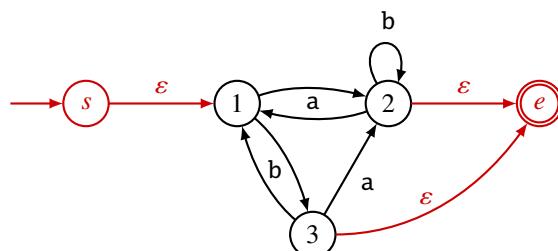
Die beiden regulären Ausdrücke  $1^*0(1|01^*0)^*$  und  $(1|01^*0)^*01^*$  sind offensichtlich verschieden. Man kann sich aber sofort überlegen, dass sie die Sprache  $L$  korrekt beschreiben. Der Ausdruck in der Klammer akzeptiert immer eine gerade Anzahl Nullen, außerhalb der Klammer gibt es genau eine weitere Null. Der Ausdruck akzeptiert also ein Wort mit einer ungeraden Anzahl Nullen. Die Einsen können in beliebiger Zahl und an beliebiger Stelle auftreten.  $\circ$

#### 4.4.4 Beispiel 1

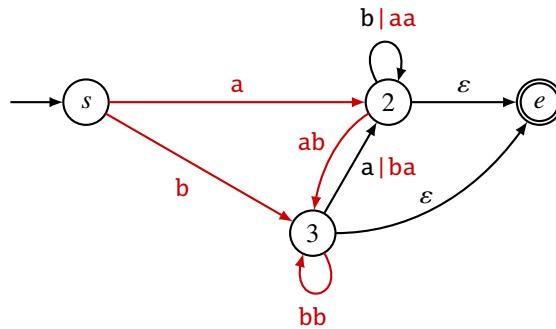
In diesem etwas umfangreicheren Beispiel soll ein regulärer Ausdruck gefunden werden, der die gleiche Sprache akzeptiert wie der Automat



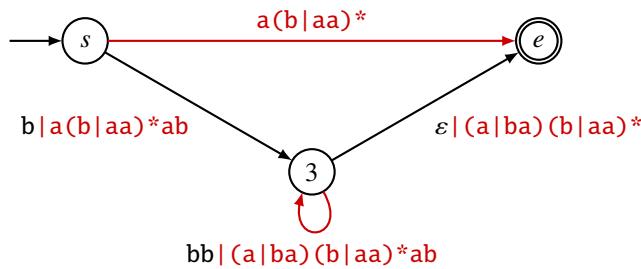
Die sich in jedem Schritt ergebenden Änderungen werden jeweils rot hervorgehoben. Im ersten Schritt wird ein neuer Startzustand und ein Akzeptierzustand hinzugefügt, es entsteht der Automat



Im zweiten Schritt werden jetzt die Zwischenzustände in der Reihenfolge erst  $q_1$ ,  $q_2$ ,  $q_3$  entfernt. Entfernung von  $q_1$ :



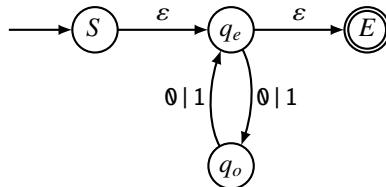
Entfernung von  $q_2$ :



Durch Entfernung von  $q_3$  kann jetzt der endgültige reguläre Ausdruck abgelesen werden, er ist

$$r = a(b|aa)^* \mid (b|a(b|aa)^*ab) (bb|((a|ba)(b|aa)^*ab))^* (| (a|ba)(b|aa)^*. \quad (4.13)$$

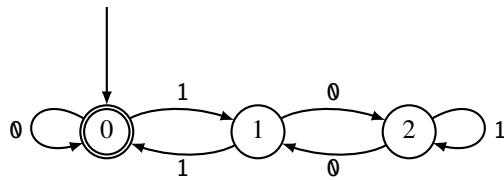
*Verständniskontrolle 4.5:* Wandeln Sie den  $\text{NEA}_\epsilon$



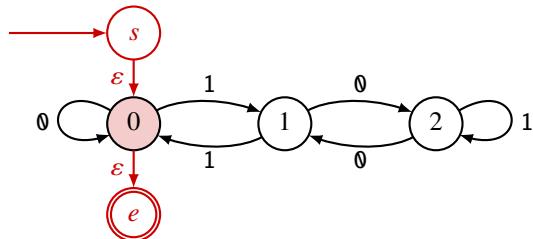
der Wörter mit einer geraden Länge akzeptiert, in einen VNEA der Form (4.12) um, indem Sie nacheinander die Zustände  $q_o$  und  $q_e$  entfernen. Lesen Sie den regulären Ausdruck ab, der die gleichen Wörter wie der  $\text{NEA}_\epsilon$  akzeptiert.

#### 4.4.5 Beispiel 2: Durch drei teilbare Binärzahlen

Wir wandeln den Automaten

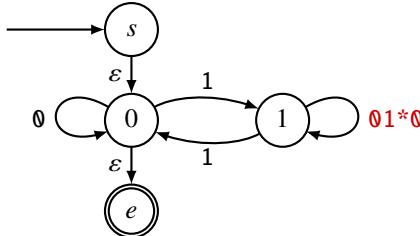


für durch drei teilbare Binärzahlen in einen regulären Ausdruck um. Dazu muss der Automat zunächst standardisiert werden, was den Automaten

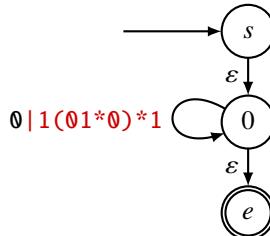


ergibt. Der Akzeptierzustand 0 ist zu einem gewöhnlichen Zustand geworden.

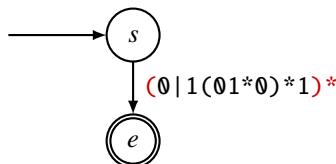
Jetzt können die Zwischenzustände in der Reihenfolge 2, 1 und 0 entfernt werden.  
Nach der Entfernung des Zustands 2 bleibt



Entfernung von 1 ergibt



Nach Entfernung des Zustands 0 kann man aus dem Automaten



den regulären Ausdruck

$$r = (0|1(1|01^*0)^*1)^* \quad (4.14)$$

ablesen.

## 4.5 Anwendungen

Reguläre Ausdrücke findet der Benutzer in vielen Anwendungen als ein Feature unter vielen anderen. In diesem Abschnitt seien aber drei Programme besonders hervorgehoben, in denen reguläre Ausdrücke eine ganz zentrale Rolle spielen.

### 4.5.1 Scanner-Generator **flex**

Ein Compiler oder Interpreter liest Code in einer formalen Sprache und versucht, ihn in ausführbaren Code umzuwandeln oder auszuführen. Der erste Schritt ist dabei, eine lexikalische Analyse durchzuführen und den Datenstrom in sinnvolle Einheiten, sogenannte Tokens, zu zerlegen. Zum Beispiel lassen sich numerische Konstanten oder die reservierten Wörter der Programmiersprache besonders effizient mit regulären Ausdrücken spezifizieren und mit einem endlichen Automaten erkennen. Es ist sinnvoll, für diese Standardaufgabe geeignete Werkzeuge herzustellen. Im Idealfall nimmt ein solches Werkzeug eine Spezifikation der akzeptierten Tokens mit regulären Ausdrücken entgegen und erzeugt daraus Code, der die Tokens erkennt und automatisch mit geeigneten Aktionen verbindet.

Für die Programmiersprache C wurde ein solches Werkzeug bereits in den Siebzigerjahren von Mike Lesk an den Bell Labs geschrieben. Es ist heute nur noch wenig verwendet und wurde weitgehend durch das schnellere und modernere Programm **flex**(1) ersetzt. Seine Funktion soll in diesem Abschnitt an einem Beispiel erläutert werden.

**flex**(1) akzeptiert eine Spezifikation eines lexikalischen Analysators in Form eines Files ähnlich dem Beispiel in Abbildung 4.5. Nach einem Vorspann aus Kommentaren enthält so ein meist mit einer Endung .1 versehenes File eine Liste von regulären Ausdrücken. In geschweiften Klammern können dem Scanner auch Aktionen mitgegeben werden, die aufgerufen werden sollen, wenn das entsprechende Token erkannt worden ist. Im Beispiel enthält die Zeile  $r_1$  einen regulären Ausdruck für Gleitkommazahlen in der Notation, die in C und ähnlichen Sprachen üblich ist.  $r_2$  beschreibt nichtnegative Ganzzahlen im Hexadezimalsystem. Der reguläre Ausdruck  $r_3$  beschreibt zwei Kürzel für die beiden Bücher des Autors<sup>1</sup> und  $r_4$  ist eine Alternative von verschiedenen denkbaren reservierten Wörtern.

Aus dem Input in Abbildung 4.5 erzeugt **flex**(1) eine C-Quelltextdatei, die eine Implementation eines deterministischen endlichen Automaten enthält. Er ist im Wesentlichen in einer großen Tabelle codiert, die die Übergangsfunktion  $\delta$  implementiert. **flex**(1) verwendet zur Konstruktion des Automaten die in diesem Kapitel behandelte Theorie. Um dies zu verifizieren wurden die Tabellen aus dem erzeugten C-Code extrahiert und mit dem Graphvisualisierungswerkzeug **graphviz** dargestellt. Der Übersichtlichkeit halber wurden alle Übergänge, die der Fehlerbehandlung dienen, weggelassen und ähnlich der Vorgehensweise bei der Formulierung nichtdeterministischer endlicher Automaten nur noch

---

<sup>1</sup>Das Buch [38] über lineare Algebra erschien 2023

```
%{
/*
 * sample.1
 */
%}
%%
r1: [-+]?[0-9]+(\.[0-9]*)?(e[-+]?[0-9]+)
r2: 0x[0-9A-F]+
r3: AutoSpr|LinAlg
r4: if|for|foreach|while|repeat
```

Abbildung 4.5: Sourcecode-Beispiel für den Scannergenerator flex. Die vier farbig hervorgehobenen regulären Ausdrücke auf den Zeilen  $r_1$  bis  $r_4$  werden von Programm **flex(1)** in die endlichen Automaten in Abbildung 4.6 übersetzt.

diejenigen Pfade dargestellt, die zu Akzeptierzuständen führen. Das Resultat ist in Abbildung 4.6 dargestellt. Die Akzeptierzustände sind mit den gleichen Farben eingefärbt wie die regulären Ausdrücke im Quelltext. Die gelben Akzeptierzuständen erkennen die verschiedenen reservierten Wörter und der blaue Akzeptierzustand die beiden alternativen Buchkürzel. Die grünen Akzeptierzustände stehen für die verschiedenen möglichen Formen, die Gleitkommazahlen annehmen können. Folgt nach einer führenden  $\emptyset$  ein  $x$  muss eine Hexadezimalzahl folgen, welche im roten Akzeptierzustand erkannt wird.

Die Information über die von **flex(1)** erzeugten Automaten kann man auch aus dem Trace-File herauslesen, welches erzeugt wird, wenn man die Kommandozeilenoption `--trace` verwendet. Das verlinkte Code-Beispiel lässt **flex(1)** aus den regulären Ausdrücken für die drei möglichen Dreierreste einer Binärzahl einen endlichen Automaten erzeugen. Sowohl der erzeugte  $NEA_e$  wie auch der daraus abgeleitete  $DEA$  sind mit ihrem Zustandsdiagramm visualisiert. Im Zustandsdiagramm des  $NEA_e$  kann man die Automatenkonstruktionen für die regulären Operationen wiedererkennen. Der  $DEA$  ist nicht minimiert, aber man kann leicht zeigen, dass sich beim Minimieren der bekannte  $DEA$  für den Dreierrest ergibt.



[autospr.ch/c/2](http://autospr.ch/c/2)

## 4.5.2 Ragel, ein Zustandsmaschinencompiler

Das Programm Ragel von Adrian D. Thurston [58] kompiliert Beschreibungen von Zustandsmaschinen in ausführbaren Code. Die Maschinen können mit einer sehr flexiblen und vielfältigen Syntax spezifiziert werden. Die Maschinen können als Zustände und Übergangsfunktion, durch reguläre Ausdrücke oder durch Mischung dieser beiden Paradigmen beschrieben werden. Der Ragel-Compiler erzeugt aus im Quellcode eingebetteten Maschinenspezifikationen ausführbaren Code in der Sprache des umgebenden Programms.



[autospr.ch/c/3](http://autospr.ch/c/3)

Der Compiler kann zu Debugging- und Dokumentationszwecken auch ein Zustandsdiagramm der erzeugten Maschine im Dot-Format ausgeben, welches mithilfe von Graphviz dargestellt werden kann.

Als Beispiel für die Arbeitsweise von Ragel betrachten wir den endlichen Automaten

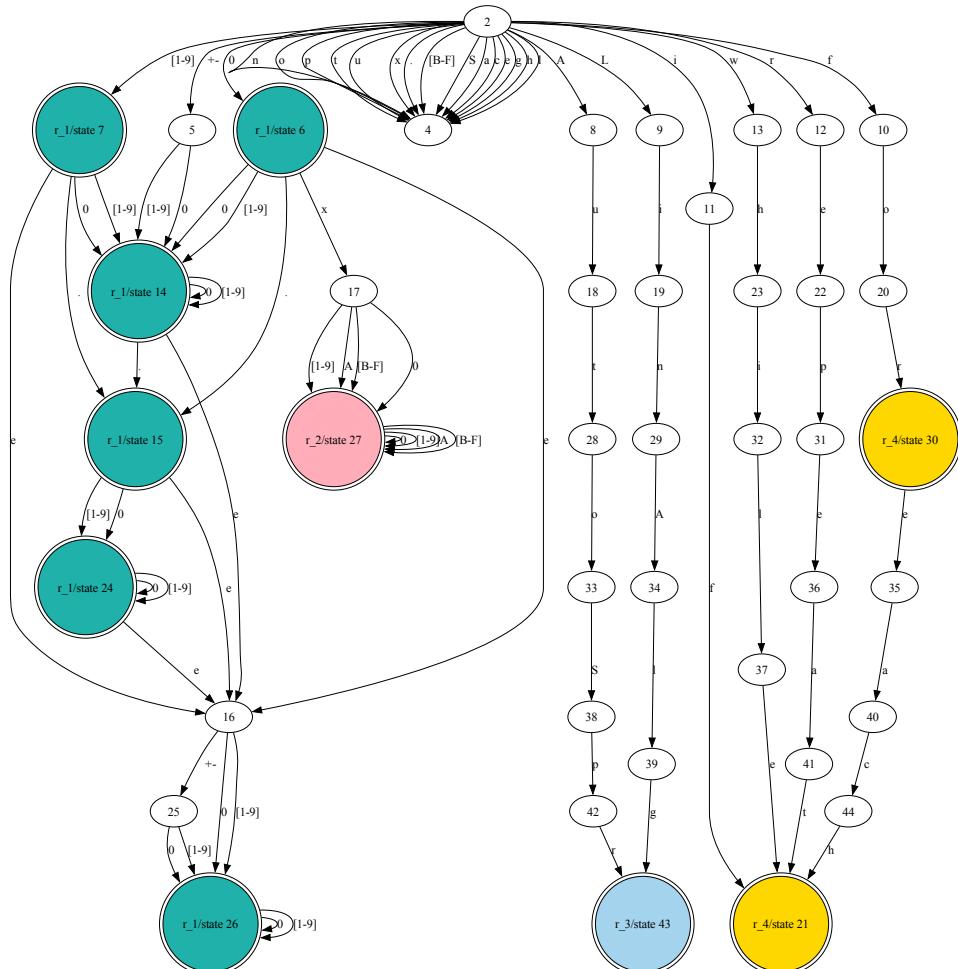


Abbildung 4.6: Von **flex(1)** aus dem Sourcecode von Abbildung 4.5 erzeugter endlicher Automat. Die Farbe der Akzeptierzustände zeigt an, welcher reguläre Ausdruck an dieser Stelle akzeptiert wird.

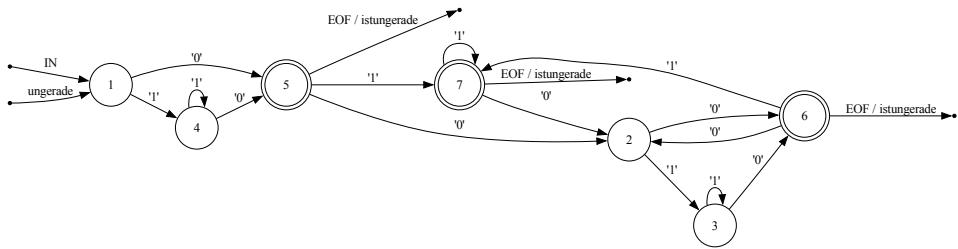
Ragel Quellcode für regulären Ausdruck für Wörter mit ungerader Anzahl 0:

```
%{
    machine ungerade;
    action istungerade { res = 1; }
}%

%% write data;

%{
ungerade := ('1'*'0'('1'|'0'('1')*'0')*)
write init;
write exec;
}%
```

Nicht minimierter Automat für Wörter mit ungerader Anzahl 0:



Minimaler deterministischer endlicher Automat für Wörter mit ungerader Anzahl 0:

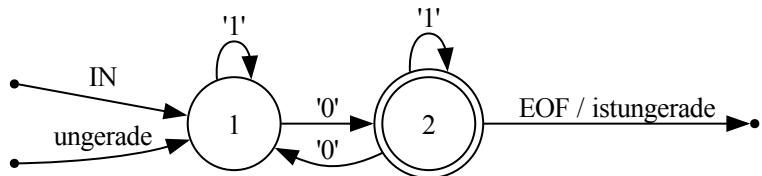
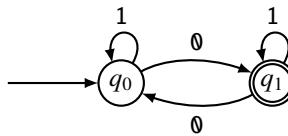


Abbildung 4.7: Von Ragel erzeugter endlicher Automat zum Erkennen von Wörtern mit einer ungeraden Anzahl 0 (oben), der nicht minimierte Automat (Mitte) und der minimierte Automat (unten).



der die Sprache

$$L = \{w \in \{0, 1\}^* \mid |w|_0 \equiv 1 \pmod{3}\} \quad (4.15)$$

akzeptiert. Mit dem Verfahren von Abschnitt 4.4 kann man den regulären Ausdruck

$$r = 1^* 0 (1 | 0(1)^* 0)^* \quad (4.16)$$

finden, der die gleiche Sprache akzeptiert. In Abbildung 4.7 ist der Ragel-Code für einen VNEA mit dem regulären Ausdruck (4.16) dargestellt. Ragel erzeugt daraus einen deterministischen endlichen Automaten. Mit Kommandozeilenoptionen kann man steuern, ob der erzeugte Automat minimiert werden soll. In Abbildung 4.7 ist das Zustandsdiagramm des nicht minimierten Automaten in der Mitte dargestellt, darunter der minimierte Automat, der wieder aussieht wie der ursprüngliche.

Der gleiche Prozess kann natürlich auch auf den im Abschnitt 4.4.5 hergeleiteten regulären Ausdruck (4.14) für durch 3 teilbare Binärzahlen angewendet werden. Abbildung 4.8 zeigt, dass auch in diesem Fall der ursprüngliche deterministische endliche Automat wiederhergestellt wird.

### 4.5.3 Eingebettete lexikalische Analysatoren mit re2c(1)

**re2c(1)** [49] ist ein Generator für lexikalische Analysatoren, der in der Art, wie er arbeitet, ungefähr zwischen **flex(1)** und **ragel(1)** liegt. Wie bei **flex(1)** werden die regulären Ausdrücke in einer Quellcodedatei direkt mit den Aktionen verknüpft, die bei Übereinstimmung ausgeführt werden sollen. Wie bei **ragel(1)** können die regulären Ausdrücke aber an beliebiger Stelle im Quellcode in jeder der unterstützten Sprachen C, Go und Rust auftauchen. Der **re2c(1)**-Compilerbettet die Implementation des endlichen Automaten direkt in den Quellcode ein.

Der **re2c(1)** Compiler kann mit der Kommandozeilen-Option `--emit-dot` ein Zustandsdiagramm des erzeugten endlichen Automaten ausgeben. Abbildung 4.9 zeigt links den ausgegebenen endlichen Automaten zum Quellcode

```

/*!re2c
([0]|([1]([0][1]*[0])*[1]))*  {}
*/
  
```

mit dem regulären Ausdruck für durch 3 teilbare Binärzahlen. Pfeile ohne Beschriftung sind als  $\epsilon$ -Übergänge zu lesen, dazu gehören die Übergänge  $1 \rightarrow 2$ ,  $7 \rightarrow 6$ , die Übergänge zu 5 und der Übergang von 2 zum mit `teilbar.re:2` beschrifteten Zustand. Die Beschriftung verrät, dass dies der Akzeptierzustand für den regulären Ausdruck auf Zeile 2 des Quellcodefiles mit dem Namen `teilbar.re` steht. Der Automat in gewohnter Notation ist in der Mitte dargestellt.

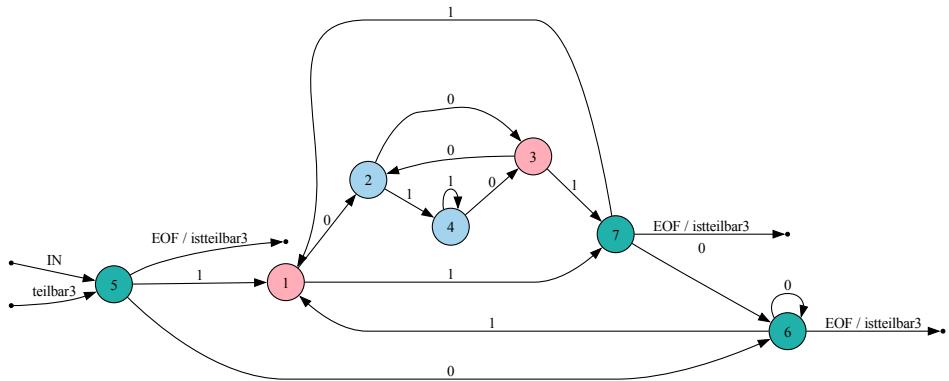
Ragel Quellcode für regulären Ausdruck für durch 3 teilbare Binärzahlen:

```
%%{
    machine teilbar3;
    action istteilbar3 { res = 1; }
}%%

%% write data;

%%{
teilbar3 := ('0' | '1'('0'('1'*))'*'1')* %istteilbar3;
write init;
write exec;
}%%
```

Nicht minimierter endlicher Automat für durch 3 teilbare Binärzahlen:



Minimaler deterministischer endlicher Automat für durch 3 teilbare Binärzahlen:

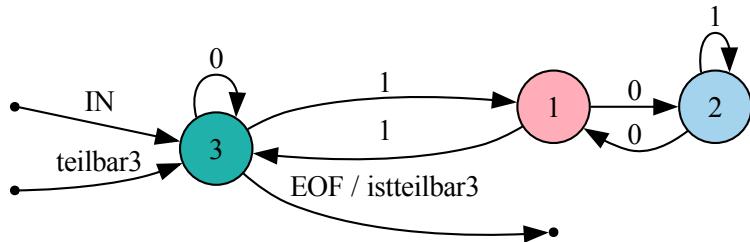


Abbildung 4.8: Von Ragel erzeugter endlicher Automat zum Erkennen von durch 3 teilbaren Binärzahlen (oben), der nicht minimierte Automat (Mitte) und der minimierte Automat (unten).

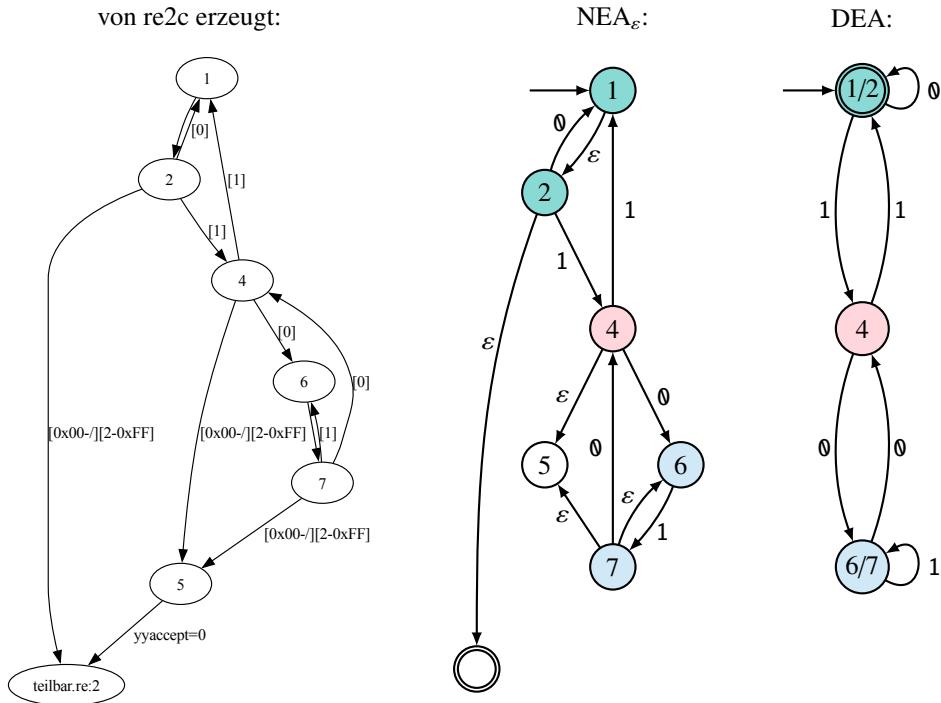


Abbildung 4.9: Von **re2c(1)** erzeugter endlicher Automat zum Erkennen von durch 3 teilbaren Binärzahlen (links), als NEA<sub>ε</sub> geschrieben (Mitte) und umgewandelt in einen DEA (rechts). Der  $\varepsilon$ -Übergang zwischen den Zuständen 1 und 2 bedeutet, dass diese im DEA zusammengefasst werden müssen, desgleichen für die Zustände 6 und 7. Der  $\varepsilon$ -Übergang vom Zustand 2 zum Akzeptierzustand macht aus dem kombinierten Zustand 1/2 einen Akzeptierzustand.

Der Algorithmus zur Umwandlung eines NEA<sub>ε</sub> in einen NEA verlangt, dass man jedes Mal, wenn man auf einem Zustand ankommt, auch noch die von dort ausgehenden  $\varepsilon$ -Übergänge nimmt. Jedes Mal, wenn man im Zustand 1 ankommt, könnte man also auch im Zustand 2 sein. Die Zustände 1 und 2 müssen zusammengefasst werden. Dasselbe gilt für die Zustände 6 und 7.

Der  $\varepsilon$ -Übergang vom Zustand 2 zum Akzeptierzustand bedeutet, dass ein Wort akzeptiert werden kann, wenn es den Automaten in den Zustand 1/2 bringt. Der Übergang macht den kombinierten Zustand 1/2 zu einem Akzeptierzustand.

Der Zustand 5 wird erreicht, wenn andere Zeichen als 0 und 1 im Input angetroffen werden. Der gezeigte Übergang zum Akzeptierzustand ist mit der Beschriftung "yyaccept=0" ausgestattet, die bedeutet, dass das Wort nicht akzeptiert werden darf. Im NEA<sub>ε</sub> und DEA wird das Alphabet {0, 1} verwendet, die von **re2c(1)** erzeugten Übergänge für andere Zeichen können daher ignoriert werden.

Der von **re2c(1)** erzeugte endliche Automat stimmt also trotz der etwas abweichenden

Darstellungsweise mit dem ursprünglichen Automaten für durch 3 teilbare Binärzahlen überein.

## Übungsaufgaben

**4.1.** Konstruieren Sie für jede der folgenden Sprachen über dem Alphabet  $\Sigma = \{0, 1\}$  einen NEA, ohne den Produktautomaten zu verwenden. Reduzieren Sie diesen für die Teilaufgaben a)-d) auf einen minimalen DEA.

- a)  $L_1 = \{w \in \Sigma^* \mid w \text{ enthält mindestens zwei } 1 \text{ nacheinander.}\}$
- b)  $L_2 = \{w \in \Sigma^* \mid w \text{ ist eine durch drei teilbare Binärzahl.}\}$
- c)  $L_3 = L_1 \mid L_2$
- d)  $L_4 = L_1 L_2$
- e)  $L_5 = L_1^*$

*Hinweis.* Die Teilaufgaben a) und b) waren schon Teil der Aufgabe 3.2 von Kapitel 3.

**4.2.** Verwandeln Sie die folgenden regulären Ausdrücke in NEAs über dem Alphabet  $\Sigma = \{a, b\}$ :

- a)  $a(ab\bar{b})^* \mid b$
- b)  $a+ \mid (ab)^+$

**4.3.** Mitglieder sogenannter Netzgruppen (netgroups) in Unix sind Tripel bestehend aus Hostname, Username und Domain. Es wird die übliche mathematische Notation für Tripel verwendet, zum Beispiel sind

```
( asterix, hans, )
( obelix , heiri, )
( asterix,, ost.ch )
```

solche Tripel, sie werden auch Netgrouptriples genannt. Die Namen können Buchstaben, Ziffern, Punkt und Unterstrich enthalten, die Komponenten können auch leer sein.

- a) Ist die Sprache  $L_1$  der Netgrouptriples regulär?
- b) Zusätzlich wird jetzt verlangt, dass mindestens eine der drei Komponenten nicht leer ist, wir nennen diese neue Sprache  $L_2$ . Ist  $L_2$  regulär?

**4.4.** Ein Wort über dem Alphabet  $\Sigma = \{0, 1\}$  heißt 3-periodisch, wenn es nur aus Wiederholungen der ersten drei Zeichen besteht, also zum Beispiel

**011011011, 100100100, 101101, 111,**

nicht aber

$$00, \quad 1001, \quad 101100, \quad 111000.$$

Ist die Sprache  $L$  der 3-periodischen Wörter regulär?

**4.5.** Finden Sie einen regulären Ausdruck für die Sprache  $L$  über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  bestehend aus den Wörtern, die eine ungerade Anzahl 1 enthalten.

**4.6.** Finden Sie einen regulären Ausdruck sowie einen deterministischen endlichen Automaten für die Sprache  $L$  über dem Alphabet  $\Sigma = \{a, b\}$  bestehend aus Wörtern, in denen der Buchstabe  $a$  nie einzeln dasteht, sondern immer nur in Gruppen von mindestens 2 Zeichen  $a$ .

**4.7.** In der Astronomie werden Punkte am Himmel mithilfe der Winkel der geographischen Länge und Breite angegeben, die im astronomischen Zusammenhang auch Rektaszension und Deklination genannt werden. Allerdings sind viele verschiedene Formate für die Winkelangabe gebräuchlich, manchmal werden sogar im gleichen Sternkatalog verschiedene Formate verwendet. Astronomen haben eine gute Intuition für einen Winkel in Minuten, da der Vollmonddurchmesser ziemlich genau 30 Winkelminuten beträgt. Daher werden Formate, die Winkelminuten zeigen, Dezimalbrüchen vorgezogen. Übliche Formate sind

Format	Beispiel
dezimale Grade	65.4321
Grad, dezimale Minuten	65 25.926
Grad, Minuten, dezimale Sekunden	65 25 55.56

Natürlich kann der Nachkommanteil auch fehlen, und es ist auch möglich, dass ein Winkel negativ ist. Stellen Sie einen regulären Ausdruck auf, der genau diese Formate akzeptiert.

**4.8.** Finden Sie einen regulären Ausdruck für die Sprache

$$L = \{w \in \Sigma^* \mid |w|_0 \text{ ist gerade.}\}$$

über dem Alphabet  $\Sigma = \{\emptyset, 1\}$ .

**4.9.** IPv6-Adressen bestehen aus acht Gruppen von jeweils 16 Bits. Eine Gruppe wird als maximal vier Hex-Ziffern 0 – 9, a – f codiert, führende Nullen sind in jeder Gruppe erlaubt. Die Gruppen werden durch Doppelpunkt : getrennt. Eine typische IPv6-Adresse ist

2001:db8:85a3:0:0:8a2e:370:7334

Folgen mehr als eine Gruppe aus lauter Nullen aufeinander, im obigen Beispiel die Zeichenfolge :0:0:, können diese durch einen doppelten Doppelpunkt :: abgekürzt werden, aber nur einmal, da zum Beispiel in der Adresse 1::1::1 nicht klar wäre, wie lange die Null-Gruppen sind und damit an welcher Position in der Adresse die mittlere 1 eigentlich steht. Die obige Adresse kann also zu

2001:db8:85a3::8a2e:370:7334

abgekürzt werden.

Finden Sie einen regulären Ausdruck, mit dem Eingabefelder für IPv6-Adressen plausibilisiert werden können.

Lösungen: <https://autospr.ch/uebungen/AutoSpr-104.pdf>



## Kapitel 5

# Kontextfreie Grammatiken und Sprachen

Die Sprache  $\{0^n 1^n \mid n \geq 0\}$  hat sich bereits im Kapitel 1 als nicht regulär herausgestellt. Mit einem regulären Ausdruck sind ihre Wörter nicht zu erkennen. Dasselbe gilt auch für die Sprache der korrekt geschachtelten Klammerausdrücke, wie sie in jeder Programmiersprache vorkommen. Es braucht also zum Beispiel beim Bau eines Parsers ein Werkzeug, mit dem dieser Art von Sprachen beizukommen ist. Die in diesem Kapitel vorgestellten kontextfreien Grammatiken und die im Kapitel 7 entwickelten Stackautomaten ermöglichen dies. Sie bilden auch die Basis für die Konstruktion eines Syntaxbaumes, der wiederum für die Auswertung von Ausdrücken oder für die Codegenerierung in einem Compiler verwendet werden kann.

Der minimale DEA hat ermöglicht, DEAs zu optimieren und zu vergleichen. Die Chomsky-Normalform (Abschnitt 5.5) stellt zwar auch sicher, dass eine Grammatik nur sehr einfache Regeln verwendet, ist aber nicht eindeutig und damit auch keine geeignete Basis für einen Vergleich von Grammatiken. Es wird sich später in Kapitel 11 sogar zeigen, dass es keinen Algorithmus gibt, mit dem entschieden werden kann, ob zwei Grammatiken die gleiche Sprache produzieren.

### 5.1 Kontextfreie Grammatik und kontextfreie Sprache

In diesem Abschnitt wird das Konzept einer kontextfreien Grammatik und der davon produzierten Sprache eingeführt. Die meisten Programmiersprachen können mindestens zu großen Teilen von einer kontextfreien Grammatik beschrieben werden, worauf auch der XKCD-Comic von Abbildung 5.1 anspielt.



Abbildung 5.1: Rätselhafter XKCD-Comic #1090 vom 3. August 2012 [34]. Der mouseover-Text verrät, worum es gehen könnte: *What just happened? There must be some context we're missing.* Das Wort *Grammatik*, welches das Strichmännchen ganz ohne Kontext im zweiten Bild schreit, ist ein Wortspiel auf den Begriff der kontextfreien Grammatik, die in diesem Kapitel als ein Werkzeug der Theorie der formalen Sprachen entwickelt wird. © 2012 xkcd.com

### 5.1.1 Warum reguläre Sprachen nicht reichen

Fast jede Programmiersprache hat die Möglichkeit, Blöcke von Anweisungen zu bilden. In der Familie der von C abstammenden Sprachen<sup>1</sup> (C, C++, Java, JavaScript) geschieht dies mithilfe von geschweiften Klammern { und }. Pascal verwendet die speziellen Schlüsselwörter begin und end. Arithmetische Ausdrücke brauchen runde Klammern, um die Reihenfolge der Operationen bei der Auswertung festzulegen. Diese Gruppierungsmethoden können jeweils mindestens im Prinzip beliebig tief geschachtelt werden. Ihnen ist gemeinsam, dass es immer gleich viele öffnende Klammern wie schließende geben muss.

Über dem Alphabet der Klammern  $\Sigma = \{(),\}$  ist die Sprache

$$L = \{w \in \Sigma^* \mid w \text{ ist ein korrekter Klammerausdruck}\}$$

eine Sprache, die man als Teilsprache in vielen Programmiersprachen finden wird. Sie ist aber nicht regulär, wie man mit dem Pumping-Lemma 2.3 zeigen kann.

*Beweis.* 1. Wir nehmen an, die Sprache  $L$  sei regulär.

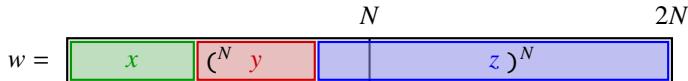
2. Nach dem Pumping-Lemma gibt es eine Zahl  $N \in \mathbb{N}$ , die Pumping-Length.
3. Wir konstruieren das Wort  $w = (^N)^N \in L$  aus  $N$  öffnenden Klammern gefolgt von  $N$  schließenden Klammern.  $w$  ist sicher ein korrekter Klammerausdruck, den wir als

$$w = \boxed{\begin{array}{|c|c|} \hline & (^N) & \\ \hline \end{array}} \quad \begin{matrix} N & & 2N \end{matrix}$$

<sup>1</sup>Die erste Verwendung von geschweiften Klammern geht auf die Sprache BCPL zurück, die 1966 entwickelt wurde. Richtig populär gemacht wurden die geschweiften Klammern aber vor allem durch die Sprache C.

visualisieren können.

4. Das Wort  $w$  hat die Länge  $|w| = 2N > N$ , es muss also nach dem Pumping-Lemma pumpbar sein. Es gibt eine Zerlegung  $w = xyz$



in Teile mit den Eigenschaften  $|xy| \leq N$  und  $|y| > 0$  derart, dass  $xy^kz \in L$  ist für alle  $k \in \mathbb{N}$ .

5. Wegen  $|xy| \leq N$  folgt, dass der Teil  $y$  nur aus öffnenden Klammern bestehen kann. Die gepumpten Wörter

$$xy^2z = \boxed{x} \boxed{(C^N y)} \boxed{y} \boxed{(C^N y)^N}$$

$$xy^0z = \boxed{x} \boxed{(C^N z^N)}$$

oder allgemein  $xy^kz$  enthalten daher  $N + (k - 1)|y|$  öffnende und  $N$  schließende Klammern.

6. Für  $k \neq 1$  sind die gepumpten Wörter nicht mehr Wörter in der Sprache  $L$ , im Widerspruch zur Aussage des Pumping-Lemmas. Dieser Widerspruch zeigt, dass die Sprache nicht regulär sein kann.  $\square$

## 5.1.2 Klammerausdrücke erzeugen

Die Sprache der Klammerausdrücke ist nicht regulär und somit nicht mit regulären Ausdrücken oder endlichen Automaten erkennbar. Korrekte Klammerausdrücke können aber nach einfachen Regeln aufgebaut werden:

1. Das leere Wort ist ein korrekter Klammerausdruck.
2. Aus einem Klammerausdruck entsteht ein neuer Klammerausdruck, indem er “eingeklammert” wird:  $K \rightarrow (K)$ .
3. Zwei korrekte Klammerausdrücke können zu einem neuen korrekten Klammerausdruck verkettet werden.

*Beispiel 5.1.* Der Klammerausdruck  $((\circ(\circ)))$  besteht aus einer eingeklammerten Verkettung der kleineren Ausdrücke  $\circ$  und  $(\circ)$ , die ihrerseits durch, im Falle von  $(\circ)$  wiederholtes, Einklammern des leeren Wortes entstehen.  $\bigcirc$

Etwas formaler kann man die Entstehung eines Wortes durch die wiederholte Anwendung der drei Regeln

1.  $K \rightarrow \varepsilon$
2.  $K \rightarrow (K)$
3.  $K \rightarrow KK$

auf die Variable  $K$  darstellen. Die äußerste Klammer in  $((\cdot))$  entsteht durch die Klammerungsregel 2 aus  $K$ . Durch Anwendung der dritten Regel wird aus  $(K)$  der Ausdruck

$(KK),$

der die Variable  $K$  zweimal enthält. Auf jede dieser Variablen kann man jetzt die Klammerungsregel 2 anwenden. Tut man dies einmal für das linke  $K$  und zweimal für das rechte  $K$  ergibt sich das Wort

$((K)((K))).$

Bis auf die Symbole  $K$  ist dies der angestrebte Klammerausdruck. Die  $K$  können mit der ersten Regel in leere Wörter umgewandelt und damit zum Verschwinden gebracht werden.

Etwas kompakter kann man die Entstehung des Wortes  $((\cdot))$  aus  $K$  durch die folgenden Transformationsschritte darstellen:

$K \rightarrow (K)$	Regel 2
$\rightarrow (KK)$	Regel 3
$\rightarrow ((K)K)$	Regel 2 auf erstes $K$
$\rightarrow ((K)(K))$	Regel 2 auf zweites $K$
$\rightarrow ((K)((K)))$	Regel 2 auf zweites $K$
$\rightarrow ((\cdot))$	Regel 1 auf erstes $K$
$\rightarrow ((\cdot))$	Regel 1 auf zweites $K$ .

*Verständniskontrolle 5.1:* Gegeben sind die Regeln

$$S \rightarrow SS \quad S \rightarrow a \quad S \rightarrow bb \quad S \rightarrow ccc$$

Leiten Sie damit die folgenden Wörter aus der Variablen  $S$  ab.

- a) abbaccc
- b) cccabba



[autospr.ch/v/5.1.pdf](http://autospr.ch/v/5.1.pdf)

### 5.1.3 Formale Definitionen

Das Beispiel der korrekten Klammerausdrücke suggeriert, was für eine allgemein nützliche Definition alles nötig ist. Wir brauchen mindestens ein Platzhalter-Symbol, welches für die Art der Wörter steht, die erzeugt werden sollen. Da sich komplizierte Wörter aus einfacher zu verstehenden Teilen zusammensetzen können, erlauben wir zusätzliche Platzhalter. Zum Beispiel kann man sagen, dass sich Wörter gerader Länge aus Paaren von Zeichen zusammensetzen. Es ist daher naheliegend, neben einem Symbol  $G$  für Wörter gerader Länge ein weiteres Symbol  $P$  für Paare von Zeichen zu verwenden. Diese Platzhaltersymbole heißen auch *Variablen*.

Es braucht Regeln, nach denen mit den Variablen Wörter zusammengesetzt werden können. Die Regel  $K \rightarrow KK$  hat zum Beispiel ausgedrückt, dass ein korrekter Klammerausdruck dadurch entstehen kann, dass man zwei korrekte Klammerausdrücke verkettet. Die Regel  $G \rightarrow GP$  besagt, dass ein Wort gerader Länge dadurch entsteht, dass man ein Wort gerader Länge um ein Paar von Zeichen verlängert.

Am Ende des Prozesses der Regelanwendung bleibt nur das Wort übrig, welches erzeugt werden sollte. Es besteht ausschließlich aus Zeichen des Alphabets der Sprache, die Variablen sind verschwunden. Die Zeichen des Alphabets unterscheiden sich also von den Variablen dadurch, dass die Regelanwendung bei ihnen endet. Sie heißen daher im vorliegenden Zusammenhang auch *Terminalsymbole*.

Schließlich muss festgelegt werden, welche Art von Wörtern erzeugt werden soll. Wenn mehrere Variablen verwendet werden, stehen die meisten für Teile von Wörtern oder Alternativen, nur eine der Variablen steht für alle in Frage kommenden Wörter. Diese Variable heißt die *Startvariable*, aus ihr können alle interessierenden Wörter durch Regelanwendung hervorgebracht werden.

## Kontextfreie Grammatik

Damit erhalten wir die folgende formale Definition.

**Definition 5.2** (Kontextfreie Grammatik). *Eine kontextfreie Grammatik (context free grammar, CFG) ist ein Quadrupel  $G = (V, \Sigma, R, S)$  bestehend aus*

1. den Variablen  $V$ ,
2. den Terminalsymbolen  $\Sigma$ ,
3. den Regeln  $R$ , die die Form  $A \rightarrow u_1u_2 \dots u_n$  haben, wobei die Symbole  $u_i$  Variablen oder Terminalsymbole sein können, und
4. der Startvariablen  $S \in V$ .

Häufig wird eine Grammatik nicht explizit als 4-Tupel geschrieben, sondern einfach als eine Liste von Regeln, aus der man die Variablen und Terminalsymbole ebenfalls ablesen kann. Die Startvariable steht dabei immer in der ersten Zeile.

Die rechten Seiten von Regeln sind Ketten von Symbolen, die sowohl Variablen als auch Terminalsymbole sein können. Sie sind also Wörter über dem Alphabet  $V \cup \Sigma$ , oder in der Schreibweise der Definition 5.2,  $u_1 \dots u_n \in (V \cup \Sigma)^*$ .

## Wörter ableiten

Ausgehend von einer kontextfreien Grammatik und einer Variablen  $A \in V$  können Wörter dadurch gebildet werden, dass wiederholt Regeln angewendet werden. Die folgende Definition beschreibt diesen Prozess formal.

**Definition 5.3** (Ableitung von Wörtern). *Enthält das Wort  $w$  die Variable  $A$ , ist also  $w$  von der Form  $w = xAy$ , dann ergibt die Anwendung der Regel der Form  $A \rightarrow u$  mit  $u \in (V \cup \Sigma)^*$  das Wort  $xuy$ . Dies wird auch  $xAy \Rightarrow xuy$  geschrieben. Ein Wort  $w$  heißt aus dem Wort  $v$  ableitbar, wenn durch eine Folge von Regelanwendungen das Wort  $v$  in das Wort  $w$  umgewandelt werden kann, wenn also*

$$v \Rightarrow \dots \Rightarrow w$$

*gilt. Dies wird auch als  $v \xrightarrow{*} w$  abgekürzt.*

## Warum “kontextfrei”?

Der Name *kontextfrei* leitet sich aus der speziellen Form der Regeln ab. Auf der linken Seite einer Regel  $A \rightarrow u_1 \dots u_n$  steht immer nur eine einzige Variable. Regeln der Form  $AB \rightarrow u_1 \dots u_n$  oder  $cA \rightarrow u_1 \dots u_n$  heißen kontextsensitiv und würden bedeuten, dass die Umwandlung des Symbols  $A$  in die Zeichenkette  $u_1 \dots u_n$  im ersten Fall nur dann möglich ist, wenn  $A$  von  $B$  gefolgt wird, oder im zweiten Fall nur dann, wenn vor dem  $A$  ein  $c$  steht. Es käme also auf den Kontext an, in dem die Variable  $A$  im bereits erzeugten Wort steht. Eine kontextfreie Grammatik erlaubt also, Regeln auf Variablen immer unabhängig vom Kontext, in dem sie auftreten, anzuwenden.

## Kontextfreie Sprache

Eine kontextfreie Sprache besteht aus den Wörtern, die sich mithilfe einer kontextfreien Grammatik ableiten lassen.

**Definition 5.4** (Erzeugte Sprache). *Die Menge der Wörter, die von einer kontextfreien Grammatik  $G$  aus der Startvariablen abgeleitet werden können, wird mit  $L(G)$  bezeichnet und heißt die von  $G$  erzeugte Sprache.*

**Definition 5.5** (Kontextfreie Sprache). *Eine Sprache  $L$  heißt kontextfrei (context free language, CFL), wenn es eine kontextfreie Grammatik gibt, die die Sprache  $L = L(G)$  erzeugt.*

Insbesondere ist die Sprache der korrekten Klammerausdrücke kontextfrei.

### 5.1.4 Grammatik für die Sprache $\emptyset^n 1^n$

Die Sprache  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  wurde bereits in Kapitel 1 als nicht regulär erkannt. Die Wörter kann man als eine Art Klammerausdrücke verstehen, wobei  $\emptyset$  als “öffnende Klammer” und  $1$  als “schließende Klammer” dient. Da die Reihenfolge der Nullen und Einsen vorgegeben ist, reicht sogar die noch einfachere Grammatik

$$S \rightarrow \epsilon \tag{5.1}$$

$$S \rightarrow \emptyset S 1 \quad (5.2)$$

zur Erzeugung aller Wörter der Sprache  $L$ . In der Tat wird das Wort  $\emptyset^n 1^n$  durch  $n$ -malige Anwendung der Regel (5.2) auf die Startvariable  $S$  und anschließende Anwendung der Regel (5.1) auf die in der Mitte zwischen  $\emptyset^n$  und  $1^n$  stehende Variable  $S$  erzeugt.

Unter den kontextfreien Sprachen finden sich also einige nicht reguläre Sprachen. Wir werden später in Abschnitt 5.4 zeigen, dass sich für alle regulären Sprachen kontextfreie Grammatiken konstruieren lassen, jede reguläre Sprache ist also auch kontextfrei. In diesem Licht zeigt das Beispiel  $\emptyset^n 1^n$  also, dass die kontextfreien Sprachen eine echte Erweiterung der regulären Sprachen bilden.

### 5.1.5 Grammatik für die Sprache $1^n + 1^m = 1^{m+n}$

Die Sprache

$$L = \{1^n + 1^m = 1^{m+n} \mid n, m \geq 0\}$$

über dem Alphabet  $\Sigma = \{1, +, =\}$  besteht aus korrekten Additionen von unär dargestellten Zahlen (mehr zur unären Darstellung auf Seite 210). Die Regelmengen der Grammatik können in den verschiedenen Formen

$$R = \left\{ \begin{array}{l} S \rightarrow 1S1 \\ \rightarrow +A \\ A \rightarrow 1A1 \\ \rightarrow = \end{array} \right\} = \left\{ \begin{array}{l} S \rightarrow 1S1 \\ S \rightarrow +A \\ A \rightarrow 1A1 \\ A \rightarrow = \end{array} \right\} = \left\{ \begin{array}{l} S \rightarrow 1S1 \mid +A \\ A \rightarrow 1A1 \mid = \end{array} \right\} \quad (5.3)$$

geschrieben werden. Für die bessere Übersicht ist es zulässig, die Variable auf der linken Seite wegzulassen, wenn sie mit der vorangegangenen Regel übereinstimmt. Für eine kompaktere Darstellung dürfen mehrere rechte Seiten auf einer Zeile mit dem Vertikalstrich für die Alternative kombiniert werden.

Die Variable  $A$  erzeugt Wörter der Form  $1^m = 1^m$ . Die Variable  $S$  erzeugt Wörter der Form  $1^n + A1^n$ , zusammen ergeben sich genau die Wörter von  $L$ .

Man kann mit dem Pumping-Lemma zeigen, dass die Sprache  $L$  nicht regulär ist (Übungsaufgabe 2.9). Mit dem später in Kapitel 8 bewiesenen Pumping-Lemma für kontextfreie Sprachen kann man zeigen, dass die Sprache nicht mehr kontextfrei ist, wenn man die Addition durch die Multiplikation ersetzt.

---

**Verständniskontrolle 5.2:** Geben Sie eine kontextfreie Grammatik für eine Sprache über dem Alphabet  $\Sigma = \{a, b, c\}$  an, die Wörter gerader Länge produziert.



autospr.ch/v/5.2.pdf

## 5.2 Syntaxbaum

Die Darstellung einer Ableitung  $S \xrightarrow{*} w$  eines Wortes aus der Startvariablen durch Auflisten der der Reihe nach nötigen Regelanwendungen ist nicht sehr übersichtlich. Außer-

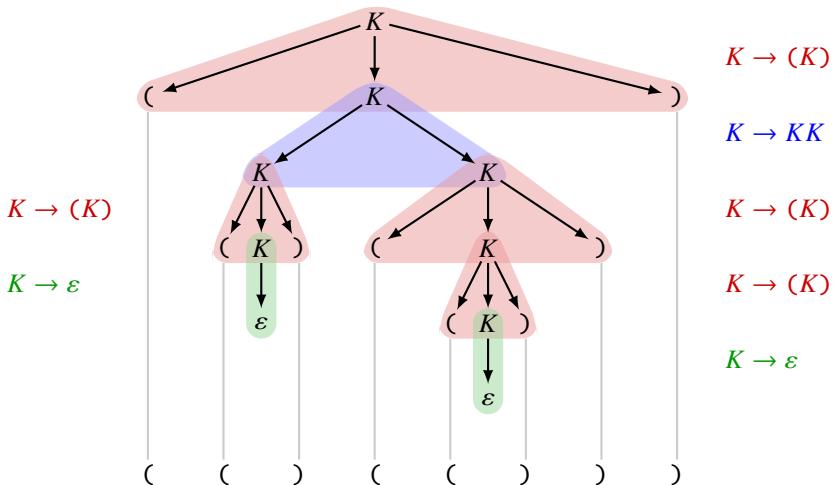


Abbildung 5.2: Syntaxbaum (Parse-Tree) für die Ableitung des Klammerausdrucks  $((()))$  aus der Startvariablen  $K$  mit Hilfe der farbig eingetragenen Regeln.

dem trägt sie der Tatsache nicht Rechnung, dass es zum Beispiel bei der Ableitung von  $((()))$  nach der Verwendung der Regel  $K \rightarrow KK$  nicht darauf ankommt, ob man erst das linke  $K$  in  $(()$  oder erst das rechte  $K$  in  $(()$  umwandelt. In beiden Vorgehensweisen werden die gleichen Regeln auf die gleichen Symbole angewendet. Der Unterschied in der Reihenfolge hat nichts mit der Struktur des erzeugten Wortes zu tun. Es sollte daher eine Darstellungsform gewählt werden, welche nicht von solchen ephemeren Eigenschaften der Ableitung abhängig ist.

### 5.2.1 Konstruktion eines Syntaxbaumes

Wir stellen die Ableitung des Klammerausdrucks  $((()))$  in der Form eines (umgekehrten) Baumes dar, wobei wir die Startvariable  $K$  an der Spitze und die Terminalsymbole am unteren Rand darstellen (Abbildung 5.2). Die drei in der Grammatik zur Verfügung stehenden Regeln sind in der Abbildung farblich hervorgehoben. Der Baum besteht aus Dreiecken, die die angewendete Regel enthalten. An der Spitze des Dreiecks steht jeweils die Variable, die von der Regel umgewandelt wird. An der Basis des Dreiecks findet man die rechte Seite der Regel. Die Ableitung terminiert in dem Moment, wo keine Variablen mehr zur Verfügung stehen. Dieser Baum heißt der *Syntaxbaum* oder *Parse-Tree* des Wortes in der Grammatik.

Die Reihenfolge der Ableitungen im unteren Teil des Baumes ist jetzt nicht mehr wichtig. Ob man erst den linken oder den rechten Teilbaum ableitet, spielt keine Rolle, es ergibt sich immer das gleiche erzeugte Wort.

*Verständniskontrolle 5.3:* In Verständniskontrolle 5.2 wurde die Grammatik



$$\begin{aligned} S &\rightarrow \varepsilon \\ S &\rightarrow SP \\ P &\rightarrow ZZ \\ Z &\rightarrow a \mid b \mid c \end{aligned}$$

für Wörter mit gerader Länge gefunden. Bestimmen Sie den Syntaxbaum für die Ableitung der folgenden Wörter

- a) abccba
- b) ccccccaa

### 5.2.2 Nicht eindeutige Syntaxbäume

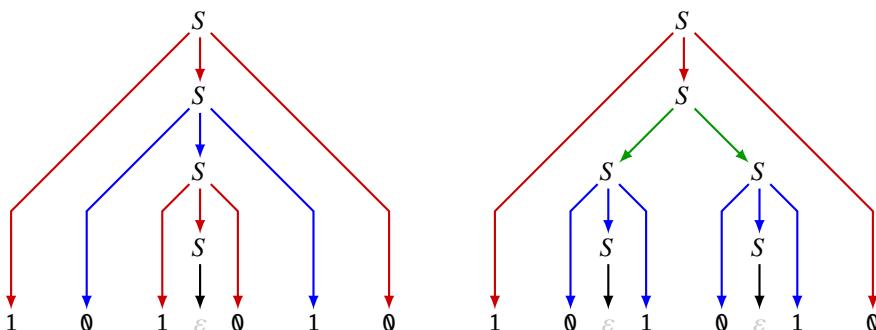
Der Syntaxbaum zeigt, welche Ableitungsschritte in ihrer Reihenfolge vertauscht werden können. Solange die Schritte in verschiedenen Ästen des Baumes stattfinden, spielt es keine Rolle, welcher Schritt zuerst ausgeführt wird, solange die Voraussetzungen für die Regelanwendung erfüllt sind. Dies garantiert aber noch nicht, dass es für die Ableitung eines Wortes nur einen einzigen Syntaxbaum gibt.

**Die Sprache**  $|w|_0 = |w|_1$

Als Beispiel betrachten wir die Grammatik

$$\begin{aligned} S &\rightarrow \emptyset S 1 \\ &\rightarrow 1 S \emptyset \\ &\rightarrow S S \\ &\rightarrow \varepsilon, \end{aligned}$$

die die Sprache  $L = \{ w \in \{\emptyset, 1\}^* \mid |w|_0 = |w|_1 \}$  erzeugt. Die beiden Syntaxbäume



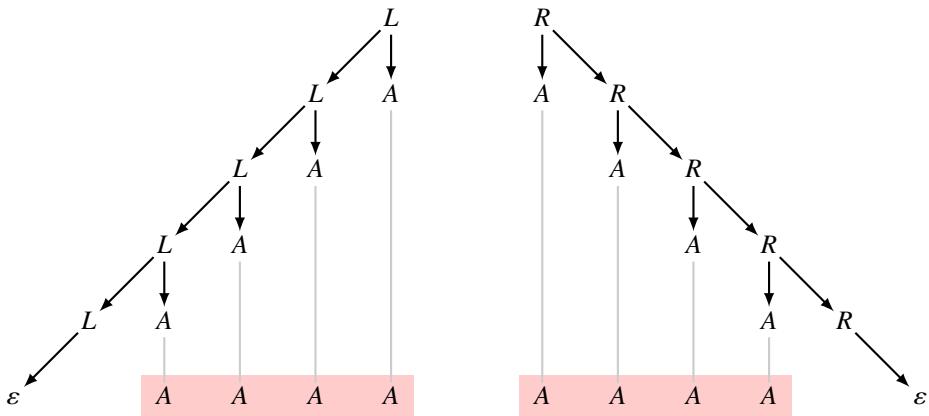
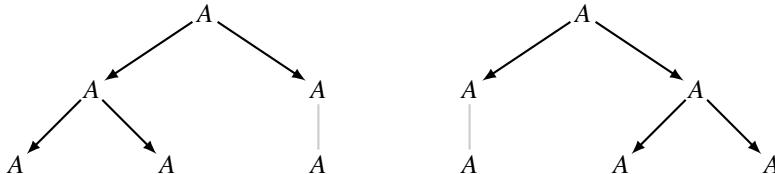


Abbildung 5.3: Syntaxbäume des Wortes AAAA für die beiden Grammatiken (5.4). Die linke Grammatik in (5.4) ist linksrekursiv, die rechte ist rechtsrekursiv.

erzeugen das gleiche Wort 101010, verwenden aber nicht die gleichen Regeln der Grammatik, wie man mithilfe der Farbcodierung der Regeln besonders gut erkennen kann. Man sagt, die Grammatik ist *nicht eindeutig*, sie erlaubt mehrere Syntaxbäume für die gleichen Wörter.

### Rechts- und Linksrekursion

Regeln der Form  $A \rightarrow AA$  ermöglichen, beliebig lange Ketten von Variablen  $A$  zu erzeugen. Die zugehörigen Syntaxbäume sind aber niemals eindeutig. Bereits ein Wort aus nur drei A kann mit den zwei grundsätzlich verschiedenen Syntaxbäumen



erzeugt werden. Die Grammatikregel legt nicht fest, welche Ableitung bevorzugt werden soll. Die Regel kann auf den linken und den rechten Ast gleichermaßen angewendet werden.

Um diese Zweideutigkeit zu vermeiden, muss statt der symmetrischen Regel  $A \rightarrow AA$  eine asymmetrische Regel verwenden. Um eine Folge von beliebig vielen A zu erzeugen, könnte man zum Beispiel die Regeln

$$\begin{array}{lll} L \rightarrow LA & \text{oder alternativ} & R \rightarrow AR \\ \rightarrow \varepsilon & & \rightarrow \varepsilon \end{array} \quad (5.4)$$

verwenden. Die linke Grammatik erzwingt, wie in Abbildung 5.3 gezeigt, dass die erste Regel nur links angewendet werden kann. Der zugehörige Syntaxbaum erstreckt sich

auf dem linken Ast in die Tiefe. Die Regel  $L \rightarrow LA$  dieser Grammatik heißt *linksrekursiv*. Mit der rechten Grammatik passiert genau das umgekehrte, die Regel  $R \rightarrow AR$  heißt *rechtsrekursiv*.

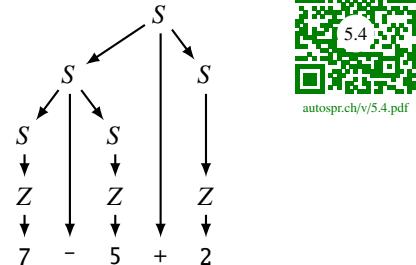
#### Verständniskontrolle 5.4: Die Grammatik

$$S \rightarrow Z$$

$$S \rightarrow S + S$$

$$S \rightarrow S - S$$

$$Z \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$



[autospr.ch/v/5.4.pdf](https://autospr.ch/v/5.4.pdf)

ist nicht eindeutig. Rechts ist ein möglicher Syntaxbaum des Ausdrucks  $7 - 5 + 2$  dargestellt. Finden Sie einen zweiten.

## 5.3 Auswertung

Ein Syntaxbaum für einen Ausdruck ist nicht nur Selbstzweck, in einem Interpreter dient die durch den Baum erfolgte syntaktische Analyse auch als Basis für die Auswertung von Ausdrücken, oder in einem Compiler zur Erzeugung von Maschinen- oder Bytecode, der zur Laufzeit den Wert berechnen kann. In diesem Abschnitt soll diskutiert werden, wie eine solche Auswertung vorzunehmen ist.

### 5.3.1 Expression-Term-Factor-Grammatik

Arithmetische Ausdrücke bilden einen Teil jeder Programmiersprachensyntax. Der Autor eines arithmetischen Ausdrucks erwartet, dass der Interpreter oder Compiler den Ausdruck nach den mathematischen Regeln auswertet, also zum Beispiel Multiplikation und Division vor Addition und Subtraktion ausführt und im Übrigen von links nach rechts rechnet. Eine sinnvolle Grammatik für arithmetische Ausdrücke muss daher Syntaxbäume erzeugen, die als Basis für die mathematisch korrekte Auswertung dienen können. Der Syntaxbaum muss zum Beispiel die Punkt- und die Strichoperationen unterscheiden. Er muss die miteinander zu verknüpfenden Terme von links nach rechts gruppieren, damit auch die Auswertung von links nach rechts erfolgen kann.

#### Die Grammatikregeln

Für arithmetische Ausdrücke wird sehr oft eine Grammatik verwendet, die ungefähr wie folgt aufgebaut ist. Die hier dargestellte Grammatik kennt der Einfachheit halber nur die Operatoren  $+$  für die Addition und  $*$  für die Multiplikation. Weitere Operationen können selbstverständlich hinzugefügt werden. Die Regeln der Grammatik sind

$$\text{expression} \rightarrow \text{expression} + \text{term} \quad (5.5)$$

$$\begin{aligned} &\rightarrow \text{term} \\ \text{term} &\rightarrow \text{term} * \text{factor} \end{aligned} \tag{5.6}$$

$$\rightarrow \text{factor}$$

$$\begin{aligned} \text{factor} &\rightarrow N \\ &\rightarrow (\text{expression}) \end{aligned} \tag{5.7}$$

$$\begin{aligned} N &\rightarrow NZ \\ &\rightarrow Z \end{aligned} \tag{5.8}$$

$$Z \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$

Die linksrekursive Regel (5.5) stellt sicher, dass die Terme von links nach rechts gruppiert werden. Sie stellt auch sicher, dass Terme nur addiert werden können. Die ebenfalls linksrekursive Regel (5.6) stellt sicher, dass die Faktoren von links nach rechts gruppiert werden und dass sie nur multipliziert werden können. Um einen Term mit einem Faktor zu multiplizieren, muss die Regel (5.7) verwendet werden, welche durch Einklammerung einen Ausdruck in einen Faktor umwandeln kann. Damit wird der Regel “Punkt vor Strich” Rechnung getragen: Ein Ausdruck, also eine Summe von Termen, wird nur dann vor der Multiplikationsoperation ausgewertet, wenn dies mit einer Klammer angezeigt wird. Die Regeln ab (5.8) dienen der Erzeugung von ganzen Zahlen ( $N$ ) aus Ziffern ( $Z$ ).

### Syntaxbaum eines Ausdrucks

Der Syntaxbaum des arithmetischen Ausdrucks

$$7 * ( 5 + 3 )$$

ist in Abbildung 5.4 dargestellt. Die Ziffern des erzeugten Wortes haben je nach Position innerhalb des Ausdrucks die Bedeutung von Ausdrücken, Termen oder Faktoren, was die langen vertikalen Ketten von Umwandlungen erklärt. Auf dem rechten Ast kann man die Anwendung der Klammerungsregel (5.7) erkennen, mit der die Summe in der Klammer in einen Faktor umgewandelt wird, damit er mit dem Term des linken Astes multipliziert werden kann.

### 5.3.2 Ausdrücke auswerten

Der Syntaxbaum von Abbildung 5.4 kann auch zur Auswertung des Ausdrucks verwendet werden. Die Werte der Ziffern in der untersten Zeile sind unmittelbar ersichtlich. Es muss jetzt ein Algorithmus formuliert werden, mit dem dem ganzen Ausdruck ein Wert zugewiesen werden kann. Dazu muss jedem Knoten des Baums ein Wert zugewiesen werden. Da jeder Knoten einer Regel der Grammatik entspricht, muss zu jeder Regel eine Vorschrift formuliert werden, die aus den Werten der Elemente der rechten Seite der Regel den Wert der Variablen auf der linken Seite ermittelt.

Für die Regeln der Form  $A \rightarrow B$  ist dies nicht schwierig, die Elemente  $A$  und  $B$  haben offenbar den gleichen Wert. Die Klammerungsregel (5.7) dient nur dazu, die Addition in der Klammer auszuführen, bevor multipliziert wird. Der Wert der Klammer ist der Wert des Ausdrucks, der darin enthalten ist. Die eigentliche Arbeit der Berechnung wird von

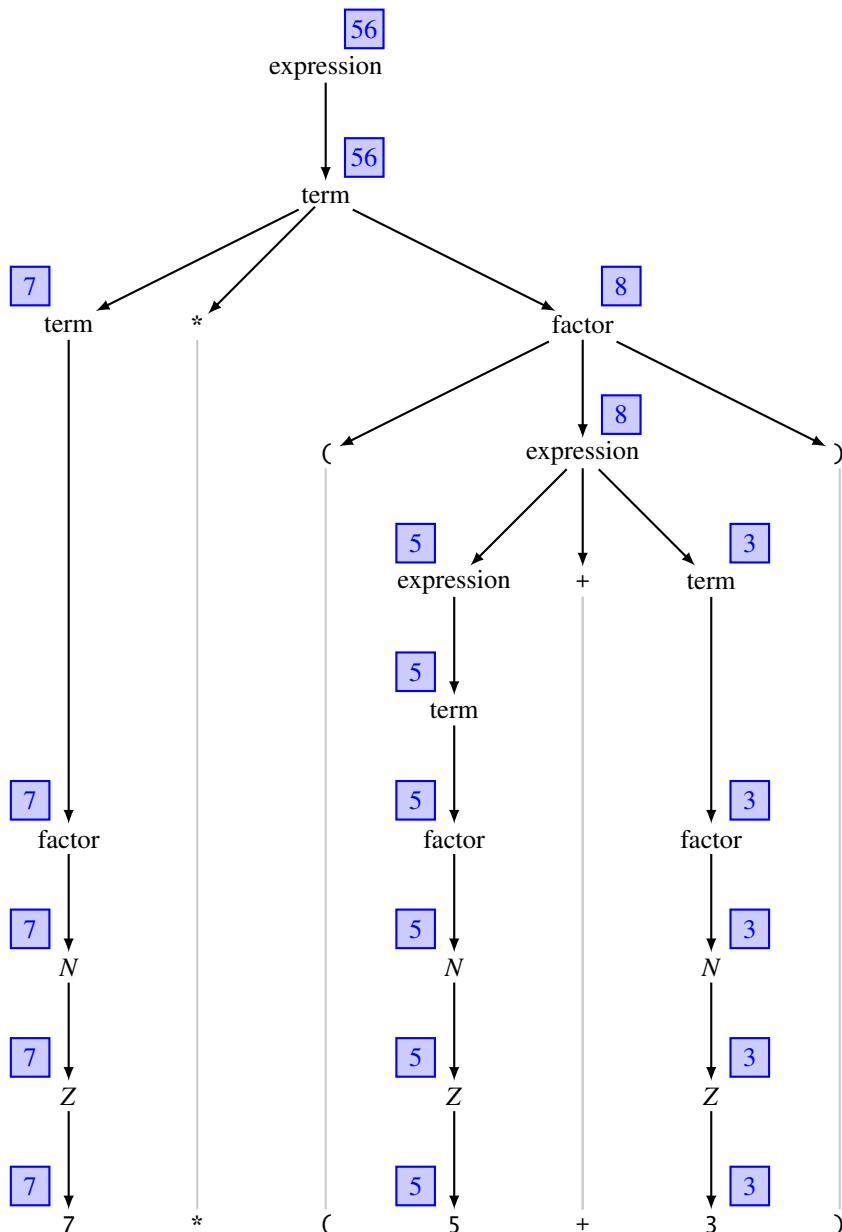
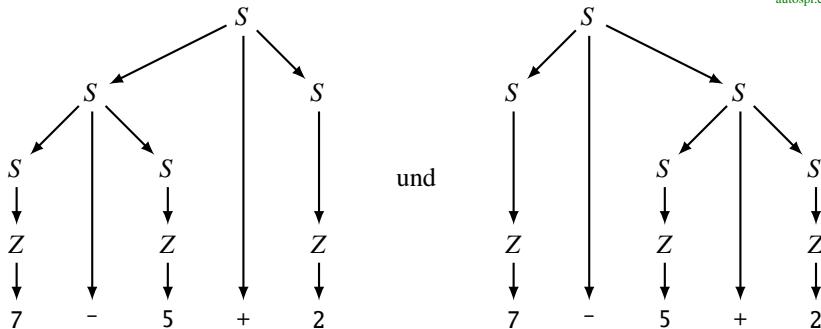


Abbildung 5.4: Syntaxbaum des Ausdrucks  $7 * (5 + 3)$  auf der Basis der Expression-Term-Factor-Grammatik. Die blauen Annotationen enthalten jeweils den Wert, der dem Knoten während der Auswertung des Ausdrucks von unten nach oben zugewiesen werden muss.

den Regeln (5.5) und (5.6) ausgeführt. Der Wert des Ausdrucks auf der linken Seite von (5.5) muss die Summe der Werte des Ausdrucks und des Terms auf der rechten Seite der Regel sein. Ebenso muss der Wert des Terms auf der linken Seite von (5.6) das Produkt der Werte des Terms und des Faktors auf der rechten Seite der Regel sein. Die nach diesem Algorithmus den Knoten zugewiesenen Werte sind in der Abbildung 5.4 als blaue Annotation hinzufügt. Es ergibt sich der korrekte Wert 56.

*Verständniskontrolle 5.5:* In Verständniskontrolle 5.4 wurde erkannt, dass die dortige Grammatik nicht eindeutig ist und die zwei Syntaxbäume



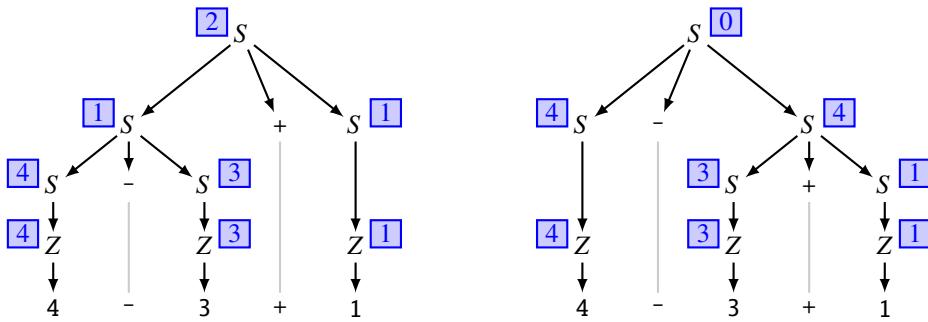
für den Ausdruck  $7-5+2$  zulässt. Formulieren Sie Regeln für die Auswertung des Syntaxbaums und bestimmen Sie die Werte des Ausdrucks für die beiden Syntaxbäume.

### Eine fehlerhafte Grammatik für Addition und Subtraktion

Die Grammatik

$$\begin{aligned}
 S &\rightarrow S + S \\
 &\rightarrow S - S \\
 &\rightarrow Z \\
 Z &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
 \end{aligned}$$

kann Ausdrücke mit Additionen und Subtraktionen von Ziffern erzeugen. Die ersten beiden Regeln erlauben aber Rechts- und Linksrekursion, was dazu führt, dass der Syntaxbaum nicht eindeutig ist. Der Ausdruck  $4-3+1$  hat die beiden Syntaxbäume



Der linke Baum liefert den korrekten Wert 2, der rechte berechnet erst die Summe  $3 + 1$  und subtrahiert sie von 4, was den falschen Wert 0 ergibt. Verwendet man also Rechtsrekursion, entspricht dies der Auswertung des Ausdrucks von rechts nach links, entgegen der üblichen mathematischen Interpretation des Ausdrucks.

Die Programmiersprache APL, die in den sechziger und siebziger Jahren des 20. Jahrhunderts einige Popularität erreicht hatte, hat tatsächlich arithmetische Ausdrücke immer von rechts nach links ausgewertet. Dieses Verhalten führt aber leicht zu Programmierfehlern, wenn der Programmierer einen Ausdruck von links nach rechts liest, die Auswertung aber in entgegengesetzter Richtung erfolgt.

### 5.3.3 Kontrollstrukturen

Ein Compiler liest den Source-Code eines Programms und wandelt ihn in Maschinencode um, der von einem Prozessor ausgeführt werden kann. Die Programmiersprache kann durch eine Grammatik beschrieben werden, die als Teil die Expression-Term-Factor-Grammatik für arithmetische Ausdrücke enthält, dazu aber noch weiter Variablen und Regeln, die Kontrollstrukturen und deren Schachtelung beschreibt. Eine vereinfachte Grammatik für die Programmiersprache C könnte ungefähr folgende Regeln enthalten:

```

anweisungen → anweisung
      → anweisungen anweisung
anweisung → ausdruck
      → funktionsaufruf
      → if-struktur
      → while-struktur
if-struktur → if ( bedingung ) block else block
while-struktur → while ( bedingung ) block
block → anweisung
      → { anweisungen }
  
```

Der Compiler erstellt aus dem Source-Code zunächst einen Parse-Tree. Für das Programm von Abbildung 5.6 ist der Syntaxbaum in Abbildung 5.5 dargestellt. Anschließend wird

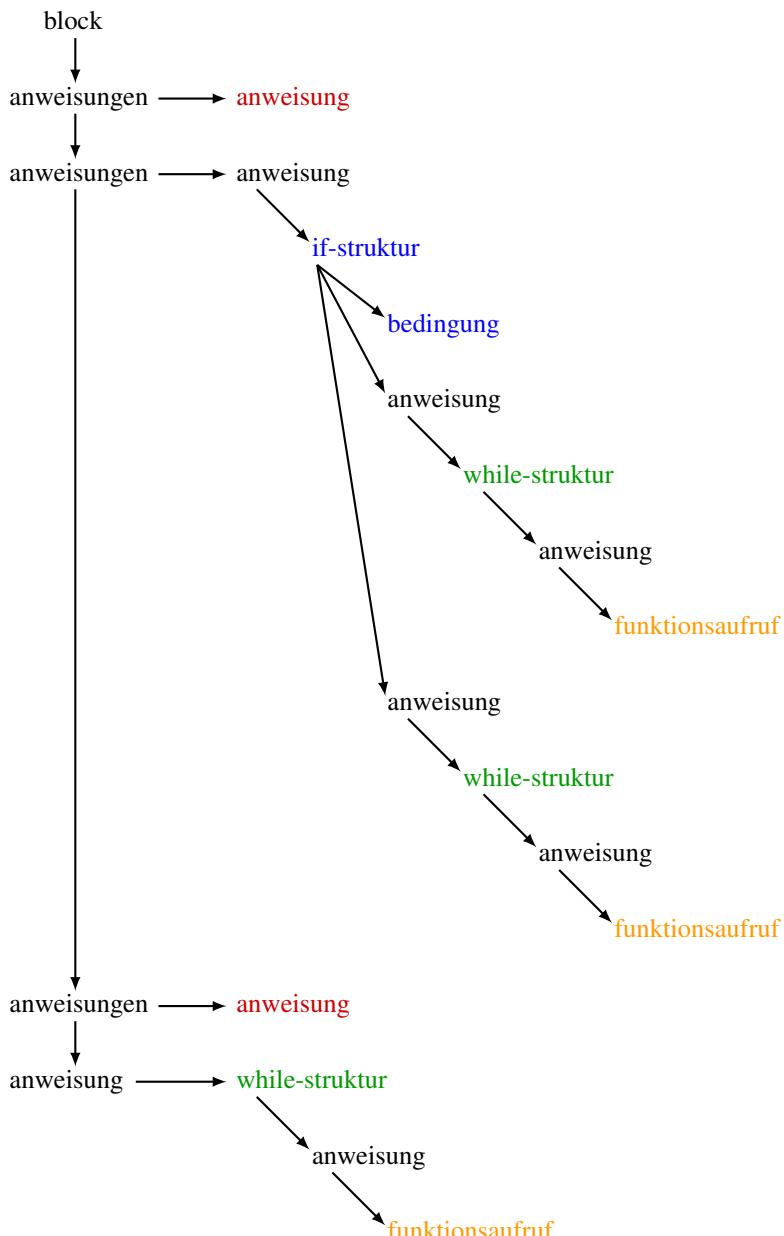


Abbildung 5.5: Leicht vereinfachter Syntaxbaum für das Programm von Abbildung 5.6. Beim Komplizieren wird jedem farbigen Knoten des Syntaxbaumes ein Assembler-Fragment zugeordnet.

```

.int      f_function(int x, int y) {
    int c = x % 10;
    if (x == 91) {
        while (c--) {
            y = g_function(y);
        }
    } else {
        while (c--) {
            y = h_function(y);
        }
    }
    c = x % 47;
    while (c--) {
        y = l_function(y);
    }
    return y;
}

    .section      __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0      sdk_version 14, 2
    .globl  _f_function ; -- Begin function f_function
    .p2align   2

_f_function:                         ; @f_function
    .cfi_startproc
; %bb.0:
    stp      x20, x19, [sp, #-32]!
    .cfi_def_cfa_offset 32
    stp      x29, x30, [sp, #16]
    add     x29, sp, #16
    .cfi_def_cfa w29, 16
    .cfi_offset w30, -8
    .cfi_offset w29, -16
    .cfi_offset w19, -24
    .cfi_offset w20, -32
    mov     x19, x0
    mov     w8, #26215
    movk   w8, #26214, lsl #16
    smull  x8, w0, w8
    lsr     x9, x8, #63
    asr     x8, x8, #34
    add     w8, w8, w9
    mov     w9, #10
    msub   w20, w8, w9, w0
    cmp     w0, #91
    b.ne   LBB0_3

; %bb.1:
    cbz   w20, LBB0_5
LBB0_2:
    sub   w20, w20, #1
    mov    x0, x1
    bl     _g_function
    mov    x1, x0
    cbnz  w20, LBB0_2
    b     LBB0_5

LBB0_3:
    cbz   w20, LBB0_5
LBB0_4:
    sub   w20, w20, #1
    mov    x0, x1
    bl     _h_function
    mov    x1, x0
    cbnz  w20, LBB0_4
LBB0_5:
    mov    w8, #16733
    movk   w8, #44620, lsl #16
    smull  x8, w19, w8
    lsr     x8, x8, #32
    add     w8, w8, w19
    asr     w9, w8, #5
    add     w8, w9, w8, lsr #31
    mov     w9, #47
    msub   w19, w8, w9, w19
    cbz   w19, LBB0_7
LBB0_6:
    sub   w19, w19, #1
    mov    x0, x1
    bl     _l_function
    mov    x1, x0
    cbnz  w19, LBB0_6
LBB0_7:
    mov    x0, x1
    ldp    x29, x30, [sp, #16]
    ldp    x20, x19, [sp], #32
    ret

    .cfi_endproc
}

    .subsections_via_symbols

```

Abbildung 5.6: C-Programm und kompilierter ARM-Maschinencode. Zu jedem Knoten des Parse-Trees wird ein gleichfarbig hinterlegtes Code-Fragment erzeugt.

jedem Knoten des Baumes ein Assembler-Fragment zugeordnet, welches die Kontrollstruktur oder Auswertung zur Laufzeit realisieren wird. In Abbildung 5.6 sind die Code-Fragmente farbig hinterlegt.

## 5.4 Reguläre Operationen

In Kapitel 4 wurden die regulären Operationen als die für die Verarbeitung von Zeichenketten bedeutungsvollen Operationen erkannt. Die Theorie der regulären Ausdrücke basierte auf ihnen. Reguläre Sprachen sind nicht nur unter regulären Operationen abgeschlossen, sie können aus Mengen von einzelnen Zeichen und regulären Operationen zusammengebaut werden. Es stellt sich damit die Frage, ob auch die kontextfreien Sprachen unter regulären Operationen abgeschlossen sind.

### 5.4.1 Reguläre Operationen für Grammatiken

Wir betrachten zwei kontextfreie Sprachen  $L_1$  und  $L_2$  über dem gemeinsamen Alphabet  $\Sigma$ . Da sie kontextfrei sind, gibt es Grammatiken

$$G_1 = (V_1, \Sigma, R_1, S_1) \quad \text{und} \quad G_2 = (V_2, \Sigma, R_2, S_2),$$

die die Sprachen  $L_1 = L(G_1)$  und  $L_2 = L(G_2)$  erzeugen. Wir nehmen außerdem an, dass die Variablen- und Regelmengen disjunkt sind, dass also  $V_1 \cap V_2 = \emptyset$  und  $R_1 \cap R_2 = \emptyset$ . Ohne diese Annahme könnten die vereinigten Regelmengen ganz neue Wörter erzeugen, die nichts mit den ursprünglichen Sprachen zu tun haben.

Der folgende Satz beantwortet die Frage, ob die regulären Operationen angewendet auf die Sprachen  $L_1$  und  $L_2$  wieder kontextfreie Sprachen sind.

**Satz 5.6.** Seien  $G_i = (V_i, \Sigma, R_i, S_i)$ ,  $i = 1, 2$ , kontextfreie Grammatiken mit disjunkten Variablen- und Regelmengen und  $S_0$  eine Variable, die nicht in  $V_1 \cup V_2$  enthalten ist. Dann gilt:

1. Die Alternative  $L_1 \mid L_2$  ist kontextfrei mit der Grammatik

$$G = (V_1 \cup V_2 \cup \{S_0\}, \Sigma, R = R_1 \cup R_2 \cup \{\textcolor{red}{S_0 \rightarrow S_1 \mid S_2}\}, S_0).$$

2. Die Verkettung  $L_1 L_2$  ist kontextfrei mit der Grammatik

$$G = (V_1 \cup V_2 \cup \{S_0\}, \Sigma, R = R_1 \cup R_2 \cup \{\textcolor{red}{S_0 \rightarrow S_1 S_2}\}, S_0).$$

3. Die Stern-Operation  $L_1^*$  ist kontextfrei mit der Grammatik

$$G = (V_1 \cup \{S_0\}, \Sigma, R = R_1 \cup \{\textcolor{red}{S_0 \rightarrow S_0 S_1, S_0 \rightarrow \varepsilon}\}, S_0).$$

Die drei Konstruktionen unterscheiden sich vor allem durch die hinzugefügten Regeln, die rot hervorgehoben sind.

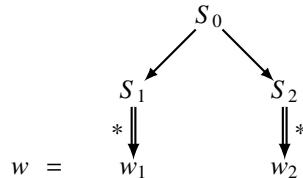
*Beweis.* In allen drei Fällen ist zu zeigen, dass die von der angegebenen Grammatik  $G$  erzeugte Sprache  $L(G)$  mit der durch die reguläre Operation erzeugte Sprache  $L$  übereinstimmt. Dazu muss man sich überlegen, dass sowohl  $L \subset L(G)$  als auch  $L(G) \subset L$  ist.

Dabei kann man verwenden, dass Ableitungen mit Regeln der Menge  $R_1 \cup R_2$  aus der Variablen  $S_1$  nur die Regeln von  $R_1$  brauchen, da die Regel- und Variablenmengen disjunkt sind. Ebenso brauchen Ableitungen aus der Variablen  $S_2$  nur die Regeln von  $R_2$ .

1. Wenn ein Wort  $w \in L_1 \cup L_2$  in  $L_1$  liegt, dann kann es mit den Regeln von  $R_1$  aus  $S_1$  abgeleitet werden. Da  $S_1$  mit der Regel  $S_0 \rightarrow S_1$  aus  $S_0$  abgeleitet werden kann, kann auch  $w$  mit den Regeln von  $R$  aus  $S_0$  abgeleitet werden. Dasselbe gilt für den Fall, dass  $w \in L_2$  ist. Somit ist  $L_1 \cup L_2 \subset L(G)$ .

Ist  $w \in L(G)$ , dann lässt sich  $w$  mit den Regeln von  $R$  aus  $S_0$  ableiten. Da  $S_0$  nur in den Regeln  $S_0 \rightarrow S_1$  und  $S_0 \rightarrow S_2$  vorkommt, muss der erste Schritt der Ableitung von  $w$  auf  $S_1$  oder  $S_2$  führen. Da  $w$  somit aus  $S_1$  oder  $S_2$  ableitbar ist folgt, dass  $w \in L_1 \cup L_2$  oder  $L(G) \subset L_1 \cup L_2$ .

2. Ein Wort  $w \in L_1 L_2$  besteht aus Teilwörtern  $w_1 \in L_1$  und  $w_2 \in L_2$ . Diese können aus  $S_1$  bzw.  $S_2$  abgeleitet werden. Also kann auch  $w_1 w_2 = w$  aus  $S_1 S_2$  abgeleitet werden und damit wegen der Regel  $S_0 \rightarrow S_1 S_2$  das Wort  $w$  aus  $S_0$ . Es folgt  $L_1 L_2 \subset L(G)$ . Diese Ableitung kann mit dem Syntaxbaum



illustriert werden.

Ist umgekehrt  $w \in L(G)$ , dann muss der erste Ableitungsschritt von  $w$  aus  $S_0$  die Regel  $S_0 \rightarrow S_1 S_2$  sein. Die nachfolgenden Ableitungsschritte erzeugen aus  $S_1$  ein Wort  $w_1 \in L_1$ , da dafür nur Regeln in  $R_1$  verwendet werden können, und aus  $S_2$  ein Wort  $w_2 \in L_2$ , da dafür nur Regeln in  $R_2$  verwendet werden können. Es folgt, dass  $w = w_1 w_2$  ist und damit  $L(G) \subset L_1 L_2$ .

3. Ein Wort in  $w \in L_1^*$  besteht aus Wörtern  $w_1 w_2 \dots w_n$  mit  $w_i \in L_1$ . Das Wort  $S_1^n$  kann durch  $n$ -fache Anwendung der Regel  $S_0 \rightarrow S_0 S_1$  und schließlich der Regel  $S_0 \rightarrow \varepsilon$  abgeleitet werden. Aus dem  $S_1$  an der  $i$ -ten Stelle kann das Wort  $w_i$  mit Regeln  $R_1$  aus  $S_1$  abgeleitet werden. Somit ist  $w \in L(G)$ . Der Syntaxbaum dieser Ableitung ist in Abbildung 5.7 dargestellt.

Umgekehrt sei  $w \in L(G)$ . Die einzigen Regeln, die  $S_0$  enthalten, sind  $S_0 \rightarrow S_0 S_1$  und  $S_0 \rightarrow \varepsilon$ , durch deren Anwendung sich eine Folge  $S_1^n$  erzeugen lässt. Die Ableitung von  $w$  aus  $S_0$  erzeugt also zunächst  $S_1^n$  und anschließend aus jedem  $S_1$  mit den Regeln von  $R_1$  je ein Wort  $w_i \in L_1$ . Somit ist  $w = w_1 \dots w_n \in L_1^*$  und damit  $L(G) \subset L_1^*$ .  $\square$

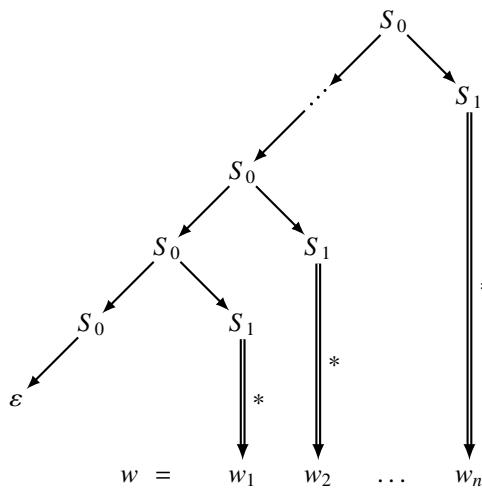


Abbildung 5.7: Syntaxbaum für die \*-Operation

In der Konstruktion der Regelmenge für die \*-Operation wurde Linksrekursion verwendet. Es wäre aber auch möglich gewesen, Rechtsrekursion zu verwenden. Die Linksrekursion führt aber zur vertrauteren Lesart von links nach rechts, daher ist ihr hier der Vorzug gegeben worden.

### 5.4.2 Reguläre Sprachen sind kontextfrei

In Kapitel 4 wurde gezeigt, dass sich jede reguläre Sprache aus dem leeren Wort, einzelnen Zeichen,  $\Sigma$  und regulären Operationen zusammenbauen lässt. Da die regulären Operationen von kontextfreien Sprachen bereits in Satz 5.6 als kontextfrei bewiesen worden sind, muss man sich nur noch überlegen, dass das leere Wort, einzelne Zeichen und  $\Sigma$  kontextfrei sind. Die kontextfreien Grammatiken

$$\begin{array}{llll} G_\varepsilon: & S_\varepsilon \rightarrow \varepsilon & \Rightarrow & L(G_\varepsilon) = \{\varepsilon\} \\ G_a: & S_a \rightarrow a & \Rightarrow & L(G_a) = \{a\} \\ G_\cdot: & S_\cdot \rightarrow t & \forall t \in \Sigma & \Rightarrow & L(G_\cdot) = \Sigma \end{array}$$

können dies leisten. Somit gilt der folgende Satz.

**Satz 5.7.** *Reguläre Sprachen sind kontextfrei.*

**Verständniskontrolle 5.6:** Die kontextfreien Grammatiken

$$\begin{array}{ll} S_1 \rightarrow S_1 P & S_2 \rightarrow \emptyset S_2 1 \\ P \rightarrow ZZ & \rightarrow \varepsilon \\ Z \rightarrow \emptyset \mid 1 & \end{array}$$



autospr.ch/v/5.6.pdf

erzeugen die Sprachen  $L_1 = \{w \in \Sigma^* \mid |w| \text{ ist gerade}\}$  und  $L_2 = \{\emptyset^n 1^n \mid n \geq 0\}$ .

- Konstruieren Sie eine Grammatik für die Sprache  $L = L_1 L_2 = \{w \in \Sigma^* \mid |w| \text{ gerade und } w \text{ endet mit } \emptyset^n 1^n, n \geq 0\}$
  - Konstruieren Sie eine Grammatik für die Sprache  $L = L_2^* = \{\emptyset^{n_1} 1^{n_1} \emptyset^{n_2} 1^{n_2} \dots \emptyset^{n_l} 1^{n_l} \mid n_i \geq 0 \forall i \leq l\}$
- 

## 5.5 Chomsky-Normalform

In Abschnitt 1.5 wurde für jeden deterministischen endlichen Automaten der Minimalautomat konstruiert, der insbesondere den Vergleich von Automaten ermöglichte, indem er eine Art ‘‘Normalform’’ dafür konstruierte. Da die Pumping-Length einer regulären Sprache die Anzahl der Zustände eines endlichen Automaten ist, bestimmt der Minimalautomat auch die minimale Pumping-Length. Dieser Abschnitt sucht nach einer ähnlichen Standardform für eine Grammatik.

### 5.5.1 Anforderungen an eine Normalform

Die grundlegendste Frage, die die gesuchte Normalform beantworten soll, ist die Frage, in wie vielen Schritten ein Wort mit einer Grammatik abgeleitet werden kann. Die folgenden Arten von Regeln machen die Beantwortung dieser Frage schwierig.

#### Unit-Rules

Regeln der Form

$$A \rightarrow B$$

heißen *Unit-Rules*. Wenn Unit-Rules eine Schleife  $A \rightarrow B \rightarrow C \rightarrow \dots \rightarrow A$  bilden, dann kann eine Ableitung beliebig verlängert werden. Idealerweise sollte also die Normalform einer Grammatik frei von Unit-Rules sein.

#### $\epsilon$ -Regeln

Regeln der Form  $A \rightarrow \epsilon$  heißen  *$\epsilon$ -Regeln*. Sie haben das Potential, neue Unit-Rules zu erzeugen, wie das Beispiel

$$\left. \begin{array}{l} A \rightarrow BC \\ B \rightarrow \epsilon \end{array} \right\} \quad \Rightarrow \quad A \rightarrow C$$

zeigt.  $\epsilon$ -Regeln sind daher ebenfalls zu vermeiden.

Nicht alle  $\epsilon$ -Regeln sind vermeidbar. Wenn die Grammatik das leere Wort produzieren kann, dann ist dafür mindestens eine  $\epsilon$ -Regel nötig. Mit der Regel  $S \rightarrow \epsilon$  für die Startvariable  $S$  der Grammatik ist dies möglich.

### Startvariable

Damit die Regel  $S \rightarrow \varepsilon$  nicht wieder zum unerwünschten Auftreten von Unit-Rules führt, muss sichergestellt sein, dass die Variable  $S$  in keiner anderen Regel auf der rechten Seite vorkommen kann.

Wenn keine  $\varepsilon$ -Regeln vorhanden sind, die in einer Ableitung angewendet werden können, dann kann die Länge eines Wortes während der Ableitung auch nicht mehr abnehmen. In jedem Ableitungsschritt bleibt die Länge des Wortes gleich oder nimmt zu.

### “Lange” Regeln

Die Regeln der Form  $A \rightarrow BCDEF$  lassen ein Wort sehr rasch explodieren und erschweren damit die Bestimmung der Länge der Ableitung. Es ist daher wünschbar, dass die Wortlänge in jedem Ableitungsschritt nur um ein Zeichen wachsen kann. Durch Zerlegen der Regel in kleinere Schritte nach dem Muster

$$A \rightarrow BCDEF \quad \Rightarrow \quad \left\{ \begin{array}{l} A \rightarrow BB_1 \\ B_1 \rightarrow CC_1 \\ C_1 \rightarrow DD_1 \\ D_1 \rightarrow EF \end{array} \right.$$

lässt sich dies erreichen.

### Umwandlung in Terminalsymbole

Die Ableitung eines Wortes aus der Startvariablen  $S$  erzeugt mithilfe von Regeln, die mehr als ein Zeichen auf der rechten Seite haben, ein ausreichend langes Wort. Regeln können aber nur auf Variablen auf der rechten Seite angewendet werden. Spätestens am Ende der Ableitung müssen alle diese Variablen durch Terminalsymbole ersetzt werden. Für die Bestimmung der Länge der Ableitung ist es wichtig, diese beiden Schritte zu trennen. Die Regel der Form  $A \rightarrow bC$  kann in die beiden Schritte

$$\left. \begin{array}{l} A \rightarrow BC \\ B \rightarrow b \end{array} \right\} \quad \Rightarrow \quad A \Rightarrow BC \Rightarrow bC$$

zerlegt werden.

### Chomsky-Normalform

Die in den vorangegangenen Abschnitten zusammengetragenen wünschbaren Eigenschaften einer Grammatik führen auf die idealisierte Form der Grammatik der folgenden Definition.

**Definition 5.8** (Chomsky-Normalform). *Eine kontextfreie Grammatik  $G = (V, \Sigma, R, S)$  ist in Chomsky-Normalform (CNF), wenn sie die folgenden Eigenschaften hat:*

1. Die Startvariable  $S$  kommt nicht auf der rechten Seite einer Regel vor.

2. Alle Regeln, mit der Ausnahme in 3., sind von der Form

$$A \rightarrow BC \quad \text{oder} \quad A \rightarrow a,$$

sie haben also genau zwei Variablen oder genau ein Terminalsymbol auf der rechten Seite.

3. R kann als einzige  $\varepsilon$ -Regel die Regel  $S \rightarrow \varepsilon$  enthalten, wenn die Sprache  $L(G)$  das leere Wort enthält.

Für eine Grammatik in Chomsky-Normalform lässt sich jetzt die Frage nach der Länge einer Ableitung vollständig beantworten.

**Satz 5.9.** Ist  $G$  eine kontextfreie Grammatik in Chomsky-Normalform, dann wird ein Wort  $w \in L(G)$  in  $2|w| - 1$  Schritten abgeleitet.

*Beweis.* Zur Erzeugung eines Wortes der Länge  $n = |w|$  ist die Anwendung von  $n - 1$  Regeln der Form  $A \rightarrow BC$  notwendig. Das so erzeugte Wort besteht aus  $n$  Variablen, die jetzt durch Anwendung von  $n$  Terminalsymbolregeln der Form  $T \rightarrow t$  in Terminalsymbole umgewandelt werden müssen. Damit ist die Ableitung in  $n + n - 1 = 2|w| - 1$  Schritten erfolgt.  $\square$

## 5.5.2 Umwandlung in Chomsky-Normalform

Die Eigenschaften der Chomsky-Normalform können durch einfache Transformationen der Regeln erreicht werden, wie in den folgenden Abschnitten illustriert werden soll. Wir gehen aus von einer Grammatik  $G = (V, \Sigma, R, S)$ , die nicht in Chomsky-Normalform zu sein braucht.

### 0. Schritt: Startvariable

Kommt die Startvariable  $S$  auf der rechten Seite einer Regel vor, dann kann durch Hinzufügen einer neuen Startvariable  $S_0$  zu den Variablen und einer neuen Regel  $S_0 \rightarrow S$  zu den Regeln eine neue Grammatik  $G' = (V \cup \{S_0\}, \Sigma, R \cup \{S_0 \rightarrow S\}, S_0)$  konstruiert werden, die die gleiche Sprache erzeugt. Der Schritt erzeugt eine neue Unit-Rule und muss daher erfolgen, bevor versucht wird, später im Schritt 2 Unit-Rules zu entfernen.

### 1. Schritt: $\varepsilon$ -Regeln

Eine  $\varepsilon$ -Regel  $A \rightarrow \varepsilon$  bedeutet, dass die Variable  $A$  auf der rechten Seite einer Regel weggelassen werden kann. Die  $\varepsilon$ -Regel ist nicht mehr nötig, sobald die Möglichkeit,  $A$  auf der rechten Seite wegzulassen, in allen Regeln angewendet worden ist. Im folgenden Beispiel sind die neuen Regeln rot hervorgehoben:

$$\left. \begin{array}{l} A \rightarrow \varepsilon \\ B \rightarrow ABC \\ C \rightarrow AA \end{array} \right\} \quad \text{wird zu} \quad \left\{ \begin{array}{l} B \rightarrow ABC \mid \textcolor{red}{BC} \\ C \rightarrow AA \mid \textcolor{red}{A} \mid \varepsilon. \end{array} \right.$$

Die letzte Regel zeigt, dass durch das Entfernen von  $\varepsilon$ -Regeln andere  $\varepsilon$ -Regeln ebenso wie neue Unit-Rules entstehen können. Dieser Schritt muss daher mehrmals wiederholt werden, bis höchstens die Regel  $S \rightarrow \varepsilon$  übrig bleibt, die sich nicht mehr entfernen lässt, die aber in der Chomsky-Normalform zulässig ist. Erst dann darf dazu übergegangen werden, Unit-Rules zu entfernen.

## 2. Schritt: Unit-Rules

Eine Unit-Rule  $A \rightarrow B$  bedeutet, dass man aus  $A$  alles ableiten kann, was man auch aus  $B$  ableiten kann. Man kann die Unit-Rule also zum Verschwinden bringen, indem man die rechte Seite durch all das ersetzt, was Regeln mit  $B$  als linker Seite ermöglichen, zum Beispiel

$$\left. \begin{array}{l} A \rightarrow B \\ B \rightarrow U | AV | UVW \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} A \rightarrow U | AV | UVW \\ B \rightarrow U | AV | UVW. \end{array} \right.$$

Durch den Prozess können neue Unit-Rules wie im Beispiel  $A \rightarrow U$  entstehen, der Schritt muss also wiederholt werden, bis alle Unit-Rules eliminiert sind.

## 3. Schritt: Komplexe Regeln

Regeln mit mehr als einem Zeichen auf der rechten Seite sollen nur Variablen auf der rechten Seite enthalten. Falls eine solche Regel auf der rechten Seite ein Terminalsymbol  $u$  enthält, dann kann es durch eine neue Variable  $U$  ersetzt werden, welches mit einer neuen Terminalsymbolregel  $U \rightarrow u$  in das Zielsymbol umgewandelt werden kann.

Falls auf der rechten Seite einer Regel mehr als zwei Variablen stehen, dann muss sie in mehrere einzelne Schritte zerlegt werden:

$$A \rightarrow U_1 U_2 \dots U_n \Rightarrow \left\{ \begin{array}{l} A \rightarrow U_1 A_1 \\ A_1 \rightarrow U_2 A_2 \\ \vdots \\ A_{n-3} \rightarrow U_{n-2} A_{n-2} \\ A_{n-2} \rightarrow U_{n-1} U_n. \end{array} \right.$$

Damit lassen sich alle Regeln mit mehr als zwei Symbolen auf der rechten Seite auf Regeln mit genau zwei Variablen auf der rechten Seite reduzieren.

Durch Anwendung der Schritte 0 bis 3 lässt sich jede kontextfreie Grammatik in Chomsky-Normalform bringen. Dies beweist den folgenden Satz.

**Satz 5.10** (Chomsky-Normalform). *Ist  $G$  eine kontextfreie Grammatik, dann gibt es eine kontextfreie Grammatik  $G'$  in Chomsky-Normalform, die die gleiche Sprache  $L(G') = L(G)$  produziert.*

Die Chomsky-Normalform einer Grammatik ist nicht eindeutig. Die Reihenfolge, in der die einzelnen  $\varepsilon$ -Regeln und Unit-Rules entfernt werden, beeinflusst das Resultat.

### 5.5.3 Beispiel für die Umwandlung in Chomsky-Normalform

Das folgende Beispiel illustriert den Algorithmus für die Umwandlung der Grammatik in Chomsky-Normalform. Die Grammatik, die umgewandelt werden soll, hat die Regeln

$$\begin{aligned} S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \varepsilon. \end{aligned} \tag{5.9}$$

#### 0. Schritt: Startvariable

Die Startvariable  $S$  kommt auf der rechten Seite vor, daher muss die neue Startvariable  $S_0$  und die Regel  $S_0 \rightarrow S$  hinzugefügt werden:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \varepsilon. \end{aligned} \tag{5.10}$$

#### 1. Schritt: $\varepsilon$ -Regel entfernen

In der Chomsky-Normalform ist nur die  $\varepsilon$ -Regel  $S_0 \rightarrow \varepsilon$  zulässig. Die Grammatik (5.10) enthält die  $\varepsilon$ -Regel  $B \rightarrow \varepsilon$ , die entfernt werden muss:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid aB \mid \textcolor{red}{a} \\ A &\rightarrow B \mid \textcolor{red}{\varepsilon} \mid S \\ B &\rightarrow b. \end{aligned} \tag{5.11}$$

Die neu entstandene  $\varepsilon$ -Regel  $A \rightarrow \varepsilon$  muss ebenfalls entfernt werden:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid \textcolor{red}{SA} \mid \textcolor{red}{AS} \mid aB \mid \textcolor{red}{a} \\ A &\rightarrow B \mid S \\ B &\rightarrow b. \end{aligned} \tag{5.12}$$

Beim Weglassen beider  $A$  in der zweiten Regel entsteht auch die nutzlose Regel  $S \rightarrow S$ , die weggelassen werden kann. Damit sind alle  $\varepsilon$ -Regeln verschwunden. Offenbar ist es mit dieser Grammatik nicht möglich, das leere Wort abzuleiten.

#### 2. Schritt: Unit-Rules entfernen

Die Grammatik (5.12) enthält die drei Unit-Rules  $A \rightarrow B$ ,  $A \rightarrow S$  und  $S_0 \rightarrow S$ . Im Allgemeinen kommt es auf die Reihenfolge an, in der die Regeln eliminiert werden, wir wählen die oben angegebene Reihenfolge.

Die Regel  $A \rightarrow B$  bedeutet, dass man alles aus  $A$  machen kann, was man auch aus  $B$  machen kann. Elimination von  $A \rightarrow B$  ergibt daher

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid SA \mid AS \mid aB \mid a \\ A &\rightarrow b \mid S \\ B &\rightarrow b. \end{aligned} \tag{5.13}$$

Die Regel  $A \rightarrow S$  bedeutet, dass aus  $A$  alles produziert werden kann, was aus  $S$  produziert werden kann. Damit bleiben die Regeln

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid SA \mid AS \mid aB \mid a \\ A &\rightarrow ASA \mid SA \mid AS \mid aB \mid a \mid b \\ B &\rightarrow b. \end{aligned} \tag{5.14}$$

Es bleibt die Regel  $S_0 \rightarrow S$ , die bedeutet, dass aus  $S_0$  alles produziert werden kann, was aus  $S$  produziert werden kann. Damit werden die Regeln zu

$$\begin{aligned} S_0 &\rightarrow ASA \mid SA \mid AS \mid aB \mid a \\ S &\rightarrow ASA \mid SA \mid AS \mid aB \mid a \\ A &\rightarrow ASA \mid SA \mid AS \mid aB \mid a \mid b \\ B &\rightarrow b. \end{aligned} \tag{5.15}$$

### 3. Schritt: Komplexe Regeln

Die Grammatik (5.15) enthält in den ersten drei Regeln rechte Seiten mit drei Variablen oder mit einer Mischung von Terminalsymbolen und Variablen. Die rechte Seite  $ASA$  kann mit einer neuen Variable  $A_1$  und der Regel  $A_1 \rightarrow SA$  erzeugt werden. Das Terminalsymbol in der rechten Seite  $aB$  kann durch eine neue Variable  $U$  und die Regel  $U \rightarrow a$  erzeugt werden. Damit erreichen wir die Chomsky-Normalform

$$\begin{aligned} S_0 &\rightarrow AA_1 \mid SA \mid AS \mid UB \mid a \\ S &\rightarrow AA_1 \mid SA \mid AS \mid UB \mid a \\ A &\rightarrow AA_1 \mid SA \mid AS \mid UB \mid a \mid b \\ A_1 &\rightarrow SA \\ U &\rightarrow a \\ B &\rightarrow b \end{aligned} \tag{5.16}$$

der Grammatik.

**Verständniskontrolle 5.7:** Üben Sie die Regeln zur Reduktion auf Chomsky-Normalform an den folgenden Beispielen:

- a) Wenden Sie die  $\varepsilon$ -Regel  $A \rightarrow \varepsilon$  auf die folgenden Regeln an:

$$B \rightarrow AC \mid CA \mid ACA$$

$$D \rightarrow AAAA$$

$$A \rightarrow \varepsilon$$



autospr.ch/v/5.7.pdf

- b) Eliminieren Sie die Unit-Rule  $A \rightarrow B$  aus den folgenden Regeln

$$C \rightarrow AC$$

$$B \rightarrow AB \mid BB \mid C$$

$$A \rightarrow B$$

- c) Reduzieren Sie die folgenden Regeln auf Chomsky-Normalform

$$A \rightarrow AbC \mid e$$

$$C \rightarrow Ad$$

## Übungsaufgaben

**5.1.** Formulieren Sie eine kontextfreie Grammatik für die folgenden Sprachen über den Terminalsymbolen  $\Sigma = \{\emptyset, 1\}$ :

- a)  $L = \Sigma^*$
- b) Wörter, die mit dem gleichen Symbol enden, mit dem sie beginnen.
- c)  $L$  besteht aus den Wörtern gerader Länge.
- d)  $L = \{\emptyset^n 1^m \mid n > m\}$

**5.2.** Für arithmetische Ausdrücke mit den Grundoperationen + und - stehen die folgenden zwei kontextfreie Grammatiken zur Auswahl. Einerseits eine vereinfachte Variante der in Abschnitt 5.3.1 besprochenen Expression-Term-Faktor-Grammatik:

$$\begin{aligned} \text{Ausdruck} &\rightarrow \text{Zahl } '+' \text{ Ausdruck} \\ &\rightarrow \text{Zahl } '-' \text{ Ausdruck} \\ &\rightarrow \text{Zahl}, \end{aligned}$$

die Rechtsrekursion statt Linksrekursion verwendet. Andererseits die folgende Grammatik:

$$\text{Ausdruck} \rightarrow \text{Term Termsequenz}$$

Termsequenz  $\rightarrow '+'$  Term Termsequenz  
 $\rightarrow '-'$  Term Termsequenz  
 $\rightarrow \varepsilon$   
 Term  $\rightarrow$  Zahl.

Die beiden Grammatiken verwenden natürlich die gleichen Regeln für das nichtterminale Symbol ‘Zahl’.

- a) Erstellen Sie für jede Grammatik den Syntaxbaum für den Ausdruck

$$47 - 1291 + 1848$$

- b) Formulieren Sie Regeln, wie der arithmetische Ausdruck auf der Basis des Syntaxbaums auszuwerten ist.  
 c) Welche Grammatik ist vorzuziehen?

**5.3.** In einer SQL INSERT Query müssen Spaltennamen der Datenbank und Werte in gleicher Zahl angegeben werden:

```
INSERT INTO blubb(a1, b2, c3) VALUES('wert1', 'wert2', 1291);
```

Es stellt sich die Frage, ob man bereits durch die Grammatik sicherstellen kann, dass gleich viele Spaltennamen wie Werte angegeben werden. Betrachten Sie dazu die Sprache über dem Alphabet  $\Sigma = \{(), ,\}$ , bestehend aus Wörtern der Form

$$w = (\underbrace{, , \dots, }_n) (\underbrace{, , \dots, }_n)$$

wobei sich zwischen den beiden Klammern gleich viele Kommata befinden. Ist diese Sprache kontextfrei?

**5.4.** Betrachten Sie die kontextfreie Grammatik

$R \rightarrow 'a' \mid 'b' \mid 'c' \mid \dots \mid 'z' \mid '.'$   
 $\rightarrow R \cdot^*$   
 $\rightarrow R \cdot \mid R$   
 $\rightarrow RR$   
 $\rightarrow (' R )'$

über dem Alphabet bestehend aus allen Zeichen, die in der Grammatik durch einfache Anführungszeichen als Terminalssymbole gekennzeichnet wurden. Finden Sie die Ableitung für die folgenden Wörter mit dieser Grammatik:

- a)  $\cdot^*$   
 b)  $(a \mid b)$

c)  $(ab)^* | (cd)^*$

### 5.5. Die kontextfreie Grammatik

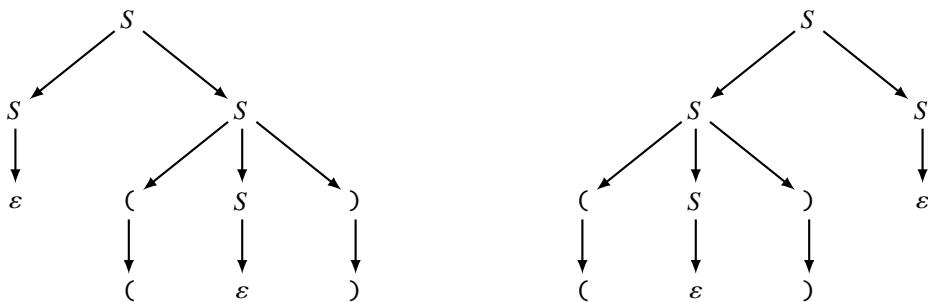
$$\begin{aligned} S &\rightarrow A \\ &\rightarrow Bc \\ A &\rightarrow AB \\ &\rightarrow ABC \\ &\rightarrow a \\ B &\rightarrow b \\ C &\rightarrow cc \end{aligned}$$

hat nicht Chomsky-Normalform. Finden Sie eine äquivalente Grammatik in Chomsky-Normalform.

### 5.6. Für die Sprache der korrekt geschachtelten Klammerausdrücke wurde die Grammatik

$$S \rightarrow (S) \mid SS \mid \varepsilon \quad (5.17)$$

gefunden, die sich jedoch als nicht eindeutig herausgestellt hat. Das Wort  $()$  hat zum Beispiel die beiden völlig verschiedenen Parse-Trees



Betrachten Sie als Alternative die Grammatik

$$\begin{aligned} S &\rightarrow (S)S \\ &\rightarrow \varepsilon \end{aligned} \quad (5.18)$$

- Finden Sie den Parse-Tree von  $()$  in der Grammatik (5.18).
- Ist die Grammatik (5.18) eindeutig?
- Finden Sie die Chomsky-Normalform dieser Grammatik.



# Kapitel 6

## Parsing

Eine kontextfreie Grammatik ist zwar in der Lage, die Syntax einer Programmiersprache mathematisch zu beschreiben, für die Praxis erwartet man aber mehr. Die mathematische Formulierung der Grammatik einer Programmiersprache ist nicht maschinenlesbar. Sie ermöglicht also nicht, die Grammatik als Sourcecode für die Erzeugung eines Compilers oder Interpreters für die Sprache zu verwenden. Dafür sind Formate wie die Extended Backus-Naur-Form entwickelt worden, über deren Geschichte der Abschnitt 6.1 berichtet. Ausgehend von der Grammatik soll anschließend in Abschnitt 6.2 der Cocke-Younger-Kasami-Algorithmus konstruiert werden, welcher die Ableitbarkeit eines Wortes aus der Grammatik feststellen und einen Syntaxbaum konstruieren kann. Für die praktische Durchführung ist die Formulierung von Abschnitt 6.2 allerdings aus Laufzeitgründen unzweckmäßig. Die auf *dynamic programming* basierende Implementation von Abschnitt 6.3 erreicht Laufzeit  $O(|w|^3)$ .

### 6.1 Spezifikation von Programmiersprachen

Programmiersprachen spielten in der Entwicklung der Informatik in der Mitte des 20. Jahrhunderts eine besondere Rolle. Es wurde erkannt, dass sie sich mit kontextfreien Grammatiken beschreiben lassen, aus denen im Idealfall sogar Code für einen Parser oder Compiler generiert werden kann. In diesem Abschnitt soll ein kurzer historischer Überblick über diese Entwicklung gegeben werden.

#### 6.1.1 Geschichte

Programme für die ersten Computer wurden meist als Folgen von Maschineninstruktionen direkt in die Computer eingegeben. Sie waren spezifisch für die zur Verfügung stehende CPU. Sie waren nicht portabel, konnten also nur auf einem Maschinentyp laufen. Inves-

titionen in Software waren damit beim Wechsel auf eine andere Hardware gefährdet. Außerdem war die Programmierung auf Maschinenebene fehleranfällig und zeitaufwendig.

## Frühe Programmiersprachen

Das Konzept einer Programmiersprache taucht erstmals im Werk des deutschen Computerpioniers Konrad Zuse auf. Sein *Plankalkül* beschreibt die Operationen eines Computers symbolisch, also in Form eines Textes. Weitere frühe Programmiersprachen waren ebenfalls im Wesentlichen Assembler-Sprachen, also symbolische Darstellungen der Maschinieninstruktionen.

Die Sprache FORTRAN entstand auf Anregung von John Backus bei IBM als praktischere Alternative für die bis dahin übliche Entwicklung in der Assembler-Sprache für die Rechner der IBM 704-Reihe. Das Projekt brachte 1957 den ersten kommerziell erhältlichen Compiler für eine Programmiersprache hervor. Der FORTRAN-Compiler übersetzt den Sourcecode in den meisten Fällen in effizienten Maschinencode, so dass Entwickler auf Assembler verzichten konnten. FORTRAN-Programme bestehen aus einer Folge von Anweisungen, die meist Zeile für Zeile übersetzt werden konnten. Arithmetische Ausdrücke wie auch Kontrollstrukturen konnten nicht geschachtelt werden.

Die Sprache FORTRAN macht kaum Annahmen über die Hardware, auf der das Programm läuft. Ein FORTRAN-Programm kann auf einer neuen Maschine einfach neu kompiliert werden. Damit wird die Investition in die Erstellung des Programms geschützt. Einzige Voraussetzung ist die Verfügbarkeit eines Compilers. Genau dies war auch eine der treibenden Kräfte hinter der Entwicklung des portablen GNU C-Compilers. Eine gemeinsame höhere Programmiersprache macht Open Source Software erst möglich.

## ALGOL 58 und ALGOL 60

Die Programmiersprache ALGOL 58 war das Resultat von Bemühungen, eine internationale algorithmische Sprache zu erschaffen, in der numerische Methoden kommuniziert und Rechenprozesse implementiert werden konnten. John Backus hat die Sprache in Form einer Grammatik beschrieben, die allerdings nicht vielen Lesern bekannt war. Peter Naur stellt erst bei der Lektüre der Grammatik fest, dass seine Interpretation der Sprache in einigen Punkten wesentlich von der Interpretation von Backus abwich. Er schloss daraus, dass Programmiersprachen in Form einer Grammatik spezifiziert werden sollten, um solchen Missverständnissen vorzubeugen.

ALGOL 58 verwendet die Expression-Term-Factor-Grammatik für arithmetische Ausdrücke, in der Klammern mindestens im Prinzip beliebig tief geschachtelt werden können. ALGOL 58 führte außerdem das Konzept der *compound statements* ein. Damit wurde es möglich, Anweisungen zusammenzufassen und es waren die Voraussetzungen geschaffen, dass auch Kontrollstrukturen geschachtelt werden konnten.

ALGOL 60 führte schließlich die Blockstrukturen ein, die in jeder modernen Programmiersprache zur Verfügung stehen.

## BNF

Peter Naur formulierte die Grammatik in einem Format, das er Backus-Normal-Form (BNF) nannte, die später auf Anregung von Donald Knuth in Backus-Naur-Form umbenannt wurde. Die BNF verwendet Zeichenketten in spitzen Klammern in der Form `<variable>` als Variablennamen, Regeln werden mit `::=` anstelle des Pfeils dargestellt, der Vertikalstrich drückt die Alternative aus.

Die Grammatik

$$S \rightarrow \varepsilon \mid \emptyset S 1$$

für die Sprache  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  kann daher in BNF als

$$\langle \text{string} \rangle ::= '' \mid \emptyset \langle \text{string} \rangle 1$$

formuliert werden.

In der Spezifikation von ALGOL 60 findet man die Expression-Term-Factor-Grammatik in der leicht gekürzten Form

```

<simple arithmetic expression> ::= <term> | <adding operator> <term>
    | <simple arithmetic expression> <adding operator> <term>
<adding operator> ::= + | -
<term> ::= <factor> | <term> <multiplying operator> <factor>
<multiplying operator> ::= × | / | ÷
<factor> ::= <primary> | <factor> | <factor> ↑ <primary>
<primary> ::= <unsigned number> | ( <arithmetic expression> )
<unsigned number> ::= <decimal number>

```

Weitere Regeln drücken aus, was genau `<decimal number>` alles sein kann. Der Zeilenumbruch in der ersten Regel ist für die Darstellung in diesem Buch verwendet worden, er ist in BNF nicht möglich.

### 6.1.2 Extended Backus-Naur-Form

Aus den Ideen der Backus-Naur-Form entwickelte sich später eine erweiterte Syntaxnotation, die einige Schwächen der ursprünglichen BNF beheben soll. Einige Änderungen gegenüber BNF werden in Tabelle 6.1 zusammengestellt. Die EBNF wurde als ISO/EC 14977:1996 standardisiert.

### 6.1.3 Beispiele von Grammatiken moderner Programmiersprachen

Ein paar Beispiele sollen zeigen, wie die Spezifikation von Programmiersprachen mithilfe von Grammatiken allgemein üblich ist. Dabei zeigt sich, dass für arithmetische Ausdrücke immer wieder die bekannte Expression-Term-Factor-Grammatik verwendet wird, die schon in Abschnitt 5.3.1 erfolgreich war.

Funktion	EBNF	BNF
Variable	variable	<variable>
→	=	::=
Verkettung	,	(nicht nötig)
Regelende	;	(nicht nötig)
Alternative		
Terminalsymbol	'a'	a
Zeichenkette	"text"	"text"
Wiederholung	{ ... }	
Option	[ ... ]	
Kommentar	(* ... *)	
Ausnahme	-	

Tabelle 6.1: Einige der in EBNF üblichen Symbole mit ihrer Entsprechung in BNF.

## Pascal

Eines der frühen Beispiele einer Programmiersprache, die mithilfe einer Grammatik spezifiziert worden ist, ist die von Niklaus Wirth entworfene Sprache Pascal. Die Expression-Term-Factor-Grammatik taucht als Teilgrammatik in der leicht gekürzten Form

```

simple-expression =
    [ sign ] term { addition-operator term } .
term =
    factor { multiplication-operator factor } .
factor =
    variable | number | string | set | nil | constant-identifier |
    bound-identifier | function-designator | "(" expression ")" |
    not factor .
relational-operator =
    "=" | "<>" | "<" | "<=" | ">" | ">=" | "in" .
addition-operator =
    "+" | "-" | or .
multiplication-operator =
    "*" | "/" | div | mod | and .

```

auf. Es ist erkennbar, dass diese Spezifikation, die im ISO-Standard 7185 für Pascal auf-taucht, von den Konventionen der EBNF abweicht.

Pascal hat eine relativ große Auswahl an Kontrollstrukturen, die beliebig geschachtelt werden können. Der folgende Ausschnitt aus der Grammatik zeigt, dass jede Kontroll-struktur auch ein **statement** ist, welches in einem **compound-statement** eingeschach-telt sein kann:

```

statement-sequence =
    statement { ";" statement } .
statement =
    [ label ":" ] (simple-statement | structured-statement) .
...

```

```

structured-statement =
    compound-statement | repetitive-statement | conditional-statement |
    with-statement .
compound-statement =
    "begin" statement-sequence "end" .
repetitive-statement =
    while-statement | repeat-statement | for-statement .
while-statement =
    "while" expression "do" statement .
repeat-statement =
    "repeat" statement-sequence "until" expression .
for-statement =
    for variable-identifier ":" initial-expression (to | downto)
    final-expression do statement .
....
conditional-statement =
    if-statement | case-statement .
if-statement =
    "if" expression "then" statement [ "else" statement ] .
case-statement =
    "case" expression "of" case-limb { ";" case-limb } [ ";" ] "end" .
case-limb =
    case-label-list ":" statement .
case-label-list =
    constant { "," constant } .
with-statement =
    "with" record-variable { "," record-variable } "do" statement .

```

## Die Sprachen der C-Familie

Ausgehend von der Sprache C, die 1972 von Dennis Ritchie entworfen wurde, entstand eine ganze Familie von Sprachen, die einen beträchtlichen Teil der Grammatik gemeinsam haben. Dazu gehören neben C und C++ auch PHP, JavaScript und Java. Sie verwendet geschweifte Klammern zur Bildung von Blöcken von Anweisungen, die in Kontrollstrukturen verwendet werden können und ihrerseits wieder Kontrollstrukturen enthalten können. C verfügt über eine große Zahl von Operatoren, was zu komplexen Regeln für die Auswertungsreihenfolge führt. Die Grammatik enthält als Teil eine deutlich erweiterte Variante der Expression-Term-Factor-Grammatik, die den zusätzlichen Operatoren Rechnung trägt:

```

<conditional-expression> ::= <logical-or-expression>
                           | <logical-or-expression> ? <expression> : <conditional-expression>

<logical-or-expression> ::= <logical-and-expression>
                           | <logical-or-expression> || <logical-and-expression>

<logical-and-expression> ::= <inclusive-or-expression>
                           | <logical-and-expression> && <inclusive-or-expression>

<inclusive-or-expression> ::= <exclusive-or-expression>
                           | <inclusive-or-expression> | <exclusive-or-expression>

```

```

<exclusive-or-expression> ::= <and-expression>
                           | <exclusive-or-expression> ^ <and-expression>

<and-expression> ::= <equality-expression>
                     | <and-expression> & <equality-expression>

<equality-expression> ::= <relational-expression>
                         | <equality-expression> == <relational-expression>
                         | <equality-expression> != <relational-expression>

<relational-expression> ::= <shift-expression>
                           | <relational-expression> < <shift-expression>
                           | <relational-expression> > <shift-expression>
                           | <relational-expression> <= <shift-expression>
                           | <relational-expression> >= <shift-expression>

<shift-expression> ::= <additive-expression>
                      | <shift-expression> << additive-expression>
                      | <shift-expression> >> <additive-expression>

<additive-expression> ::= <multiplicative-expression>
                           | <additive-expression> + <multiplicative-expression>
                           | <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <cast-expression>
                               | <multiplicative-expression> * <cast-expression>
                               | <multiplicative-expression> / <cast-expression>
                               | <multiplicative-expression> % <cast-expression>

<cast-expression> ::= <unary-expression>
                     | ( <type-name> ) <cast-expression>

<unary-expression> ::= <postfix-expression>
                      | ++ <unary-expression>
                      | -- <unary-expression>
                      | <unary-operator> <cast-expression>
                      | sizeof <unary-expression>
                      | sizeof <type-name>

```

Den Ausdrücken der Expression-Term-Factor-Grammatik entsprechen die **additive-expressions**, die **multiplicative-expressions** sind Terme und den Faktoren entsprechen die **cast-expressions**.

## XPath

XPath ist eine Syntax, mit der Knoten im DOM eines XML-Dokuments spezifiziert werden können. Als Teil von XPath können auch arithmetische Ausdrücke berechnet werden, wofür man in der Spezifikation [8] die folgenden Grammatik-Regeln findet:

```

[25] AdditiveExpr      ::=  MultiplicativeExpr
                           | AdditiveExpr '+' MultiplicativeExpr
                           | AdditiveExpr '-' MultiplicativeExpr
[26] MultiplicativeExpr ::=  UnaryExpr
                           | MultiplicativeExpr MultiplyOperator UnaryExpr
                           | MultiplicativeExpr 'div' UnaryExpr
                           | MultiplicativeExpr 'mod' UnaryExpr
[27] UnaryExpr         ::=  UnionExpr
                           | '-' UnaryExpr

```

Auch dies ist eine Form der Expression-Term-Factor-Grammatik.

### 6.1.4 Syntaxanalyse auf der Basis einer Grammatik

Da Programmiersprachen, wie die Beispiele eindrücklich gezeigt haben, durch Grammatiken nach oft ähnlichen Prinzipien spezifiziert werden können, bietet sich die Grammatik auch als die Basis für die Entwicklung eines Compilers an. Dazu ist notwendig, direkt aus der Grammatik einen Algorithmus abzuleiten, der einen Syntaxbaum des zu kompilierenden Sourcecodes erstellen kann. Diese Aufgabe soll im nächsten Abschnitt gelöst werden.

## 6.2 Der Cocke-Younger-Kasami-Algorithmus

Gesucht ist ein Algorithmus, der die Frage beantworten kann, ob ein Wort  $w \in \Sigma^*$  aus einer gegebenen Grammatik  $G = (V, \Sigma, R, S)$  abgeleitet werden kann, ob also  $S \xrightarrow{*} w$ .

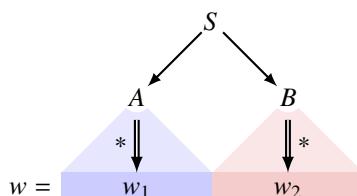
### 6.2.1 Ideen

Die Chomsky-Normalform von Abschnitt 5.5 vereinfacht die Regeln, die in einer Grammatik vorkommen können. Im Folgenden soll daher angenommen werden, dass die Grammatik bereits Chomsky-Normalform hat.

Besonders einfach ist die Antwort auf die eingangs gestellte Frage, wenn  $w = \varepsilon$  ist. Das leere Wort kann von einer Grammatik in Chomsky-Normalform genau dann produziert werden, wenn es die Regel  $S \rightarrow \varepsilon \in R$  gibt.

Wörter der Länge 1 dürfen in der Ableitung keine Regeln der Form  $A \rightarrow BC$  verwenden, da diese längere Wörter produzieren würden. Ist  $w$  also ein einzelnes Terminalsymbol, dann ist  $w$  genau dann ableitbar, wenn es eine Regel der Form  $S \rightarrow w \in R$  gibt.

Um ein längeres Wort abzuleiten, muss mindestens eine Regel der Form  $S \rightarrow AB$  angewendet werden. Durch weitere Anwendungen von Regeln entsteht anschließend aus  $A$  und  $B$  je ein Teilwort von  $w$ . Es gibt also eine Unterteilung  $w = w_1w_2$  mit  $A \xrightarrow{*} w_1$  und  $B \xrightarrow{*} w_2$ . Die Regelanwendung kann auch durch



visualisiert werden. Die Frage nach der Ableitbarkeit von  $w$  aus  $S$  wird damit reduziert auf die Frage der Ableitbarkeit der kleineren Wörter  $w_1$  und  $w_2$  aus  $A$  bzw.  $B$ . Die Frage, die der Algorithmus beantworten soll, wird damit etwas allgemeiner. Nicht nur soll er entscheiden können, ob  $S \xrightarrow{*} w$  gilt für die Startvariable, er soll allgemeiner  $A \xrightarrow{*} w$  für jede beliebige Variable  $A \in V$  entscheiden können.

```

boolean ableitbar(variable A, String w) {
    if (w.length() == 0) {
        return A → ε ∈ R;                                Fall: w = ε
    }
    if (w.length() == 1) {
        return A → w ∈ R;                               Fall: w ein Terminalsymbol
    }
    foreach Unterteilung w = w1w2 {                Fall: |w| > 1
        foreach A → BC ∈ R {
            if (ableitbar(B, w1) && ableitbar(C, w2))
                return true;
        }
    }
    return false;                                       sonst
}

```

Abbildung 6.1: Pseudocode des Cocke-Younger-Kasami-Algorithmus. Die Farbcodierung der Fälle wird auch in Abbildung 6.2 verwendet.

## 6.2.2 Pseudocode

Mit den Ideen aus Abschnitt 6.2.1 ist es jetzt möglich, einen rekursiven Algorithmus zu formulieren. Um zu entscheiden, ob das Wort  $w$  aus der Variablen  $A$  abgeleitet werden kann, geht man wie folgt vor:

1. Falls  $|w| = 0$  ist, ist  $w = \epsilon$  genau dann aus  $A$  ableitbar, wenn  $A = S$  ist und die Regel  $S \rightarrow \epsilon \in R$  ist.
2. Falls  $|w| = 1$  ist, ist  $w$  ein Terminalsymbol und  $w$  ist genau dann aus  $A$  ableitbar, wenn es die Regel  $A \rightarrow w \in R$  gibt.
3. Falls  $|w| > 1$  ist, müssen alle Unterteilungen  $w = w_1w_2$  mit  $|w_1| > 0$  und  $|w_2| > 0$  und alle Regeln der Form  $A \rightarrow BC$  durchprobiert werden und es muss rekursiv gefragt werden, ob  $B \xrightarrow{*} w_1$  und  $C \xrightarrow{*} w_2$ . Falls beide Ableitungen möglich sind, ist eine Ableitung des Wortes  $w$  gefunden.

In Abbildung 6.1 ist der Pseudocode dieses nach Cocke, Younger und Kasami benannten Algorithmus dargestellt.

Der Algorithmus löst das Problem, aber als rekursiver Algorithmus muss im schlimmsten Fall mit einer Laufzeit gerechnet werden, die exponentiell von der Länge des Wortes abhängt. In dieser Form ist der Algorithmus also noch nicht praktisch einsetzbar.

*Beispiel 6.1.* Als Beispiel betrachten wir die Ableitbarkeit des Wortes 000111 aus der

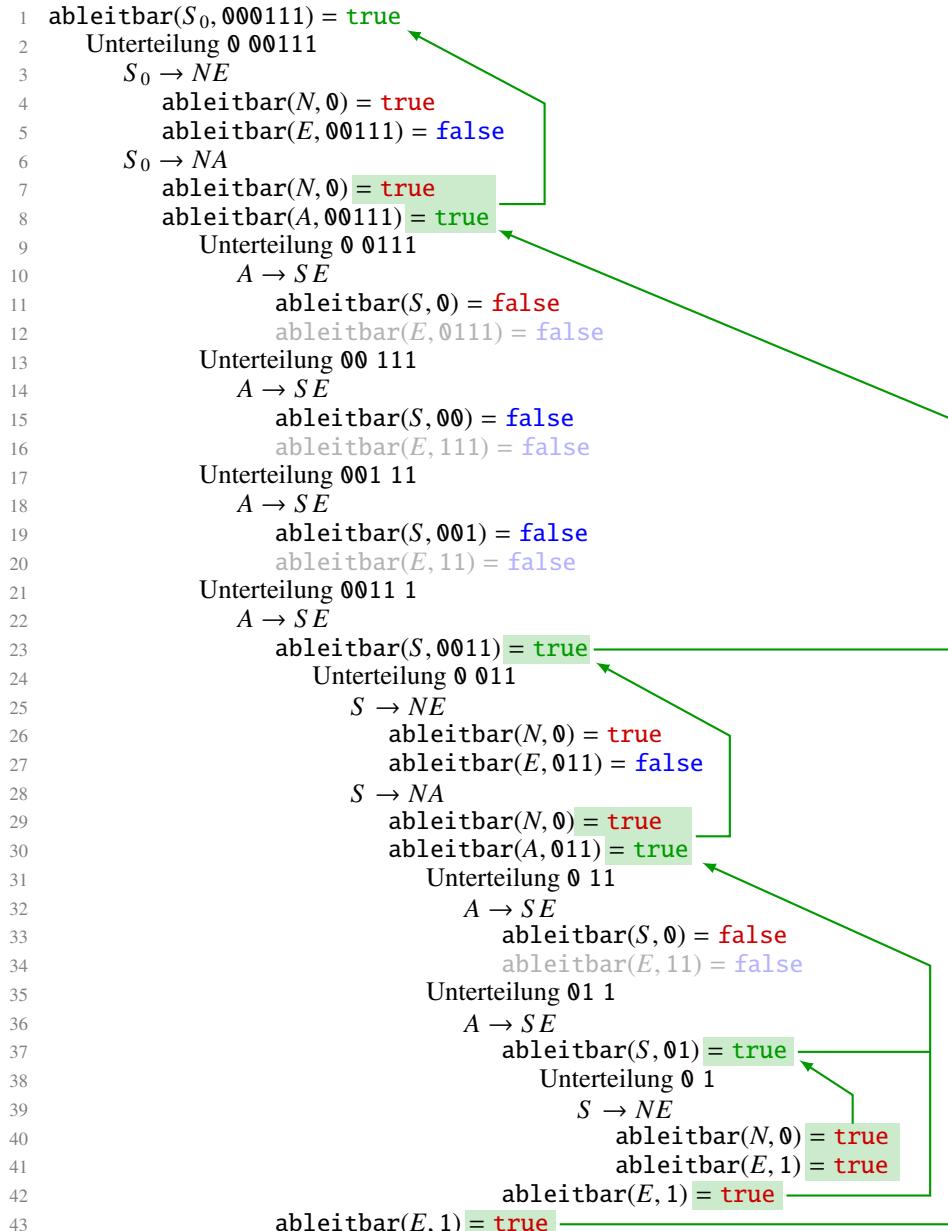


Abbildung 6.2: Durchführung des CYK-Algorithmus zur Bestimmung der Ableitbarkeit des Wortes  $\text{000111}$ . Die grünen Pfeile bezeichnen die Rückkehr von einem Aufruf der Funktion aufgrund zweier wahrer Rückgabewert nach rekursiven Aufrufen der Funktion `ableitbar()`. Rote Rückgabewerte entstehen aus Situationen, wo das Argument  $w$  Länge 1 hat, blaue Rückgabewerte folgen aus Aufrufen mit Variablen  $E$  oder  $N$ , die nur ein einzelnes Terminalsymbol erzeugen können.

## Grammatik

$$S \rightarrow \emptyset S 1 | \varepsilon \quad \text{mit Chomsky-Normalform} \quad \left\{ \begin{array}{l} S_0 \rightarrow NE | NA | \varepsilon \\ S \rightarrow NE | NA \\ A \rightarrow NA \\ N \rightarrow \emptyset \\ E \rightarrow 1 \end{array} \right. \quad (6.1)$$

der Sprache  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  (siehe auch Abbildung 6.2). Dazu wird die Funktion  $\text{ableitbar}(S_0, \emptyset\emptyset\emptyset 111)$  aufgerufen. Da das Wort  $\emptyset\emptyset\emptyset 111$  Länge  $6 > 1$  hat, werden alle Unterteilungen und alle Regeln mit  $S_0$  auf der linken Seite untersucht.

Als erste Unterteilung innerhalb des Aufrufs  $\text{ableitbar}(S_0, \emptyset\emptyset\emptyset 111)$  wird  $\emptyset \emptyset\emptyset 111$  untersucht (Zeile 2). Die Regel  $S_0 \rightarrow NE$  (Zeile 3) führt aber nicht zum Erfolg, weil der rekursive Aufruf  $\text{ableitbar}(E, \emptyset\emptyset 111)$  (Zeile 5) nicht zu einem wahren Resultat führt, da es von  $E$  ausgehend keine Regeln mit zwei Variablen auf der rechten Seite gibt.

Die nächste Regel  $S_0 \rightarrow NA$  (Zeile 6) wird schließlich erfolgreich sein, das wird aber erst nach vielen weiteren rekursiven Aufrufen klar. Im Aufruf  $\text{ableitbar}(A, \emptyset\emptyset 111)$  sind die Unterteilungen  $\emptyset \emptyset 111$  (Zeile 9),  $\emptyset \emptyset 1 11$  (Zeile 13) und  $\emptyset \emptyset 1 1 1$  (Zeile 17) jeweils bereits im ersten rekursiven Aufruf für das erste Teilwort nicht erfolgreich, so dass der zweite Aufruf nicht einmal mehr nötig ist. Er ist daher in Abbildung 6.2 jeweils grau dargestellt.

Erst die Unterteilung  $\emptyset 11 1$  auf Zeile 21 führt zum Erfolg. Zwar schlägt der Aufruf mit der Regel  $S \rightarrow NE$  (Zeile 25) wieder fehl, aber die Regel  $S \rightarrow NA$  (Zeile 28) führt mit der Aufteilung  $\emptyset 11$  auf den rekursiven Aufruf  $\text{ableitbar}(A, 11)$ . Dieser ist schließlich für die weitere Unterteilung  $1 1$  (Zeile 35) und die Regel  $A \rightarrow SE$  (Zeile 36) erfolgreich.

Die grünen Pfeile zeigen, wie die Wahrheitswerte `true` an die Aufrüfer zurückgegeben werden, was schließlich dazu führt, dass die Funktion  $\text{ableitbar}(S_0, \emptyset\emptyset\emptyset 111)$  ebenfalls `true` zurückgibt.  $\circ$

Das Beispiel zeigt auch, wie in den rekursiven Aufrufen der Funktion die gleichen Fragen nach Ableitbarkeit immer wieder gestellt werden. Die Aufrufe  $\text{ableitbar}(N, \emptyset)$  und  $\text{ableitbar}(E, 1)$  erfolgen gleich dreimal, auch  $\text{ableitbar}(S, \emptyset)$  wird zweimal erfolglos aufgerufen. Es ist daher zu vermuten, dass der Algorithmus nicht sehr effizient ist.

---

**Verständniskontrolle 6.1:** Die Grammatik (6.1) für Wörter der Form  $\emptyset^n 1^n$  hat Chomsky-Normalform. Überzeugen Sie sich mithilfe des Pseudocode von Abbildung 6.1, dass das Wort  $\emptyset 11$  nicht ableitbar ist.



### 6.3 Der CYK-Algorithmus in Tabellenform

Das Beispiel in Beispiel 6.1 hat gezeigt, dass die unbefriedigende, potenziell exponentielle Laufzeit des Cocke-Younger-Kasami-Algorithmus vor allem daher röhrt, dass viele

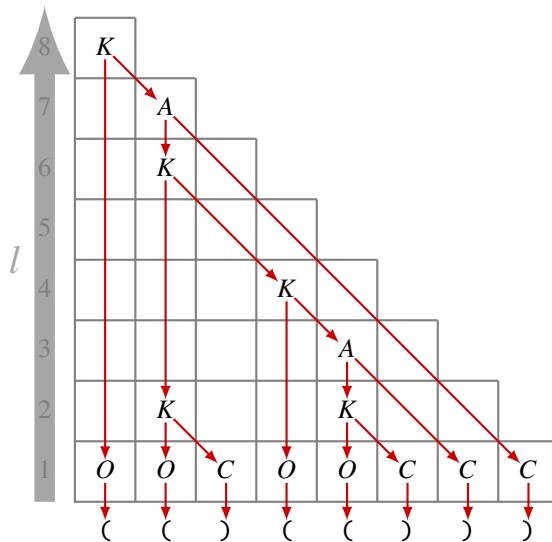


Abbildung 6.3: Tabelle für die Durchführung des CYK-Algorithmus mit einem Syntaxbaum für den Klammerausdruck  $((()))$  unter Verwendung der Chomsky-Normalform der Klammerausdrucksgrammatik in (6.2).

Anfragen immer wieder erneut gestellt werden. Eine wesentliche Verbesserung kann sich ergeben, wenn die Resultate der rekursiven Anfrage gespeichert werden können.

### 6.3.1 Ableitungstabelle

Der Syntaxbaum eines Wortes, das aus einer kontextfreien Grammatik in Chomsky-Normalform ableitbar ist, kann übersichtlich in einer quadratischen Tabelle dargestellt werden. Als Beispiel verwenden wir Klammerausdrücke mit der Grammatik

$$\left. \begin{array}{l} K \rightarrow KK \\ \quad \quad \quad \rightarrow (K) \\ \quad \quad \quad \rightarrow \varepsilon \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} S \rightarrow KK \mid OA \mid OC \mid \varepsilon \\ K \rightarrow KK \mid OA \mid OC \\ A \rightarrow KC \\ O \rightarrow ( \\ C \rightarrow ) \end{array} \right. \quad (6.2)$$

mit der Chomsky-Normalform auf der rechten Seite. Ein Syntaxbaum der Ableitung des Wortes  $((()))$  ist in Abbildung 6.3 eingetragen.

#### Aus einem Feld ableitbare Wörter

Aus einer Variablen in der Zeile  $l$  der Tabelle lässt sich ein Wort der Länge  $l$  ableiten, welches genau senkrecht unter der Variablen beginnt.

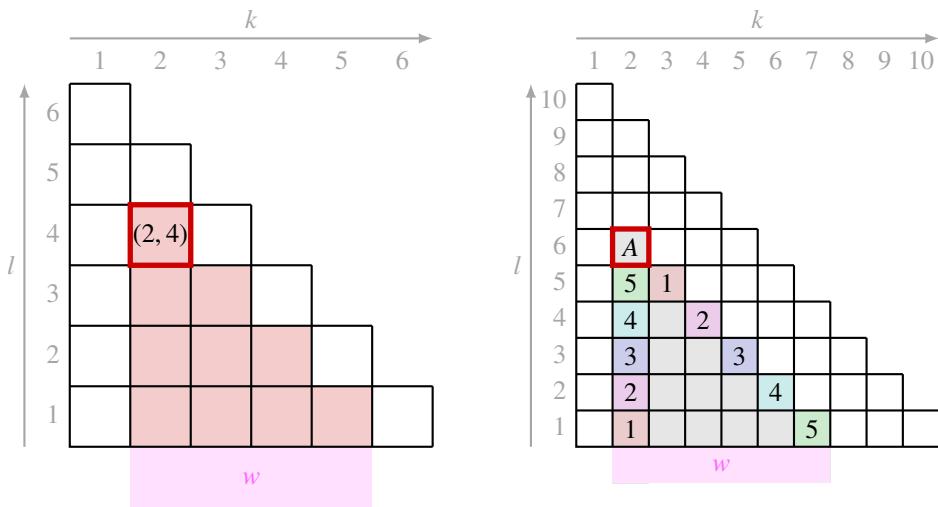


Abbildung 6.4: Links: Vom Feld  $(2, 4)$  der Tabelle aus kann das Wort der Länge 4 erzeugt werden, welches in der Position vertikal unter dem Feld beginnt (links). Das Feld  $(2, 4)$  definiert das rote Ableitungsdreieck, welches das Wort  $w$  erzeugt. Rechts: Um mit einer Regel der Form  $A \rightarrow BC$  ausgehend von der Variablen  $A$  im Feld  $(2, 6)$  das Wort  $w$  zu erzeugen, müssen die Variablen  $B$  und  $C$  in korrespondierenden Feldern (Definition 6.3) mit der gleichen Nummer eingetragen sein (rechts).

**Definition 6.2.** Ein Ableitungsdreieck ist eine gestufte Tabelle, die alle Felder umfasst, die die Erzeugung eines Wortes beeinflussen können. Das vom Ableitungsdreieck erzeugte Wort ist das Wort an der Basis des Dreiecks.

Die Tabelle in Abbildung 6.3 ist selbst ein Ableitungsdreieck für das Wort  $((\cdot))(\cdot)$ . Das zum Feld  $(2, 4)$  gehörige Ableitungsdreieck in Abbildung 6.4 links ist rot hervorgehoben und erzeugt das Wort  $w$ . Ein Ableitungsdreieck enthält von jedem Feld  $(k, l)$  ausgehend ein Ableitungsdreieck, welches das Teilwort der Länge  $l$  beginnend beim  $k$ -ten Zeichen des Wortes an der Basis erzeugt.

Das Ziel der folgenden Abschnitte ist, einen Algorithmus zu formulieren, wie die Felder der Ableitungsdreiecks für das Wort  $w$  mit Variablensymbolen gefüllt werden müssen, damit man die Ableitbarkeit und den Syntaxbaum aus der Tabelle ablesen kann.

## Regeln

Einzelne Terminalsymbole des abzuleitenden Wortes werden von Variablen in der untersten Zeile erzeugt. Dazu werden die Regeln der Form  $A \rightarrow a$  der Grammatik verwendet, die Variablen in Terminalsymbole umsetzen. In der untersten Zeile der Tabelle müssen also Variablen eingetragen werden, die die darunter stehenden Terminalsymbole erzeugen können.

Zur Erzeugung längerer Wörter sind Regeln der Form  $A \rightarrow BC$  nötig. Um das Teilwort der Länge  $l$  beginnend beim Zeichen  $k$  von  $w$  zu erzeugen, muss die Variable  $A$  im Feld

$(k, l)$  eingetragen werden. Sie steht also an der Spitze des vom Feld  $(k, l)$  ausgehenden Ableitungsdreiecks. Die Variable  $B$  erzeugt ein Teilwort  $w_1$  von  $w$ , welches ebenfalls an der Stelle  $k$  beginnt und daher in Spalte  $k$  unter der Variablen in Feld  $(k, l_1)$  eingetragen werden muss. Damit die Variable  $C$  den Rest des Wortes erzeugt, muss sie in Feld  $(k + l_1, l - l_1)$  eingetragen werden.

**Definition 6.3.** Die Felder  $(k, l_1)$  und  $(k + l_1, l - l_1)$  heißen korrespondierende Felder im Ableitungsdreieck des Feldes  $(k, l)$ .

**Beispiel 6.4.** Als Beispiel betrachten wir den Spezialfall  $l = 2$ : die einzigen korrespondierenden Felder sind in diesem Fall  $(k, 1)$  und  $(k + 1, l - 1) = (k + 1, 1)$ . Die korrespondierenden Felder unter  $(k, 2)$  sind also die beiden benachbarten Felder  $(k, 1)$  und  $(k + 1, 1)$  der untersten Zeile.  $\circlearrowright$

Wenn für die Regel  $A \rightarrow BC$  die Variable  $A$  in das Feld  $(k, l)$  eingetragen wird, dann müssen die Variablen  $B$  und  $C$  in korrespondierende Felder des von  $(k, l)$  ausgehenden Ableitungsdreiecks eingetragen werden. Die korrespondierenden Paare von Feldern für die Variablen  $B$  und  $C$  sind in Abbildung 6.4 rechts durch gleiche Farbe und gleiche Nummer gekennzeichnet.

### 6.3.2 Rekursion wird zu Iteration

Der Algorithmus muss herausfinden, ob es Regeln gibt, mit denen man in der Tabelle einen Syntaxbaum aufbauen kann, der im obersten Feld mit der Startvariable  $S$  beginnt.

#### Iteration über Unterteilungen und Regeln

Der Cocke-Younger-Kasami-Algorithmus iteriert über alle Unterteilungen des Wortes und über alle Regeln der Form  $S \rightarrow AB$ . Dies ist gleichbedeutend damit zu fragen, ob sich die Variablen  $A$  und  $B$  in korrespondierende Felder eines Paares wie in Abbildung 6.4 rechts eintragen lassen, so dass sich die zugehörigen Teilwörter aus  $A$  und  $B$  ableiten lassen. Die Unterdreiecke ausgehend von  $A$  und  $B$  müssen also für sich wieder korrekte Ableitungen enthalten. Dies ist das in Abschnitt 6.2 diskutierte rekursive Vorgehen.

Statt alle Regeln und Platzierungen ausgehend vom obersten Feld durchzuprobieren, kann man auch umgekehrt vorgehen und beginnend in der untersten Zeile alle Variablen eintragen, mit denen das zugehörige Wort erzeugt werden kann. In der untersten Zeile müssen also alle Variablen eingetragen werden, für die es eine Terminalsymbolregel gibt, die aus der Variablen das Zeichen an der entsprechenden Stelle des Wortes erzeugt. Es ist also durchaus möglich, dass in ein Feld mehrere Variablen eingetragen werden müssen.

In die Felder ab der zweiten Zeile von unten werden alle Variablen  $A$  eingetragen, für die es eine Regel  $A \rightarrow BC$  gibt, wobei  $B$  und  $C$  bereits in korrespondierenden Feldern wie in Abbildung 6.4 eingetragen sind. Auch hier ist es möglich, dass mehrere Variablen eingetragen werden müssen.

Jede Variable in einem Feld bedeutet, dass das zugehörige Wort aus der Variablen abgeleitet werden kann. Die Frage, ob das ganze Wort aus der Startvariablen abgeleitet werden kann, ist daher gleichbedeutend mit der Frage, ob im obersten Feld die Startvariable eingetragen werden kann.

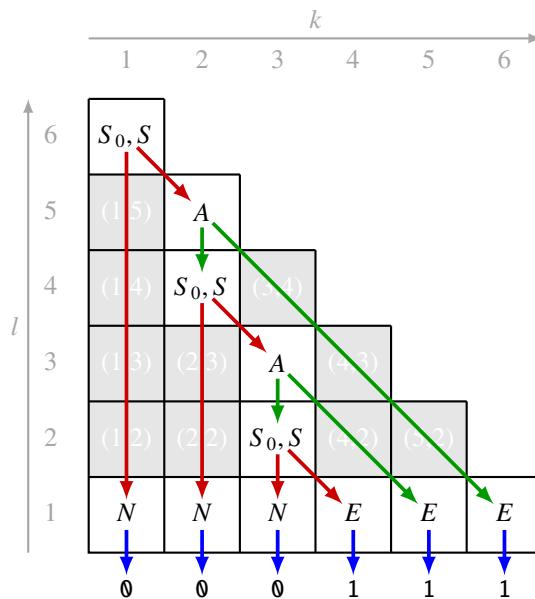


Abbildung 6.5: Durchführung des CYK-Algorithmus mithilfe der Tabelle für die Sprache  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  und das Wort  $w = \emptyset\emptyset\emptyset 111$ .

## Der Algorithmus

Der Cocke-Younger-Kasami-Algorithmus, der entscheiden soll, ob aus der Variablen  $S$  das Wort  $w = a_1 a_2 \dots a_n$  ableitbar ist, läuft jetzt wie folgt ab.

1. In das Feld  $k$  der untersten Zeile der Tabelle werden alle Variablen  $A$  eingetragen, für die es eine Regel  $A \rightarrow a_k$  in der Regelmenge der Grammatik gibt.
2. In den Feldern der höheren Zeilen werden von unten nach oben jeweils alle Variablen  $A$  eingetragen, für die es eine Regel  $A \rightarrow BC$  so gibt, so dass in zwei korrespondierenden Feldern die Variablen  $B$  und  $C$  bereits eingetragen sind.
3. Wenn im obersten Feld die Variable  $S$  eingetragen werden kann, dann ist  $w$  aus  $S$  ableitbar, sonst nicht.

## Beispiel: Die Sprache $L = \{\emptyset^n 1^n \mid n \geq 0\}$

Die Sprache  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  hat die Grammatik

$$\left. \begin{array}{l} S \rightarrow \emptyset S 1 \\ \rightarrow \varepsilon \end{array} \right\} \quad \text{mit Chomsky-Normalform}$$

$$\left\{ \begin{array}{l} S_0 \rightarrow NE \mid NA \mid \varepsilon \\ S \rightarrow NE \mid NA \\ A \rightarrow SE \\ N \rightarrow \emptyset \\ E \rightarrow 1. \end{array} \right.$$

Es soll entschieden werden, ob das Wort  $000111$  aus  $S_0$  abgeleitet werden kann. Das Resultat der Durchführung des Algorithmus ist in Abbildung 6.5 dargestellt.

Im ersten Schritt des Algorithmus werden die Felder der untersten Zeile mit den Variablen  $N$  und  $E$  gefüllt, denn nur für diese Variablen gibt es die blauen Regeln  $N \rightarrow \emptyset$  und  $E \rightarrow 1$ , die die Terminalssymbole erzeugen.

Im zweiten Schritt werden die Felder  $(1, 2)$ ,  $(2, 2)$  usw. mit Variablen gefüllt, die als Ausgangspunkt für Regeln mit zwei Variablen auf der rechten Seite stehen. Das Feld  $(1, 2)$  kann nur eine Variable  $X$  enthalten, für die es eine Regel  $X \rightarrow NN$  gibt, denn in beiden korrespondierenden Feldern  $(1, 1)$  und  $(2, 1)$  stehen die Variablen  $N$ . Eine solche Regel gibt es aber in der Grammatik nicht, daher bleibt das Feld leer. Dasselbe gilt für das Feld  $(2, 2)$ .

Das Feld  $(3, 2)$  muss diejenigen Variablen  $X$  enthalten, für die es eine Regel  $X \rightarrow NE$  gibt, da  $N$  und  $E$  die Variablen in den korrespondierenden Feldern  $(3, 1)$  und  $(4, 1)$  sind. Es gibt zwei solche Regeln, nämlich  $S_0 \rightarrow NE$  und  $S \rightarrow NE$ , so dass in das Feld die Variablen  $S_0$  und  $S$  eingetragen werden müssen.

Das Feld  $(2, 3)$  muss diejenigen Variablen  $X$  enthalten, für die es eine Regel  $X \rightarrow NS_0$  oder  $X \rightarrow NS$  gibt. Solche Regeln gibt es nicht, so dass das Feld leer bleiben muss.

Im Feld  $(3, 3)$  wird  $A$  eingetragen, weil die korrespondierenden Felder  $(3, 2)$  und  $(5, 1)$  die Variablen  $S$  bzw.  $E$  enthalten und es eine Regel  $A \rightarrow SE$  gibt. Wir müssen aber auch noch das andere Paar von korrespondierenden Feldern  $(3, 1)$  und  $(4, 2)$  untersuchen. Da das Feld  $(4, 2)$  leer ist, liefert dieses Paar von korrespondierenden Feldern keine neuen Variableneinträge für das Feld  $(3, 3)$ .

Die Felder  $(2, 4)$  und  $(3, 2)$  enthalten sowohl die Variable  $S_0$  wie auch die Variable  $S$ . Nur  $S$  wird aber in der Regel  $A \rightarrow SE$  gebraucht. Die Variablen  $S_0$  in diesen Feldern werden also nie gebraucht. Die von diesen Feldern erzeugten Teilwörter  $01$  und  $0011$  könnten zwar aus  $S_0$  abgeleitet werden, aber  $S_0$  kann selbst nicht auf der rechten Seite einer Regel vorkommen.

Dieses Vorgehen wird fortgesetzt, bis auch das oberste Feld ausgefüllt ist. Da es möglich war, im obersten Feld  $(1, 6)$  die Startvariable  $S_0$  einzutragen, lässt sich das Wort aus der Startvariablen ableiten.

---

*Verständniskontrolle 6.2:* Verwenden Sie die Tabellendurchführung des CYK-Algorithmus, um sich zu überzeugen, dass das Wort  $00111$  von der Grammatik (6.1) *nicht* produziert werden kann.



### 6.3.3 Laufzeit

Wir untersuchen die Laufzeit des Algorithmus für ein Wort der Länge  $n = |w|$ . Die eben beschriebene Implementation des Cocke-Younger-Kasami-Algorithmus bearbeitet einmal von unten nach oben alle Felder der Tabelle, von denen es  $O(n^2)$  gibt. In jedem Feld müssen alle vorhandenen Regeln überprüft werden, der Aufwand dafür ist immer gleich groß. Zu jedem Ausgangsfeld gibt es  $O(n)$  mögliche korrespondierende Felder, die als Zielfel-

der für die Regeln der Form  $A \rightarrow BC$  in Frage kommen. Die Laufzeit für die Bearbeitung jedes Feldes ist daher von der Größenordnung  $O(n)$ . Die Gesamlaufzeit des Algorithmus wird daher  $O(n^2) \cdot O(n) = O(n^3)$ .

Für praktische Anwendungen ist selbst diese wesentlich verbesserte Laufzeit von  $O(n^3)$  unzureichend. Beim Einsatz in einem Compiler würde dies zum Beispiel bedeuten, dass mit jeder Verdoppelung der Anzahl Zeilen in einem Sourcefile die Laufzeit für den Compiler achtmal größer wird.

## Übungsaufgaben

**6.1.** LDAP-Filter werden verwendet, um aus einem LDAP-Verzeichnis eine Menge von Knoten aufgrund von Werten einzelner Attribute auszuwählen. Elementare LDAP-Suchfilter haben die Form

`attribut=wert`

Sie selektieren genau die Knoten, deren Attribut `attribut` den Wert `wert` hat. Für die Zwecke dieser Aufgabe sind Attribute und Werte nichtleere Strings aus Buchstaben und Ziffern. Daraus lassen sich mithilfe von Klammern und logischen Verknüpfungen beliebige LDAP-Filter aufbauen. Die logischen Verknüpfungen verwenden eine Präfix-Notation:

UND:	<code>(&amp;(attr1=wert1)(attr2=wert2))</code>
ODER:	<code>( (attr1=wert1)(attr2=wert2))</code>
NICHT:	<code>(!(attr=wert))</code>

Bei der UND- und der ODER-Verknüpfung dürfen beliebig viele Filter miteinander verknüpft werden.

- Gibt es einen regulären Ausdruck, der LDAP-Filter akzeptiert?
- Bilden die syntaktisch korrekten LDAP-Filter eine kontextfreie Sprache?

**6.2.** Die Sprache Lisp hat eine etwas eigene Syntax, Larry Wall hat sie einst mit “Haferbrei vermischt mit abgeschnittenen Fingernägeln” verglichen [61]. In Lisp gibt es Atome:

- numerische Atome, d. h. Zahlen: 1 2 3 -4 3.14 -7.5 6.02E+23
- Strings, also Zeichenfolgen in Anführungszeichen:

`"ein String" "&]"$787?" "setq" "12"`

- Symbole, alle anderen Zeichenketten, die von Zahlen und Strings unterscheidbar sind und kein Klammern und Anführungszeichen enthalten.

Aus den Atomen können jetzt Listen kombiniert werden. Eine Liste besteht aus einer öffnenden Klammer, einer Folge von Listenelementen und einer schließenden Klammer. Listenelemente sind entweder Atome oder Listen. Listen können auch leer sein, ebenso Strings. Ein typisches Lisp-Programm ist

```
(defun print-quadratzahlen (x)
  (cond ((plusp x)
          (print (* x x))
          (print-quadratzahlen (- x 1)))))
```

- a) Formulieren Sie eine kontextfreie Grammatik für Lisp.
- b) Falls Ihre Grammatik nicht Chomsky-Normalform hat: markieren Sie alle Regeln, die der Normalform widersprechen.

*Hinweis.* Zur Vereinfachung gehen Sie davon aus, dass Zahlen positive natürliche Zahlen sind, dass Strings nur Kleinbuchstaben und Ziffern enthalten, und dass ein Symbol eine Folge von Kleinbuchstaben (keine Ziffern) ist.

**6.3.** Verwenden Sie die Expression-Term-Factor-Grammatik von Abschnitt 5.3.1 und den Cocke-Younger-Kasami-Algorithmus, um den Syntaxbaum des Ausdrucks  $7^*(5+3)$  zu ermitteln.

Lösungen: <https://autospr.ch/uebungen/AutoSpr-106.pdf>



# Kapitel 7

## Stackautomaten

Die Geschwindigkeit des Cocke-Younger-Kasami-Algorithmus ist für praktische Zwecke ungenügend. In diesem Kapitel wird daher ein alternativer Ansatz ausprobiert. Ein deterministischer endlicher Automat wird mit einem Stackspeicher zu einem Stackautomaten erweitert. Dadurch wird er in die Lage versetzt, kontextfreie Sprachen zu parsen. Allerdings sind diese Automaten nicht deterministisch. Für den Einsatz in praktisch nützlicher Software werden zusätzliche Voraussetzungen an die Sprache nötig, die einen deterministisch arbeitenden, stackbasierten Algorithmus ermöglichen. Solche LR-Parser werden von Parser-Generatoren verwendet, um aus einer Grammatik funktionierende Software zu erzeugen. Nicht nur gibt es zu jeder Grammatik einen Stackautomaten, auch die Umkehrung gilt: Jeder Stackautomat akzeptiert eine kontextfreie Sprache, es lässt sich also eine Grammatik aus dem Stackautomaten ableiten.

### 7.1 Stackspeicher

In diesem Abschnitt wird zunächst erklärt, was ein Stack ist. In Abschnitt 7.1.2 wird dann gezeigt, wie man einen endlichen Automaten um einen Stack erweitert. Solche Stackautomaten können Sprachen akzeptieren, die nicht regulär sind. Sie sind aber meistens nicht deterministisch.

#### 7.1.1 Stackoperationen

Ein Stack, auch Kellerspeicher genannt, ist ein Speicher, der beliebig viele Elemente eines Alphabets  $\Gamma$ , des Stackalphabets, speichern kann. Er gestattet jedoch immer nur Zugriff auf das zuletzt gespeicherte Element, wie in Abbildung 7.1 dargestellt.

Auch die Notation, die wir brauchen, um Stackoperationen auszudrücken, ist bereits in der Abbildung 7.1 angedeutet. Wenn das Zeichen F auf dem Stack gespeichert werden

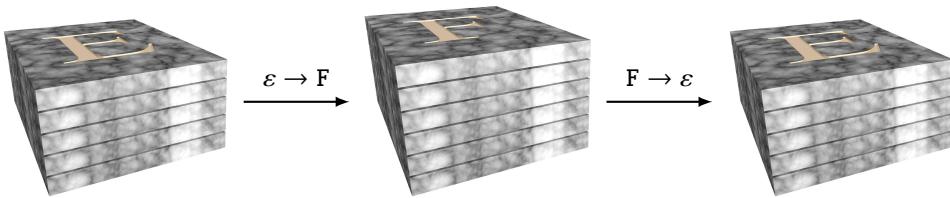


Abbildung 7.1: Der Stack ist ein Speicher, der immer nur Zugriff auf das zuletzt gespeicherte Element gestattet.

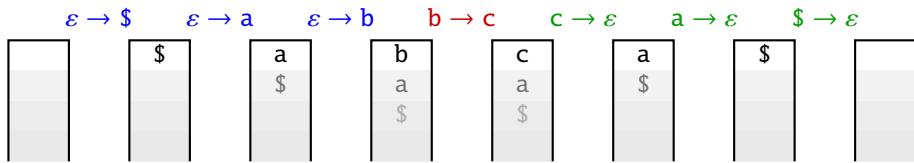


Abbildung 7.2: Stackoperationen: Zunächst werden die Zeichen \$, a und b auf den Stack gelegt (blaue Operationen). Dann wird das Zeichen b oben auf dem Stack durch c ersetzt (rot). Schließlich wird der Stack durch Entfernung der Zeichen in umgekehrter Reihenfolge wieder geleert (grüne Operationen). Dass der Stack leer ist, wird mit dem speziellen Zeichen \$ erkannt.

soll, schreiben wir das als Stack-Übergang  $\varepsilon \rightarrow F$ . Das  $\varepsilon$  ist zu lesen als: “es ist egal, was vor dieser Operation zuoberst auf dem Stack liegt”.

Der Übergang  $F \rightarrow \varepsilon$  beschreibt das Entfernen des Zeichens F vom Stack. Hier ist das  $\varepsilon$  zu lesen als: “Nach dem Entfernen des Zeichens belassen wir den Stack so, wie er ist”. Man kann das F auch als Bedingung lesen. Der Stackübergang ist nur möglich, wenn tatsächlich ein F zuoberst auf dem Stack liegt.

Es ist auch die kombinierte Operation möglich. Die Schreibweise  $b \rightarrow c$  bedeutet dann, dass dieser Stackübergang möglich ist, wenn ein b zuoberst auf dem Stack liegt. Dieses b wird dann entfernt und dafür das c auf den Stack gelegt. Alle Operationen sind auch in Abbildung 7.2 illustriert.

**Definition 7.1** (Stackoperation). Eine Stackoperation über dem Alphabet  $\Gamma$ , die das Zeichen v vom Stack entfernt und dafür das Zeichen n auf den Stack legt, ist ein Paar  $(v, n)$  welches auch

$$v \rightarrow n, \quad \text{mit } v, n \in \Gamma_\varepsilon$$

geschrieben wird.

Der Stack hat keinen “eingebauten” Mechanismus, mit dem man feststellen kann, ob der Stack leer ist. Bei einem Stapel Zettel auf dem Schreibtisch erkennt man auch nicht am Stapel selbst, dass er leer ist, sondern daran, dass man nur die Tischplatte sieht. Die Tischplatte ist sozusagen das Element, mit dem man den Stapel begonnen hat. Aus diesem Bild können wir einen eigenen Mechanismus konstruieren, um erkennen zu können, wann der Stack leer ist. Bevor wir Zeichen auf den Stack legen, legen wir ein spezielles Erkennungszeichen für das Stackende auf den Stack. Dieses Zeichen übernimmt die Rolle der

Tischplatte: Sobald wir zuoberst auf dem Stack dieses Zeichen sehen, wissen wir, dass der Stack geleert worden ist.

In den nachfolgenden Beispielen verwenden wir jeweils, wenn nichts anderes vermerkt ist, das Zeichen  $\$$  zur Markierung des Stackanfangs<sup>1</sup>. Mit der Stackoperation  $\varepsilon \rightarrow \$$  wird das Zeichen  $\$$  auf den Stack gelegt, mit der Operation  $\$ \rightarrow \varepsilon$ , die nur möglich ist, wenn tatsächlich das  $\$$  zuoberst auf dem Stack liegt, wird es wieder entfernt und so erkannt, dass der Stack leer geworden ist.

## 7.1.2 Stackautomaten

Mit den im vorangegangenen Abschnitt besprochenen Stackoperationen lässt sich ein Stackautomat jetzt durch Erweiterung der Definition eines endlichen Automaten um Stackoperationen im Sinne der Definition 7.1 konstruieren. Dies hat vor allem Einfluss auf die Definition der Übergangsfunktion, die jetzt auch die Stackoperation beinhalten muss.

**Definition 7.2** (Stackautomat). *Ein Stackautomat oder Pushdown-Automat (PDA) ist ein 6-Tupel*

$$P = (Q, \Sigma, \Gamma, \delta, q_0, F).$$

*Die Elemente der endlichen Menge  $Q$  heißen Zustände,  $F \subset Q$  sind die Akzeptierzustände. Das Alphabet  $\Sigma$  enthält die Zeichen, die in Inputwörtern vorkommen können, das Stackalphabet  $\Gamma$  enthält die Zeichen, die auf dem Stack gespeichert werden können.*

$$\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon) : (q, a, s) \mapsto \delta(q, a, s) \ni (q', s')$$

*ist die Übergangsfunktion und  $q_0$  der Startzustand. Die Stackoperation  $s \rightarrow s'$  ist mit dem Zustandsübergang von  $q$  nach  $q'$  verbunden, der das Zeichen  $a$  verarbeitet.*

Man beachte, dass die Definition eines Stackautomaten grundsätzlich nicht deterministisch ist, es wird gar nicht versucht, deterministische Stackautomaten systematisch zu definieren und zu untersuchen. Es zeigt sich nämlich, dass dies nicht zu einer nützlichen Theorie führt. Dies hat natürlich zur Konsequenz, dass ein Stackautomat ohne zusätzliche, vereinfachende Anforderungen an die Sprache keine gute Grundlage für die Konstruktion deterministischer Software sein kann. Wie solche zusätzlichen Eigenschaften aussehen könnten, wird in Abschnitt 7.3 diskutiert.

In einem Zustandsdiagramm beschriften wir die Übergänge zusätzlich zum verarbeiteten Zeichen (welches auch  $\varepsilon$  sein kann) mit dem damit einhergehenden Stackübergang. In Abbildung 7.3 sind viele der möglichen Kombinationen dargestellt.

Die Verarbeitung von Wörtern durch einen Stackautomaten ist ganz ähnlich der Vorgehensweise eines endlichen Automaten. Bei jedem Übergang müssen zusätzlich die Stackoperationen ausgeführt werden. Beachtet man in einem Zustandsdiagramm nur die Zeichen vor dem Komma, bleibt ein nichtdeterministischer endlicher Automat übrig. Ein Wort  $w \in \Sigma^*$  definiert im Allgemeinen einen Pfad durch diesen endlichen Automaten. Das Wort definiert einen Pfad durch den Stackautomaten, wenn der Pfad durch den endlichen Automaten auch mit den Stackoperationen kompatibel ist.

<sup>1</sup>Die Wahl des Zeichens  $\$$  scheint historisch gleich begründet worden zu sein wie die Wahl dieses Zeichens als Delimiter für mathematische Formeln im TeX-Satzsystem von Donald Knuth. Da Mathematiker ohnehin kein Interesse an der Realität und damit an Geld hätten, haben sie auch keine andere Verwendung für das Zeichen  $\$$ .

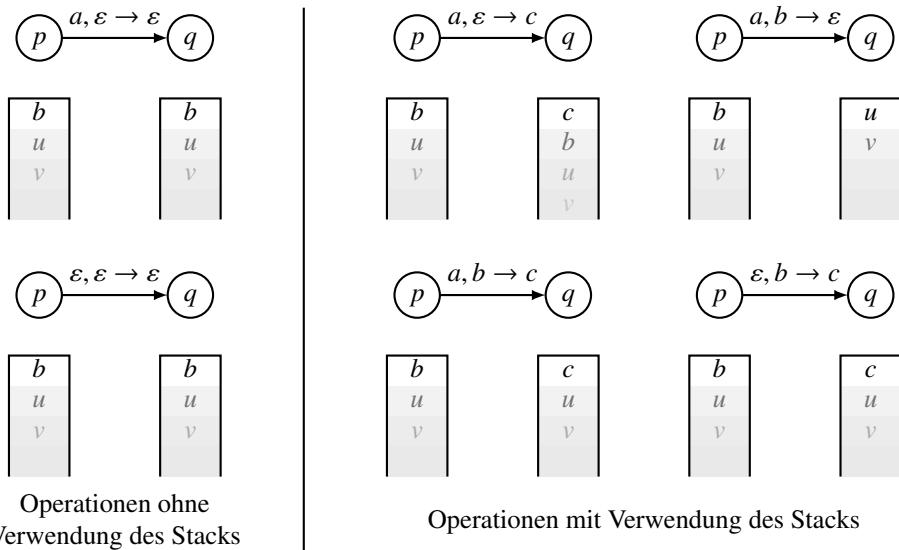


Abbildung 7.3: Zustandsübergänge in einem Stackautomaten. Die Schreibweise  $a, b \rightarrow c$  bedeutet, dass der Übergang von  $p$  nach  $q$  zur Verarbeitung des Zeichens  $a$  möglich ist, wenn sich zuoberst auf dem Stack das Zeichen  $b$  befindet, welches dann durch  $c$  ersetzt wird.

**Definition 7.3** (Wort akzeptieren und akzeptierte Sprache). Ein Stackautomat  $P$  akzeptiert das Wort  $w$ , wenn es zu dem Wort einen Pfad aus Stackautomatenübergängen vom Startzustand  $q_0$  zu einem Akzeptierzustand gibt. Die Menge

$$L(P) = \{w \in \Sigma^* \mid P \text{ akzeptiert } w.\}$$

heißt die von  $P$  akzeptierte Sprache.

Die Beispiele der nachfolgenden Abschnitte können helfen, die Definitionen und Notationen verständlicher zu machen.

### 7.1.3 Die Sprachen $\emptyset^n 1^n$ und $\{w \mid |w|_0 = |w|_1\}$

Die Sprachen

$$L_1 = \{\emptyset^n 1^n \mid n \geq 0\} \quad \text{und} \quad L_2 = \{w \in \{\emptyset, 1\}^* \mid |w|_0 = |w|_1\}$$

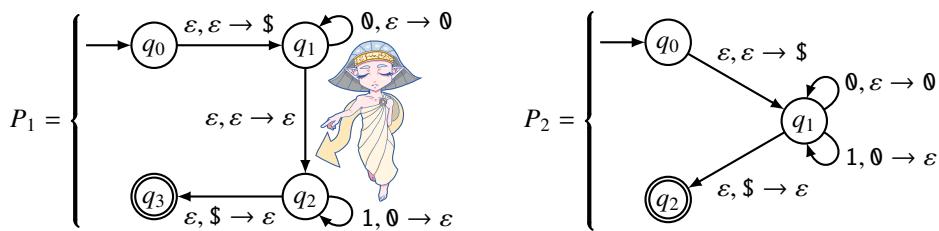
wurden schon früher als nicht regulär erkannt. Es ist also nicht möglich, sie mit einem endlichen Automaten zu akzeptieren. Der Grund dafür lag in der Tatsache, dass eine geeignete Maschine in der Lage sein muss, die Anzahl der  $\emptyset$  zu zählen und mit der Anzahl der 1 zu vergleichen. Genau dies ist mit den endlich vielen Zuständen eines endlichen Automaten nicht möglich.

Ein Stackautomat kann zwar auch nicht zählen, aber er kann für jede  $\emptyset$  ein Zählzeichen auf dem Stack zwischenspeichern und bei der Verarbeitung einer 1 wieder entfernen. Ist

der Stack am Ende des Wortes leer, war die Zahl der Nullen und Einsen gleich. Dies macht plausibel, dass  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  von einem Stackautomaten akzeptiert werden kann.

Der Stack implementiert also nicht einen Zähler, vielmehr dient er als Zwischenspeicher für Zeichen, die bei der späteren Verarbeitung wieder entfernt werden können. Um zu erkennen, ob der Stack bereits geleert wurde, verwenden wir wieder das Zeichen \$, das zu Beginn auf den Stack gelegt wird und vor dem Akzeptieren des Wortes wieder entfernt werden muss. Für die Sprache  $L_2$  spielt die Reihenfolge der Zeichen keine Rolle, es kommt nur darauf an, dass es zu jeder 0 auch eine 1 gibt.

Die beiden Stackautomaten



sind Versuche, die beiden Sprachen mit einem Stackautomaten zu akzeptieren. Tatsächlich ist  $L_1 = L(P_1)$ .

Der zweite Automat funktioniert dagegen nicht. Der Übergang  $1, 0 \rightarrow \varepsilon$  bedeutet, dass eine 1 nur dann verarbeitet werden kann, wenn es bereits mindestens eine 0 auf dem Stack hat. Zum Beispiel ist das Wort  $10 \in L_2$ , aber da 1 vor einer 0 kommt, kann es nicht akzeptiert werden. Es ist also  $L_2 \neq L(P_2)$ .

Eine korrekte Lösung für einen Stackautomaten, der die Sprache  $L_2$  akzeptiert, wird in Verständniskontrolle 7.1 gesucht.

## 7.1.4 Palindrome

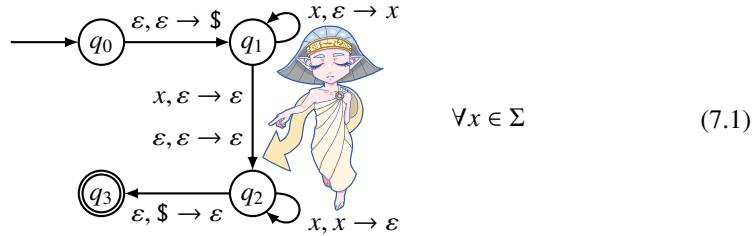
Der Stack kann nicht nur dazu verwendet werden, Zeichen im Input zu zählen. Er kann auch verwendet werden, um Zeichen miteinander zu vergleichen, wie in der Sprache

$$L = \{w \in \Sigma^* \mid w \text{ ist ein Palindrom.}\}.$$

Ein Palindrom gerader Länge enthält in der zweiten Hälfte die gleichen Zeichen wie in der ersten, aber in umgekehrter Reihenfolge. Ein Palindrom ungerader Länge enthält noch ein weiteres nicht gepaartes Zeichen in der Mitte.

Um ein Palindrom zu akzeptieren, muss man sich die gepaarten Zeichen merken, zum Beispiel indem man sie auf dem Stack ablegt. Das mittlere Zeichen muss nicht auf den Stack, es kann einfach übersprungen werden. Bei der Verarbeitung der zweiten Hälfte des Wortes müssen alle Zeichen im Input mit den auf dem Stack gespeicherten Zeichen übereinstimmen.

Der Stackautomat



implementiert diese Idee und akzeptiert daher genau die Sprache  $L$  der Palindrome. Die Zeichen  $x$  in den Übergängen sind dabei so zu lesen, dass dieser Übergang für jedes Zeichen  $x \in \Sigma$  des Inputalphabets möglich ist.

Das Wort

$$w = \underbrace{0101\dots010}_{n} \underbrace{1010\dots10}_{n}$$

ist ein Palindrom der Länge  $2n + 1$ . Das Zeichen  $\emptyset$  in der Mitte muss als das  $x$  für den Übergang von  $q_1$  nach  $q_2$  verarbeitet werden. Es gibt aber keine Möglichkeit dieses  $\emptyset$  von irgendeinem anderen  $\emptyset$  im Wort  $w$  zu unterscheiden. Auch einem deterministischen Stackautomaten fehlt die Information dazu. Dies zeigt, dass man nicht erwarten darf, beliebige kontextfreie Sprachen mit deterministischen Stackautomaten zu akzeptieren.

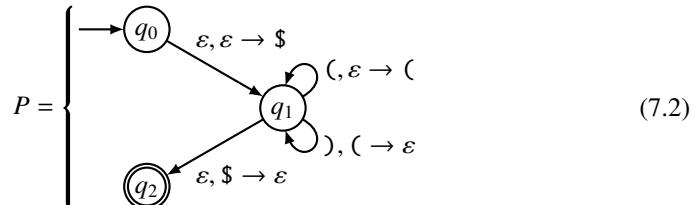
### 7.1.5 Klammerausdrücke

Klammerausdrücke haben in Abschnitt 5.1.1 als Beispiel dafür gedient, dass eine größere Sprachfamilie nötig ist, die wir dann mit einer Grammatik beschrieben haben. Wir hoffen, dass Stackautomaten das Werkzeug sein könnten, die kontextfreien Sprachen mit einer Maschine zu akzeptieren. Wir versuchen dies für die Sprache

$$L = \{w \in \{\(\), \)\}^* \mid w \text{ ist ein korrekter Klammerausdruck.}\}$$

der korrekten Klammerausdrücke, für die wir die Grammatik bereits aus Abschnitt 5.1.2 kennen.

Zu jeder öffnenden Klammer in einem Klammerausdruck gibt es eine zugehörige schließende Klammer. Die öffnende Klammer muss immer vor der schließenden Klammer kommen. Dies ist genau das Verhalten, welches den Automaten  $P_2$  in Abschnitt 7.1.3 ungeeignet gemacht hat für die Sprache  $L_2 = \{w \mid |w|_\emptyset = |w|_1\}$ . Für die Klammerausdrücke erhalten wir daher den Automaten



der die Sprache  $L = L(P)$  der Klammerausdrücke akzeptiert.

*Verständniskontrolle 7.1:* Finden Sie einen Stackautomaten, der die Sprache

$$L = \{w \in \{\emptyset, 1\}^* \mid |w|_0 = |w|_1\}$$



autospr.ch/v/7.1.pdf

akzeptiert. In Abschnitt 7.1.3 wurde ein Stackautomat diskutiert, der nicht funktioniert hat. Der Grund dafür war, dass der Stackautomat verlangt hat, dass an jedem Punkt in der Verarbeitung die Anzahl der verarbeiteten  $\emptyset$  mindestens so groß wie die Anzahl der verarbeiteten 1 sein musste. Ein funktionierender Automat muss also damit umgehen können, dass während der Verarbeitung die 1 die Oberhand haben können.

## 7.2 Grammatik und Stackautomat

Das Beispiel von Abschnitt 7.1.5 zeigt, dass die kontextfreie Sprache der Klammerausdrücke mit einem Stackautomaten akzeptiert werden kann. Tatsächlich funktioniert dies auch allgemein. In diesem Abschnitt soll gezeigt werden, dass sich aus jeder kontextfreien Grammatik ein Stackautomat konstruieren lässt, der die gleiche Sprache akzeptiert, wie sie die Grammatik produziert.

### 7.2.1 Stack als Zwischenspeicher für die Regelanwendung

Die Grammatik

$$\begin{aligned} G : \quad K &\rightarrow (\textcolor{red}{K}) \\ &\rightarrow \textcolor{green}{KK} \\ &\rightarrow \textcolor{blue}{\varepsilon} \end{aligned}$$

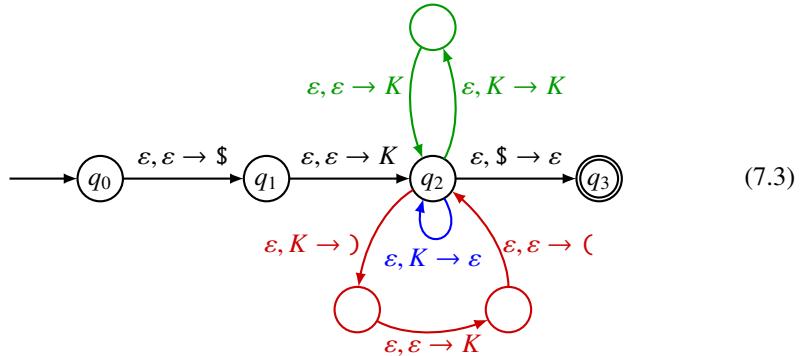
erzeugt die Sprache der Klammerausdrücke. Bei der Erzeugung eines Wortes der Sprache  $L(G)$  werden mehrfach Regeln der Grammatik angewendet, bis nur noch Terminalsymbole übrig bleiben. Zum Beispiel wird der Ausdruck  $(())$  durch die Regeln

$$K \rightarrow (\textcolor{red}{K}) \rightarrow (\textcolor{green}{KK}) \rightarrow ((\textcolor{red}{K})K) \rightarrow ((K)(\textcolor{red}{K})) \rightarrow ((\textcolor{blue}{\varepsilon})(K)) \rightarrow ((\textcolor{blue}{\varepsilon})) \rightarrow (())$$

erzeugt. Als Arbeitsspeicher für die Erzeugung der Wörter könnte man den Stack eines Stackautomaten rekrutieren. Dazu muss natürlich zunächst die Startvariable auf den Stack gebracht werden.

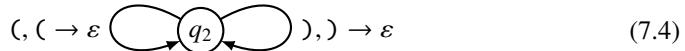
Als erste Regel wird  $K \rightarrow (K)$  angewendet. Auf dem Stack liegen anschließend drei Zeichen. Da zuoberst sicher eine Klammer liegt, ist die Anwendung weiterer Regeln blockiert, da Regeln nur auf Variablen anwendbar sind. Die Klammer muss daher entfernt werden. Die Klammer als Terminalsymbol wird aber auch im Input auftreten, also können wir diese entfernen, sobald sie auf dem Stack erscheint. Die Zeichen auf dem Stack müssen daher nach der Regelanwendung so abgelegt werden, dass die öffnende Klammer zuoberst liegt. Jede Regel der Grammatik wird daher in eine Folge von Übergängen übersetzt, die die rechte Seite der Regel von rechts nach links auf den Stack legt. Für die

Klammergrammatik tut der Stackautomat



dies. Die farbigen Übergänge sind nur möglich, wenn die Variable  $K$  zuoberst auf dem Stack liegt.

Auf dem Stack entstehen außer den Variablen auch Terminalsymbole, also im vorliegenden Fall Klammern. Diese müssen zu Zeichen aus dem Inputwort passen. Es sind daher noch zwei Regeln hinzuzufügen, die Terminalsymbole auf dem Stack zusammen mit passenden Symbolen aus dem Input entfernen. Diese Übergänge beim Zustand  $q_2$  sind:



Die Übergänge von (7.3) und (7.4) zusammen definieren einen Stackautomaten, der genau die Sprache der Klammerausdrücke akzeptiert.

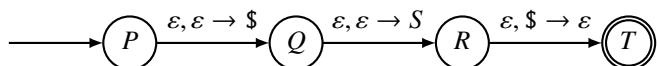
## 7.2.2 Stackautomat zu einer Grammatik

Der Prozess, der in Abschnitt 7.2.1 die Konstruktion eines Stackautomaten für die Sprache der korrekten Klammerausdrücke aus der Grammatik erlaubt hat, lässt sich allgemein formulieren und mithilfe der Chomsky-Normalform noch etwas vereinfachen.

**Satz 7.4.** *Sei  $G$  eine kontextfreie Grammatik, die die Sprache  $L(G)$  erzeugt, dann gibt es einen Stackautomaten  $P$ , der die gleiche Sprache  $L(P) = L(G)$  akzeptiert.*

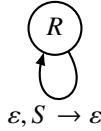
*Beweis.* Wir dürfen ohne Einschränkung der Allgemeinheit annehmen, dass die Grammatik bereits in Chomsky-Normalform ist und die Startvariable  $S$  hat. Den Stackautomaten  $P$  konstruieren wir jetzt in fünf Schritten.

1. Schritt: Das Grundgerüst des Stackautomaten ist



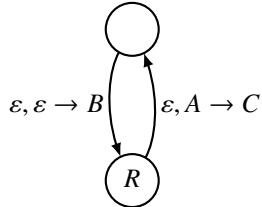
Es legt zuerst das Stackendsymbol  $\$$  auf den Stack und anschließend die Startvariable  $S$ . Akzeptiert werden kann nur, wenn nach den in den folgenden Schritten beim Zustand  $R$  angefügten Übergängen der Stack leer ist und damit das Symbol  $\$$  zuoberst auf dem Stack liegt.

2. Schritt: Falls die Regel  $S \rightarrow \epsilon$  zur Grammatik gehört, wird beim Zustand  $R$  der Übergang



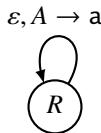
hinzugefügt. Er entfernt die Startvariable vom Stack. Dies ermöglicht, das leere Wort zu akzeptieren.

3. Schritt: Für jede Regel der Form  $A \rightarrow BC$  wird ein neuer Hilfszustand erzeugt und es werden die beiden Übergänge



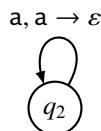
hinzugefügt. Sie entfernen  $A$  vom Stack und legen stattdessen zuerst  $C$  und dann  $B$  auf den Stack.

4. Schritt: Für jede Terminalsymbolregel der Form  $A \rightarrow a$  wird ein Übergang der Form



hinzugefügt. Er ersetzt die Variable  $A$  auf dem Stack durch das Terminalsymbol  $a$ .

5. Schritt: Für jedes Terminalsymbol  $a$  wird ein Übergang der Form



hinzugefügt. Er entfernt das Zeichen  $a$  vom Stack, wenn auch im Input ein  $a$  folgt. □

Jede kontextfreie Sprache lässt sich also durch einen Stackautomaten akzeptieren. Der konstruierte Automat ist jedoch nichtdeterministisch und daher wenig geeignet als Basis

für die Implementation eines Programms, welches Wörter der Sprache akzeptieren soll.

**Verständniskontrolle 7.2:** Verwenden Sie die kontextfreie Grammatik

$$S \rightarrow \emptyset S 1 \mid \epsilon$$

um einen Stackautomaten zu konstruieren, der die Sprache  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  akzeptiert.



## 7.3 LR-Parser

Es ist zwar möglich, zu jeder Grammatik einen Stackautomaten zu konstruieren. Da er nicht deterministisch ist, kann man davon keine performante Parser-Implementation erwarten. Dies hat aber viel grundsätzlichere Gründe, die in Abschnitt 7.3.1 untersucht werden. Die Syntaxanalyse ist daher nur für einen Teil aller kontextfreien Sprachen effizient möglich. Die in Abschnitt 7.3.2 beschriebene LR-Parser-Konstruktion zeigt einen verbreitet eingesetzten Lösungsansatz.

### 7.3.1 Warum Syntaxanalyse viel schwieriger ist

Deterministische endliche Automaten sind die Basis für die Implementation effizienter Regex-Engines. Sogar für nichtdeterministische endliche Automaten war dank der Konstruktion des Thompson-NEA immer eine Realisierung möglich, die in einer zur Wortlänge proportionalen Laufzeit entscheiden kann, ob ein Wort akzeptiert wird.

Für kontextfreie Grammatiken ist die bisher optimale Lösung der CYK-Algorithmus mit der Laufzeit  $O(|w|^3)$ . Er basiert auf einer rekursiven Analyse des ganzen Wortes, welches dazu aber erst gelesen werden muss. Je mehr Quelltext gelesen wird, desto umfangreicher werden auch die Analyseschritte, was sich in der immer größer werdenden Ableitungstabelle (Abschnitt 6.3.1) äußert. Für die Syntaxanalyse großer Mengen von Quellcode, wie sie ein Compiler durchführen muss, ist  $O(|w|^3)$  viel zu langsam. Es würde bedeuten, dass eine Verdoppelung der Größe des Quellcodes mit einer Verachtfachung der Compilezeit einhergeht. Gefragt ist ein Algorithmus, der die Syntaxanalyse in einer Laufzeit  $O(|w|)$  durchführen kann, also im Wesentlichen laufend während des Lesens des Quellcodes.

Die bereits in Abschnitt 7.1.4 eingeführte Sprache

$$L = \{w \in \Sigma^* \mid w \text{ ist ein Palindrom.}\}$$

der Palindrome zeigt jedoch, dass man für beliebige Sprachen keine allgemeine Implementation erwarten kann, die ähnlich schnell wie ein endlicher Automat sein könnte. Die Wörter der Form

$$w = \underbrace{1010 \dots 10}_{2n} \underbrace{1010101 \dots 01}_{2n}$$

sind sicher Palindrome. Die Grammatik

$$P \rightarrow \emptyset P \emptyset \mid 1 P 1 \quad (7.5)$$

$$\rightarrow \emptyset \mid 1 \mid \varepsilon \quad (7.6)$$

der Palindrome zeigt, dass die mittlere 1 anders behandelt wird als alle anderen. Die mittlere 1 wird durch die Regel (7.6) erzeugt, die anderen 1 durch die Regel (7.5). Es gibt jedoch keine Möglichkeit, gewöhnliche 1 von der mittleren 1 zu unterscheiden, wenn nicht bereits das ganze Wort gelesen worden ist.

Noch dramatischer wird das Problem für die Sprache  $L^*$  von beliebigen Verkettungen von Palindromen. Von der Syntaxanalyse einer Zeichenkette mit der Grammatik

$$\begin{aligned} S &\rightarrow S P \mid P \\ P &\rightarrow \emptyset P \emptyset \mid 1 P 1 \\ &\rightarrow \emptyset \mid 1 \mid \varepsilon \end{aligned} \quad (7.7)$$

erwartet man eine Zerlegung in Palindrome. Da einzelne Zeichen Palindrome sind, ist  $L^* = \Sigma^*$ . Somit hat sogar jedes Palindrom auch noch eine alternative Darstellung als Verkettung von Einzelzeichen, die alle als Palindrome betrachtet werden können. Das letzte Zeichen einer Zeichenkette kann die syntaktische Analyse verändern, wie das Beispiel

$$0101x = \begin{cases} \textcolor{brown}{0} \textcolor{blue}{1} \textcolor{brown}{0} \textcolor{blue}{1} = \textcolor{blue}{0} \textcolor{brown}{1} \textcolor{blue}{0} \textcolor{brown}{1} = \textcolor{brown}{0} \textcolor{blue}{1} \textcolor{brown}{0} \textcolor{blue}{1} = \textcolor{blue}{0} \textcolor{brown}{1} \textcolor{blue}{0} \textcolor{brown}{1} = \dots & \text{für } x = 0 \\ \textcolor{brown}{0} \textcolor{blue}{1} \textcolor{brown}{0} \textcolor{blue}{1} = \textcolor{blue}{0} \textcolor{brown}{1} \textcolor{blue}{0} \textcolor{brown}{1} = \textcolor{brown}{0} \textcolor{blue}{1} \textcolor{brown}{0} \textcolor{blue}{1} = \textcolor{blue}{0} \textcolor{brown}{1} \textcolor{blue}{0} \textcolor{brown}{1} & \text{für } x = 1 \end{cases} \quad (7.8)$$

zeigt, in dem die möglichen Aufteilungen in Palindrome durch die alternierende farbige Hinterlegung gekennzeichnet sind. Es ist somit nicht möglich, eine vollständige syntaktische Analyse eines Teilwortes durchzuführen. Die Grammatik (7.7) bevorzugt keine dieser Zerlegungen und ist daher nicht eindeutig.

Die Palindrome sind ein etwas pathologischer Fall, der die Situation bei der syntaktischen Analyse eines Programmtextes nicht wiedergibt. Ein typisches Programm besteht aus der Aneinanderreihung einzelner Anweisungen, die in Blöcken zusammengefasst sind. Die Anweisungen können Kontrollstrukturen sein, die wieder geschachtelte Blöcke von Anweisungen enthalten können. Diese Grobstruktur ist für den Programmierer sehr leicht zu erkennen. In den von C abstammenden Sprachen (C, C++, Java, JavaScript) wird die Blockstrukturierung mit geschweiften Klammern { und } durchgeführt. Der Programmierer ist sich gewohnt, den Text jeweils nach Paaren von öffnenden und schließenden Klammern abzusuchen, wobei er durch Konventionen über Einrückungen solcher Klammerpaare noch unterstützt wird. Da für die Blockstrukturierung ein Zeichen reserviert ist, wird die syntaktische Analyse in viel einfachere Teilaufgaben aufgeteilt. Was nach einer schließenden Klammer } kommt, hat keinen Einfluss auf die Analyse des Inhalts eines Blocks. Der in (7.8) illustrierte Fall wird also bei einer Programmiersprache nicht eintreten.

### 7.3.2 Ein deterministischer $O(n)$ -Parser

Die Sprache  $L = \{\emptyset^n 1^n \mid n > 0\}$  wird von der Grammatik

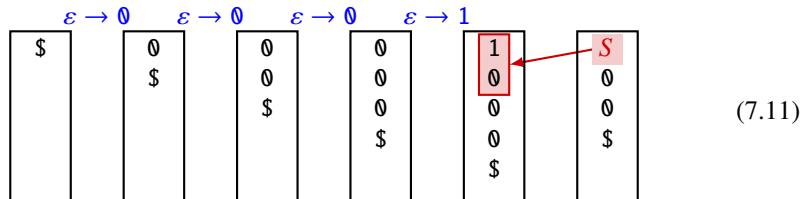
$$S \rightarrow \emptyset S 1 \quad (7.9)$$

$$S \rightarrow \emptyset 1 \quad (7.10)$$

erzeugt. Wir wissen bereits, wie ein nichtdeterministischer Stackautomat gefunden werden kann, der  $L$  akzeptiert. Er basiert auf der Idee, die Regeln mehr oder weniger blind auf den Stackinhalt anzuwenden und darauf zu hoffen, dass die Inputzeichen zu diesen Regelanwendungen passen. Eine deterministische Vorgehensweise müsste die Wahl der anzuwendenden Regeln vom Input steuern lassen.

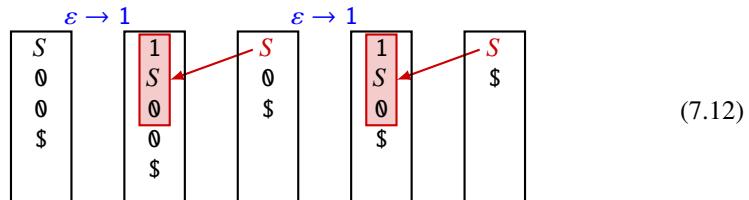
### Syntaxanalyse von $0^n 1^n$

Als Beispiel dafür, wie dies möglich werden könnte, versuchen wir das Wort  $000111$  mit der Grammatik (7.9)–(7.10) zu analysieren. Da wir nach einem einzelnen Zeichen  $0$  noch nicht wissen können, wann wir die zugehörige  $1$  finden werden, schieben wir zunächst jedes Zeichen auf den Stack. Nach vier Zeichen erreichen wir die Situation



Auf dem Stack erkennen wir von unten nach oben gelesen die Zeichenfolge  $01$ , welche auch die rechte Seite einer einzigen Regel ist, nämlich (7.10). Wir können daher  $01$  durch  $S$  ersetzen.

Da sich keine weiteren Muster von rechten Seiten von Regeln auf dem Stack erkennen lassen, schieben wir das nächste Inputzeichen ebenfalls auf den Stack. So entsteht auf dem Stack



von unten nach oben die rechte Seite  $0S1$ , die wir nach Regel (7.9) wieder durch  $S$  ersetzen können. Dies wiederholt sich, bis auf dem Stack nur noch die Startvariable  $S$  allein übrig bleibt, was anzeigt, dass im Input erfolgreich ein Wort gefunden wurde, welches aus  $S$  erzeugt werden kann.

### Shift-Reduce

Der Prozess der Syntaxanalyse im vorangegangenen Beispiel war vollständig deterministisch, er erfolgte nach zwei Regeln:

**Reduce:** Wenn auf dem Stack die rechte Seite einer Regel erkennbar ist, wende die Regel in umgekehrter Richtung an. Die Reduktionsoperationen sind in (7.11) und (7.12) rot dargestellt.

**Shift:** Schiebe Zeichen des Inputs auf den Stack. Die Shift-Operationen sind in (7.11) und (7.12) blau dargestellt.

Der Prozess terminiert mit der Startvariable auf dem Stack. Da die Veränderungen auf dem Stack immer nur umgekehrte Anwendungen von Regeln der Grammatik waren, können wir schließen, dass dieser Prozess genau Wörter akzeptiert, die von der Grammatik erzeugt werden.

Auch der Berechnungsaufwand ist unter Kontrolle. In jedem Schritt wird zunächst der Stack daraufhin inspiziert, ob sich die rechte Seite einer Regel dort findet. Dieser Aufwand ist von der Größenordnung der Anzahl der Regeln in der Grammatik, bleibt also immer gleich. Dann wird das nächste Zeichen auf den Stack geschoben, dieser Aufwand bleibt ebenfalls konstant.

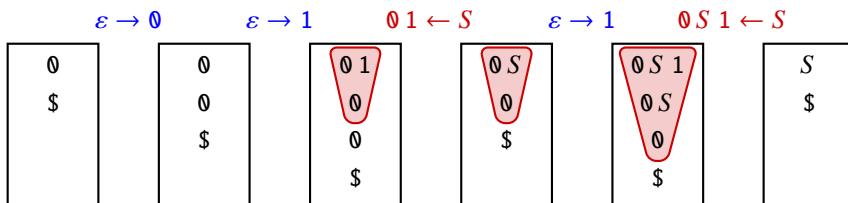
Es kann allerdings geschehen, dass mehrere Regelanwendungen möglich sind, bis wieder ein neues Zeichen auf den Stack geschoben werden muss. Da aber bei jedem Reduktionsschritt die Anzahl der Elemente auf dem Stack kleiner wird, kann es höchstens so viele Reduktionsschritte geben, wie es Inputzeichen gibt. Der beschriebene Prozess hat daher Laufzeit  $O(|w|)$ .

### LR-Operationen

Für die Ausführung der Reduce-Operation war es nötig, mehrere Elemente des Stacks zu inspizieren. Dies widerspricht der Definition eines Stacks, auf dem immer nur das oberste Element sichtbar ist. Die Reduktionsoperation entspricht auch nicht der Art und Weise, wie man mit einem Stackelement umgeht, das man wiedererkannt hat. Dazu muss jeweils die richtige Zahl von Elementen vom Stack entfernt werden, die Zahl ist abhängig vom Inhalt des Stacks.

Man kann die Operationen wieder in die Form einer Stackoperation bringen, wenn man bei jeder Shift-Operation das Element zuoberst auf dem Stack vergrößert, also etwas mehr Platz schafft. Dies setzt voraus, dass wir zwischen zwei verschiedenen Arten von Shift-Operationen unterscheiden können. Da ja bereits ein Zeichen auf dem Stack liegen kann, müssen wir entscheiden, ob das neue Symbol mit dem bereits vorhandenen zusammengefasst werden soll. Im Beispiel startet  $\emptyset$  immer eine neue Gruppe von Symbolen, aber Symbole  $S$  oder  $1$  müssen immer zusammen mit anderen Symbolen, die schon auf dem Stack liegen, wieder entfernt werden.

Statt ein Symbol, welches später zusammen mit anderen wieder entfernt wird, einfach nur den Stack zu schieben, wird mit den bereits vorhandenen Elementen ein neues Paket gebildet. In



werden die Elemente, die als Paket behandelt werden müssen, als farbiges Dreieck zusammengefasst. Die Durchführung der Reduktion entfernt also jeweils das ganze Dreieck und schiebt stattdessen die linke Seite der Regel auf den Stack. Dies kann wie bei der ersten Regelanwendung in der Mitte zur Bildung eines neuen **roten** Pakets führen. Wir werden auch Fälle sehen, wo nur ein Teil eines Paketes reduziert und mit dem Resultat ein neues Paket gebildet wird.

### Dot-Notation

In der Literatur findet man oft die folgende Notation für die Elemente, die auf dem Stack liegen. Statt eines Dreiecks wird mit einem Punkt innerhalb des Datenstroms die Stelle markiert, bis zu der die Elemente bereits verarbeitet worden sind. Die Shift-Operation bedeutet dann einfach, dass der Punkt um ein Zeichen nach rechts geschoben wird. Die Reduktionsoperation ersetzt das Element links vom Punkt durch die linke Seite der Regel und platziert den Punkt rechts davon.

Die Verarbeitung des Wortes **0011** kann in dieser Notation als

.	<b>0</b>	<b>0</b>	1	1	<i>e</i>
<b>0</b>	.	<b>0</b>	1	1	<i>e</i>
<b>0</b>	<b>0</b>	.	1	1	<i>e</i>
<b>0</b>	<b>0</b>	<b>1</b>	.	1	<i>e</i>
<b>0</b>	<b>S</b>	.	1	<i>e</i>	
<b>0</b>	<b>S</b>	<b>1</b>	.	<i>e</i>	
<b>S</b>	.	<i>e</i>			

geschrieben werden. Die rot hinterlegten Teile sind rechte Seiten einer Regel und können daher auf die roten **S** reduziert werden.

### Ein endlicher Automat zur Mustererkennung

Die Voraussetzung für den Reduktionsschritt ist, dass der Stackinhalt als die rechte Seite einer Regel erkannt wird. Es geht also um ein Mustererkennungsproblem für Wörter, die sich aus Terminalsymbolen und Variablen zusammensetzen. Wir möchten nach jedem weiteren Symbol sofort die Information erhalten können, ob eine Reduktion möglich ist. Dies ist genau die Arbeitsweise eines endlichen Automaten, der die möglichen rechten Seiten akzeptieren soll.

Für die Sprache  $0^n 1^n$  mit der Grammatik

$$\begin{aligned} S &\rightarrow 0S1 \\ &\rightarrow 01 \end{aligned}$$

müssen oben auf dem Stack die Wörter

$$\varepsilon, 0, 01, 0S, 0S1,$$

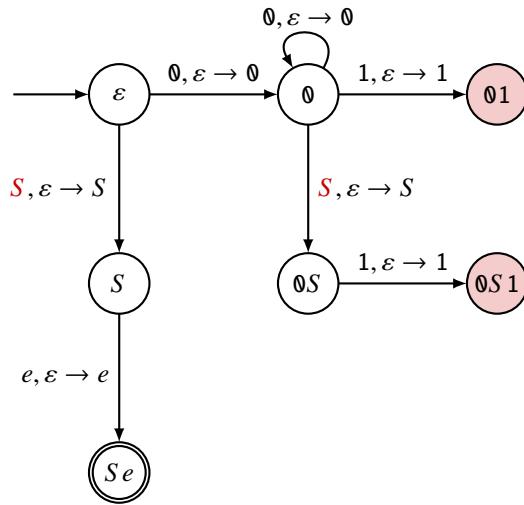


Abbildung 7.4: Shift-Reduce Automat für die Syntaxanalyse der Sprache  $\{0^n 1^n \mid n > 0\}$ . In den roten Zuständen findet eine Reduktion mit einer Grammatikregel statt. Die dabei entstehende Variable  $S$  wird je nach Stackinhalt unter der reduzierten rechten Seite für die beiden mit **rotem S** bezeichneten Übergänge verwendet. Der Automat ist insofern deterministisch, da alle Übergänge vom Input ausgelöst werden.  $e$  steht für das Ende des Inputs.

unterschieden werden können. Ein endlicher Automat dafür braucht also mindestens fünf Zustände.

Abbildung 7.4 zeigt einen Stackautomaten, der die nötigen Aktivitäten codiert. Darin fehlen allerdings die Veränderungen, die bei einer Reduktionsoperation auszuführen sind. Bei der Reduktion mit der Regel  $S \rightarrow 01$  wird  $01$  entfernt und  $S$  wieder auf den Stack gelegt. Dabei muss auch der Zustand wieder auf den Zustand des Automaten vor der Verarbeitung von  $01$  zurückgesetzt werden.

Damit dies möglich wird, wird der Stack etwas verbreitert, um jeweils auch den Zustand zu speichern, zu dem zurückgekehrt werden muss. Die grüne Spalte in

$\varepsilon \rightarrow 0$	$\varepsilon \rightarrow 1$	$01 \leftarrow S$	$\varepsilon \rightarrow 1$	$0S1 \leftarrow S$
0 \$	0 \$	01 0 0 \$	0S 0 \$	0S1 0S 0 \$
ε ε	ε ε	01 0 0 ε	0S 0 ε	0S1 0S 0 ε

enthält den Zustand, der bei einer bestimmten Stackhöhe erreicht worden ist, mit den Bezeichnungen des Zustands wie in Abbildung 7.4. Wenn das darüber liegende Element entfernt wird, muss der Automat auch auf diesen Zustand zurückgesetzt werden. Dies spezifiziert die Reduktionsübergänge in Abbildung 7.4 vollständig.

### Shift/reduce-Konflikt

Nicht jede Grammatik eignet sich für die Verarbeitung mit einem Shift-Reduce-Parser. Zum Beispiel ist die Grammatik

$$\text{expr} \rightarrow \text{expr} + \text{expr} \quad (7.13)$$

$$\rightarrow \text{expr} * \text{expr} \quad (7.14)$$

$$\rightarrow \text{zahl}$$

nicht eindeutig. Bei der Analyse des Ausdrucks  $\text{expr} + \text{expr} * \text{expr}$  gibt es zwei Möglichkeiten, die Regeln anzuwenden. Steht der Punkt in der Dot-Notation nach einem  $\text{expr}$ -Symbol, dann kann dies auf die folgenden zwei Arten auf Regeln passen:

$$\begin{aligned} \text{expr} &\leftarrow \text{expr} + \text{expr} . \ * \text{expr} \\ \text{expr} &\leftarrow \text{expr} + \text{expr} . \ * \text{expr} \end{aligned}$$

Im ersten Fall verarbeitet man die Additionsoperation. Im zweiten Fall würde man weitere Elemente verschieben, um schließlich einen Multiplikationsausdruck zu erhalten. Die Zweideutigkeit wirkt sich also auf die Reihenfolge von Addition und Multiplikation aus. Die gleiche Art von Konflikt tritt auch mit vertauschten Operationszeichen auf.

Diese Art eines Konflikts zwischen Weiterverschieben oder Reduzieren heißt *shift/reduce-Konflikt*. Man könnte den Syntaxanalysevorgang so definieren, dass immer der Shift-Operation der Vorzug gegeben wird, wie es einige Parsergeneratoren tun. Dies hat im obigen Beispiel zur Folge, dass zuerst die Multiplikation ausgeführt wird, dann die Addition. Dies ist die richtige Operationsreihenfolge. Vertauscht man die Operationszeichen, wird zuerst die Additionsoperation ausgeführt, erst dann die Multiplikation. Dies führt auf ein falsches numerisches Resultat.

### Reduce/reduce-Konflikt

Damit immer eindeutig ist, welche Regel angewendet werden soll, darf es nicht vorkommen, dass eine Regel auf der rechten Seite die rechte Seite einer anderen Regel als Anfangsteil enthält. In der Grammatik

$$\begin{aligned} S &\rightarrow S A \\ &\rightarrow S B \\ A &\rightarrow X Y \\ B &\rightarrow X Y \\ X &\rightarrow \dots \\ Y &\rightarrow \dots \end{aligned}$$

wird von den Regeln mit  $S$  eine Folge von Elementen für die Variablen  $A$  und  $B$  aufgebaut. Sowohl  $A$  als auch  $B$  können aus der Verkettung  $X Y$  reduziert werden. Es gibt aber keinen Hinweis, welche Regel gewählt werden soll. Dies ist ein *reduce/reduce-Konflikt*.

## Look-Ahead

Die Konfliktsituationen können möglicherweise entschärft oder eliminiert werden, wenn der Parser vorausschauend zusätzliche Symbole des Inputs untersuchen kann. Ein LR( $k$ )-Parser ist ein Parser wie oben, der aber für die Entscheidung über die Shift- und Reduce-Operationen  $k$  Symbole vorausschauen kann.

## Anwendungsbereich

Die LR-Technik zur Syntaxanalyse erfasst zwar nicht alle kontextfreien Sprachen, ist aber allgemein genug, dass damit die modernen Programmiersprachen analysiert werden können. Wenn es möglich ist, eine Grammatik ohne die im vorangegangenen Abschnitt behandelten Konflikte zu formulieren, dann kann ein LR-Parser dafür eingesetzt werden.

### 7.3.3 Parsergenerator Bison

Die Erzeugung eines Parsers aus den Grammatikregeln ist ein automatisierbarer Prozess. Ein Parsergenerator kann aus einer Grammatik automatisch den Quellcode eines Parsers erzeugen. Tatsächlich gibt es für fast jede denkbare Programmiersprache auch eine Auswahl von Parsergeneratoren. Die Wikipedia-Seite [9] gibt einen Überblick über die Vielfalt.

In den frühen 70er-Jahren entwickelte Stephen C. Johnson einen Parser-Generator für die Sprache B, den er Yacc (Yet-Another-Compiler-Compiler) nannte. Später wurde Yacc in C neu geschrieben und wurde Teil von Unix. Werkzeuge wie der Scannergenerator Lex und der Parsergenerator Yacc machten Unix zu einer attraktiven Softwareentwicklungsumgebung und leisteten damit einen wichtigen Beitrag zur Beliebtheit von Unix im akademischen Umfeld.

Für das GNU-Projekt hat Robert Corbett den mit Yacc weitgehend kompatiblen Parsergenerator Bison entwickelt. Viele Werkzeuge des GNU-Projektes verwenden generierte Parser, zum Beispiel die Bash [18], Octave [19], Perl [46] und PHP [47]. Bison wird oft zusammen mit dem in Abschnitt 4.5.1 beschriebenen Flex eingesetzt.

## Grammatik

Bison verwendet eine Syntax, die der BNF ähnlich ist. Die Grammatik für die Sprache  $L = \{\emptyset^n 1^n \mid n \geq 0\}$  kann in Yacc-Notation als

Grammatik:	$S \rightarrow \emptyset S 1$ $\rightarrow \epsilon$	$\rightsquigarrow$	Yacc/Bison:	S:	$'\emptyset' S '1'$   ;
------------	---	--------------------	-------------	----	-------------------------------

geschrieben werden. Ein Programm wird aber ein Inputfile mit mehreren Zeilen von Wörtern von  $L$  verarbeiten wollen, dies wird durch die Grammatik

Szeilen:

Szeilen Szeile
Szeile

```

;

Szeile: S '\n'
;

S:      '0' S '1'
|
;

```

ausgedrückt. Die Regel für die Variable `Szeile` definiert eine Zeile mit einem Wort von  $L$ , sie muss mit einem Newline '`\n`' terminiert sein. Die Regel für `Szeilen` definiert Verkettungen von beliebig vielen Zeilen.

Um ein funktionsfähiges Programm zu erhalten, muss noch ein Scanner, vorzugsweise mit Flex erzeugt, hinzugefügt werden. Im vorliegenden Fall ist er besonders einfach, weil er jedes Zeichen an den Parser weiterreichen kann. Außerdem müssen zwei Hilfsfunktionen `void yyerror(char *errmsg)` und `int yywrap()` sowie die `main()`-Funktion implementiert werden. Der gesamte Quellcode ist in Abbildung 7.5 zusammengestellt.

### Ein Calculator-Programm

Auf der Basis der Expression-Term-Factor-Grammatik von Abschnitt 5.3.1 kann man mit `flex(1)` und `bison(1)` mit recht geringem Aufwand ein Taschenrechnerprogramm entwickeln. Im verlinkten Code ist dies in mehreren Phasen dargestellt. Es beginnt mit der Definition der Tokens für einen mit `flex(1)` generierten Scanner und der Grammatikspezifikation für `bison(1)`. In späteren Phasen wird die Funktionalität erweitert, die Grammatik wird um Funktionen, Konstanten, Register und weitere Operationen erweitert. In der letzten Phase hat die Software die Fähigkeit, den Syntaxbaum darzustellen.



[autosp.ch/c/4](http://autosp.ch/c/4)

### Werte

Der von Bison erzeugte Parser soll nicht einfach nur die syntaktische Korrektheit feststellen, sondern er soll auch die Grundlage für den Aufbau eines Syntaxbaumes oder für die Evaluation eines Ausdrucks liefern. Dazu muss den Variablen einer Regel ein Wert zugewiesen werden. Bison verwendet dazu eine C-Union, deren Felder ebenfalls im Bison-Quellcode spezifiziert werden. In einem Programm, welches arithmetische Ausdrücke auswerten kann, sind die vom Scanner erkannten Tokens zum Beispiel Zahlen, Nummern von Registern oder Namen von Konstanten. Die Union-Definition

```

%union {
    int      reg;
    double   value;
    char     name[10];
}

```

ermöglicht, je nach Art des erkannten Tokens einen Zahlenwert im Feld `value`, einen Namen im Feld `name[10]` oder eine Registernummer im Feld `reg` zurückzugeben.

Bison Parser-Definition	Main File
<pre>%{ /*  * nulleins-grammar.y  */ extern void      yyerror(char *errmsg); extern int       yylex(void); %} %% Szeilen:     Szeilen Szeile       Szeile    ; Szeile: S '\n'        ; S:     '0' S '1'                 ; %%</pre>	<pre>/*  * nulleins.c  *  * (c) 2024 Prof Dr Andreas Müller  */ #include &lt;stdio.h&gt; #include &lt;stdlib.h&gt; #include &lt;getopt.h&gt; #include &lt;unistd.h&gt;  extern int      yydebug; extern int      yyparse();  void    yyerror(char *errmsg) {     fprintf(stderr, "error: %s\n", errmsg); }  int    yywrap() {     return 1; }  int    main(int argc, char *argv[]) {     yydebug = 0;     int    c;     while (EOF != (c = getopt(argc, argv, "d")))         switch (c) {             case 'd':                 yydebug = 1;                 break;         }     while (!feof(stdin)) {         yyparse();     }     return EXIT_SUCCESS; }</pre>
Flex Scanner-Definition	
<pre>%{ /*  * nulleins-tokens.l  */ %} %% .      { return yytext[0]; } \n     { return '\n'; } %%</pre>	

Abbildung 7.5: Vollständiger Quellcode für einen interaktiven Parser für die Sprache  $L = \{0^n 1^n \mid n \geq 0\}$  realisiert mit Flex und Bison.

Die Zuweisung des Datentyps zu verschiedenen Tokens erfolgt über `%token`-Direktiven. Durch

```
%token <value>NUMBER
%token <reg>REGISTER
%token <name>CONSTANT
```

wird deklariert, dass die vom Scanner erkannten NUMBER-Tokens eine Gleitkommazahl als Wert haben, dass REGISTER-Tokens eine Registernummer liefern und dass CONSTANT-Tokens den Namen einer Konstanten mitbringen.

## Aktionen

Der Wert eines Ausdrucks wird ermittelt, indem man jeder Regel eine Aktion zur Bestimmung des Wertes der linken Seite aus den Werten der Komponenten der rechten Seite

zuordnet, wie dies in Abschnitt 5.3 gezeigt wurde. In Phase 11 entsteht der Syntaxbaum, indem man bei jeder Regelanwendung einen Knoten des Baums hinzufügt.

In der Bison-Syntax kann einer Regel ein beliebiger Block mit C-Code hinzugefügt werden. Wird die Regel bei der Reduktion verwendet, wird der Code vom erzeugten Parser ausgeführt. Bei der Reduktion sind auch die Werte der Variablen auf der rechten Seite bekannt, diese werden von Bison über den Platzhalter  $\$n$  für die Variable an der  $n$ -ten Stelle bereitgestellt. Der Platzhalter für die linke Seite der Regel ist  $\$\$$ . Die Auswertung arithmetischer Ausdrücke kann daher mit dem Code

```

expr:      term          { $$ = $1; }
          | expr '+' term { $$ = $1 + $3; }
          | expr '-' term { $$ = $1 - $3; }
;
term:      term '*' factor { $$ = $1 * $3; }
          | term '/' factor { $$ = $1 / $3; }
          | factor           { $$ = $1; }
;
factor:    '(' expr ')'   { $$ = $2; }
          | NUMBER          { $$ = $1; }
;

```

erfolgen. Für die Zuweisung der Werte muss der Parsergenerator auch wissen, welchen Datentyp die Variablen haben. Dazu dient die Deklaration

```
%type <value>expr term factor
```

die den drei Variablen `expr`, `term` und `factor` den Typ einer Gleitkommazahl gibt.

Da die Grammatik bereits dafür sorgt, dass Punktoperationen vor Strichoperationen ausgeführt werden, ist nur noch die eigentliche Rechnung zu codieren.

### Shift/Reduce für die Expression-Term-Factor-Grammatik

Der von Bison erzeugte Parser kann im Debug-Mode betrieben werden, indem die Variable `yydebug` auf 1 gesetzt wird. Dann gibt der Parser ein umfangreiches Log aller Operationen aus. Für den Ausdruck  $(1+3)^*7$  ergibt sich der in Tabelle 7.1 dargestellte Ablauf.

Man kann in der Tabelle auch erkennen, dass bei einer Reduktion nicht immer alle bisher auf den Stack geschobenen Elemente reduziert werden müssen. Bei der Reduktion von  $expr + term$  auf  $expr$  wird die Klammer, die bereits auf den Stack geschoben worden ist, nicht reduziert. Sie wird erst zwei Zeilen später reduziert, wenn die Regel  $factor \rightarrow (expr)$  angewendet wird.

### Syntaxbäume für arithmetische Ausdrücke

Statt nur die Werte bei der Auswertung einer Regel zu bestimmen und an die darüberliegenden Knoten des Syntaxbaumes weiterzugeben, kann man auch im Speicher eine Darstellung des Syntaxbaums aufbauen. Dies ist in Phase 11 des Projektes realisiert. Damit wird es möglich, einen früheren Ausdruck erneut auszuführen oder auch nur die Interpretation des Ausdrucks durch die Grammatik zu überprüfen. Im nächsten Abschnitt wird dies

Aktion	Regel	LR-Elemente
Input		( 1 + 3 ) * 7 \n
Shifting token '('		( . 1 + 3 ) * 7 \n
Shifting NUMBER		( 1 . + 3 ) * 7 \n
Reducing by rule 10	factor → NUMBER	( factor . + 3 ) * 7 \n
Reducing by rule 8	term → factor	( term . + 3 ) * 7 \n
Reducing by rule 3	expr → term	( expr . + 3 ) * 7 \n
Shifting '+'		( expr + . 3 ) * 7 \n
Shifting NUMBER		( expr + 3 . ) * 7 \n
Reducing by rule 10	factor → NUMBER	( expr + factor . ) * 7 \n
Reducing by rule 8	term → factor	( expr + term . ) * 7 \n
Reducing by rule 4	expr → expr + term	( expr . ) * 7 \n
Shifting ')'		( expr ) . * 7 \n
Reducing by rule 9	factor → ( expr )	factor . * 7 \n
Reducing by rule 8	term → factor	term . * 7 \n
Shifting '**'		term * . 7 \n
Shifting NUMBER		term * 7 . \n
Reducing by rule 10	factor → NUMBER	term * factor . \n
Reducing by rule 6	term → term * factor	term . \n
Reducing by rule 3	expr → term	expr . \n
Shifting '\n'		expr \n .
Reducing by rule 1	exprline → expr \n	exprline .

Tabelle 7.1: Shift/Reduce-Operation zur Syntaxanalyse des Ausdrucks  $(1+3)/7$ . Die Shift-Operationen sind blau hinterlegt, die Reduce-Operationen hellrot. Die von der Regel erfasste rechte Seite ist in der Spalte der LR-Elemente jeweils dunkler rot hinterlegt, die resultierende Variable auf der nächsten Zeile in roter Schrift gesetzt.

für eine Grammatikänderung ausprobiert, die durch verschiedene Interpretationen eines Internet-Memes suggeriert wird.

### Multiplikation ohne Multiplikationszeichen

In der Mathematik ist es üblich, das Produkt zweier Variablen  $a$  und  $b$  ohne Multiplikationszeichen als  $ab$  zu schreiben. Diese Schreibweise hat zu epischen Diskussionen in den sozialen Medien über Memes wie jenes in Abbildung 7.6 geführt. Sie hat auch Fehler in verbreiteten Taschenrechneranwendungen zutage gefördert: Tippt man die Aufgabe genau so wie im Meme ein, ergeben sich je nach Rechnerversion zum Teil skurrile Resultate.

Die Schreibweise  $8 \div 2(4 - 2)$  ist in der Mathematik nicht üblich und wird von der Expression-Term-Factor-Grammatik gar nicht erfasst. Ersetzt man das Geteiltzeichen  $\div$  durch den üblicheren Schrägstrich, erhält man  $8/2(4-2)$ . Die Notation  $a/bc$  wird in der Mathematik typischerweise als  $a/(bc)$  interpretiert, die Operationen werden in diesem Fall also nicht von links nach rechts ausgeführt, was  $(a/b)c$  ergäbe. In dieser Interpretation müsste das Meme den Wert

$$8/2(4-2) = \frac{8}{2(4-2)} = \frac{8}{2 \cdot 2} = 2 \quad (7.15)$$

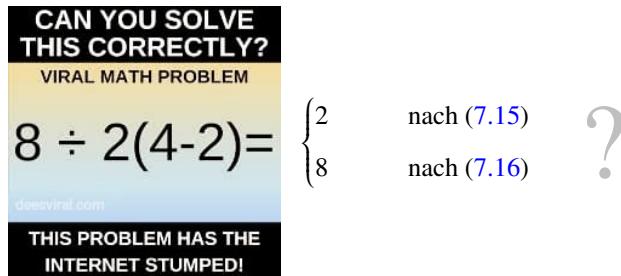


Abbildung 7.6: Internet-Meme mit einer arithmetischen Aufgabe, die für epische Diskussionen und Flame-Wars in den sozialen Medien sorgte. Der Grund dafür ist, dass die Aufgabe schlecht gestellt ist und nur mit einer erweiterten Grammatik syntaktisch analysiert werden kann. Sie führt auch dazu, dass Ausdrücke je nach Schreibweise anders ausgewertet werden.

ergeben.

Die Expression-Term-Factor-Grammatik erlaubt nicht, den Ausdruck auszuwerten. Die Klammer müsste ein *factor* sein, die Zahl davor ein *factor* oder ein *term*, aber die Kombination *factor factor* kommt in der Grammatik nicht vor. Es bleibt also nichts anderes, als das Multiplikationszeichen explizit zu schreiben, dann entsteht der Ausdruck  $8/2^*(4-2)$ . Die Expression-Term-Factor-Grammatik behandelt Division und Multiplikation auf der gleichen Stufe, die Auswertung erfolgt daher von links nach rechts und ergibt

$$8/2^*(4-2) = \frac{8}{2} \cdot (4 - 2) = \frac{8}{2} \cdot 2 = 8. \quad (7.16)$$

Dies ist nicht das Resultat, das der Mathematiker in (7.15) erhalten hat!

Man könnte versucht sein, die Grammatik zu erweitern und eine Regel

$$\text{factor} \rightarrow \text{factor factor} \quad (7.17)$$

hinzuzufügen. In Yacc-Notation könnte dies in der Form

```
factor: '(' expr ')'           { $$ = $2; }
      | NUMBER                 { $$ = $1; }
      | factor factor          { $$ = $1 * $2; }
      ;
```

erfolgen. Dies ist im Calculator-Projekt in Phase 10 durchgeführt. Bison warnt aber sofort vor einer großen Zahl von shift/reduce-Konflikten. Dies ist nicht überraschend, denn schon bei der Klammergrammatik haben wir gesehen, dass Regeln der Form  $K \rightarrow KK$  zu nicht eindeutigen Syntaxbäumen führen. Genau so eine Regel haben wir jetzt für Faktoren eingeführt. Der Schaden hält sich aber in Grenzen, weil die Multiplikation assoziativ ist und damit die Reihenfolge der Auswertungen der Multiplikationen keine Rolle spielt.

Tatsächlich funktioniert die Berechnung auch, die Syntaxbäume für die Berechnung des Ausdrucks mit (links) und ohne (rechts) Multiplikationszeichen sind in Abbildung 7.7

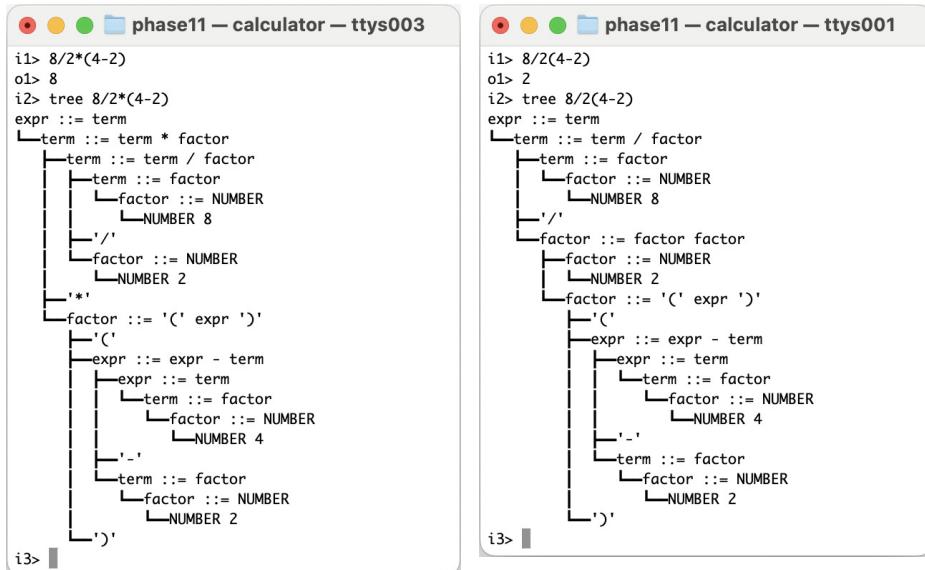


Abbildung 7.7: Verschiedene Syntaxbäume für zwei verschiedene Formen der Aufgabe des Memes in Abbildung 7.6. Links wird die normale Expression-Term-Factor-Grammatik verwendet, die nach dem Multiplikationszeichen verlangt, rechts die um die Regel  $\text{factor} \rightarrow \text{factor factor}$  erweiterte, nicht mehr eindeutige Grammatik, die die direkte Auswertung ermöglicht. Der linke Syntaxbaum wertet erst die Division 8/2 aus, dann erst wird das Produkt mit dem Klammerausdruck gebildet. Der rechte Syntaxbaum berechnet erst das Produkt von 2 mit der Klammer, dann folgt erst die Division.

dargestellt. Im linken Syntaxbaum wird erst der Quotient 8/2 ermittelt, dann erst das Produkt mit der Klammer. Der rechte Syntaxbaum beginnt mit der Multiplikation von 2 mit der Klammer, erst dann wird die Division ausgeführt.

Für eine Programmiersprache ist es unerwünscht, dass es verschiedene Notationen mit und ohne Multiplikationszeichen gibt, die in der mathematischen Intuition zum gleichen Resultat führen sollten. Die Gefahr schwer zu erkennender Softwarefehler ist sehr groß. Daher wird die Grammatikerweiterung in keiner Programmiersprache verwendet.

## Klammern

Für Klammerausdrücke haben wir die zwei Grammatiken

$$\left. \begin{array}{l} K \rightarrow (K) \\ \quad \rightarrow KK \\ \quad \rightarrow \varepsilon \end{array} \right\} \quad \text{und} \quad \left\{ \begin{array}{l} K \rightarrow (K)K \\ \quad \rightarrow \varepsilon \end{array} \right. \quad (7.18)$$

kennengelernt. In der Yacc-Syntax erhalten die Grammatiken die Form

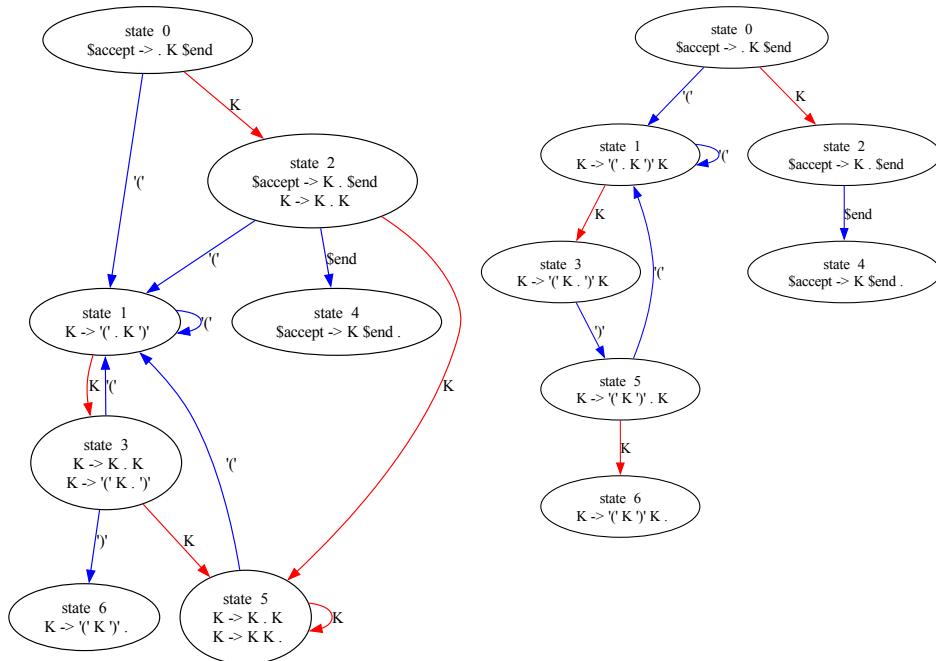


Abbildung 7.8: Stackautomat erzeugt von Bison für die Klammergrammatiken (7.18). Blaue Übergänge sind Shift-Operationen, rote Pfeile gehören zu Reduce-Operationen. Die linke Grammatik ist nicht eindeutig, die Konflikte sind daran erkennbar, dass die Zustände 2, 3 und 5 mehrfache Beschriftungen haben.

$$\begin{array}{ll}
 \text{K:} & \text{K:} \\
 & '(', K ')', \\
 & | \quad K K \\
 & | \\
 & ; & | \\
 & & ; \\
 \text{und} & 
 \end{array}$$

Die linke Grammatik ist nicht eindeutig und führt wie erwartet auf 7 shift/reduce-Konflikte und 3 reduce/reduce-Konflikte. Die rechte Grammatik ist eindeutig und wird von Bison ohne Fehler- oder Warnmeldungen verarbeitet.

Bison kann den erzeugten Automaten im vcg-Format ausgeben, welches in das verbreitete Dot-Format von Graphviz konvertiert werden kann. Die beiden Stackautomaten sind in Abbildung 7.8 dargestellt. Die blauen Übergänge sind Shift-Operationen, die roten werden für Reduktionsoperationen gebraucht. Mehrfache Beschriftungen eines Zustands deuten auf Konflikte hin.

### Palindrome

Man kann auch versuchen, mit Bison einen Parser für Palindrome zu konstruieren. Die Grammatik

Aktion	Regel	LR-Elemente
Input		a b a \n
Shifting 'a'		a . b a \n
Shifting 'b'		a b . a \n
Shifting 'a'		a b a . \n
Reducing by rule 6	$P \rightarrow a$	a b P . \n
error: syntax error		

Abbildung 7.9: Verarbeitung des Palindroms aba mit einem Parser, der von Bison aus der Palindrom-Grammatik abgeleitet wurde. Da der Parser die Shift-Operation bevorzugt, führt die Reduktion im vierten Schritt auf eine Situation, in der keine weiteren Reduktionen mehr möglich sind. Es wäre nötig gewesen, als dritten Schritt das b auf  $P$  zu reduzieren, dann wäre eine weitere Reduktion möglich gewesen. Dies ist aber nur zu erkennen, wenn der Parser mehr als ein Element vorausschauen kann.

Grammatik	Yacc/Bison-Format:
$\begin{aligned} P &\rightarrow a P a \mid b P b \\ &\rightarrow a \mid b \\ &\rightarrow \epsilon \end{aligned}$	$\begin{aligned} P: & 'a' P 'a' \mid 'b' P 'b' \\ &\mid 'a' \mid 'b' \\ &\mid ; \end{aligned}$

ist mehrdeutig und verursacht eine große Zahl von Konflikten, aber bison versucht trotzdem einen Parser zu generieren. Wir wissen aber bereits, dass Palindrome nur korrekt analysiert werden können, wenn das ganze Wort gelesen wurde. Es reicht nicht, nur wenige Elemente look-ahead zu lesen. Da Bison nur ein einziges Element look-ahead zur Verfügung steht, können Palindrome mit mehr als einem Zeichen nicht erkannt werden.

In der in Abbildung 7.9 gezeigten Verarbeitung des Palindroms aba reduziert der Parser das letzte Element im vierten Schritt mit der Regel  $P \rightarrow a$ , was auf eine Situation führt, in der kein Palindrom mehr erkannt werden kann. Der Parser meldet daher einen Syntaxfehler. Wäre im dritten Schritt das Element b auf  $P$  reduziert worden, hätte die Syntaxanalyse erfolgreich sein können. Dies wäre aber nur erkennbar gewesen, wenn der Parser mehr als nur ein Zeichen hätte vorausschauen können.

Neuere Versionen des Bison Parsergenerators, die allerdings in den Linux-Distributionen noch kaum Eingang gefunden haben, sind nicht auf LR(1)-Parser beschränkt. Insbesondere können GLR-Parser erzeugt werden, die für alle kontextfreien Sprachen funktionieren können. Allerdings ist die Laufzeit dieser Parser im schlimmsten Fall wieder  $O(|w|^3)$ , also so langsam wie beim CYK-Algorithmus.

## 7.4 Grammatik eines Stackautomaten

Nach Satz 7.4 kann jede kontextfreie Sprache von einem Stackautomaten akzeptiert werden. Es ist aber immer noch denkbar, dass Stackautomaten auch Sprachen akzeptieren können, die nicht von einer Grammatik produziert werden können. In diesem Abschnitt soll gezeigt werden, wie man zu einem Stackautomaten auch immer eine kontextfreie Grammatik konstruieren kann, die die gleiche Sprache erzeugt.

### 7.4.1 Regeln als Wege durch das Zustandsdiagramm

Die Variablen einer Grammatik stehen für Teile eines Wortes. In der Expression-Term-Factor-Grammatik

$$\begin{aligned} \textit{expression} &\rightarrow \textit{expression} + \textit{term} \mid \textit{expression} - \textit{term} \mid \textit{term} \\ \textit{term} &\rightarrow \textit{term} * \textit{factor} \mid \textit{term} / \textit{factor} \mid \textit{factor} \\ \textit{factor} &\rightarrow ( \textit{expression} ) \mid \textit{number} \\ \textit{number} &\rightarrow \dots \end{aligned}$$

stehen die Variablen für Teile von arithmetischen Ausdrücken, die auch in der Mathematik die entsprechenden Namen tragen.

In einem Stackautomaten führt ein Wortteil von einem Zustand zu einem anderen, wobei sich auch der Stackinhalt ändern kann. Diese Analogie legt nahe, Paare von Zuständen des Automaten als Variablen einer Grammatik zu betrachten. Die Regeln der Grammatik entstehen dann daraus, dass sich der Weg zwischen zwei Zuständen im Automaten durch Zwischenzustände in Teile aufteilen lässt, die natürlich auch wieder Wege zwischen zwei Zuständen sind. Ein Wort entsteht also durch Zusammensetzung der Wörter, die zu den durch die Zwischenzustände gebildeten Teilwegen gehören.

Damit ein Paar von Zuständen als eigenständige Variable einer kontextfreien Grammatik betrachtet werden kann, darf das, was zwischen den beiden Zuständen auf dem Stack passiert, nicht von Dingen, die vorher oder nachher auf den Stack gelegt werden, beeinflusst sein. Ein Paar von Zuständen  $p$  und  $q$  kann also nur für Wörter stehen, die von  $p$  nach  $q$  führen, wenn man in  $p$  mit leerem Stack beginnt und am Ende wieder einen leeren Stack vorfindet.

**Definition 7.5.** Sei  $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$  ein Stackautomat, dann ist  $A_{pq}$  definiert als die Variable, die für die Wörter steht, die den Stackautomaten mit leerem Stack vom Zustand  $p$  in den Zustand  $q$  überführt.

### 7.4.2 Vorbereitung des Stackautomaten

Die Definition 7.5 legt fest, wie aus einem Stackautomaten Variablen für eine kontextfreie Grammatik abgeleitet werden sollen. Für beliebige Stackautomaten führt die Definition jedoch auf Inkonsistenzen, die zu pathologischen Grammatiken Anlass geben.

#### Startvariable

Die Startvariable einer Grammatik steht für beliebige Wörter der erzeugten Sprache. Wäre die Definition 7.5 anwendbar, müsste auch die einzige Startvariable der Grammatik von der Form  $A_{pq}$  sein, wobei  $p$  der Startzustand des Stackautomaten und  $q$  ein Akzeptierzustand ist. Hat der Stackautomat mehrere Akzeptierzustände, ergäben sich so mehrere mögliche Startvariablen. Damit die Definition die Konstruktion einer sinnvollen Grammatik ermöglicht, darf es nur einen Akzeptierzustand geben.

Ein Stackautomat mit mehreren Akzeptierzuständen kann wie in Abbildung 7.10 in einen Stackautomaten umgewandelt werden, der nur einen Akzeptierzustand hat. Dazu

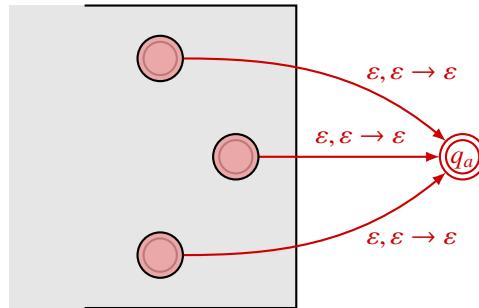
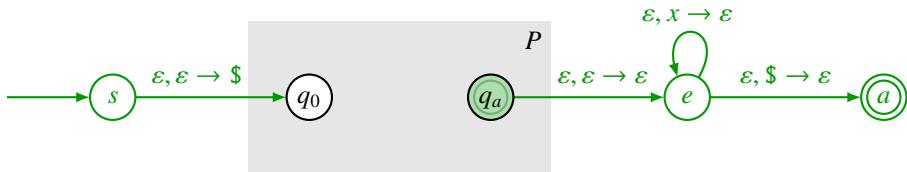


Abbildung 7.10: Umwandlung eines Stackautomaten in einen, der nur noch einen einzigen Akzeptierzustand hat. Es werden  $\varepsilon$ -Übergänge zu einem neuen Akzeptierzustand  $q_a$  hinzugefügt und die ursprünglichen Akzeptierzustände zu gewöhnlichen Zuständen abgewertet.

wird ein neuer Akzeptierzustands  $q_a$  und  $\varepsilon$ -Übergänge von den bereits vorhandenen Akzeptierzuständen zu  $q_a$  hinzugefügt. Die bisherigen Akzeptierzustände werden zu gewöhnlichen Zuständen heruntergestuft.

### Stack leeren

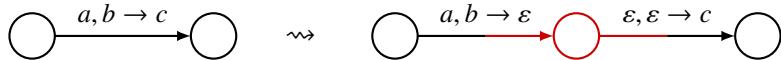
Die Definition der Variablen  $A_{pq}$  verlangt, dass der Stack bei der Ankunft im Zustand  $q$  leer sein muss. Insbesondere muss dies für die Startvariable  $A_{q_0 q_a}$  gelten. Von einem beliebigen Stackautomaten kann man nicht erwarten, dass er den Stack leert, dies ist daher als nächstes sicherzustellen. Der Stackautomat mit Startzustand  $q_0$  und Akzeptierzustand  $q_a$  von Abbildung 7.10 wird mit den grünen Elementen wie folgt zu



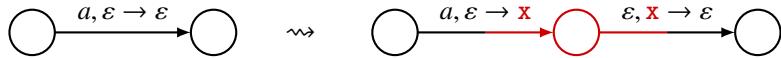
erweitert. Dabei ist das  $x$  im Übergang  $\varepsilon, x \rightarrow \varepsilon$  beim Zustand  $e$  so zu lesen, dass für jedes Zeichen  $x \in \Gamma$  ein solcher Übergang hinzuzufügen ist. Mit dieser Modifikation ist jetzt garantiert, dass der Stack beim Akzeptieren eines Wortes im Zustand  $a$  immer leer ist. Damit kann  $A_{sa}$  als Startvariable verwendet werden.

### Stackoperationen

Die Variable  $A_{pq}$  beschreibt Wörter, die von  $p$  nach  $q$  führen, aber mit leerem Stack. Bei der Ableitung der Grammatikregeln muss auch beobachtet werden, was auf dem Stack passiert. Es vereinfacht die Analyse, wenn es nur wenige mögliche Situationen gibt. Wir verlangen daher, dass bei jedem Übergang entweder ein Zeichen auf den Stack gelegt wird oder eines entfernt wird. Jeder Übergang, der ein Zeichen auf dem Stack auswechselt, muss nach dem Muster

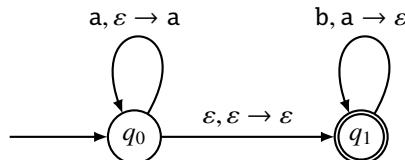


mit einem Hilfszustand in zwei Übergänge zerlegt werden. Übergänge, die den Stack gar nicht beanspruchen, werden ebenfalls zerlegt und es wird ein neues Zeichen  $x$  auf den Stack gelegt und gleich wieder entfernt:



In beiden Fällen kann  $a$  auch das leere Wort sein.

**Verständniskontrolle 7.3:** Standardisieren Sie den folgenden Stackautomaten für die Sprache  $L = \{a^n b^m \mid n \geq m\}$  als Vorbereitung darauf, die Grammatik abzulesen.



### 7.4.3 Ablesen von Regeln

Nach den Anpassungen des Stackautomaten gemäß den Vorgaben von Abschnitt 7.4.2 ist sichergestellt, dass es nur einen Akzeptierzustand gibt, dass der Stack vor dem Akzeptieren immer geleert wird und dass bei jedem Übergang der Stack um ein Zeichen wächst oder schrumpft.

#### Visualisierung des Stacks

Die Veränderungen des Stacks können als die blaue Kurve in Abbildung 7.11 dargestellt werden, die die Höhe des Stacks wiedergibt. Da mit jedem Übergang ein Zeichen auf den Stack kommt oder eines entfernt wird, besteht die Kurve nur aus ansteigenden und abfallenden Segmenten, die alle gleich steil sind. Es gibt keine horizontalen Segmente. Wege mit leerem Stack beginnen und enden auf der Nulllinie, ohne je darunter zu fallen.

#### Wege mit zwischenzeitlich leerem Stack

Wir nehmen an, das Wort  $w$  führe den Stackautomaten vom Zustand  $p$  in den Zustand  $q$  mit leerem Stack. Im Zwischenzustand  $r$  werde der Stack erneut leer, wie in Abbildung 7.12. Das Wort besteht also aus zwei Teilen  $w = uv$ , wobei  $u$  den Automaten vom Zustand  $p$  in den Zustand  $r$  bringt mit leerem Stack, während  $v$  ihn vom Zustand  $r$  in den Zustand  $q$  bringt, ebenfalls mit leerem Stack. Für die Teilwörter  $u$  und  $v$  stehen die Variablen  $A_{pr}$  und  $A_{rq}$ . Somit gilt die Regel

$$A_{pq} \rightarrow A_{pr} A_{rq}. \quad (7.19)$$

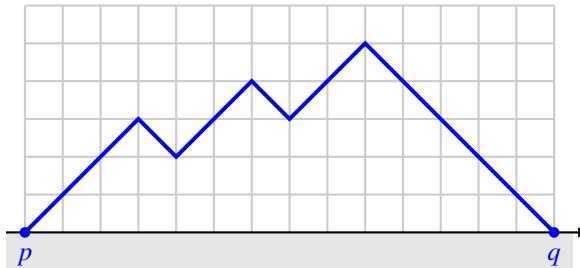


Abbildung 7.11: Veränderung der Höhe des Stacks während der Verarbeitung eines Wortes, welches den Stackautomaten vom Zustand  $p$  zum Zustand  $q$  führt mit leerem Stack.

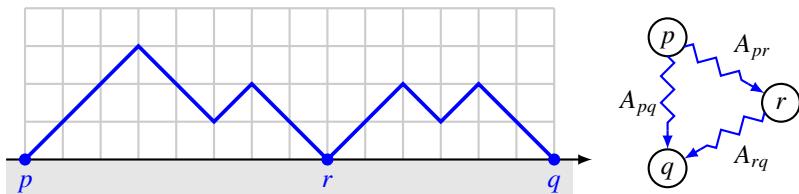


Abbildung 7.12: Regel für einen Pfad durch den Stackautomaten von  $p$  nach  $q$ , auf dem im Zwischenzustand  $r$  der Stack leer wird.

### Wege ohne leeren Stack

Wenn der Stack während der Verarbeitung des Wortes  $w$ , welches den Automaten mit leerem Stack vom Zustand  $p$  zum Zustand  $q$  führt, niemals leer wird, dann bleibt das Zeichen, welches beim ersten Übergang auf den Stack gelegt wird, bis zum Schluss auf dem Stack. Die Situation ist in Abbildung 7.13 dargestellt.

Beim ersten Übergang wird ein Zeichen  $x$  auf den Stack gelegt, welches bis zum Zustand  $s$  auf dem Stack liegen bleibt und erst beim letzten Übergang wieder entfernt wird. Der Teilpfad von  $r$  nach  $s$  führt den Stackautomaten von  $r$  zu  $s$  mit leerem Stack. Dieser Teil des Wortes wird durch die Variable  $A_{rs}$  repräsentiert.

Von den roten Übergängen in Abbildung 7.13 kann man auch ablesen, dass beim aufsteigenden Übergang ein  $a$  im Input verarbeitet wird, während beim absteigenden Über-

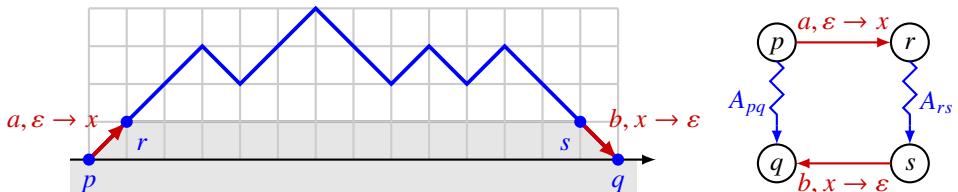


Abbildung 7.13: Regel für einen Pfad durch den Stackautomaten von  $p$  nach  $q$ , auf dem der Stack erst am Ende leer wird.

gang am Schluss ein  $b$  im Input verarbeitet werden muss. Das Wort, welches von  $p$  nach  $q$  führt, beginnt also mit  $a$  und endet mit  $b$ . Dazwischen steht ein Wort, welches durch die Variable  $A_{rs}$  dargestellt wird. Damit haben wir die Regel

$$A_{pq} \rightarrow aA_{rs}b \quad (7.20)$$

gefunden.  $a$  und  $b$  können auch  $\varepsilon$  sein.

### $\varepsilon$ -Regeln

Es ist immer möglich, mit leerem Stack vom Zustand  $p$  zum Zustand  $p$  zu kommen, indem man nichts tut. Die Variable  $A_{pp}$  steht daher mindestens für das leere Wort  $\varepsilon$ . Zur Grammatik müssen daher immer auch die Regeln

$$A_{pp} \rightarrow \varepsilon \quad (7.21)$$

hinzugenommen werden. Die Regel (7.21) sagt, dass die Variable  $A_{pp}$  wenn nötig weggelassen werden kann.

### Die Grammatik eines Stackautomaten

Die Regeln von (7.19), (7.20) und (7.21) ergeben zusammen die Regeln einer Grammatik für die Wörter, die der Stackautomat akzeptiert.

**Satz 7.6.** Ein Stackautomat  $P$ , der wie in Abschnitt 7.4.2 vorbereitet worden ist, akzeptiert eine Sprache, die auch von der Grammatik mit der Startvariablen  $A_{sa}$  und den Regeln

$$R = \left\{ \begin{array}{l} A_{pq} \rightarrow A_{pr}A_{rq} \\ \quad \rightarrow aA_{rs}b \\ A_{pp} \rightarrow \varepsilon \end{array} \middle| \begin{array}{l} p, q, r \text{ und } s \text{ sind Zustände} \\ \text{wie in den Abbildungen} \\ 7.12 \text{ und } 7.13. \end{array} \right\} \quad (7.22)$$

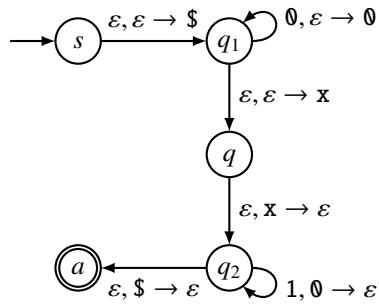
produziert wird.

Wie das nachfolgende Beispiel zeigt, können die Regeln (7.22) sehr unübersichtlich und schwer verständlich sein. Zum Teil ist dies den nicht sehr sprechenden Bezeichnungen der Variablennamen  $A_{pq}$  geschuldet. Durch Umbenennung und Vereinfachung der Regeln kann meistens eine übersichtlichere und leichter verständliche Form der Grammatik gefunden werden.

### 7.4.4 Beispiele

Die nachfolgenden Beispiele sollen den Prozess illustrieren, wie aus einem Stackautomaten eine Grammatik abgeleitet werden kann.

*Beispiel 7.7.* In Abschnitt 7.1.3 wurde das Zustandsdiagramm eines Stackautomaten  $P_1$  für die Sprache  $L_1 = \{0^n 1^n \mid n \geq 0\}$  konstruiert. Der Stackautomat erfüllt noch nicht alle Bedingungen. Nach den Regeln in Abschnitt 7.4.2 muss der  $\varepsilon$ -Übergang zwischen  $q_1$  und  $q_2$  in zwei Schritte aufgeteilt werden, in denen  $x$  auf den Stack gelegt und gleich wieder entfernt wird. So entsteht der Automat



Die Startvariable der zugehörigen Grammatik ist  $A_{sa}$ .

Zwischen den Zuständen  $q_1$  und  $q_2$  wird das Zeichen  $\$$  nicht entfernt. Dies ist die Situation von Abbildung 7.13. Somit muss der Grammatik die Regel

$$A_{sa} \rightarrow \varepsilon A_{q_1 q_2} \varepsilon = A_{q_1 q_2} \quad (7.23)$$

hinzugefügt werden.

Zwischen  $q_1$  und  $q_2$  gibt es mehrere Pfade mit leerem Stack. Zunächst gibt es den direkten Weg, der im ersten Schritt  $q_1 \rightarrow q$  ein  $x$  auf den Stack legt und im Schritt  $q \rightarrow q_2$  wieder entfernt. Die zugehörige Regel ist

$$A_{q_1 q_2} \rightarrow \varepsilon A_{qq} \varepsilon = A_{qq}.$$

Natürlich gibt es dafür noch die Regel  $A_{pp} \rightarrow \varepsilon$ , mit  $p = q$ , die weiter unten als (7.25) hinzugefügt wird. Es gibt aber auch noch den Pfad, auf dem zuerst ein Zeichen  $0$  auf den Stack gelegt wird, welches erst ganz am Schluss wieder entfernt wird. Die zugehörige Regel ist

$$A_{q_1 q_2} \rightarrow 0 A_{q_1 q_2} 1. \quad (7.24)$$

Insgesamt haben wir die Grammatikregeln

$$A_{as} \rightarrow A_{q_1 q_2} \quad (7.23)$$

$$A_{q_1 q_2} \rightarrow 0 A_{q_1 q_2} 1 \quad (7.24)$$

$$\rightarrow \varepsilon \quad (7.25)$$

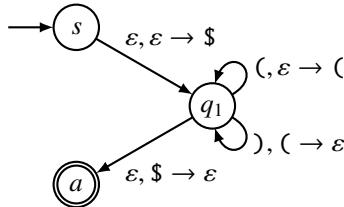
gefunden. Offenbar wird  $A_{as}$  nur in der ersten Regel gebraucht, die Variable  $A_{q_1 q_2}$  ist genauso gut als Startvariable geeignet. Schreibt man  $S = A_{q_1 q_2}$ , wird aus den letzten zwei Regeln die viel einfachere Grammatik

$$\begin{aligned} S &\rightarrow 0 S 1 \\ &\rightarrow \varepsilon, \end{aligned}$$

die wir bereits in Abschnitt 5.1.4 als Grammatik für die Sprache  $L_1$  kennengelernt haben.

○

*Beispiel 7.8.* Der Stackautomat (7.2) für Klammerausdrücke erfüllt bereits die Bedingungen von Abschnitt 7.4.2. Mit den Konventionen dieses Abschnitts geschrieben lautet er



Daraus lassen sich die Regeln

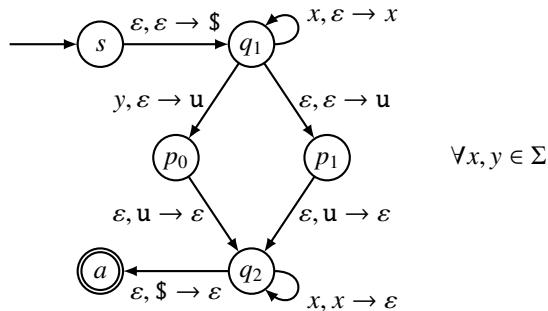
$$\begin{aligned} A_{sa} &\rightarrow \varepsilon A_{q_1 q_1} \varepsilon = A_{q_1 q_1} \\ A_{q_1 q_1} &\rightarrow A_{q_1 q_1} A_{q_1 q_1} \\ &\rightarrow (A_{q_1 q_1}) \\ &\rightarrow \varepsilon \end{aligned}$$

ablesen. Wieder kann man die erste Regel weglassen und die Startvariable als  $K = A_{q_1 q_1}$  schreiben. So erhält man die Grammatik

$$K \rightarrow KK \mid (K) \mid \varepsilon.$$

Dies ist die bekannte, mehrdeutige kontextfreie Grammatik für Klammerausdrücke.  $\circlearrowright$

*Beispiel 7.9.* In Abschnitt 7.1.4 wurde der Stackautomat (7.1) für Palindrome vorgestellt. Die Vorbereitungen von Abschnitt 7.4.2 fügen zwei Zustände  $p_0$  und  $p_1$  hinzu. Der Automat bekommt so die Form



Die daraus abgelesenen Regeln sind

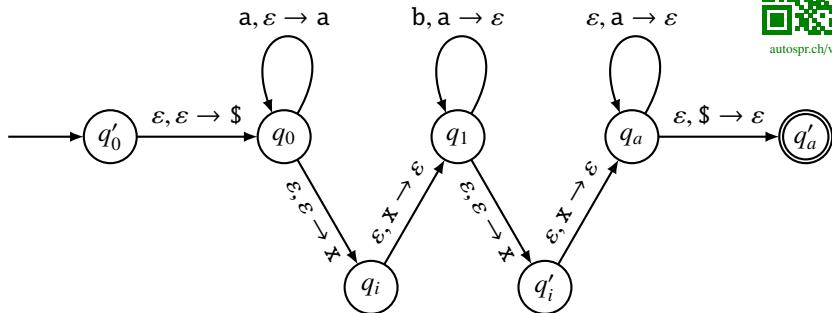
$$\begin{aligned} A_{as} &\rightarrow \varepsilon A_{q_1 q_2} \varepsilon = A_{q_1 q_2} \\ A_{q_1 q_2} &\rightarrow x A_{q_1 q_2} x \\ &\rightarrow y A_{p_0 p_0} \varepsilon = y A_{p_0 p_0} \rightarrow y \\ &\rightarrow \varepsilon A_{p_1 p_1} \varepsilon = A_{p_1 p_1} \rightarrow \varepsilon \end{aligned}$$

Schreibt man  $P = A_{q_1 q_2}$ , vereinfacht sich dies zur bekannten Grammatik

$$\begin{aligned} P &\rightarrow x P x & x \in \Sigma \\ &\rightarrow y & y \in \Sigma \\ &\rightarrow \varepsilon \end{aligned}$$

für Palindrome. Sie stimmt mit (7.5)–(7.6) überein. ○

**Verständniskontrolle 7.4:** Lesen Sie die Grammatik aus dem folgenden Stackautomaten für die Sprache  $L = \{a^n b^m \mid n \geq m\}$  ab:



## Übungsaufgaben

**7.1.** Konstruieren Sie einen Stackautomaten, der die Sprache

$$L = \{a^n b^m c^m d^n \mid m, n \geq 0\}.$$

über dem Alphabet  $\Sigma = \{a, b, c, d\}$  akzeptiert.

**7.2.** Konstruieren Sie einen Stackautomaten, der die durch die Grammatik

$$\begin{aligned} S &\rightarrow \varepsilon \\ &\rightarrow S S \\ &\rightarrow G S U \\ G &\rightarrow 0 \mid 2 \mid 4 \mid 6 \mid 8 \\ U &\rightarrow 1 \mid 3 \mid 5 \mid 7 \mid 9 \end{aligned}$$

definierte Sprache akzeptiert.

*Hinweis.* Versuchen sie zuerst die Sprache zu verstehen, die von dieser Grammatik erzeugt wird und erstellen Sie dann einen “passenden” Stackautomaten.

**7.3.** Konstruieren Sie einen Stackautomaten, der die Sprache

$$L = \{a^i b^j \mid i \neq j\}$$

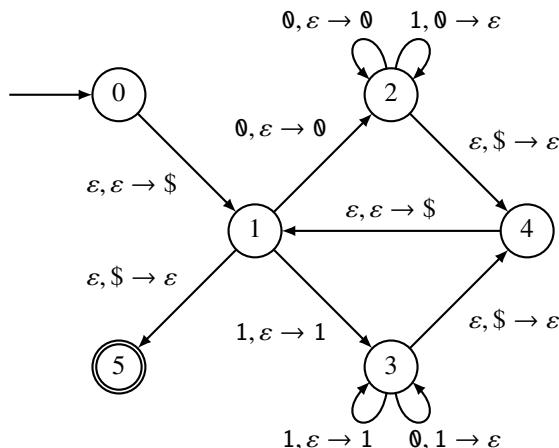
akzeptiert.

**7.4.** Finden Sie einen Stackautomaten, der

$$L = \{\varnothing^i 1^j \varnothing^i \mid j > 0 \wedge i \geq 0\}$$

akzeptiert.

**7.5.** Man finde eine Grammatik für die Sprache  $L = \{w \in \Sigma^* \mid |w|_1 = |w|_\varnothing\}$  über dem Alphabet  $\Sigma = \{\varnothing, 1\}$ , indem man sie aus dem folgenden Stackautomaten ableitet:



*Hinweis.* Die Konstruktion dieses Automaten wird in der Lösung von Verständniskontrolle 7.1 erläutert. Die Idee des Automaten ist, dass der obere Zweig mit dem Zustand 2 verwendet wird, solange mehr  $\varnothing$  als 1 gelesen wurden. Der untere Zweig mit dem Zustand 3 wird dagegen verwendet, wenn mehr 1 als  $\varnothing$  gelesen wurden. Im oberen Zweig werden die Nullen auf den Stack gelegt und der Stack wird mit Einsen wieder abgebaut. Im unteren Zweig werden die Einsen auf den Stack gelegt und der Stack wird mit Nullen wieder abgebaut. Es ist also möglich, dass während der Verarbeitung eines Wortes mehrmals zwischen dem oberen und unteren Teilautomaten hin- und hergewechselt wird. Akzeptabel ist ein Wort, welches den Stack leer lässt. Im Zustand 1 und 4 sind immer gleiche Einsen wie Nullen gelesen worden.

Lösungen: <https://autospr.ch/uebungen/AutoSpr-107.pdf>



# Kapitel 8

# Nicht kontextfreie Sprachen

Mithilfe eines Stacks wurde es möglich, Zeichen wie öffnende Klammern zu zählen und die dazugehörigen schließenden Klammern zu identifizieren. Mit dem Abbau des Stacks geht aber auch die Erinnerung an die Anzahl verloren und steht nicht mehr für einen weiteren Vergleich zur Verfügung. Es kann daher vermutet werden, dass die Sprache  $L = \{a^n b^n c^n \mid n \geq 0\}$ , in der drei verschiedene Zeichen gezählt werden müssen, nicht kontextfrei sein wird. Mit dem in diesem Kapitel beschriebenen Pumping-Lemma für kontextfreie Sprachen kann dies auch streng bewiesen werden.

## 8.1 Pumping-Lemma

Das Pumping-Lemma für reguläre Sprachen formulierte eine “schöne” Eigenschaft, die alle regulären Sprachen haben. Kann man zeigen, dass eine Sprache diese Eigenschaft nicht hat, dann kann sie auch nicht regulär sein. Die Pumpeigenschaft regulärer Sprachen sagt, dass es in langen Wörtern immer Wortteile gibt, die wiederholt in das Wort eingesetzt werden können, ohne dass man damit die Sprache verlässt.

Für kontextfreie Sprachen suchen wir jetzt eine ähnliche Eigenschaft. Kontextfreie Grammatiken können die Schachtelung von Klammern oder Kontrollstrukturen beschreiben. Für genügend komplexe Wörter, die auch tatsächlich solche Klammern enthalten, lassen sich immer zusammengehörige Klammern finden. Sie können beliebig oft paarweise erneut eingesetzt werden, so dass das Wort länger wird, aber in der Sprache bleibt. Eine Pumpeigenschaft darf also auch für kontextfreie Sprachen erwartet werden.

### 8.1.1 Syntaxbäume

Die Produktion eines Wort  $w \in \Sigma^*$  aus der Startvariablen einer Grammatik  $G = (V, \Sigma, R, S)$  kann als Syntaxbaum dargestellt werden. Für eine beliebige Grammatik kann die Breite des Baumes sprunghaft wachsen und dann lange Zeit gleich bleiben. Um dies zu vermei-

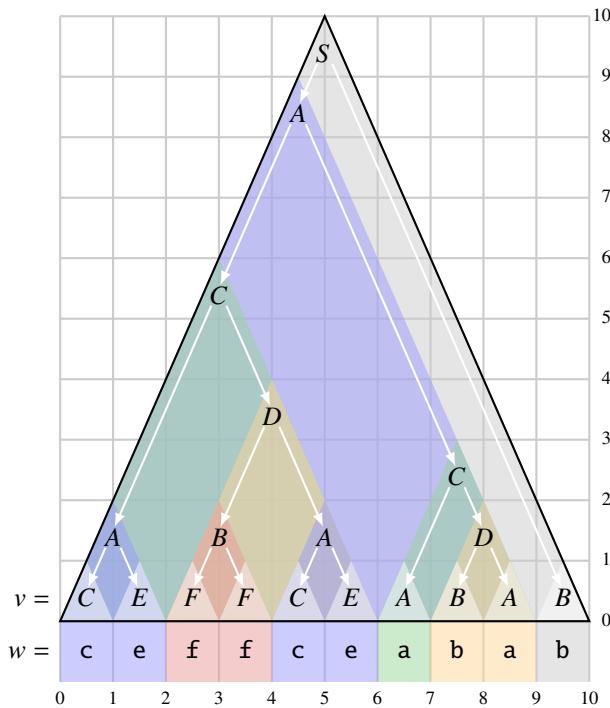


Abbildung 8.1: Syntaxbaum und Syntaxdreieck für die Erzeugung eines Wortes  $w \in L$  aus der Startvariablen einer Grammatik  $G = (V, \Sigma, R, S)$ .

den, verwenden wir wie in Kapitel 6 im Folgenden immer eine Grammatik in Chomsky-Normalform.

### Syntaxdreieck

Der Syntaxbaum eines Wortes  $w$  kann als *Syntaxdreieck* wie in Abbildung 8.1 dargestellt werden. Dank der Chomsky-Normalform lässt sich der Syntaxbaum immer in zwei Teile aufteilen. Im oberen Teil werden die Regeln der Form  $A \rightarrow BC$  angewendet. So entsteht eine Kette von Variablen  $v$  in der untersten Zeile des Dreiecks. Erst am Ende werden die Terminalsymbolregeln der Form  $A \rightarrow a$  angewendet und die Kette  $v$  von Variablen in eine Zeichenkette  $w$  von Terminalsymbolen umgewandelt. Im Inneren des Dreiecks stehen die Zweierregeln, an der Basis die Terminalsymbolregeln. Eine ähnliche Struktur ist auch in der Tabellendarstellung des Cocke-Younger-Kasami-Algorithmus in Abschnitt 6.3 verwendet worden.

Abbildung 8.1 zeigt auch, wie von jeder Variablen in der Ableitung des Wortes  $w$  ein eigenes Syntaxdreieck ausgeht, an dessen Spitze die Variable steht, die das Teilwort an der Basis erzeugt. Die Syntaxdreiecke sind immer so geschachtelt, dass eine der Seiten auf die entsprechende Seite des übergeordneten Dreiecks fällt.

## Wortlänge

Abbildung 8.1 zeigt auch, dass das Syntaxdreieck immer so hoch ist wie das erzeigte Wort lang ist. Da im Dreieck immer Zweierregeln angewendet werden, enthält jedes Syntaxdreieck zwei geschachtelte Syntaxdreiecke, die die Seiten des großen Dreiecks berühren und deren Ecken an der Basis zusammentreffen. Steht an der Spitze des großen Dreiecks die Variable  $A$  und an den Spitzen der geschachtelten Dreiecke die Variablen  $B$  und  $C$ , dann stellt das große Dreieck die Anwendung der Regel  $A \rightarrow BC$  dar<sup>1</sup>.

Die Summe der Höhen der geschachtelten Syntaxdreiecke ist die Höhe des gesamten Dreiecks. Daher hat eines der geschachtelten Dreiecke ein Höhe  $\leq |w|/2$ , während das andere geschachtelte Dreieck eine Höhe  $\geq |w|/2$  hat.

Ein spezieller Fall ist ein Wort der Länge  $2^n$  in  $L$ , in dessen Syntaxdreiecken immer jeweils zwei genau gleich große Syntaxdreiecke geschachtelt sind. In diesem Fall gibt es  $2^n$  Syntaxdreiecke der Höhe 1,  $2^{n-1}$  Syntaxdreiecke der Höhe 2,  $2^{n-2}$  Syntaxdreieck der Höhe 4, usw. und ein Syntaxdreieck der Höhe  $2^n$ . Im ganzen Syntaxdreieck stehen also

$$2^n + 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = \sum_{k=0}^n 2^k = 2^{n+1} - 1$$

Variablen.

## Unterdreiecke

Da im Inneren eines Syntaxdreiecks immer nur Regeln der Form  $A \rightarrow BC$  angewendet werden, enthält jedes Syntaxdreieck der Höhe  $h$  zwei unmittelbare Unterdreiecke mit den Höhen  $h'$  und  $h''$  mit  $h = h' + h''$ . Wenn eines der Unterdreiecke sehr klein ist, ist das andere umso größer. Es lassen sich also immer geschachtelte Unterdreiecke finden, die nicht zu schnell klein werden. Genauer drückt dies der folgende Satz aus. Der Sachverhalt ist auch in Abbildung 8.2 dargestellt.

**Satz 8.1** (Existenz von großen Unterdreiecken). *Erfüllt die Höhe  $h$  des Syntaxdreiecks die Ungleichung  $2^n < h \leq 2^{n+1}$ , dann enthält das Dreieck ein Unterdreieck mit Höhe  $h'$ , die  $2^{n-1} < h' \leq 2^n$  erfüllt.*

*Beweisidee.* Abbildung 8.2 illustriert die Idee des Beweises. Jedes Syntaxdreieck enthält zwei unmittelbare Unterdreiecke, die aber die Bedingungen des Satzes noch nicht erfüllen müssen. Das Dreieck mit Spitze 1 hat die Unterdreiecke mit Spitzen 2 und 6, beide erfüllen die Bedingungen nicht.

Das größere Dreieck ist zu hoch, man kann darin wieder zwei Unterdreiecke finden, die man auf die gleiche Weise untersucht. So kann man eine Folge von kleiner werdenden Unterdreiecken finden, bis man schließlich ein Unterdreieck findet, dessen Unterdreiecke beide die Höhen  $\leq 2^n$  haben. In Abbildung 8.2 ist dies das Unterdreieck mit Spitze 3.

---

<sup>1</sup>In der Diskussion des Cocke-Younger-Kasami-Algorithmus in Abschnitt 6.3 wurden die Felder, die die Variablen  $B$  und  $C$  in der Tabelle enthielten, korrespondierende Felder genannt. Sie waren durch die gleiche Bedingung bestimmt, dass nämlich die von diesen Variablen ausgehenden Syntaxbäume das ganze Wort an der Basis erzeugen müssen.

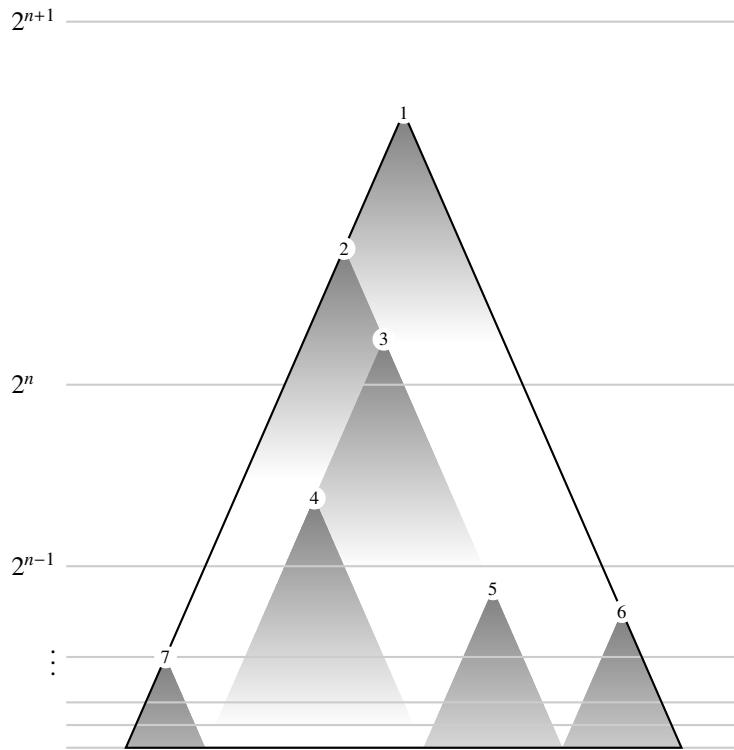


Abbildung 8.2: Jedes Syntaxdreieck mit Höhe zwischen  $2^n$  und  $2^{n+1}$ , im Bild die Dreiecke mit Spitzen 1, 2 und 3, enthält ein Unterdreieck mit Höhe zwischen  $2^{n-1}$  und  $2^n$ , im Bild das Dreieck mit Spitze 4.

Die Unterdreiecke mit Spitzen 4 und 5 können nicht beide eine zu geringe Höhe  $\leq 2^{n-1}$  haben, da sie sonst zusammen nicht die Höhe der Spitze 3 erreichen, die höher als  $2^n$  ist.  $\square$

*Beweis.* Seien  $h'$  und  $h''$  die Höhen der beiden unmittelbaren Unterdreiecke. Falls  $h' > 2^n$  oder  $h'' > 2^n$  ist, ist die Höhe des jeweils anderen Unterdreiecks zwar höchstens  $2^n$  wie verlangt, aber man kann nicht schließen, dass sie auch  $> 2^{n-1}$  ist. Wir arbeiten daher weiter mit dem Unterdreieck, dessen Höhe größer ist als  $2^n$ . Dieses Unterdreieck hat wieder zwei Unterdreiecke, mit denen wir gleich verfahren: Solange die Höhe größer als  $2^n$  ist, arbeiten wir mit dem größeren Unterdreieck weiter.

Auf diese Weise erreichen wir ein Unterdreieck mit Höhe  $> 2^n$ , welches nur Unterdreiecke mit Höhen  $h'$  und  $h''$  hat, die  $\leq 2^n$  sind. Es ist natürlich immer noch  $h' + h'' \geq 2^n$ . Beide Unterdreiecke kommen also für das gesuchte Unterdreieck in Frage, sofern ihre Höhen  $> 2^{n-1}$  sind. Es kann aber nicht sein, dass beide Höhen  $\leq 2^{n-1}$  sind, denn dann wäre

$$h' + h'' \leq 2^{n-1} + 2^{n-1} = 2^n$$

im Widerspruch zu  $h' + h'' > 2^n$ . Es folgt daher, dass eines der Unterdreiecke eine Höhe  $> 2^{n-1}$  hat.  $\square$

### Pfade in einem Syntaxdreieck

Ein Pfad in einem Syntaxdreieck ist eine Folge von Pfeilen des Syntaxbaumes, der von den Variablen an der Spitze des Dreiecks zu einer Variablen an der Basis führt. Der erste Schritt führt von den Variablen an der Spitze zu einer der Variablen an den Spitzen der beiden unmittelbaren Unterdreiecke.

Ein Pfad in einem Syntaxdreieck der Höhe  $|w|$  kann nicht mehr als  $|w|$  Variablen enthalten. Im kürzesten Fall ist ein Pfad nur ein Segment lang, dies ist der Fall  $S \rightarrow AB$  mit dem Segment  $S \rightarrow B$  am rechten Rand des Syntaxdreiecks in Abbildung 8.1. Es gibt aber immer auch längere Pfade, wie der folgende Satz garantiert.

**Satz 8.2** (Existenz langer Pfade). *Im Syntaxdreieck des Wortes  $w$  gibt es immer einen Pfad der Länge  $\geq \log_2 |w|$ .*

*Beweis.* Sei  $n \in \mathbb{N}$  mit  $2^{n+1} \geq |w| > 2^n$ , d. h.  $n+1 \geq \log_2 |w| > n$ . Nach Satz 8.1 gibt es eine Folge von Unterdreiecken, deren Höhe  $h$  jeweils  $2^k \geq h > 2^{k-1}$  erfüllt mit  $k = n, \dots, 1$ . Insbesondere gibt es eine Folge von  $n$  geschachtelten Unterdreiecken. Sie bilden einen Pfad der gewünschten Art.  $\square$

### Pfade mit wiederholten Variablen

In jedem Syntaxdreieck eines Wortes der Länge  $|w|$  gibt es einen Pfad, der mindestens  $\log_2 |w|$  Schritte umfasst. Auf so einem Pfad stehen also  $1 + \log_2 |w|$  Variablen. Da die Grammatik nur  $|V|$  verschiedene Variablen hat, muss auf so einem Pfad eine Variable mindestens zweimal vorkommen, wenn  $\log_2 |w| \geq |V|$  oder  $|w| \geq 2^{|V|}$  ist. Sobald also ein Wort mindestens die Längen  $2^{|V|}$  hat, kommt eine Variable auf dem Pfad mehr als einmal vor.

**Satz 8.3** (Unterdreiecke mit wiederholten Variablen). *Sei  $G = (V, \Sigma, R, S)$  eine kontextfreie Grammatik in Chomsky-Normalform und  $N = 2^{|V|+1}$ . Ist  $w \in L(G)$  ein von der Grammatik erzeugtes Wort mit Länge  $|w| \geq N$ , dann gibt es ein Unterdreieck der Höhe  $\leq N$  mit einem Pfad, auf dem eine Variable zweimal vorkommt.*

*Beweis.* Nach Satz 8.1 gibt es im Syntaxdreieck des Wortes immer ein Unterdreieck der Höhe  $h$  mit

$$\frac{N}{2} = 2^{|V|} < h \leq 2^{|V|+1} = N.$$

Nach Satz 8.2 gibt es in einem solchen Unterdreieck immer einen Pfad der Länge  $\geq |V|$ . Ein Pfad der Länge  $l$  enthält  $l+1$  Variablen, also muss der Pfad mindestens  $|V|+1$  Variablen enthalten, was nur möglich ist, wenn eine Variable zweimal auftritt.  $\square$

## 8.1.2 Scherenschnitt in Syntaxbäumen

In einem Syntaxdreieck mit Höhe mindestens  $N$  kann man also immer einen Pfad finden, der wiederholte Variablen enthält und außerdem in einem Unterdreieck der Höhe  $\leq N$

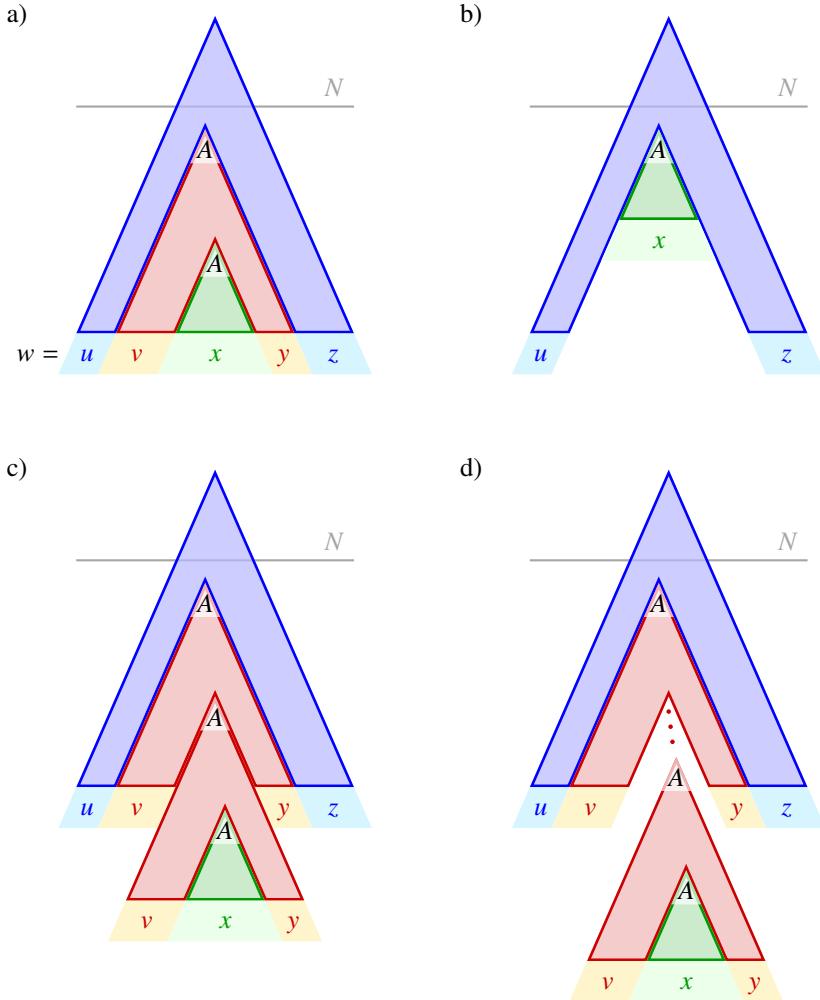


Abbildung 8.3: Scherenschnitt in einem Syntaxdreieck: Das Syntaxdreieck a) erzeugt das Wort  $w$  und enthält zwei geschachtelte Unterdreiecke mit der gleichen Variable  $A$  an der Spitze. Ersetzt man das **rote** Dreieck durch das **grüne**, entsteht ein Syntaxbaum, der nur noch das Wort  $uxz$  erzeugt. Das rote Syntaxdreieck kann auch mehrfach eingesetzt werden, wie in der unteren Zeile. So können Syntaxbäume für alle Wörter der Form  $uv^kxy^kz$  mit  $k \in \mathbb{N}$  erzeugt werden.

enthalten ist. Die wiederholten Variablen erzeugen zwei geschachtelte Unterdreiecke wie in Abbildung 8.3 dargestellt, wo die wiederholten Variablen mit A bezeichnet sind. Die Unterdreiecke zerlegen das erzeugte Wort in fünf Teile  $w = uvxyz$ . Die Tatsache, dass die gleichen Variablen auf verschiedener Höhe liegen, bedeutet, dass  $|vy| > 0$  ist. Da das **rote** Dreieck Höhe  $\leq N$  hat, ist  $|vxy| \leq N$ .

Die Variable an der Spitze des **roten** und des **grünen** Dreiecks sind gleich. Daher kann das **rote** Dreieck ausgeschnitten und durch das kleine **grüne** Dreieck ersetzt werden (Abbildung 8.3 b)). Die vom **roten** Dreieck erzeugten Teile des Wortes fallen daher weg und es bleibt nur noch  $uxz$ .

Man kann aber anstelle des kleinen **grünen** Dreiecks auch eine Kopie des **roten** Dreiecks mit eingesetztem kleinem **grünem** Dreieck einsetzen. Dadurch werden die **roten** Teile zweimal erzeugt, man erhält  $uv^2xy^2z$  (Abbildung 8.3 c)). Iteriert man diese Konstruktion, entstehen nacheinander alle Wörter der Form  $uv^kxy^kz$  mit  $k \in \mathbb{N}$  (Abbildung 8.3 d)). Alle diese Wörter haben einen Syntaxbaum, sie werden also aus der gleichen Grammatik erzeugt und gehören damit alle zur Sprache  $L(G)$ . Damit ist der folgende Satz gezeigt.

**Satz 8.4** (Pumpeigenschaft für kontextfreie Grammatiken). *Sei  $G$  eine kontextfreie Grammatik, welches das Wort  $w$  produziert. Kommt auf einem Pfad im Syntaxdreieck des Wortes  $w$  eine Variable mehr als einmal vor, dann lässt sich das Wort  $w$  in fünf Teile  $w = uvwyz$  zerlegen mit  $|vy| > 0$ , derart, dass die Wörter  $uv^kxy^kz \in L$  für alle  $k \in \mathbb{N}$  sind.*

### 8.1.3 Pumpeigenschaft von kontextfreien Sprachen

Mit den oben zusammengetragenen Eigenschaften von Syntaxdreiecken lässt sich jetzt das folgende Pumping-Lemma für kontextfreie Sprachen beweisen.

**Satz 8.5** (Pumping-Lemma für kontextfreie Sprachen). *Ist  $L$  eine kontextfreie Sprache, dann gibt es eine natürliche Zahl  $N \in \mathbb{N}$ , die Pumping-Length, mit der folgenden Eigenschaft. Ist  $w \in L$  mit  $|w| \geq N$ , dann gibt es eine Aufteilung  $w = uvxyz$  mit*

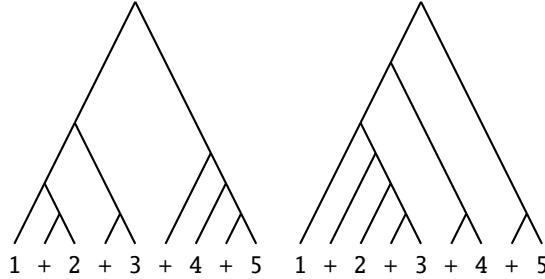
1.  $|vxy| \leq N$
2.  $|vy| > 0$
3. Für alle  $k \in \mathbb{N}$  ist  $uv^kxy^kz \in L$ .

*Beweis.* Da  $L$  kontextfrei ist, gibt es eine Grammatik  $G = (V, \Sigma, R, S)$ , die  $L = L(G)$  erzeugt. Wir dürfen zudem annehmen, dass die Grammatik Chomsky-Normalform hat. Nach Satz 8.3 setzen wir  $N = 2^{|V|+1}$ . Sei  $w \in L$  mit  $|w| \geq N$ . Nach Satz 8.3 gibt es im Syntaxdreieck von  $w$  ein Unterdreieck der Höhe  $\leq N$  mit einem Pfad, auf dem eine Variable  $A$  zweimal vorkommt. Das Unterdreieck erzeugt ein Wort  $w'$  der Länge  $|w'| \leq N$ , welches als Teilwort in  $w = u_0w'z_0$  enthalten ist. Nach Satz 8.4 zerfällt das Wort  $w'$  in fünf Teile,  $w' = u'vxyz'$  wobei die Teile  $v$ ,  $x$  und  $y$  die Eigenschaften 1. und 2. haben. Die Wörter  $u'v^kxy^kz'$  können für alle  $k \in \mathbb{N}$  von der Variable an der Spitze des Unterdreiecks abgeleitet werden. Mit  $u = u_0u'$  und  $u = u_0u'$  folgt, dass die Wörter  $uv^kxy^kz \in L$  sind für alle  $k \in \mathbb{N}$ .  $\square$

**Verständniskontrolle 8.1:** Das Wort  $1+2+3+4+5$  hat zwei mögliche Parse-Trees mit der untenstehenden Grammatik. Ergänzen Sie die Variablen und finden Sie Unterteilungen des Wortes gemäß dem Beweis des Pumping-Lemmas.



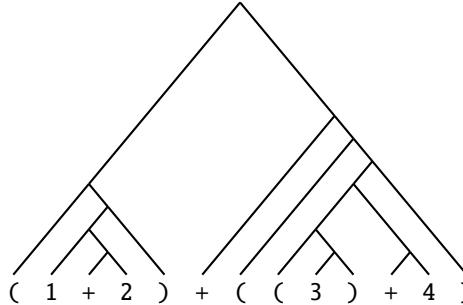
$$\begin{aligned} A &\rightarrow AB \\ &\rightarrow \emptyset \mid 1 \mid \dots \mid 9 \\ B &\rightarrow PA \\ P &\rightarrow + \end{aligned}$$



**Verständniskontrolle 8.2:** Das Wort  $(1+2)-((3)+4)$  kann mit untenstehendem Parse-Tree aus der Grammatik erzeugt werden. Ergänzen Sie die Variablen und finden Sie die Unterteilungen des Wortes gemäß dem Beweis des Pumping-Lemmas, mit der sich Klammern aufpumpen lassen.



$$\begin{aligned} A &\rightarrow AB \mid OC \\ &\rightarrow \emptyset \mid 1 \mid \dots \mid 9 \\ B &\rightarrow ZA \\ C &\rightarrow AS \\ Z &\rightarrow + \mid - \\ O &\rightarrow ( \\ S &\rightarrow ) \end{aligned}$$



## 8.2 Die Sprache $a^n b^n c^n$

Als Beispiel für die Anwendung des Pumping-Lemmas 8.5 für kontextfreie Sprachen zeigen wir, dass die Sprache aus Wörtern der Form  $a^n b^n c^n$  nicht kontextfrei sein kann.

**Satz 8.6.** *Die Sprache*

$$L = \{a^n b^n c^n \mid n \geq 0\}$$

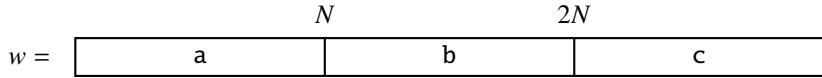
*ist nicht kontextfrei.*

*Beweis.* Wir führen wie üblich bei Anwendungen des Pumping-Lemmas einen Beweis mit Widerspruch durch.

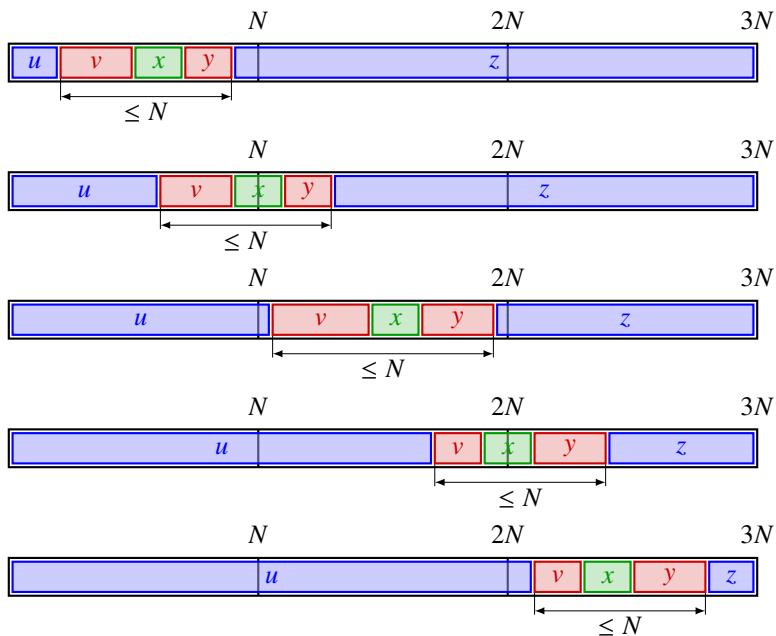
1. Wir nehmen an, dass  $L$  kontextfrei ist.

2. Nach dem Pumping-Lemma gibt es eine Zahl  $N$ , die Pumping-Length, so dass Wörter mit mindestens dieser Länge die Pumpeneigenschaft für kontextfreie Sprachen haben.

3. Wir wählen das Wort  $w = a^N b^N c^N \in L$ :



4. Es gibt viele mögliche Aufteilungen des Wortes  $w$ , die mit den Bedingungen vereinbar sind, die das Pumping-Lemma formuliert. Zum Beispiel:



Einige dieser Aufteilungen erstrecken sich über mehr als einen Block von  $N$  gleichen Zeichen. Die Länge der mittleren drei Teile ist  $|vxy| \leq N$ .

5. Allen Aufteilungen ist gemeinsam, dass die aufpumpbaren Teile  $v$  und  $y$  sich über höchstens zwei der Blöcke gleicher Zeichen erstrecken können. Es können also nur zwei Arten von Zeichen in  $v$  und  $y$  vertreten sein. Beim Pumpen ändert die Anzahl dieser Zeichen, nicht aber die Anzahl der dritten Art von Zeichen. Das gepumpte Wort  $uv^k xy^k z$ , kann nicht mehr in  $L$  sein.
6. Dieser Widerspruch zur Aussage des Pumping-Lemmas zeigt, dass die Annahme von Schritt 1 nicht zutreffen kann. Die Sprache  $L$  kann daher nicht kontextfrei sein.

□

**Verständniskontrolle 8.3:** Verwenden Sie das Pumping-Lemma für kontextfreie Sprachen, um zu zeigen, dass die Sprache  $L = \{(c^n + c^m)^n \mid m < n\}$  nicht kontextfrei ist. Die Wörter bestehen aus  $n$ -fach geschachtelten Klammern, die zuinnerst eine kleinere Anzahl  $m$  von Pluszeichen enthalten.



**Hinweis.** Beachten Sie, dass es mindestens zwei Arten von Unterteilungen gibt, die beim Pumpen auf unterschiedliche Art zu einem Widerspruch führen.

## Übungsaufgaben

**8.1.** In Abschnitt 8.2 wurde gezeigt, dass die Sprache  $\{a^n b^n c^n \mid n \geq 0\}$  über dem Alphabet  $\Sigma = \{a, b, c\}$  nicht kontextfrei ist. Wenn man die Bedingungen etwas lockert, und nur noch verlangt, dass die Anzahl der verschiedenen Zeichen übereinstimmt, die Reihenfolge aber beliebig sein darf, erhält man die Sprache

$$L = \{w \in \Sigma^* \mid |w|_a = |w|_b = |w|_c\}.$$

Ist  $L$  kontextfrei?

**8.2.** Will man eine Matrix wie

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

in einem Computerprogramm verwenden, braucht man irgendeine Syntax, mit der man Matrizen eingeben kann. Octave und Matlab verwenden zum Beispiel:

```
A = [
    1, 2, 3;
    4, 5, 6;
    7, 8, 9;
```

Zeilen werden durch Strichpunkt getrennt, Element in einer Zeile durch Komma. Jede Zeile muss gleich viele Elemente enthalten. Können Sie eine Grammatik formulieren, welche genau korrekte Matrix-Definitionen produziert?

**8.3.** In einer HTML-Tabelle sollte man in jedem `<tr>`-Element gleich viele `<td>`-Elemente angeben (sofern man nicht `<td>`-Elemente verwendet, die sich über mehrere Spalten oder Zeilen erstrecken). Es stellt sich die Frage, ob man dies durch eine kontextfreie Grammatik sicherstellen könnte. Betrachten Sie dazu die folgende vereinfachte Sprache  $L$  über dem Alphabet  $\Sigma = \{r, d\}$ . Die Buchstaben sollen dabei für `<tr>`-Elemente und `<td>`-Elemente stehen. Die Wörter von  $L$  sollen die Form

$$(rd^n)^k$$

haben. Damit ist gemeint, dass jedes Wort aus  $k$  Teilstücken besteht, die alle identisch sind und jeweils aus einem  $r$  und  $n$  Zeichen  $d$  bestehen. Die Wörter

$rrr, \quad rddrddrdd, \quad rdddddddrddddddd, \quad rddddddddd$

gehören also zu  $L$ , die Wörter

$rdr, \quad rdrddrdd, \quad rdddddddr, \quad dddddddddd$

aber nicht. Gibt es für diese Sprache eine kontextfreie Grammatik?

#### 8.4. Der Sprachforscher Stuart M. Shieber hat aus Beispielen wie dem Satz

De Jan säit das mer d'Chind em Hans es Hus händ wele laa hälfe aastriche.

abgeleitet, dass das Schweizerdeutsch grammatische Konstruktionen zulässt, die auf Wörter der Form

$$wa^m b^n xc^m d^n y \quad (8.1)$$

hinaus laufen [54]. Zeigen Sie, dass die Sprache aus Wörtern der Form (8.1) nicht kontextfrei ist.

#### 8.5. In Abschnitt 5.1.5 wurde gezeigt, dass die Sprache

$$L = \{1^n + 1^m = 1^{m+n} \mid n, m \geq 0\}$$

kontextfrei ist. In einer Bemerkung wurde darauf hingewiesen, dass die Sprache

$$L = \{1^n * 1^m = 1^{m+n} \mid n, m \geq 0\}$$

nicht kontextfrei sei. Beweisen Sie dies.

Lösungen: <https://autospr.ch/uebungen/AutoSpr-108.pdf>



# Kapitel 9

## Abzählbar und überabzählbar unendlich

Die Menge  $\Sigma^*$  aller Wörter ist offenbar unendlich, ebenso die Menge aller Sprachen. Zwar sind endliche Automaten oder Grammatiken endliche Objekte, aber die akzeptierten bzw. erzeugten Sprachen können unendlich viele Wörter enthalten. Es gibt wohl unendlich viele Grammatiken, doch ist diese Menge mit der Menge aller Sprachen vergleichbar? In diesem Kapitel wird gezeigt, welchen Unterschied es zwischen abzählbar und überabzählbar unendlich großen Mengen gibt und was für Konsequenzen dies für die theoretische Informatik hat.

### 9.1 Eine unendliche Geschichte

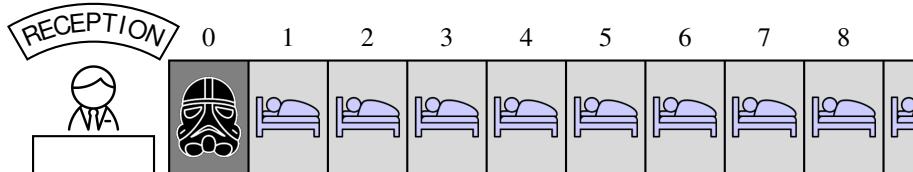
Vor unendlich langer Zeit in einer Galaxie unendlich weit weg von hier spielte sich die folgende Geschichte ab ...

Ein einsamer Reisender landet mit seinem Raumschiff auf dem berühmten Wüstenplaneten mit der Doppelsonne<sup>1</sup>. Um sich von den Strapazen seiner Reise zu erholen, möchte er sich ein paar Tage Ruhe gönnen. Das berühmte “unendliche Hotel” ist ihm als Unterkunft besonders empfohlen worden, weil dort jeder Gast willkommen sei und für jeden Neuankömmling immer ein Zimmer bereit sei. Der Reisende betritt die Lobby und steuert auf den Empfang zu, wo der Hotelmanager, ein uralt aussehender, kleiner Mann gerade mit dem Sortieren von Dokumenten beschäftigt ist. Der Fremde spricht ihn an und fragt: “In der ganzen Galaxie ist euer Hotel für seine Gastfreundschaft bekannt, darf ich für meinen Aufenthalt auf Unterkunft bei euch zählen?”. Der Hotelmanager blickt von seinen Papieren auf, betrachtet den neuen Gast lange und sagt dann: “Das unendliche Hotel wir-

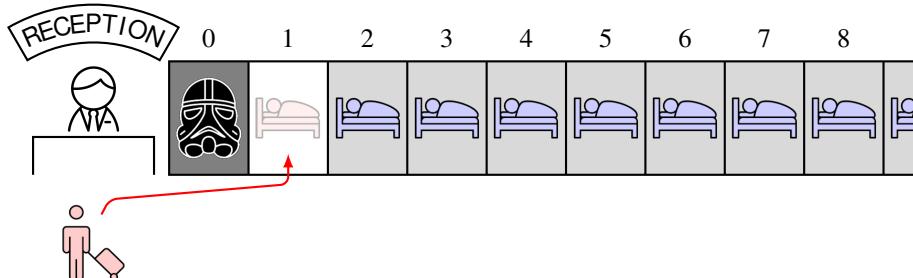
---

<sup>1</sup>Wegen Bedenken der Rechtsabteilung von Lucasfilm Ltd darf der genaue Ort der Begebenheit hier leider nicht verraten werden, mehr dazu in [42].

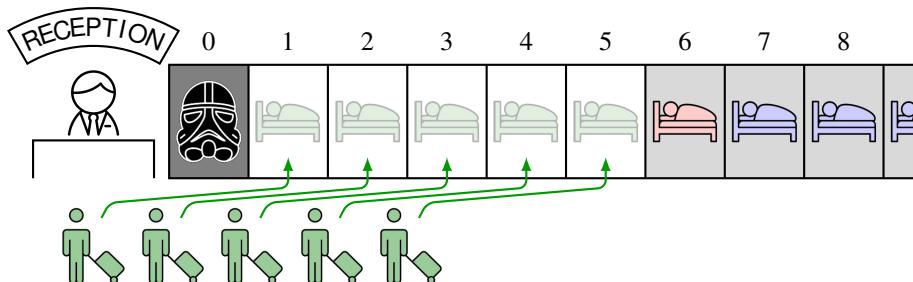
a) Das Hotel ist immer voll:



b) Unterbringung eines neuen Gastes:



c) Unterbringung endlich vieler neuer Gäste:



d) Unterbringung abzählbar unendlich vieler neuer Gäste:

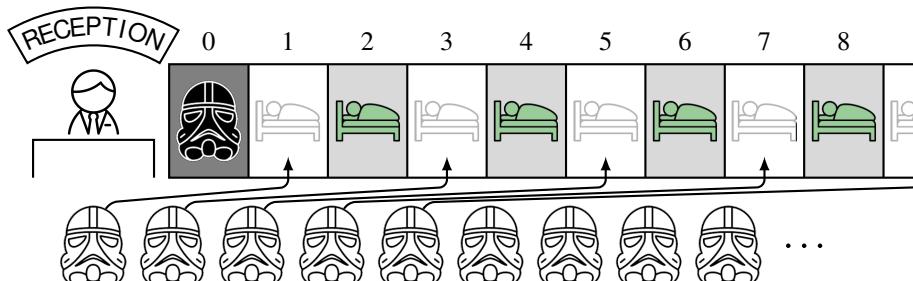


Abbildung 9.1: a) Das unendliche Hotel auf dem Wüstenplaneten mit der Doppelsonne ist immer voll, hat aber immer Platz für neu ankommende Gäste: b) Ein neuer Guest, c) endlich viele neue Gäste, d) abzählbar unendlich viele neue Gäste.

sind, immer ausgebucht wir sind, aber immer Platz wir haben". Tatsächlich zeigt die beeindruckende, große Tafel an der Rückwand des Empfangs alle unendlich vielen Zimmer mit ihrer natürlichen Zimmernummer. Alle Zimmer scheinen belegt zu sein. Der Neuankömmling ist sich nicht sicher, ob er die eigentümliche Sprache des Hotelmanagers richtig verstanden hat und will sich schon zum Gehen wenden. Da wird er Zeuge einer eindrücklichen Transformation. Der Hotelmanager hebt seinen Arm, zeigt damit in den langen Gang mit den unendlich vielen Zimmertüren. Ein leichtes Rumpeln ist zu hören und die Anzeigetafel verändert sich. Das muss "die Macht" gewesen sein, denkt sich der neue Gast, von den magischen Kräften des Managers hatte er auch schon wilde Geschichten gehört, die er für Produkte einer überaktiven Phantasie hielt. Auf der Anzeigetafel haben sich alle Gäste außer dem Gast im Zimmer 0 in ein Zimmer mit einer um eins höheren Nummer verschoben. Die Tür zu Zimmer 1 öffnet sich und auf der Anzeigetafel erscheint der Name des neuen Gastes unter der Nummer 1. Der Manager weist ihm den Weg in sein Zimmer und schärft ihm noch ein, den Gast im Zimmer 0 nicht zu stören, da dieser sich immer schnell aufrege, was man an seinem keuchenden Atem erkennen könne.

Kurze Zeit später trifft eine kleine Gruppe von fünf Raumreisenden ein und verlangt ebenfalls Unterkunft. Dieselbe Prozedur spielt sich erneut ab. Der Hotelmanager hebt den Arm, auf der Anzeigetafel verschieben sich alle Gäste außer dem Gast im Zimmer 0 in ein Zimmer mit einer um 5 größeren Nummer, die Namen der neuen Gäste erscheinen unter den Nummern 1–5 und die Türen der Zimmer 1–5 gehen auf. Der Hotelmanager wünscht seiner Kundschaft einen angenehmen Aufenthalt.

Als wichtiger Knotenpunkt in der Galaxie kehrt im unendlichen Hotel auf dem Wüstenplaneten selten Ruhe ein. Eine unendlich große Reisegesellschaft dringt plappernd in die Hotelhalle ein. Der Reiseleiter bestürmt den Manager mit seiner Reservationsbestätigung für unendlich viele Zimmer. Der Hotelmanager ist nicht aus der Ruhe zu bringen, zeigt wieder mit seinem Arm den langen Gang hinunter. Die bereits vorhandenen Namen verschieben sich zur jeweils doppelt so großen Zimmernummer und alle ungeraden Zimmertüren öffnen sich. Der Gast im Zimmer 0 ist nicht gestört worden. Die Reisegesellschaft zieht schnatternd in die frei gewordenen Zimmer ein.

Seit einiger Zeit herrscht Krieg zwischen der Rebellenallianz und der Armee des Imperators. Mit seiner neuen, unendlich großen Armee hofft er, die Rebellen endlich besiegen zu können, trotz der mangelnden Treffsicherheit seiner Stormtrooper, die deswegen schon zum Gespött der Galaxie geworden sind. Die neue Armee wird auf unendlich vielen, mit natürlichen Zahlen nummerierten Raumschiffen transportiert. Jedes Schiff fasst eine Division mit jeweils unendlich vielen Troopern, die ebenfalls eine natürliche Zahl als Erkennungsnummer haben. Auf ihrem Feldzug kommt die Armee auch auf unserem Planeten vorbei. Der imperiale Quartiermeister verlangt ultimativ für alle Soldaten Unterkunft im Hotel. Der Hotelmanager stellt mit der inzwischen bekannten Methode die ungeraden Zimmer bereit und fordert den Quartiermeister auf leicht passiv aggressive Art heraus, selbst eine Zuordnung der Soldaten zu den Zimmern zu finden. Der Quartiermeister lässt dazu die einzelnen Divisionen nebeneinander jeweils in Einerkolonne auf dem unendlich großen Vorplatz des unendlichen Hotels antreten (Abbildung 9.2). Dann teilt er den Soldaten die Zimmer zu, indem er in Schlangenlinie durch die Reihen der Kämpfer schreitet. Jeder bekommt ein Zimmer zugewiesen und nach kurzer Zeit ist der Vorplatz des Hotels leer und alle Zimmer des Hotels sind wieder belegt.

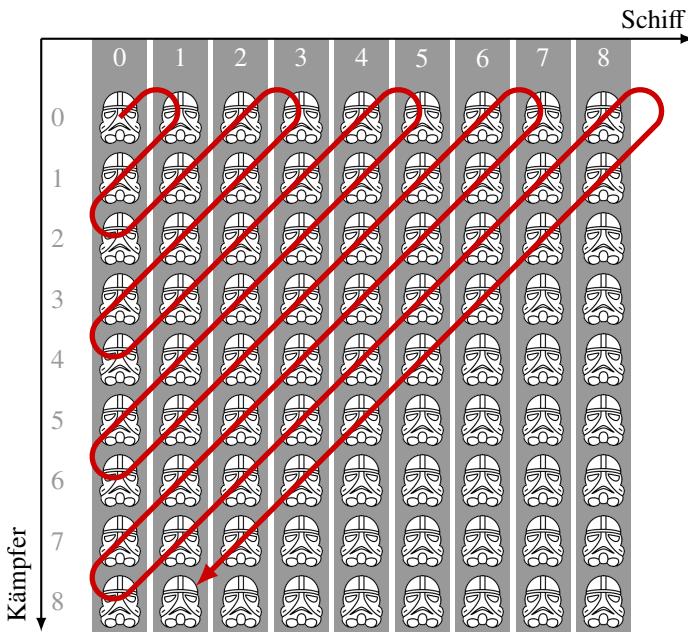


Abbildung 9.2: Exzerzierübung der Truppen des Imperators zur Bestimmung der Zimmernummer für jeden einzelnen Kämpfer. Die Truppen stehen in Einerkolonne pro Schiff ein und werden dann in der Reihenfolge der roten Schlangenlinie in die Zimmer geschickt.

Leider hat die Geschichte kein gutes Ende. Aus einer anderen Galaxie tritt plötzlich Q vom Q Kontinuum auf den Plan. Das Kontinuum zeichnet sich dadurch aus, dass die Individuen des Kontinuums nicht mit natürlichen, sondern mit reellen Zahlen nummeriert sind. An dieser Herausforderung scheitert der sonst nie um eine Lösung verlegene Hotelmanager. Er wird zum Aussteiger und beginnt ein Einsiedlerleben, das er nur selten unterbricht, um junge Raumreisende in der Anwendung “der Macht” zu unterweisen. Einer seiner Schüler ist besonders berühmt geworden, noch heute werden seine Abenteuer erzählt, Hollywood hat sie sogar verfilmt.

## 9.2 Abzählbar und überabzählbar unendliche Mengen

Die Geschichte des unendlichen Hotels<sup>2</sup> hat illustriert, dass unendliche Mengen einige Überraschungen bereithalten können, auf die die mit endlichen Mengen trainierte Intuition nicht vorbereitet ist. In diesem Abschnitt wird daher die Theorie der abzählbar unendlichen Mengen etwas genauer entwickelt.

<sup>2</sup>Die Geschichte geht auf David Hilbert zurück, der das unendliche Hotel in einer Vorlesung 1925 eingeführt hat. Wir haben eine an die heutige Zeit angepasste Variante davon präsentiert.

Mächtigkeit	Menge	Symbol
0	$\emptyset$	$[0]$
1	$\{\emptyset\}$	$[1]$
2	$\{\emptyset, \{\emptyset\}\}$	$[2]$
3	$\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$	$[3]$
$\vdots$	$\vdots$	$\vdots$
$n$	$A$	$[n]$
$n + 1$	$A \cup \{A\}$	$[n + 1]$
$\vdots$	$\vdots$	$\vdots$

Tabelle 9.1: Konstruktion endlicher Mengen jeder beliebigen Mächtigkeit aus der leeren Menge. Die Nachfolgermenge enthält jeweils die letzte Menge als letztes Element (**rot** hervorgehoben).

### 9.2.1 Endlich und abzählbar unendlich

Als erstes müssen wir lernen, endliche und unendliche Mengen voneinander zu unterscheiden.

#### Die natürlichen Zahlen und die Axiome von Peano

Die natürlichen Zahlen dienen zum Zählen. Bei diesem Prozess ist die Tatsache zentral, dass es zu jeder natürlichen Zahl einen Nachfolger gibt. Giuseppe Peano hat daher die folgenden Axiome aufgestellt, die die natürlichen Zahlen beschreiben sollen:

1. Null ist eine natürliche Zahl:  $0 \in \mathbb{N}$ .
2. Jede natürliche Zahl  $n$  hat einen Nachfolger  $n'$ .
3. 0 ist nicht Nachfolger einer natürlichen Zahl.
4. Natürliche Zahlen mit gleichem Nachfolger sind gleich.
5. Enthält die Menge  $X$  die 0 und mit jeder natürlichen Zahl  $n$  auch ihren Nachfolger  $n'$ , dann enthält  $X$  alle natürlichen Zahlen:  $\mathbb{N} \subset X$ .

#### Die natürlichen Zahlen und die Mächtigkeit endlicher Mengen

Die natürlichen Zahlen  $\mathbb{N}$  werden oft als Abstraktion der Mächtigkeit endlicher Mengen konstruiert. Zwei Mengen sind gleich mächtig, wenn es eine bijektive Abbildung zwischen ihnen gibt. Die leere Menge  $\emptyset$  hat die Mächtigkeit 0, die Menge  $\{\emptyset\}$  enthält nur ein einziges Element, nämlich die leere Menge, sie hat die Mächtigkeit 1. Nach diesem Muster lassen sich endliche Mengen beliebig großer Mächtigkeit konstruieren, die den natürlichen Zahlen entsprechen. Die Tabelle 9.1 zeigt die Konstruktion. Die  $n$ -elementigen Mengen in der mittleren Spalte werden mit  $[n]$  bezeichnet. Die Menge  $[n + 1]$  entsteht als Menge

$$[n + 1] = \{[0], [1], \dots, [n]\}$$

aller früheren Mengen. Das “letzte” Element der Menge  $[n]$  ist also jeweils  $[n - 1]$ .

Für die Menge

$$\mathbb{N} = \underbrace{\emptyset}_0, \underbrace{\{\emptyset\}}_1, \underbrace{\{\emptyset, \{\emptyset\}\}}_2, \dots, \underbrace{A}_n, \underbrace{A \cup \{A\}}_{n+1}, \dots$$

aller Mengen in der mittleren Spalte von Tabelle 9.1 kann man verifizieren, dass sie die Axiome von Peano erfüllen. Zum Beispiel zeigen die Zeilen  $n$  und  $n+1$ , wie der Nachfolger konstruiert wird. Das Axiom 1 folgt aus der Identifikation von 0 mit der leeren Menge. Da der Nachfolger als Menge aller früher konstruierten Mengen entsteht, kann die leere Menge nicht ein Nachfolger sein.

### Endliche und abzählbar unendliche Mengen

Endliche Mengen sind Mengen, für die es eine Bijektion mit einer der Mengen der Tabelle 9.1 gibt. Eine solche Bijektion ist automatisch auch injektiv. Wir überlegen uns nun, dass solch eine injektive Abbildung einer endlichen Menge immer surjektiv ist.

**Satz 9.1.** *Ist  $A$  eine endliche Menge und  $f: A \rightarrow A$  eine Injektion, dann ist  $f$  auch eine Bijektion.*

*Beweis.* Da  $f$  eine Injektion ist, sind die Elemente  $f(a)$  für  $a \in A$  alle verschieden. Die Menge  $f(A) = \{f(a) \mid a \in A\}$  hat also gleich viele Elemente wie  $A$ . Da sie endlich ist, muss sie mit  $A$  übereinstimmen.  $\square$

In dieser Vorstellung einer endlichen Menge stützen wir uns darauf, dass die Elemente einer Menge gezählt werden können. Wir verwenden explizit die natürlichen Zahlen  $\mathbb{N}$  zum Vergleich. Eine Kontraposition von Satz 9.1 ist, dass eine Menge, für die es eine Injektion gibt, die nicht eine Bijektion ist, auch nicht endlich sein kann.

Wie die Geschichte von Abschnitt 9.1 gezeigt hat, gibt es für die Menge  $\mathbb{N}$  der natürlichen Zahlen eine Injektion

$$f: \mathbb{N} \rightarrow \mathbb{N} : n \mapsto n'$$

( $n'$  ist der in den Peano-Axiomen postulierte Nachfolger von  $n$ ), die keine Bijektion ist, denn  $0 \in \mathbb{N}$  kann ja nicht Nachfolger einer Zahl sein. Die Menge  $\mathbb{N}$  unterscheidet sich also grundsätzlich von den endlichen Mengen.

Da eine Menge nicht endlich sein kann, wenn es eine Injektion der Menge in sich gibt, die keine Bijektion ist, kann man diese Idee auch als von den natürlichen Zahlen unabhängige Definition der Eigenschaft, endlich zu sein, verwenden.

**Definition 9.2** (Endliche Menge). *Eine Menge  $A$  heißt endlich, wenn jede Injektion  $A \rightarrow A$  auch eine Bijektion ist.*

Die Zimmer des unendlichen Hotels bilden keine endliche Menge, denn für jeden neuen Gast wendet der Hotelmanager eine Injektion an, die keine Bijektion ist.

**Definition 9.3** (abzählbar unendlich). *Die Menge  $\mathbb{N}$  und jede dazu gleichmächtige Menge heißt abzählbar unendlich.*

Eine Menge  $A$  ist also abzählbar unendlich, wenn es eine bijektive Abbildung  $\mathbb{N} \rightarrow A$  gibt. Man kann auch sagen, dass es eine nummerierte Liste aller Elemente von  $A$  gibt. Wir sagen auch, man kann die Menge  $A$  durchnummernieren.

## 9.2.2 Mengenoperationen

Die Vereinigung, die Schnittmenge und das kartesische Produkt endlicher Mengen sind wieder endliche Mengen. Die Kombinatorik lehrt, wie man in vielen Fällen die Anzahl der Elemente berechnen kann. Für abzählbar unendliche Mengen erwarten wir ähnliche Regeln, müssen uns aber mit einer separaten Überlegung versichern, dass uns unsere endliche Intuition nicht in die Irre führt.

### Endliche Vereinigung

Ist die Vereinigung endlich vieler abzählbar unendlicher Mengen auch wieder abzählbar unendlich? Seien  $A_1, \dots, A_n$  abzählbar unendliche Mengen. Jede dieser Mengen kann durchnummeriert werden:

$$\begin{aligned}A_1 &= \{a_{10}, a_{11}, a_{12}, \dots\} \\A_2 &= \{a_{20}, a_{21}, a_{22}, \dots\} \\&\vdots \quad \vdots \\A_n &= \{a_{n0}, a_{n1}, a_{n2}, \dots\}.\end{aligned}$$

Daraus kann jetzt die Durchnummerierung

$$\bigcup_{k=1}^n A_k = \{a_{10}, a_{20}, \dots, a_{n0}, a_{11}, a_{21}, \dots, a_{n1}, \dots\}$$

oder

$$= \{a_0, a_1, \dots\}$$

mit

$$a_{n(k-1)+i} = a_{ik}$$

von  $A_1 \cup \dots \cup A_n$  konstruiert werden. Damit ist der folgende Satz gezeigt.

**Satz 9.4.** *Die endliche Vereinigung abzählbar unendlicher Mengen ist wieder abzählbar unendlich.*

### Ganze Zahlen

Die ganzen Zahlen

$$\mathbb{Z} = \mathbb{N} \cup \{-n \mid n \in \mathbb{N}\}$$

sind die Vereinigung von zwei abzählbar unendlichen Mengen, also ist auch  $\mathbb{Z}$  abzählbar unendlich.

*Verständniskontrolle 9.1:* Sei

$$A_q = \left\{ \frac{a}{q} \mid a \in \mathbb{Z} \right\}$$

die Menge aller Brüche mit Nenner  $q$ .



a) Zeigen Sie, dass  $A_q$  abzählbar unendlich ist.

b) Zeigen Sie: Die Menge

$$B_n = \left\{ \frac{a}{b} \mid a \in \mathbb{Z} \wedge b \in \mathbb{N} \setminus \{0\} \wedge b \leq n \right\}$$

der Brüche mit Nenner  $\leq n$  ist abzählbar unendlich.

---

### Abzählbare Vereinigung und kartesisches Produkt

Die imperiale Armee aus der Geschichte wurde auf abzählbar unendlich vielen Schiffen verschifft, wobei jedes Schiff eine Division mit abzählbar vielen Stormtroopern fasste. Es sind also abzählbar unendlich viele Mengen  $A_k$ ,  $k \in \mathbb{N}$ , gegeben, die alle abzählbar unendlich sind. Die Geschichte zeigt, dass die Vereinigung

$$A = \bigcup_{k=0}^{\infty} A_k$$

wieder abzählbar unendlich ist. Dazu muss eine Zimmerzuordnung für die Kämpfer konstruiert werden, also eine Abbildung  $\mathbb{N} \rightarrow A$ . Die Konstruktion begann damit, die Elemente von  $A$  auf dem Vorplatz anzurufen. Jeder Soldat hat dabei den Platz mit den Koordinaten  $(k, p)$  eingenommen, der durch die Nummer  $k$  des Schiffs und die persönliche Nummer  $p$  des Soldaten bestimmt ist (Abbildung 9.3). Die Aufgabe ist also gleichbedeutend damit, für jedes Paar  $(k, p)$  eine Zimmernummer zu vergeben. Die Vereinigung  $A$  ist somit gleichmächtig mit dem kartesischen Produkt  $\mathbb{N} \times \mathbb{N}$ .

**Satz 9.5.**  $\mathbb{N} \times \mathbb{N}$  ist abzählbar unendlich.

*Beweis.* Zum Beweis müssen die Elemente  $(k, p) \in \mathbb{N}^2$  durchnummieriert werden. Wir verwenden die Nummerierung in Abbildung 9.3. Die Zahlen in der ersten Zeile sind die Dreieckszahlen

$$t_k = \sum_{i=0}^k = \frac{k(k+1)}{2}.$$

Die Punkte mit Koordinaten  $(k, 0)$  erhalten somit die Nummer  $t_k$ . Der Punkt mit Koordinaten  $(k, p)$  liegt im blauen Streifen mit der Nummer  $s = k + p$ . Der Punkt am rechten oberen Ende dieses Streifens hat die Nummer  $t_{k+p}$ , die anderen Punkte haben mit  $p$  aufsteigende Nummern. Somit ist

$$(k, p) \mapsto t_{k+p} + p = \frac{(k+p)(k+p+1)}{2} + p$$

die Nummerierung des Punktes mit Koordinaten  $(k, p)$ . Damit ist eine Nummerierung von  $\mathbb{N}^2$  konstruiert.  $\square$

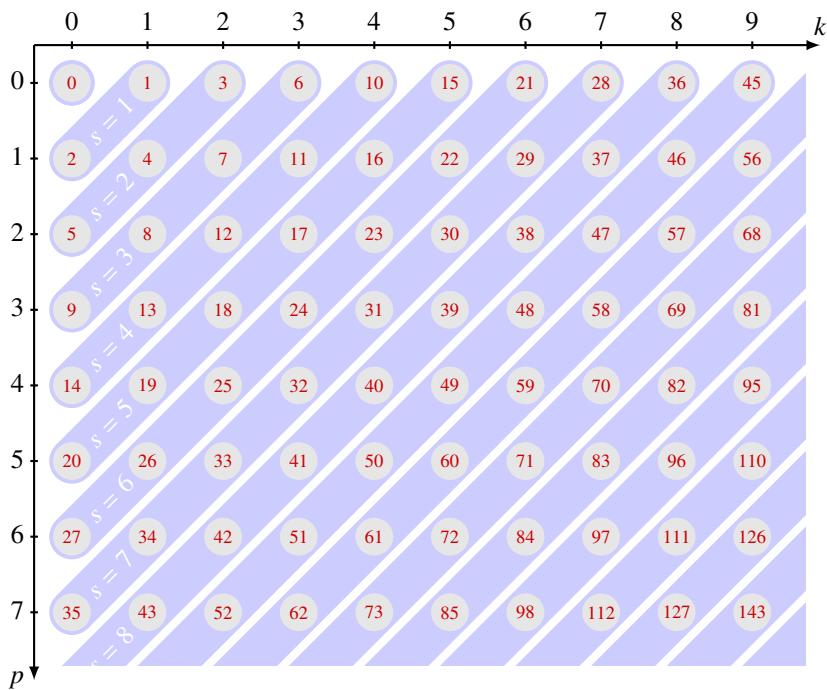


Abbildung 9.3: Die Nummerierung der Punkte des kartesischen Produktes zeigt, dass  $\mathbb{N}^2$  abzählbar unendlich ist.

## Rationale Zahlen

Für die rationalen Zahlen

$$\mathbb{Q} = \left\{ \frac{p}{q} \mid p \in \mathbb{Z} \wedge q \in \mathbb{N} \setminus \{0\} \right\}$$

gibt es eine Abbildung

$$f: \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{Q} : (a, b) \mapsto \frac{a}{b+1}.$$

Da die linke Seite  $\mathbb{Z} \times \mathbb{N}$  abzählbar unendlich ist, ist auch  $\mathbb{Q}$  abzählbar unendlich.

**Verständniskontrolle 9.2:** Eine ganze gaußsche Zahl ist eine komplexe Zahlen  $a + bi$  mit  $a, b \in \mathbb{Z}$ . Die Menge der ganzen gaußschen Zahlen wird mit  $\mathbb{Z}[i]$  bezeichnet. Sie können als Gitterpunkte der komplexen Ebene dargestellt werden. Zeigen Sie, dass  $\mathbb{Z}[i]$  abzählbar unendlich ist.



autospr.ch/v/9.2.pdf

## Die Menge aller Wörter

Die in Abschnitt 1.1.2 eingeführte Menge

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k = \{\varepsilon\} \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots \cup \Sigma^k \cup \dots$$

aller Wörter ist eine abzählbar unendliche Vereinigung der Mengen  $\Sigma^k$ , die jeweils  $|\Sigma^k| = |\Sigma|^k$  Elemente enthalten, also alle endlich sind. Als abzählbar unendliche Vereinigung endlicher Mengen ist die Menge  $\Sigma^*$  abzählbar unendlich.

Der Spezialfall  $\Sigma = \{1\}$  ist besonders interessant. In diesem Fall ist

$$\Sigma^* = \{\varepsilon, 1, 11, 111, \dots\} = \{1^k \mid k \in \mathbb{N}\},$$

die Abbildung  $k \mapsto 1^k$  ist also eine Bijektion von  $\mathbb{N} \rightarrow \Sigma^*$ . Man nennt die Wörter von  $\{1\}^*$  die *unäre Darstellung* der natürlichen Zahlen.

## Die Menge aller deterministischen endlichen Automaten

In Abschnitt 1.2.2 wurde auch die Tabellendarstellung eines deterministischen endlichen Automaten vorgestellt. Es gibt viele verschiedene Möglichkeiten, Tabellen als Zeichenketten zu beschreiben, zum Beispiel könnte man sie als HTML-Tabelle oder als JSON-Struktur beschreiben. Welche Codierung man auch wählt, jeder endliche Automat hat eine Darstellung als eine Zeichenkette. Die Menge der endlichen Automaten lässt sich also in  $\Sigma^*$  einbetten. Da  $\Sigma^*$  abzählbar unendlich ist, ist auch die Menge der deterministischen endlichen Automaten abzählbar unendlich.

Das Argument funktioniert für fast alle Konstruktionen der vorangegangenen Kapitel. Die Menge aller nichtdeterministischen endlichen Automaten, die Menge der regulären Ausdrücke, die Menge aller kontextfreien Grammatiken und die Menge der Stackautomaten sind alle abzählbar unendlich.

### 9.2.3 Überabzählbar unendliche Mengen

Alle abzählbar unendlich großen Mengen sind gleichmächtig wie die Menge  $\mathbb{N}$  der natürlichen Zahlen, sie sind in gewissem Sinne "gleich groß". Es ist nicht offensichtlich, dass es Mengen gibt, die noch größer sind, für die es also keine nummerierte Auflistung gibt.

#### Definition

Wenn sich eine Menge nicht durchnummerieren lässt, dann muss sie größer sein als jede abzählbar unendliche Menge.

**Definition 9.6** (überabzählbar unendlich). *Eine unendliche Menge A heißt überabzählbar unendlich, wenn es keine bijektive Abbildung  $f: \mathbb{N} \rightarrow A$  gibt.*

Eine Menge ist überabzählbar unendlich, wenn in jeder numerierten Liste von Elementen von A mindestens ein Element fehlt.

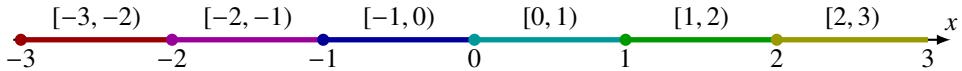


Abbildung 9.4: Die reellen Zahlen als abzählbare Vereinigung der Intervalle  $[k, k + 1) = k + I$  müsste abzählbar unendlich sein, wenn das Intervall  $I = [0, 1)$  nur abzählbar unendlich viele verschiedene reelle Zahlen enthielte.

## Reelle Zahlen

In der Geschichte wusste der Hotelmanager nicht mehr weiter, als er sich mit dem Kontinuum der reellen Zahlen konfrontiert sah. Der Grund dafür ist der folgende Satz.

**Satz 9.7.** *Die reellen Zahlen  $\mathbb{R}$  sind überabzählbar unendlich.*

*Beweis.* Es genügt zu zeigen, dass bereits die reellen Zahlen im halboffenen Intervall  $I = [0, 1)$  überabzählbar unendlich sind. Wären diese nämlich abzählbar unendlich, dann müsste auch die abzählbare Vereinigung

$$\mathbb{R} = \bigcup_{k \in \mathbb{Z}} [k, k + 1) = \bigcup_{k \in \mathbb{Z}} (k + I)$$

aller halboffenen Intervalle, die bei ganzen Zahlen beginnen, abzählbar unendlich sein (Abbildung 9.4).

Die Zahlen in  $r \in I$  können im Binärsystem in der Form

$$r = 0.r_1r_2r_3\dots$$

dargestellt werden, wobei die  $r_i$  Binärziffern in  $\{0, 1\}$  sind. Nehmen wir an, es gäbe eine vollständige Auflistung aller reellen Zahlen des Intervalls  $I$  wie in Abbildung 9.5. Dann lässt sich wie folgt eine Zahl  $r \in I$  finden, die in der Auflistung fehlt. Die erste Nachkommastelle der Zahl  $r$  ist 1 minus die erste Nachkommastelle der ersten Zahl der Liste. Die zweite Nachkommastelle von  $r$  ist 1 minus die zweite Nachkommastelle der zweiten Zahl der Liste. Und so weiter, die  $k$ -te Nachkommastelle von  $r$  ist 1 minus die  $k$ -te Nachkommastelle der  $k$ -ten Zahl der Liste. Da die Zahl  $r$  sich an der  $k$ -ten Stelle von der  $k$ -ten Zahl der Liste unterscheidet, unterscheidet sich  $r$  von jeder Zahl der Liste.  $r$  kommt in der Liste nicht vor. Dass  $r$  in der Liste fehlt, widerspricht der Annahme, dass die Auflistung vollständig war.  $\square$

---

Verständniskontrolle 9.3: Zeigen Sie: Die Menge aller Potenzreihen

$$f(x) = \sum_{k=0}^{\infty} a_k x^k$$



mit beschränkten ganzzahligen Koeffizienten  $a_k \in \mathbb{Z}$ ,  $|a_k| < M$  ist überabzählbar unendlich.

---

$r_0 = 0$	.	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	0	...	
$r_1 = 0$	.	1	0	0	0	1	1	1	1	0	0	1	0	0	0	1	0	0	0	...	
$r_2 = 0$	.	0	0	0	1	0	1	0	1	1	0	0	0	0	1	0	0	1	1	...	
$r_3 = 0$	.	1	0	1	1	1	0	0	1	1	1	0	0	0	1	0	0	1	1	...	
$r_4 = 0$	.	1	0	1	0	1	1	0	1	0	0	0	1	0	1	1	1	1	1	...	
$r_5 = 0$	.	0	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0	1	0	...	
$r_6 = 0$	.	0	1	0	1	1	0	0	0	0	1	1	0	0	1	1	0	1	1	...	
$r_7 = 0$	.	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	...	
$r_8 = 0$	.	1	1	1	1	0	0	1	1	1	0	1	1	1	0	1	0	1	0	...	
$r_9 = 0$	.	1	0	0	0	0	1	0	0	0	0	1	1	0	1	1	0	0	0	...	
$r_{10} = 0$	.	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	1	0	0	1	...
$r_{11} = 0$	.	0	0	1	1	1	0	1	1	1	1	0	0	0	1	1	1	1	0	0	...
$r_{12} = 0$	.	0	0	1	0	1	1	1	1	1	0	0	0	0	1	1	0	1	1	0	...
$r_{13} = 0$	.	1	1	0	1	1	0	1	1	1	1	1	0	0	0	0	0	1	0	0	...
$\vdots$	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	...	
$r = 0$	.	1	1	1	0	0	0	1	1	1	1	1	0	0	1	0	1	0	1	...	

Abbildung 9.5: Aufzählung der reellen Zahlen im halboffenen Intervall  $[0, 1)$ . Die **rote** Zahl  $r$  unterscheidet sich aufgrund der Konstruktion jeweils in der  $k$ -Stelle von  $r_k$ . Somit kann  $r$  nicht in der Liste vorkommen.

### Die Potenzmenge

Die reellen Zahlen sind kein Spezialfall. Überabzählbar unendlich große Mengen sind in der Tat sehr verbreitet und auch das Argument im Beweis von Satz 9.7 ist wieder anwendbar.

**Satz 9.8.** Ist  $A$  eine abzählbar unendlich große Menge, dann ist ihre Potenzmenge  $P(A)$  überabzählbar unendlich.

*Beweis.* Der Beweis von Satz 9.7 kann wie folgt auf die vorliegende Situation verallgemeinert werden. Zunächst wählen wir eine Aufzählung  $A = \{a_0, a_1, a_2, \dots\}$ , weil  $A$  nach Voraussetzung abzählbar ist. Sie ist in Abbildung 9.6 als Kopfzeile der Tabelle dargestellt.

Wir nehmen jetzt an, wir hätten eine vollständige Aufzählung  $A_k$  der Teilmengen von  $A$ . Um eine Teilmenge  $A_k$  von  $A$  zu spezifizieren, müssen wir angeben, welches der Elemente von  $A$  in der Menge enthalten ist. In der Tabelle wird dies durch Kreuze gemacht.

Die rote Menge  $\tilde{A}$  soll jetzt das Element  $a_k$  genau dann enthalten, wenn  $A_k$  das Element nicht enthält, also

$$a_k \in \tilde{A} \quad \Leftrightarrow \quad a_k \notin A_k, \quad k \in \mathbb{N}.$$

Die Menge  $\tilde{A}$  unterscheidet sich von jeder der Mengen  $A_k$  durch das Element  $a_k$ , sie fehlt

$A$	$a_0$	$a_1$	$a_2$	$a_3$	$a_4$	$a_5$	$a_6$	$a_7$	$a_8$	$a_9$	$a_{10}$	$a_{11}$	$a_{12}$	$a_{13}$	$a_{14}$	$a_{15}$	$a_{16}$	$a_{17}$	...
$A_0$	x	x	x	x	x		x	x	x	x			x	x				...	
$A_1$	x	x				x	x	x	x	x			x		x			...	
$A_2$		x	x		x		x	x	x	x			x		x	x	x	...	
$A_3$	x		x	x		x	x	x	x			x		x	x	x	x	...	
$A_4$	x		x		x	x		x		x		x	x	x	x	x	x	...	
$A_5$			x	x		x	x			x		x	x	x	x	x		...	
$A_6$		x		x	x		x		x	x		x	x		x	x	x	...	
$A_7$	x				x		x		x	x		x	x		x	x	x	...	
$A_8$	x	x	x	x			x	x	x	x		x	x	x	x	x		...	
$A_9$	x				x			x		x		x	x		x	x		...	
$A_{10}$	x			x	x		x	x	x	x		x	x	x	x		x	...	
$A_{11}$			x	x	x		x	x	x	x		x	x	x	x			...	
$A_{12}$	x	x		x	x		x	x	x	x		x	x	x	x			...	
$A_{13}$													x			x			...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\tilde{A}$	x	x	x					x	x	x				x		x	x	...	

Abbildung 9.6: Aufzählung der Potenzmenge einer abzählbar unendlichen Menge  $A$ . Die Teilmengen  $A_k$ ,  $k \in \mathbb{N}$ , enthalten die Elemente, in deren Spalte ein  $\times$ -Symbol steht. Die rote Menge  $\tilde{A}$  unterscheidet sich aufgrund der Konstruktion jeweils im Element  $a_k$  der Menge  $A_k$ . Somit kann  $\tilde{A}$  nicht in der Liste vorkommen.

also in der Aufzählung. Dieser Widerspruch zeigt, dass es eine Aufzählung von  $P(A)$  nicht geben kann.  $\square$

## Die Menge aller Sprachen

Früher wurde gezeigt, dass die Menge  $\Sigma^*$  aller Wörter abzählbar unendlich ist. Sprachen sind Teilmengen von  $\Sigma^*$ , die Menge aller Sprachen ist also die Potenzmenge  $P(\Sigma^*)$  von  $\Sigma^*$ . Nach Satz 9.8 ist sie überabzählbar unendlich. Es gibt also viel mehr Sprachen als es endliche Automaten, reguläre Ausdrücke oder kontextfreie Grammatiken gibt (Abbildung 9.7). Das “Sprachuniversum” ist viel größer, als die Konstruktionen der vorangegangenen Kapitel vermuten lassen.

## Übungsaufgaben

### 9.1. Sind die folgenden Mengen abzählbar oder überabzählbar unendlich?

- Die Menge  $G_n$  aller linearen Gleichungssysteme mit  $n$  Gleichungen und  $n$  Unbekannten und ganzzahligen Koeffizienten und rechten Seiten.

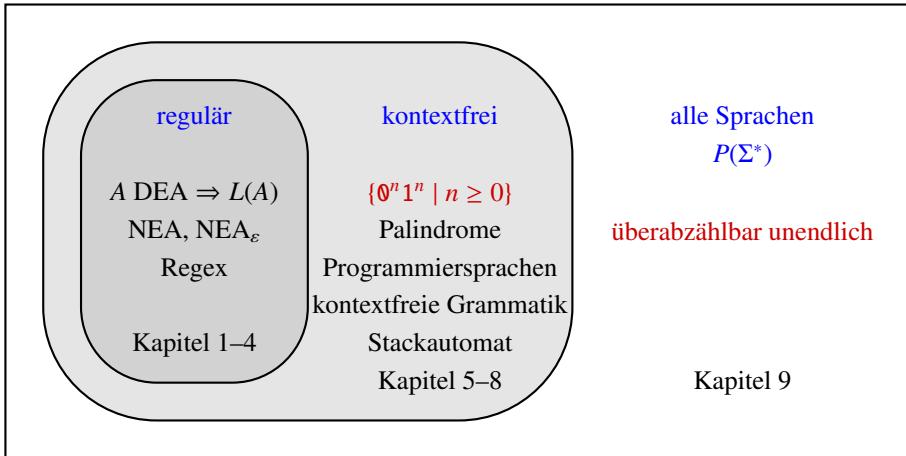


Abbildung 9.7: Die bisher untersuchten Sprachen sind abzählbar unendlich große Familien von Sprachen (Mengen mit dunklem Hintergrund). Die Menge aller Sprachen ist aber überabzählbar unendlich.

- b) Die Menge  $G$  aller linearen Gleichungssysteme mit gleich vielen Gleichungen wie Unbekannten und ganzzahligen Koeffizienten.
- c) Die Menge  $\mathbb{Z}[X]_n$  aller Polynome
 
$$a_n X^n + a_{n-1} X^{n-1} + \dots + a_2 X^2 + a_1 X + a_0$$
 vom Grad  $n$  mit ganzzahligen Koeffizienten.
- d) Die Menge  $\mathbb{Z}[X]$  aller Polynome beliebigen Grades  $n$  mit ganzzahligen Koeffizienten.
- e) Eine Zahl  $x \in \mathbb{R}$  heißt algebraisch, wenn sie Nullstelle eines Polynoms mit ganzzahligen Koeffizienten ist. Die Zahl  $\sqrt{2}$  ist eine Nullstelle von  $X^2 - 2$  und ist daher algebraisch. Die Zahlen  $\pi$  und  $e$  sind dagegen nicht algebraisch. Ist die Menge  $\mathbb{A}$  aller algebraischen Zahlen abzählbar oder überabzählbar unendlich?

**9.2.** Welche der folgenden Mengen sind abzählbar unendlich, welche sind überabzählbar unendlich?

- a) Die Menge aller regulären Ausdrücke.
- b) Die Menge aller regulären Sprachen.
- c) Die Menge aller kontextfreien Grammatiken.
- d) Die Menge aller kontextfreien Sprachen.
- e) Die Menge aller Stackautomaten.



# Kapitel 10

## Turing-Maschinen

Endliche Automaten und Stackautomaten sind nicht in der Lage, gewisse Sprachen zu erkennen, für die jeder Programmieranfänger mit Leichtigkeit ein Programm schreiben kann. Moderne Computer sind ganz offensichtlich leistungsfähiger als diese einfachen Typen von Maschinen. In diesem Kapitel steht im Vordergrund, wie die bisherigen Maschinenkonzepte so verallgemeinert werden können, dass auch Sprachen wie  $a^n b^n c^n$  verarbeitet werden können. Die Verallgemeinerung soll einerseits so einfach wie möglich sein, damit es weiterhin möglich ist, allgemeingültige Aussagen abzuleiten. Andererseits sollen die erweiterten Maschinen Fähigkeiten bekommen, die mit modernen Computern vergleichbar sind, damit die Schlussfolgerungen für die Praxis relevant bleiben. Die Turing-Maschine, die in Abschnitt 10.1 konstruiert wird, verallgemeinert alle früheren Maschinenkonzepte und kommt einem modernen Computer bereits sehr nahe. Abschnitt 10.2 zeigt, dass verschiedene denkbare Varianten nicht grundsätzlich etwas daran ändern, was mit einer Turing-Maschine berechnet werden kann. Dies wird bestätigt durch die Existenz der universellen Turing-Maschine, die in Abschnitt 10.4 skizziert wird. In Abschnitt 10.3 wird die neue Klasse der *Turing-erkennbaren Sprachen* einführt, die sich mit Hilfe von Turing-Maschinen definieren lassen.

### 10.1 Vom Stackautomaten zur Turing-Maschine

Ein moderner Computer unterscheidet sich vor allem in zwei Eigenschaften von einem Stackautomaten: Er verfügt über einen Direktzugriffsspeicher und er läuft von selbst, auch ohne expliziten Input, wie er bei den deterministischen endlichen Automaten und etwas eingeschränkt bei den Stackautomaten nötig ist.

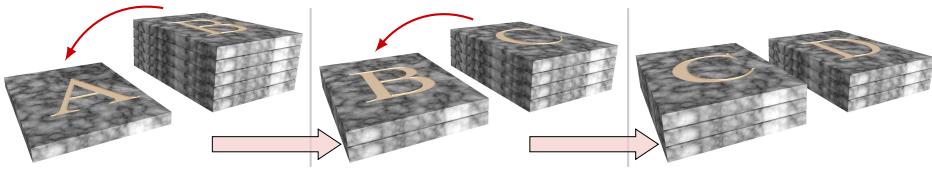


Abbildung 10.1: Zwei Stacks ergeben zusammen einen frei zugreifbaren Speicher ähnlich einem Buch, in dem beliebig geblättert werden kann.

### 10.1.1 Speicher

Ein Stackautomat hat seine über einen endlichen Automaten hinausgehenden Fähigkeiten durch den Stackspeicher erhalten. Die Limitierung eines solchen Speichers ist, dass immer nur das zuletzt gespeicherte Element verfügbar ist. Will man auf ein früher gespeichertes Element zugreifen, muss man die seither auf dem Stack abgelegten Elemente entfernen, wodurch sie verloren gehen.

#### Zustände als Speicher

Es ist denkbar, den endlichen Automaten, der zum Stackautomaten gehört, mit ausreichend vielen Zuständen auszustatten, so dass man gewisse Informationen auch in den Zuständen speichern kann. Eine moderne CPU hat zum Beispiel eine paar Register, die unmittelbar benötigte Datenelemente aufnehmen können. Sind diese aber voll, muss die CPU auf den Hauptspeicher zugreifen. Stünde ihr statt des Hauptspeichers nur ein Stack zur Verfügung, gingen beim Zugriff auf tief unten im Stack vergrabene Datenelemente die darüber liegenden verloren, weil in den CPU-Registern kein Platz mehr dafür ist.

#### Ein Buch als zwei Stacks

Es ist zu erwarten, dass ein zweiter Stack die Fähigkeiten verbessern könnte. Dies ist jedoch nicht nur ein gradueller Unterschied. Ein zweiter Stack kann nämlich dazu verwendet werden, die Elemente vom ersten Stack aufzunehmen, die dort entfernt werden müssen, um Zugriff auf Elemente weiter unten auf dem Stack zu erhalten. So gehen keine Informationen mehr verloren.

Ein Buch wie das hier vorliegende ist ein Paar von Stacks. Bevor das Buch aus der Werkstatt des Buchbinders oder moderner der automatisierten Bindemaschine kommt, besteht es nur aus einem Stapel von Blättern. Die Bindung erzwingt, dass immer nur auf die Doppelseite zugegriffen werden kann, bei der das Buch geöffnet ist. Der eine Stack enthält alle Blätter von der aktuellen Seite bis zur Buchvorderseite, der andere die Blätter bis zum hinteren Buchdeckel. Durch Umblättern verschiebt man ein Blatt des Buches vom einen Stack auf den anderen. Damit wird ein anderes Speicherelement für den Zugriff freigegeben, ohne dass Information verloren geht. Da jetzt auf jede beliebige Information zugegriffen werden kann, ohne dass Information verloren gehen kann, verhält sich ein Paar von Stacks wie ein Direktzugriffsspeicher (Abbildungen 10.1 und 10.2).



Abbildung 10.2: Ein Buch besteht aus zwei Stapeln von Blättern, die sich wie zwei gekoppelte Stackspeicher verhalten. Es ist immer nur eine Doppelseite zugreifbar, aber durch Umblättern können die Daten zwischen den beiden Stacks hin- und hergeschoben werden, so dass ein Wahlzugriffsspeicher entsteht, der keine Information verliert.

### Das Band als Direktzugriffsspeicher

Die Vorstellung eines Direktzugriffsspeichers als zwei zusammenwirkende Stackspeicher ist eher schwerfällig. Im Folgenden wird daher die Vorstellung eines Bandes von Speicherstellen verwendet, welches in beiden Richtungen unendlich ausgedehnt ist (Abbildung 10.3). Die einzelnen Speicherzellen können jeweils ein Element eines Alphabets  $\Gamma$  aufnehmen, welches auch das *Bandalphabet* genannt wird. Ein Stackautomaten hat immer auf genau eine Speicherzelle Zugriff. Daran ändert sich auch bei der Vorstellung des Bandes nichts. Die aktuelle Schreib-/Leseposition wird auch als *Schreib-/Lesekopf* bezeichnet.

Die Bezeichnung des Speichers als Band stammt natürlich aus einer Zeit, in der ein großer Speicher nur mit externen Medien zu realisieren war, typischerweise mit Magnetbandeinheiten. Die Daten werden auf solchen Geräten als Einheiten fester Größe gespeichert, die *Records* heißen. Es ist nicht möglich, weniger als einen Record zu lesen oder zu schreiben. Die Kommandozeilen-Optionen der Unix-Programm `dd(1)` [11] oder `mt(1)` [33] ermöglichen, zwischen einzelnen Files auf einem Magnetband zu navigieren bzw. auf einen bestimmten Record zuzugreifen. Ein noch älterer Vorläufer der Idee ist der Jacquard-Webstuhl, der von Lochkarten gesteuert wird, die untereinander zu einem langen Band verbunden sind. Für die Verarbeitung eines Durchschusses durch den Webstuhl werden die Löcher einer einzelnen Karte abgetastet.

Der Hauptspeicher eines modernen Computers bietet sich als Realisation des Bandes an. Dieser ist jedoch von endlicher Größe und manchmal sogar kleiner als was umfangreiche Programme benötigen. Dies ist jedoch eine leicht zu überwindende Einschränkung. Die virtuelle Speicherverwaltung spiegelt jedem Prozess einen für alle praktischen Zwecke des Prozesses unbeschränkt großen Speicher vor: Ein Programm kann innerhalb vernünftiger Grenzen Speicher anfordern und wird den Speicher auch zugeteilt erhalten. Es bleibt

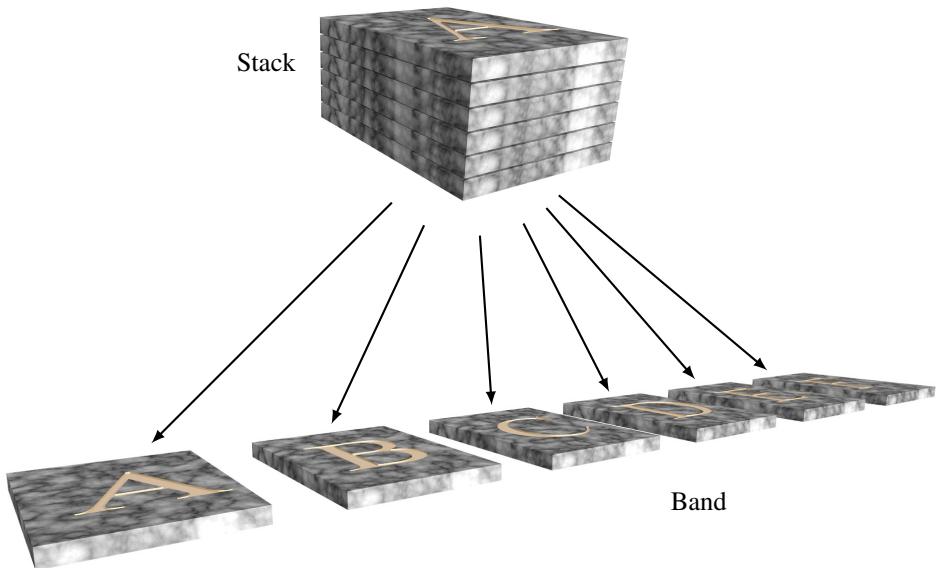


Abbildung 10.3: Da zwei Stacks einen Wahlzugriffsspeicher ergeben, kann man sich die Elemente des Stacks auch als in einem unendlich langen Band ausbreiten vorstellen.

dann die Aufgabe des Betriebssystems sicherzustellen, dass die Speicherseiten bei Bedarf durch den Prozess tatsächlich gelesen und beschrieben werden können.

### **Kein Inputwort**

In der Beschreibung der Arbeit eines endlichen Automaten oder eines Stackautomaten kam dem Inputwort eine zentrale Rolle zu. Die Maschine verarbeitet ein Zeichen des Wortes nach dem anderen. Man könnte sagen, das Inputwort ist ein weiterer Speicher, aus dem die Maschine Zeichen in vorgegebener Reihenfolge lesen kann.

Bei einer komplexen Fragestellung kann man nicht erwarten, dass die Antwort mit einem einzigen Durchgang durch die Buchstaben des Wortes gefunden werden kann. Da jetzt mit dem Band ein beliebig großer Speicher zur Verfügung steht, kann die Maschine die Input-Daten zunächst in diesen Speicher kopieren und bei Bedarf immer wieder lesen. Damit entfällt die Notwendigkeit der speziellen Rolle des Inputwortes. Selbst für Wörter, die wie mit einem deterministischen endlichen Automaten mit einmaligem Lesen verarbeitet werden könnten, werden wir immer davon ausgehen, dass das zu verarbeitende Wort bereits auf dem Band steht.

### **Maschinen = Prozesse**

Ein einzelner Prozess innerhalb eines modernen Betriebssystems ist ein besseres Bild. Die vom Betriebssystem zur Verfügung gestellten I/O-APIs geben dem Programmierer den Eindruck, als würde sein Programm auf Datenquelle außerhalb des Programms zugreifen.



Abbildung 10.4: Gruppenbild der “Computer” am Harvard College Observatory [20, public domain]. Neben ihrer Berechnungsarbeit wurden die Frauen auch für ihre astrophysikalischen Entdeckungen berühmt. Stehend: Williamina Fleming, die 1899 als eine der ersten Frauen eine Anstellung am Harvard College erhielt. Sie war jedoch schon seit 1888 mit der Anstellung und Verwaltung der Gruppe der Computer betraut. Sie gilt auch als die Entdeckerin des Pferdekopfnebels im Sternbild Orion.

Dies ist aber nur einem vom Betriebssystem vermittelte Illusion. Beim Aufruf einer I/O-Operation übernimmt der Kernel des Betriebssystems die Kontrolle, führt die Operation aus und speichert die Daten im Adressraum des aufrufenden Programms. Aus Sicht des Prozesses wird dieser einfach angehalten, bis fast magisch die Daten im Speicher auftauchen. Es ändert sicher daher nichts an den Fähigkeiten einer Maschine, wenn man ihr Eingabemöglichkeiten zur Verfügung stellt.

### 10.1.2 Berechnung

Alan Turing war in seinen Überlegungen von der Arbeitsweise menschlicher Computer inspiriert. Abbildung 10.4 zeigt die Gruppe der Computer am Harvard College Observatory, die einerseits Berechnungen wie die Auswertung von photographischen Aufnahmen des Nachhimmels durchführten, daneben aber auch eigene astronomische Forschung betrieben. Ein Computer folgt bei ihrer Berechnung schriftlichen Anweisungen und schreibt die Resultate in vorbereitete Felder von Lösungsformularen. Sie folgt dabei einfachen Regeln, ohne verstehen zu müssen, was sie genau tut.

Schon Charles Babbage hatte die Idee einer universellen Rechenmaschine entwickelt. Seine Partnerin Ada Lovelace hatte aber erkannt, dass sich damit viel mehr machen ließ. Beliebige mathematische Objekte, die für die Maschine codiert werden können, können Gegenstand von Berechnungen werden. Die Maschine von Babbage wurde allerdings nie fertig gebaut.

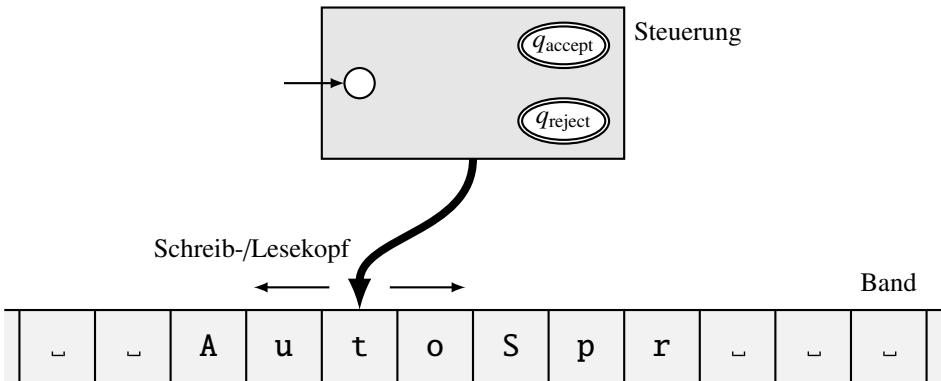


Abbildung 10.5: Eine Turing-Maschine liest mit dem Schreib-/Lesekopf ein Datenelement vom Band, führt damit eine Berechnung durch und entscheidet, ob das Datenelement auf dem Band durch ein neues ersetzt werden soll. Anschließend wird der Schreib-/Lesekopf nach links oder rechts bewegt, um den nächsten Berechnungsschritt durchzuführen.

Turing formalisierte die Berechnungsarbeit zum Konzept der Turing-Maschine, welches wir heute kennen. Die Turing-Maschine ist ein Modell für die Berechnung, welches einfach und übersichtlich genug ist, dass man darüber sinnvolle mathematische Aussagen ableiten kann, aber auch komplex genug, dass man argumentieren kann, dass jede Berechnung, die ein moderner Computer ausführen kann, auch ein Turing-Maschine durchführen kann.

Der Aufbau einer Turing-Maschine ist in Abbildung 10.5 wiedergegeben. Die Maschine wird gesteuert von einem endlichen Automaten, der in der Abbildung mit *Steuerung* bezeichnet ist. Dieser liest mithilfe eines Schreib-/Lesekopfes jeweils eine einzelne Zelle des bereits in Abschnitt 10.1.1 als natürlicher Speicher diskutierten Bandes. Das gelesene Zeichen wird verarbeitet und entschieden, ob das aktuelle Feld des Bandes verändert werden soll und mit welchem Zeichen es gegebenenfalls überschrieben werden muss. Anschließend wird der Schreib-/Lesekopf auf das Nachbarfeld links oder rechts verschoben.

Die Berechnung durch die Turing-Maschine läuft also automatisch ab, sie wird nicht wie bei einem endlichen Automaten dadurch angestoßen, dass der Maschine der Reihe nach die Zeichen des Wortes übergeben werden. Die Zustandsmaschine entscheidet ohne Einfluss von außen, wann ein Zustand eingetreten ist, mit dem die Berechnung angehalten werden kann. Grund dafür kann sowohl der erfolgreiche Abschluss einer Berechnung wie auch das Fehlschlagen der Berechnung sein.

**Verständniskontrolle 10.1:** Beschreiben Sie die Vorgehensweise einer Turing-Maschine, die Wörter über dem Alphabet  $\Sigma = \{0, 1\}$  wie folgt erkennt:

- Das Wort besteht nur aus Nullen.
- Das Wort enthält keine Nullen.
- Das Wort hat die Form  $0^n 1^m$ ,  $n, m > 0$ .



autospqr.ch/v/10.1.pdf

### 10.1.3 Zustandsdiagramm

In diesem Abschnitt soll die Steuerung der Turing-Maschine etwas genauer beschrieben werden. Wie alle früheren Automaten wird auch eine Turing-Maschine von einem endlichen Automaten gesteuert (Abbildung 10.5).

#### Übergänge

Ein Zustandsübergang in einer Turing-Maschine hängt vom aktuellen Zustand und dem Inhalt der aktuell sichtbaren Speicherzelle des Bandes ab. Beim Übergang kann der Inhalt der aktuellen Speicherstelle verändert werden. Schließlich muss entschieden werden, bei welcher Speicherzelle weitergearbeitet werden kann. Moderne Computer verwenden für diesen Zweck verschiedene Adressregister, die beliebige Zellen des Speichers adressieren können. In unserer Konstruktion des Bandes gibt es keine Möglichkeit die Zellen des Bandes absolut anzuspringen. Es liegt eher die Situation eines Buches vor, welches keine Seitenzahlen hat. Die einzige Art der Navigation im Speicher ist vor- und zurückzublättern.

Da man Blättern um mehrere Seiten nach vorne und nach hinten durch wiederholtes Blättern um einzelne Seiten realisieren kann, braucht es nur die Blätteroperationen um eine Seite. Auch eine Operation, die gar nicht blättert, ist nicht nötig. Wenn nach einem Übergang immer noch die gleiche Speicherzelle zur Verarbeitung ansteht, dann hätte man diese Arbeit auch schon im letzten Schritt auch noch erledigen können. Es bleiben also nur die Bewegungen  $L$  nach links und  $R$  nach rechts des Schreib-/Lesekopfes.

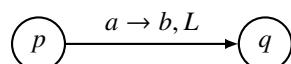
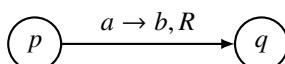
Die Einschränkung auf relative Bewegungen des Schreib-Lesekopfes ist weniger limitierend als man im ersten Moment vermuten würde. Der Maschinencode, der vom Compiler aus einer Funktionsdefinition in einer kompilierten Sprache wie C produziert wird, richtet beim Aufruf der Funktion auf dem Stack, der als Teil des Hauptspeichers realisiert ist, ein Stackframe ein, in dem alle lokalen Variablen gespeichert sind. Der Zugriff auf die lokalen Variablen erfolgt somit nicht absolut, sondern immer relativ zum Anfang des Stackframe.

Viele Compiler können auch positionsunabhängigen Code erzeugen (position independent code, PIC). Solcher Code verwendet keine absoluten Sprungadressen, sondern nur relative Sprünge. Dies vereinfacht die Verwendung des Codes in einer dynamisch ladbaren Bibliothek. Der Code funktioniert ganz unabhängig davon, wo er in den Speicher geladen worden ist. Es müssen dann nur noch die Einstiegspunkte in die Bibliothek verwaltet werden.

In Anlehnung an die Schreibweise bei Stackautomaten verwenden wir für die Zustandsübergänge die Notation

$$\delta(p, q) = (q, b, R)$$

$$\delta(p, q) = (q, b, L)$$



Damit der Übergang möglich ist, muss die aktuelle Speicherzelle das Zeichen  $a$  enthalten. Das Zeichen wird dann von  $b$  überschrieben, die Maschine geht in den Zustand  $q$  über und

der Schreib-/Lesekopf wird auf eine Nachbarzelle bewegt. Für das Zeichen  $R$  erfolgt die Bewegung nach rechts, für  $L$  nach links.

### Akzeptierzustände

Endliche Automaten und Stackautomaten konnten so viele Akzeptierzustände wie nötig beinhalten. Sie konnten zum Beispiel dazu verwendet werden, mitzuteilen, welche Art von Wort akzeptiert worden ist. Für die Ausgabe dieser Information kann jetzt aber auch das Band verwendet werden. Die verschiedenen Akzeptierzustände konnten durch Einführung zusätzlicher  $\varepsilon$ -Übergänge konsolidiert werden, wie das zum Beispiel in Abschnitt 4.4.2 bei der Konstruktion eines VNEA aus einem DEA geschah. Es gibt also keinen Grund mehr, viele verschiedene Akzeptierzustände zu verwenden.

Die früheren Automaten konnten nicht weiter arbeiten, wenn keine weiteren Inputzeichen mehr vorhanden waren. Da auch das Inputwort nur noch ein Teil des Speichers ist, der sich nicht von anderen Teilen unterscheidet, kann das Ende des Inputwortes keine geeignete Marke für das Ende der Berechnung mehr sein. Wir erwarten auch, dass die Maschine Probleme lösen kann, bei denen die Arbeit nach dem Lesen des Wortes erst richtig losgeht. Die Maschine muss lange Zeit selbstständig arbeiten können. Es braucht daher noch einen Mechanismus, mit dem der Maschine das Ende der Rechnung signalisiert werden kann. Die Akzeptierzustände einer Turing-Maschine dienen jetzt vor allem dem Zweck, die Maschine anzuhalten.

Für eine Turing-Maschine werden daher nur noch zwei Akzeptierzustände

$$q_{\text{accept}} \quad \text{und} \quad q_{\text{reject}}$$

benötigt. Der Zustand  $q_{\text{accept}}$  signalisiert, dass die Verarbeitung erfolgreich abgeschlossen wurde, während  $q_{\text{reject}}$  einen Misserfolg anzeigen.

### Initialisierung

Die Turing-Maschine muss ihre Arbeit in einem wohldefinierten Zustand aufnehmen. Dazu muss der Startzustand  $q_0$  definiert sein. Das zu verarbeitende Inputwort muss, wie früher vereinbart, auf das Band geschrieben werden, doch die Maschine muss es dort auch finden können. Daher wird der Schreib-/Lesekopf zu Beginn der Verarbeitung auf das erste Zeichen des Wortes positioniert.

Mit dem Inputwort auf dem Band und dem Schreib-/Lesekopf am richtigen Ort ist es aber noch nicht getan. Wie kann die Maschine das Ende des Inputwortes erkennen? Dies ist nur möglich, wenn die Speicherzellen außerhalb des Inputwortes mit einem Zeichen  $\sqcup$  initialisiert werden, welches nicht im Inputwort vorkommen kann. Wir müssen also das Alphabet  $\Sigma$  der Sprache, aus der das Inputwort stammt, vom Bandalphabet  $\Gamma$  unterscheiden. Die Differenz  $\Gamma \setminus \Sigma$  muss mindestens das Zeichen  $\sqcup$  enthalten.

### Bandzustände erweitern die Möglichkeiten

Da die Turing-Maschine den Bandinhalt verändern kann, hängt die weitere Verarbeitung nicht nur von den Zuständen der Steuerung ab. Man kann sich zum Beispiel auch vorstellen, die Zeichen  $0$  und  $1$  als eine Binärzahl zu interpretieren und damit einen Zähler

zu implementieren. Dazu müsste die Maschine jeweils ans Ende der Nullen und Einsen fahren und so lange nach links fahrend Einsen in Nullen verwandeln, bis eine Null oder ein  $\sqcup$  angetroffen wird, die in eine Eins zu verwandeln sind. So ein binärer Zähler kann unendlich lange arbeiten und immer wieder neue Bandinhalte produzieren. Die Möglichkeit, das Band zu überschreiben, gibt der Turing-Maschine unendlich viele verschiedene Zustände.

Die Möglichkeit der Modifikation des Bandes kann auch dazu verwendet werden, anzuzeigen, welche Zeichen bereits verarbeitet sind. Man kann sie zum Beispiel mit  $\sqcup$  oder sonst einem Zeichen  $x$  überschreiben.

---

**Verständniskontrolle 10.2:** Beschreiben Sie den Pseudocode, mit dem man nach den vereinbarten Spielregeln ein Wort  $w$  auf dem Band daraufhin testen kann, ob  $|w|_0 = |w|_1$  ist.



#### 10.1.4 Formale Definition

Die Diskussion in den Abschnitten 10.1.1 und 10.1.3 hat die Anforderungen an die Definition einer Turing-Maschine zusammengetragen und führt uns jetzt auf die folgende Definition.

**Definition 10.1** (Turing-Maschine). *Eine Turing-Maschine ist ein 7-Tupel*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

wobei  $\delta$  die Übergangsfunktion

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\} : (q, a) \mapsto (q', a', r)$$

ist. Das Bandalphabet  $\Gamma$  muss das Zeichen  $\sqcup$  enthalten, welches nicht im Inputalphabet  $\Sigma$  vorkommt:  $\sqcup \in \Gamma \setminus \Sigma$ .

Zur Verarbeitung eines Wortes  $w \in \Sigma^*$  durch die Turing-Maschine  $M$  wird das Inputwort auf das mit  $\sqcup$  initialisierte Band geschrieben, der Schreib-/Lesekopf auf das erste Zeichen positioniert und die Maschine gestartet. Die Maschine hält, sobald einer der beiden Akzeptierzustände  $q_{\text{accept}}$  oder  $q_{\text{reject}}$  erreicht wird.

**Definition 10.2** (Akzeptieren eines Wortes, erkannte Sprache). *Die Turing-Maschine  $M$  akzeptiert das Wort  $w$ , wenn sie bei der Verarbeitung im Zustand  $q_{\text{accept}}$  anhält. Sie verwirft das Wort, wenn sie im Zustand  $q_{\text{reject}}$  anhält. Die Menge*

$$L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$$

*heißt die von  $M$  erkannte Sprache.*

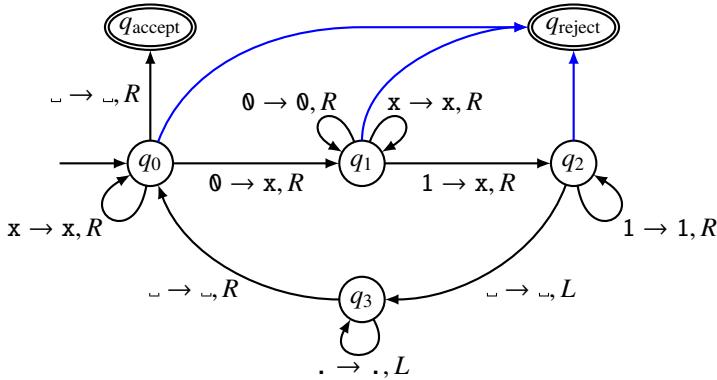


Abbildung 10.6: Turing-Maschine, die Wörter der Form  $0^n 1^n$ ,  $n \geq 0$  erkennt. Die blauen Übergänge werden für Bandzeichen genommen, für die es keine anderen Übergänge gibt. Im Zustand  $q_0$  ist das zum Beispiel der Übergang für ein 1 auf dem Band.

Man beachte, dass die Definitionen nicht verlangen, dass eine Turing-Maschine immer anhalten muss. Man kann sehr leicht Maschinen konstruieren, die niemals anhalten und manchmal passiert das in der täglichen Programmierpraxis ganz unabsichtlich und nennt sich Endlosschleife. Die Sprache  $L(M)$  umfasst aber nach Definition immer nur Wörter, auf denen  $M$  tatsächlich anhält.

### 10.1.5 Beispiele von Turing-Maschinen

Die Sprache

$$L = \{1^n * 1^m = 1^{nm} \in \{1, *, =\}^*\}$$

beschreibt korrekte Multiplikationen in unärer Darstellung. Sie ist weder regulär noch kontextfrei, wie man durch Anwendung des Pumping-Lemma zeigen kann.  $L$  illustriert, dass endliche Automaten und Stackautomaten nicht wirklich rechnen können. Wenn das Konzept Turing-Maschine in die Nähe der Fähigkeiten eines Computers kommt, dann müsste es eine Turing-Maschine geben, die  $L$  erkennen kann. Die nachfolgenden Beispiele sollen illustrieren, dass Turing-Maschinen tatsächlich viel leistungsfähiger sind.

#### Turing-Maschine für die Sprache $0^n 1^n$

Die nicht reguläre Sprache  $\{0^n 1^n \mid n \geq 0\}$  kann von einem Stackautomaten akzeptiert werden, es wäre also denkbar, einen Stackautomaten mit einer Turing-Maschine nachzubilden. Es ist aber nicht so offensichtlich, wie das gehen soll, weil der Stack irgendwo auf dem gleichen Band abgespeichert werden müsste, wo sich auch schon das zu prüfende Wort befindet. Wir gehen daher einen anderen Weg.

Die Turing-Maschine muss zwei Dinge feststellen: Zu jedem 0 gibt es eine 1 und nach den Einsen kommen keine Nullen mehr. Dabei muss sich die Turing-Maschine irgendwie merken, welche Nullen und Einsen bereits gezählt worden sind. Da der Inhalt des Bandes nicht mehr gebraucht wird, kann jedes verarbeitete Zeichen mit x überschrieben werden.

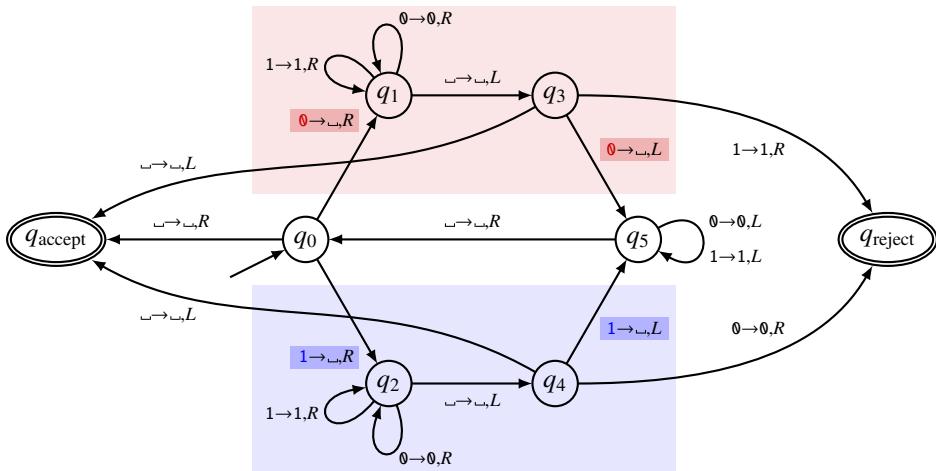


Abbildung 10.7: Zustandsdiagramm einer Turing-Maschine, die Palindrome über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  erkennt. Die Zustände im roten Bereich dienen dazu, zwei zusammengehörende  $\emptyset$  am Anfang und Ende des Wortes zu finden. Die hervorgehobenen Übergänge zeigen die beiden Nullen. Der blaue Bereich zeigt die entsprechenden Übergänge zum Detektieren zusammengehöriger Einsen.

Der folgende Pseudocode realisiert diese Idee:

1. Erste  $\emptyset$  durch  $x$  ersetzen.
2. Nach rechts fahrend alle  $\emptyset$  und  $x$  überspringen, bis die erste 1 gefunden wird.
3. 1 durch  $x$  ersetzen.
4. Nach rechts fahrend alle 1 überspringen bis  $\sqcup$  gefunden wird.
5. Nach links fahren bis zum Wortanfang.
6. Weiter bei 2.

Wenn im Schritt 4 eine  $\emptyset$  angetroffen wird, dann stimmt die Reihenfolge der Nullen und Einsen nicht und das Wort wird mit  $q_{\text{reject}}$  verworfen.

Das Zustandsdiagramm von Abbildung 10.6 implementiert den Pseudocode. Der Übergang von  $q_0$  nach  $q_1$  erkennt die nächste zu verarbeitende  $\emptyset$  und ersetzt sie durch  $x$ . Im Zustand  $q_1$  werden  $\emptyset$  und  $x$  übersprungen, erst bei einer 1 folgt der Übergang in den Zustand  $q_2$ . Wird das Wortende vorher gefunden, erfolgt der Übergang in den Zustand  $q_{\text{reject}}$ . Im Zustand  $q_2$  fährt die Turing-Maschine weiter bis ans Ende des Wortes, wobei nur 1 akzeptabel sind. Am Wortende geht die Maschine in den Zustand  $q_3$ , der dazu verwendet wird, an den Wortanfang zurückzukehren, erkennbar an der Kopfbewegung nach links. Dann beginnt der nächste Durchgang, der wieder ein Paar  $\emptyset$  und 1 findet.

## Turing-Maschine für Palindrome

Mit einem Stackautomaten war es in Abschnitt 7.1.4 möglich, die Sprache der Palindrome zu akzeptieren. Allerdings war der Stackautomat notwendigerweise nichtdeterministisch und es war unmöglich, mit einem Parsergenerator einen Parser für Palindrome zu generieren. Mit der Turing-Maschine von Abbildung 10.7 wird es jetzt möglich, Palindrome deterministisch zu erkennen. Dazu muss sie jeweils die Zeichen an beiden Enden des Wortes vergleichen und dann löschen. Stimmen die Zeichen nicht überein, liegt kein Palindrom vor und die Maschine geht in den Zustand  $q_{\text{reject}}$ . Falls auf diese Weise das ganze Wort gelöscht werden kann, liegt ein Palindrom vor und die Maschine geht in den Zustand  $q_{\text{accept}}$ .

## Binäre Addition

In der Schule lernt man den Algorithmus der schriftlichen Addition für mehrstellige Zahlen. Dieser ist natürlich auch für Binärzahlen anwendbar, zum Beispiel wird die Summe  $1291 + 719$  binär von rechts beginnend durch

$$\begin{array}{r}
 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \\
 + 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \\
 \hline
 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0
 \end{array}$$



[autospr.ch/video/1](http://autospr.ch/video/1)

berechnet. Die kleinen roten Ziffern bedeuten, dass an dieser Stelle ein Übertrag von der Stelle weiter rechts zu berücksichtigen ist.

Der Algorithmus der binären Addition lässt sich mit einer Turing-Maschine implementieren. Wir nehmen dazu an, dass zu Beginn die beiden Summanden durch genau ein Leerzeichen getrennt nebeneinander auf dem Band stehen. Damit sind die Voraussetzungen für den folgenden Turing-Maschinenalgorithmus geschaffen, der auch dem verlinkten Turing-Simulator als Beispiel beigefügt ist. Der Algorithmus geht wie folgt vor:



[autospr.ch/c/5](http://autospr.ch/c/5)

1. Fahre ans Ende des zweiten Wortes und verschiebe das zweite Wort Zeichen für Zeichen eine Speicherzelle nach rechts, indem jeweils ein Zeichen gelesen, durch ein Leerzeichen überschrieben und dann ins Feld rechts daneben geschrieben wird. Nach diesem Schritt gibt es zwei Zeichen Platz zwischen den beiden Wörtern.
2. Fahre ans Ende des zweiten Wortes, lösche das Zeichen dort und fahre ans Ende des ersten Wortes, um das letzte Zeichen des ersten Summanden zu holen und das Zeichen zu löschen. Dadurch sind jetzt drei Leerzeichen Platz entstanden. Schreibe die Summe in das mittlere leere Feld, merke Dir den Übertrag.
3. Fahre fort, jeweils das letzte Zeichen beider Summanden zu löschen, ihre Summe samt Übertrag zu berechnen und vorne an die Summe anzufügen, bis beide Summanden vollständig verarbeitet sind.
4. Die Summe der beiden Zahlen steht jetzt auf dem Band.

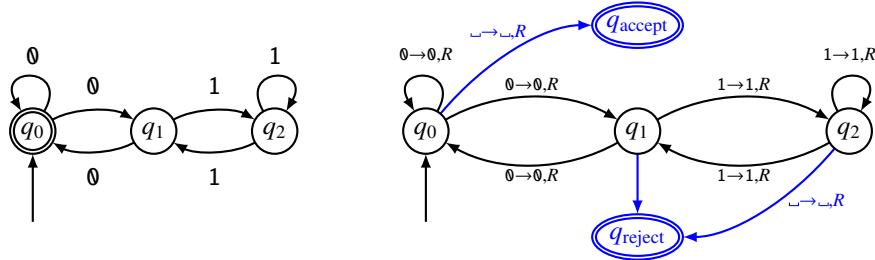


Abbildung 10.8: Verwandlung eines DEA für durch drei teilbare Binärzahlen in eine Turing-Maschine. Jeder Übergang für ein Zeichen  $a$  muss in einen Übergang der Form  $a \rightarrow a, R$  verwandelt werden. Die hinzugefügten blauen Übergänge zu den Akzeptanzzuständen sind alle mit  $u \rightarrow u, R$  beschriftet.

Das verlinkte Video zeigt den Gang der Berechnung mit einem Modell einer Turing-Maschine, die Mike Davey gebaut hat [10].

### Unäre Multiplikation

Die unäre Darstellung (Seite 210) einer Zahl verwendet ein Alphabet mit nur einem Zeichen  $\Sigma = \{1\}$ . Die Zahl  $n \in \mathbb{N}$  wird dargestellt durch das Wort  $1^n$ , also  $n$  aufeinanderfolgende Ziffern 1. In der unären Darstellung sind Additionen besonders einfach auszuführen, die Wörter werden einfach aneinandergehängt.



Die Multiplikation in der unären Darstellung muss das zweite Wort so oft kopieren, wie der erste Faktor mit der Anzahl seiner Stellen angibt. Im verlinkten Video von William Lewis [27] ist dies durchgeführt.

### Turing-Maschine für einen DEA

Jeder deterministische endliche Automat lässt sich in eine Turingmaschine umwandeln. Die Turing-Maschine verwendet im Wesentlichen das Zustandsdiagramm des endlichen Automaten, die Zeichen werden aber vom Band gelesen, sie müssen nicht verändert werden, die Kopfbewegung ist immer nach rechts. Außerdem muss die Maschine detektieren, ob das Wortende erreicht wurde und ob der erreichte Zustand ein Akzeptanzzustand ist. Der Übergang zu  $q_{\text{accept}}$  erfolgt also von Akzeptanzzuständen des DEA in dem Moment, wo ein  $u$  vom Band gelesen wird. Ebenso erfolgt der Übergang zu  $q_{\text{reject}}$  aus Nichtakzeptanzzuständen, wenn  $u$  gelesen wird. Abbildung 10.8 illustriert die Transformation für den DEA, der durch drei teilbare Binärzahlen akzeptiert.

#### 10.1.6 Berechnungsgeschichte

Der Gang der Berechnung mit einer Turing-Maschine kann dokumentiert werden, indem der Bandinhalt nach jedem einzelnen Verarbeitungsschritt protokolliert wird. In Abbil-

$w = 0110$	$w = 01010$	$w = 01000$
$q_0 \ 0$	$q_0 \ 0 \ 1 \ 0 \ 1 \ 0$	$q_0 \ 0 \ 1 \ 0 \ 0 \ 0$
$q_1 \ 1$	$q_1 \ 1 \ 1 \ 0 \ 1 \ 0$	$q_1 \ 1 \ 0 \ 0 \ 0 \ 0$
$1 \ q_1 \ 1$	$1 \ q_1 \ 0 \ 1 \ 0$	$1 \ q_1 \ 0 \ 0 \ 0$
$1 \ 1 \ q_1 \ 0$	$1 \ 0 \ q_1 \ 1 \ 0$	$1 \ 0 \ q_1 \ 0 \ 0$
$1 \ 1 \ 0 \ q_1$	$1 \ 0 \ 1 \ q_1 \ 0$	$1 \ 0 \ 0 \ q_1 \ 0$
$1 \ 1 \ q_3 \ 0$	$1 \ 0 \ 1 \ 0 \ q_1$	$1 \ 0 \ 0 \ 0 \ q_1$
$1 \ q_5 \ 1$	$1 \ 0 \ 1 \ q_3 \ 0$	$1 \ 0 \ 0 \ q_3 \ 0$
$q_5 \ 1$	$1 \ 0 \ q_5 \ 1$	$1 \ 0 \ q_5 \ 0 \ 0$
$q_0 \ 1$	$1 \ q_5 \ 0 \ 1$	$q_5 \ 1 \ 0 \ 0$
$q_2 \ 1$	$q_0 \ 1 \ 0 \ 1$	$q_5 \ 1 \ 0 \ 0$
$1 \ q_2$	$q_2 \ 0 \ 1$	$q_0 \ 1 \ 0 \ 0$
$q_4 \ 1$	$q_2 \ 0 \ 1$	$q_2 \ 0 \ 0 \ 0$
$q_5 \ q_0$	$0 \ q_2 \ 1$	$0 \ 0 \ q_2 \ 0$
$q_0 \ q_a$	$0 \ 1 \ q_2$	$0 \ 0 \ q_4 \ 0$
	$0 \ q_3$	$0 \ 0 \ q_r$
	$q_a$	

Abbildung 10.9: Berechnungsgeschichte zum Erkennen von Palindromen mit der Turing-Maschine von Abbildung 10.7. In der letzten Zeile findet man jeweils die Zustände  $q_a = q_{\text{accept}}$  bzw.  $q_r = q_{\text{reject}}$ , die bewirken, dass die Maschine anhält. Ganz rechts die Verarbeitung eines Wortes, welches wegen der blauen 0 kein Palindrom ist. Wie erwartet hält die Maschine im Zustand  $q_{\text{reject}}$ , wenn sie im Zustand  $q_4$  auf die Null statt die erwartete 1 trifft.

dung 10.9 ist dies für die Berechnung der Palindrom-Turing-Maschine von Abbildung 10.7 für drei verschiedene Wörter durchgeführt.

Um auch den Zustand der Maschine darstellen zu können, wird das Symbol für den aktuellen Zustand unmittelbar vor dem Zeichen, auf das der Schreib-/Lesekopf zeigt, eingefügt. In Abbildung 10.9 ist das Zustandssymbol zur Verdeutlichung zusätzlich rot eingefärbt. Die zu einer Berechnung gehörende Liste von Zeichenketten heißt die *Berechnungsgeschichte*.

Der zusätzliche Platz, den das Zeichen für den Zustand beansprucht, hat zur Folge, dass die Bandzeichen nicht immer in der gleichen Spalte stehen. Die grünen Spuren in Abbildung 10.9 folgen den Zeichen durch die Berechnungsgeschichte. Sie weichen immer dann nach rechts oder links ab, wenn der Schreib-/Lesekopf die Zeichenposition nach links oder rechts kreuzt.

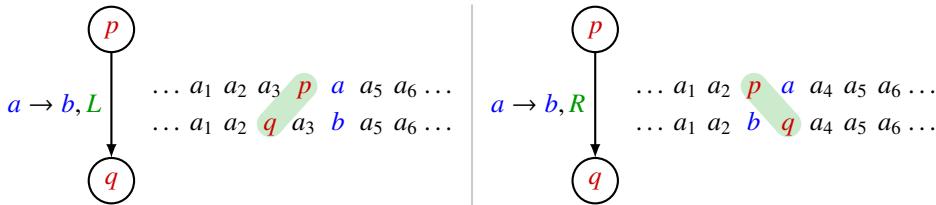


Abbildung 10.10: Zustandsübergänge der Turing-Maschine (links) und die entsprechenden Veränderungen in der Berechnungsgeschichte (rechts). In einem Schritt der Berechnungsgeschichte ändert sich nur das Zustandssymbol, es verschiebt sich um höchstens eine Stelle und es verändert sich nur das Bandalphabetzeichen, welches in der oberen Zeile rechts vom Zustandssymbol steht.

*Verständniskontrolle 10.3:* Stellen Sie die Berechnungsgeschichten auf, nach denen die Turing-Maschine für die Sprache  $L = \{\varnothing^n 1^n \mid n \geq 0\}$  mit Zustandsdiagramm in Abbildung 10.6



- a) das Wort **0011** akzeptiert und
- b) das Wort **00111** verwirft.

Die Berechnungsgeschichte ermöglicht, den Gang der Berechnung mithilfe von Vergleichen von Zeichenketten zu analysieren. Zwei aufeinanderfolgende Zeilen unterscheiden sich nur in unmittelbarer Umgebung des Symbols für den Zustand (Abbildung 10.10). Das Zustandssymbol verschiebt sich nur ein Feld nach links oder rechts und kann einen neuen Wert annehmen. Außerdem kann nur das Bandalphabetsymbol verändert werden, welches rechts vom Zustandssymbol in der Ausgangszeile steht.

## 10.2 Varianten von Turing-Maschinen

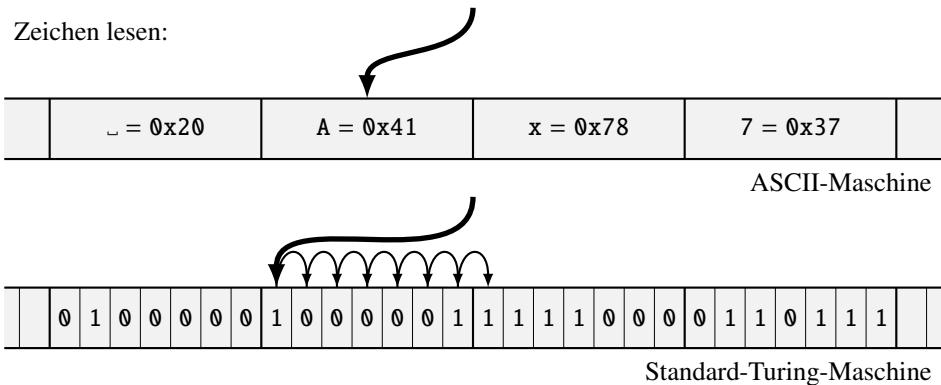
Die bisher vorgestellte Grundform einer Turing-Maschine ist noch ziemlich weit von der Architektur heutiger Computer entfernt. In diesem Abschnitt sollen daher verschiedene Varianten von Turing-Maschinen untersucht werden. Die entscheidende Frage dabei ist, ob sich die Fähigkeiten der Turing-Maschine dadurch wesentlich ändern. Gibt es Aufgabenstellungen, die mit einer primitiveren Maschine nicht gelöst werden können, die durch die Variante lösbar werden? Gibt es Aufgabenstellungen, die die Variante wesentlich schneller lösen kann?

### 10.2.1 Verschiedene Bandalphabete

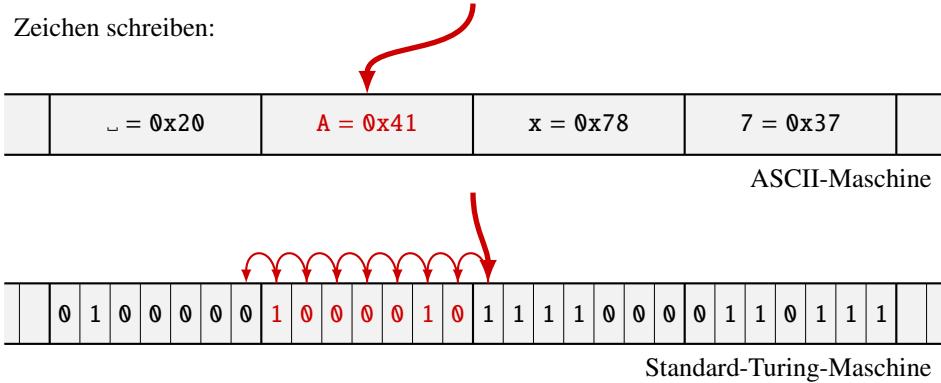
Die von Turing definierte Standard-Turing-Maschine verwendet  $\Sigma = \{\varnothing, 1\}$  als Inputalphabet und  $\Gamma = \Sigma \cup \{\_\}$  als das Bandalphabet. Jedes andere Alphabet kann binär codiert werden, zum Beispiel kann Standard ASCII mit 7 Bit codiert werden.

Eine Turing-Maschine mit ASCII als Bandalphabet kann ein aus 7 Bit zusammengesetztes Zeichen in jedem Schritt lesen. Die Standard-Turing-Maschine muss diese 7 Bit

Zeichen lesen:



Zeichen schreiben:



Bewegung des Schreib-/Lesekopfes:

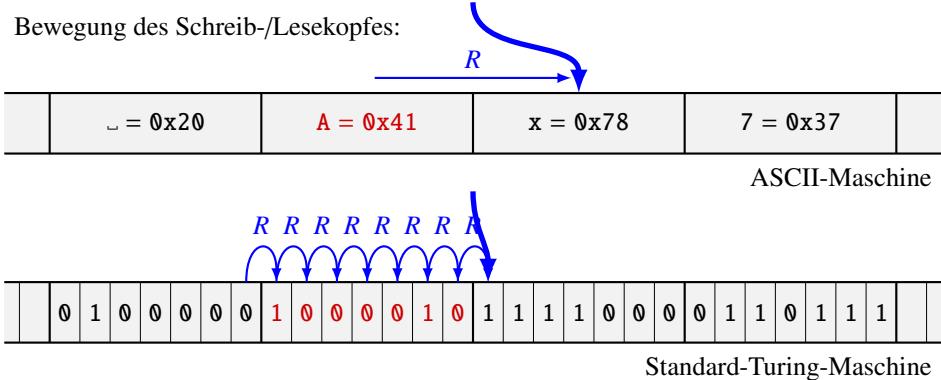


Abbildung 10.11: Verarbeitung des ASCII-Alphabets mit einer Standard-Turing-Maschine. Das Lesen und Schreiben eines Zeichens muss in mehrere Einzelbit-Operationen aufgelöst werden. Die Kopfbewegung am Ende wird zu einer Bewegung um so viele Felder, wie das Alphabet Bits für die Codierung eines Zeichens braucht.

einzelnen Zeichen lesen und in geeigneten Zuständen des endlichen Automaten der Steuerung speichern (Abbildung 10.11). Dazu sind 7 Operationen der Standard-Turing-Maschine nötig. Die Steuerung bestimmt dann die 7 Bit, mit denen die gerade gelesenen Bandzeichen überschrieben werden müssen, was erneut 7 Operationen braucht. Die Kopfbewegung nach links oder rechts muss 7 Bit nach links oder rechts überspringen, um an den Anfang der nächsten Gruppe von 7 Bit zu gelangen. Die Standard-Turing-Maschine kann daher einen einzelnen Schritt der ASCII-Maschine mit immer der gleichen Zahl von Einzelbitoperationen nachbilden. Sie kann also genau die gleichen Aufgaben bewältigen, braucht dazu aber eine um einen konstanten Faktor größere Anzahl Schritte.

### Binär oder Dezimal

Die ersten Computer waren primär dazu gebaut worden, mathematische Berechnung durchzuführen, sie waren programmierbare Rechenmaschinen. Die effiziente Codierung von Zahlen stand dabei im Vordergrund. Die rein binäre Rechnung ist zwar am effizientesten, doch am Ende der Berechnung müssen die Resultate für menschliche Leser dezimal ausgegeben werden können. Insbesondere in kommerziellen Anwendungen, z. B. in der Buchhaltung, stellt die eigentliche Rechnung nur einen kleinen Teil des Aufwandes dar, die Umwandlung für die Ausgabe beansprucht den größten Teil der Arbeit. Es liegt daher nahe, die Codierung der Zahlen so zu wählen, dass die Rechnung immer noch einigermaßen effizient durchführbar, die Darstellung aber speziell einfach wird. Dazu wurde *binary coded decimal* (BCD) verwendet. In BCD wird jede Dezimalstelle mit 4 Bit codiert. Die Zahl 1291 kann als

$$1291_{10} = \underbrace{0001}_1 \underbrace{0010}_2 \underbrace{1001}_9 \underbrace{0001}_1 \text{ BCD}$$

codiert werden.

Die Hardware zur Darstellung von BCD-codierten Daten ist sehr einfach, es gibt Logikbausteine (zum Beispiel SN7445), die die 4 Bit in 10 verschiedene Ausgänge decodieren, mit denen sich zum Beispiel die 10 Elemente einer Nixie-Röhre ansteuern lassen. Oder Bausteine wie der SN7446, die die 4 Bit in die 7 Segmente einer Segmentanzeige decodieren. Die Segmentanzeige wurde 1908 von Frank Wood erfunden und gehört zu den ältesten technischen Anzeigeelementen. Sie standen den Computerpionieren in vielfältiger Form zur Verfügung.

Heutzutage ist der Aufwand für die Datenausgabe meist viel kleiner als die Verarbeitung, so dass die für die Rechnung effizienteste Codierung das Format der Wahl ist. Moderne Computer verwenden daher binär codierte Ganzzahlen mit oder ohne Vorzeichen.

Eine ähnliche Standardisierung hat sich bei den Gleitkommaformaten ergeben. Auch hier dominiert die Norm IEEE 754, welche sowohl die Darstellung wie auch die arithmetischen Operationen für binäre Gleitkommazahlen definiert. In diesen Formaten gespeicherte Zahlen sind zwischen verschiedenen Maschinen austauschbar. Überlauf, Unterlauf und Rundung werden damit einheitlich behandelt. Moderne CPUs implementieren die Arithmetik nach IEEE-754, so dass man unabhängig von der CPU die gleichen Resultate erwarten darf.

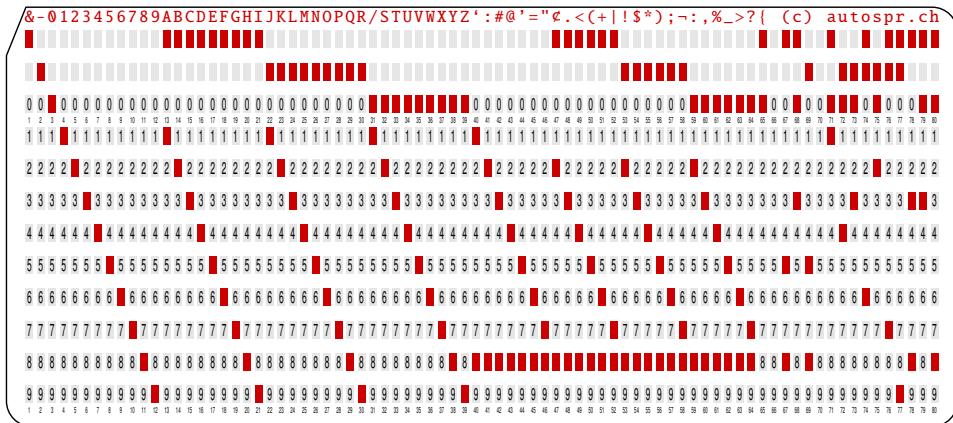


Abbildung 10.12: 80-Zeichen Lochkarte, deren Format IBM im Jahre 1928 eingeführt hatte. Die in diesem Beispiel gezeigte EBCDIC-Codierung kommt mit drei Löchern in jeder Spalte aus. Eine rein binäre Codierung würde Spalten mit vielen Löchern beinhalten und damit die mechanische Stabilität der Karte komprimieren.

### ASCII oder EBCDIC

Das ASCII-Format für die Codierung von Textzeichen entstand ursprünglich aus den Bedürfnissen der Telekommunikationsindustrie, es entwickelte sich aus den für Fernschreiber entwickelten Verfahren für die Übertragung von Text. Als traditionsreiche und standardisierte Codierung bietet sich ASCII als Bandalphabet einer Turing-Maschine an.

Wie bei den Zahlendarstellungen haben auch bei der Textdarstellung andere Use-Cases alternative Codierungen bevorzugt. Zum Beispiel hatte IBM im Jahr 1928 die 80 spaltige Lochkarte auf den Markt gebracht, die bis in die 70er Jahre die elektronische Datenverarbeitung prägte. Diese Karte verwendete für Zeichencodierung nicht einen binären Code, da durch Codewörter mit vielen Löchern die Karte mechanisch stark geschwächt und damit die Zuverlässigkeit eingeschränkt würde. Die in Abbildung 10.12 gezeigte Codierung verwendet ein Loch zur Codierung von Ziffern, genau zwei Löcher für Buchstaben und höchstens drei Löcher für Sonderzeichen. Wir ein Computer vor allem für die Verarbeitung von auf Lockkarten gespeicherten Daten konstruiert, ist es daher angezeigt, einen einfach zum Lochkartenformat konvertierbaren Code zu verwenden. Daher verwenden viele IBM-Systeme den aus der Lochkartencodierung abgeleiteten Extended Binary Coded Decimal Interchange Code (EBCDIC) als Alphabet.

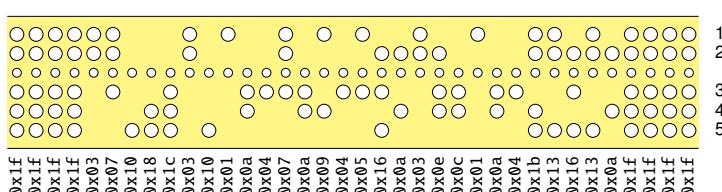
### 10.2.2 Mehrspurmaschine

Lochstreifen wurden ursprünglich für die Speicherung von Fernschreiberübermittlungen verwendet, fanden aber auch Eingang in die Datenverarbeitung. Die Lochstreifentypen in Abbildung 10.13 verwenden jeweils eine Taktspur mit kleinen Löchern und mehrere Datenspuren mit größeren Löchern. Jedes Loch codiert ein Bit, die Bits werden parallel gelesen und als ein Zeichen eines breiteren Alphabets interpretiert. Fernschreiber verwenden

## Inhalt:

shift AUTOMATEN UND SPRACHEN shift 2024

## 5 Spuren CCITT-2



## Inhalt:

Automaten und Sprachen 2024

## 8 Spuren ASCII gerade Parität

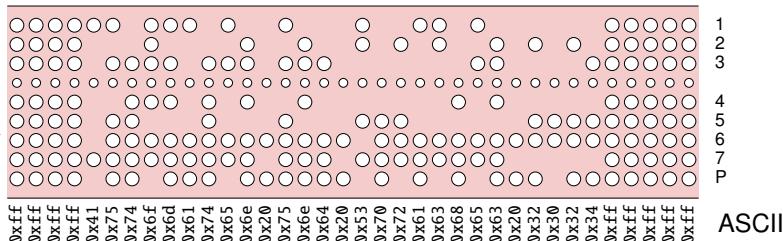


Abbildung 10.13: Die Löcher eines Lochstreifens bilden ein einfaches Alphabet, mit dem Daten auf dem Lochstreifen gespeichert werden können. Der gelbe Lochstreifen hat fünf Datenspuren, in denen Text mit dem CCITT-2 Alphabet codiert wird. Der rote hat acht Spuren und verwendet ASCII als Codierung.

nur ein mit 5 Bit codiertes Alphabet, ein fünfspuriger Lochstreifen reicht aus. Für Datenverarbeitungsanwendungen ist ein Format mit einem achtspurigen Streifen einfacher, in jeder Lochreihe auf dem Streifen ist ein Byte codiert.

## Magnetbandsysteme

Magnetische Speichermedien sind viel flexibler, sie können im Gegensatz zu Lochstreifen beliebig oft beschrieben werden, wie es für die Realisierung einer Turing-Maschine erforderlich ist. Schon früh war klar, dass man das Bandmaterial besser ausnutzen und die Transfergeschwindigkeiten steigern kann, wenn man mehrere parallel montierte Schreib-/Leseköpfe verwendet. Die IBM 2401 Bandstation aus den sechziger Jahren verwendeten neuen Spuren auf einem 0.5" (12.7 mm) breiten Band (Abbildung 10.14), acht Datenspuren und eine Paritätsspur. Die parallel geschriebenen Bits werden als ein Byte interpretiert. Während andere Bandsysteme die Bits seriell schrieben und damit Bytes verschiedener Länge erlaubten, trug das starre Format des IBM 2401 und seine Verbreitung beim Daten-austausch wesentlich dazu bei, dass heute nur noch 8 Bit lange Bytes verwendet werden.

Die Quarter-Inch-Cartridges (QIC) schrieben und lasen Daten dagegen nur mit einem einzigen Schreib-/Lesekopf serpentinartig. Das Band wurde mehrmals hin und her gespult und der Kopf jedes Mal ein Spurbreite senkrecht zur Laufrichtung des Bandes verschoben. Im ursprünglichen QIC-11 Format wurden vier Spuren. Moderne LTO Bandsysteme verwenden beide Verfahren kombiniert. Von Tausenden von Spuren werden serpentinartig von Schreib-/Leseköpfen mit 8 oder 16 Elementen 8 oder 16 Spuren parallel



Abbildung 10.14: 9-Track-Band, wie es von IBM mit der IBM 2401 Bandmaschine für das IBM System/360 eingeführt worden ist. Die Spule mit 11.5" (27.67 cm) Durchmesser kann 2400' oder etwa 730 m Magnetband aufnehmen. Auf dem 0.5" (12.7 mm) breiten Band werden die acht Bits eines Bytes und ein Paritätsbit parallel auf neun Spuren mit einer Dichte von 1600 Bits pro Zoll (bpi) geschrieben. Damit können auf einer Spule 40 MB gespeichert werden. Rechts sind die Datenspuren mit einem magnetischen Entwickler sichtbar gemacht worden.

gelesen oder beschrieben.

### Memorybusbreite

Auch das Band einer Turing-Maschine kann man sich als mehrspurig vorstellen. Der Schreib-/Lesekopf der Turing-Maschine kann gleichzeitig mehrere Zeichen lesen oder schreiben. Betrachtet man das Band als den Hauptspeicher des Computers, begann der Einsatz von Mikroprozessoren mit CPUs, die in jedem Speicherzyklus ein einzelnes Byte lesen oder schreiben konnten. Später wurde dies erweitert, die Hardware war jetzt in der Lage, 2 (16 Bit), 4 (32 Bit) oder 8 Bytes (64 Bit) in einer Operation zu lesen. Dies entspricht der Erweiterung der Spurzahl der ursprünglichen Maschine auf 2, 4 oder 8 Spuren.

### Spuren und die Fähigkeiten der Turing-Maschine

Die Spurzahl hat keinen Einfluss auf die Fähigkeiten der Turing-Maschine. Die Maschine mit der größeren Spurzahl liest breitere Wörter, dies entspricht einem größeren Bandalphabet. Die Größe des Bandalphabets spielt keine Rolle, da alle Bandalphabete letztlich auch binär codiert werden können, wie in Abschnitt 10.2.1 dargestellt.

### 10.2.3 Einseitig unendliches Band

Der Hauptspeicher einer CPU ist nicht beidseitig unendlich. Vielmehr werden die Speicherzellen durch Adressen adressiert, die bei 0 beginnen, ansonsten aber so große Werte erreichen, die man für die meisten praktischen Zwecke als beliebig groß ansehen kann. Der Hauptspeicher ist also ein einseitig unendlich großer Speicher. Auch Realisierungen

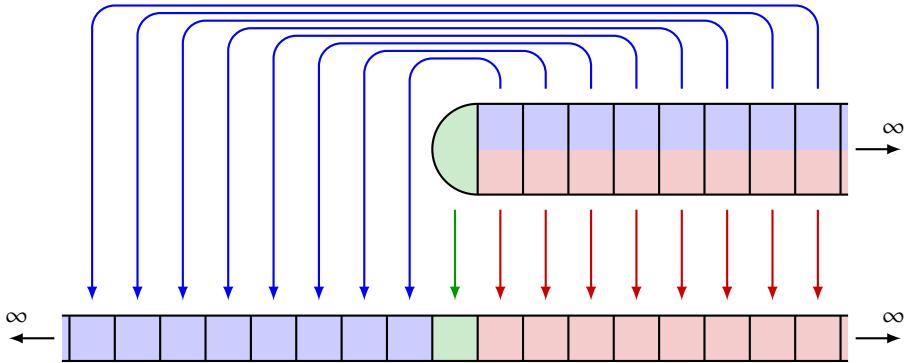


Abbildung 10.15: Ein beidseitig unendliches Band kann mit einem einseitig unendlichen Band mit doppelter Wortbreite realisiert werden. Dies zeigt, dass ein einseitig unendliches Band keine Einschränkung für eine Turing-Maschine ist.

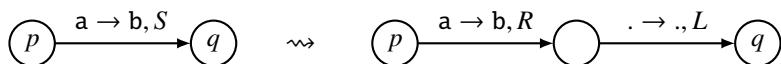
mit Magnetbändern in der Frühzeit der Datenverarbeitung gehen von einem Band aus, welches einen klar definierten Anfang hat.

Abbildung 10.15 zeigt, wie ein beidseitig unendliches Band mit einem einseitig unendlichen Band mit einem Alphabet doppelter Wortbreite realisiert werden kann. Turing-Maschinen mit einseitig unendlichem Band können daher alle Probleme auch lösen, die Turing-Maschinen mit beidseitig unendlichem Band lösen können.

#### 10.2.4 Zusätzliche Kopfbewegungen

Die Standard-Turing-Maschine bewegt den Schreib-/Lesekopf nach jeder Operation nach links oder nach rechts. Es ist wie auf Seite 221 diskutiert nicht vorgesehen, den Schreib-/Lesekopf nach einer Lese- oder Schreiboperation nicht zu bewegen. Wenn sich der Schreib-/Lesekopf nicht bewegt, dann erfolgt die nächste Operation auf der Basis der Zeichen, die der Steuerung bereits bekannt sind. Sie könnten daher auch in den vorangegangenen Schritt integriert werden. Alle Schritte, die den Schreib-/Lesekopf nicht bewegen, können zusammengefasst und in einem Schritt ausgeführt werden, zusammen mit dem nächsten Schritt, der den Schreib-/Lesekopf bewegt.

Umgekehrt kann eine Standard-Turingmaschine Übergänge, die den Schreib-/Lesekopf nicht bewegen, wie folgt simulieren. Sie kompensiert die unvermeidliche Kopfbewegung am Ende eines Übergangs durch einen zweiten Übergang, der den Bandinhalt nicht verändert, aber den Kopf in die andere Richtung bewegt. Bezeichnet man einen Übergang, der den Schreib-/Lesekopf nicht bewegt, mit dem neuen Kopfbewegungssymbol  $S$  für Stillstand, kann der Übergang  $a \rightarrow b, S$  mit der Standard-Maschine wie im Zustandsdiagramm



rechts realisiert werden. Darin ist  $. \rightarrow .$  zu lesen als ein Übergang, der für jedes Bandzeichen möglich ist und es nicht verändert.

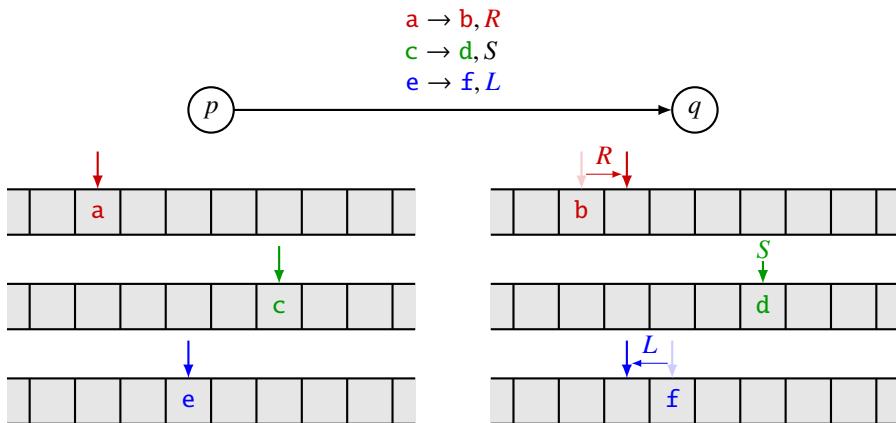


Abbildung 10.16: Übergang einer Multiband-Maschine mit drei Bändern. Der Übergang erfolgt, wenn die vorausgesetzten Zeichen in den Zellen, auf die die Schreib-/Leseköpfe zeigen, vorhanden sind. Die Zeichen werden überschrieben und die Köpfe unabhängig voneinander bewegt.

Während die Kopfbewegung  $S$  für eine Turing-Maschine mit nur einem Band nicht sinnvoll ist, wird sie für die im nächsten Abschnitt beschriebenen Mehrbandmaschinen wichtig.

### 10.2.5 Mehrbandmaschine

Die kommerzielle elektronische Datenverarbeitung mit Universalcomputern entwickelte sich schrittweise aus Maschinen, die Buchhaltungsoperationen basierend auf den auf Lochkarten gespeicherten Informationen durchführen konnten. Diese verfügten weder über Speicher noch waren sie wirklich programmierbar. Sie automatisierten vor allem die Aufbereitung von Berichten oder den Druck von Rechnungen und anderen Dokumenten. Auch frühe Datenverarbeitungsanlagen verfügten nur über sehr beschränkten Hauptspeicher und waren daher auf Algorithmen angewiesen, die Daten vor allem von Kartenstapeln oder später von Band lesen, sofort verarbeiten und die Resultate gleich wieder auf Band schreiben können. Es war also normal, mit mehreren Bandgeräten parallel zu arbeiten. Auch heute kann es bei der Verarbeitung großer Datenmengen vorteilhaft sein, mehrere Files auf verschiedenen Disks zu lesen und zu schreiben.

Im Gegensatz dazu verwendet die Grundform der Turing-Maschine nur ein einziges Band. Ändert sich die Leistungsfähigkeit einer Turing-Maschine, wenn man sie mit mehreren Bändern und zugehörigen, unabhängig voneinander steuerbaren Schreib-/Leseköpfen ausstattet?

#### Übergänge einer Mehrbandmaschine

Ein Übergang einer Turing-Maschine mit drei Bändern ist in Abbildung 10.16 dargestellt. Damit der Übergang von  $p$  nach  $q$  ausgeführt werden kann, müssen die Leseköpfe auf allen Bändern die spezifizierten Zeichen vorfinden. Jede Bandzelle wird dann durch das

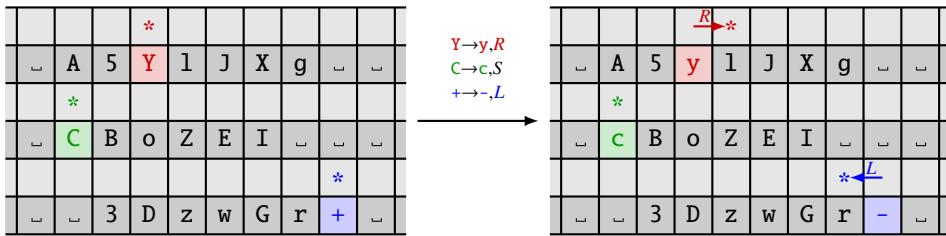


Abbildung 10.17: Simulation eines Übergangs einer Turing-Maschine mit drei Bändern auf einer Turing-Maschine mit einem sechsspurigen Band. Die hellen Spuren werden für die Markierungen verwendet, die die Position des Schreib-/Lesekopfes anzeigen.

neue Zeichen überschrieben und jeder Schreib-/Lesekopf wird nach links oder rechts verschoben, oder er bleibt stehen.

### Simulation einer Mehrbandmaschine auf einer Standardmaschine

Um eine Mehrbandmaschine mit  $n$  Bändern auf einer einfacheren Maschine zu simulieren, muss die unabhängige Bewegung der Schreib-/Leseköpfe nachgebildet werden. Dies kann mithilfe eines  $2n$ -spurigen Bandes geschehen. Die einen Spuren enthalten die Daten, die anderen Spuren einen Zeiger, der die Position des simulierten Schreib-/Lesekopfes für dieses Band darstellt (Abbildung 10.17).

Zur Simulation eines Schrittes muss die Maschine erst das Band von links nach rechts durchlesen, die Steuerung muss sich dabei die Zeichen merken, die bei den Zeigerpositionen gefunden wurden. Daraus können die neuen Zeichen ermittelt werden, die in die Zellen geschrieben werden müssen. Die Maschine fährt dann von rechts nach links nochmals durch das Band, modifiziert die Zelleninhalte bei den Zeigerpositionen und verschiebt die Markierungen nach rechts bzw. links, wie von den Übergängen verlangt.

### Harvard- und von Neumann-Architektur

Die Harvard-Architektur für eine Rechenmaschine verwendet verschiedene Datenpfade und Speicher für Instruktionen und Daten. Sie soll auf den *Harvard Mark I* Computer zurückgehen, der auf Relais-Technologie basierte, Instruktionen von Lochstreifen las und Daten in elektromechanischen Zählern speicherte.

Die Harvard-Architektur findet auch heute noch zum Beispiel in Signalprozessoren und Mikrokontrollern Anwendung. Die AVR-Mikrokontroller-Familie von Atmel verwendet intern drei unabhängige Datenpfade für den Zugriff auf den Flash-Speicher, der den Programmcode enthält, das statische RAM, welches zur Laufzeit veränderliche Daten speichert, und auf die Peripheriegeräte. Programmanweisungen können nicht im RAM stehen, sie müssen aus dem Flash-Speicher gelesen werden. Bei Mikrocontrollern, die auf ARM-Prozessorkernen basieren, ist der Datenpfad manchmal vereinheitlicht, d. h. die Instruktionen und die Daten werden über den gleichen Memory-Bus ausgetauscht. Die verschiedenen Speicherbereiche verhalten sich wie verschiedene Bänder einer Turing-Maschine.

Die von Neumann-Architektur, die von allen modernen Allzweckcomputern verwendet wird, vereinheitlichte Daten- und Programmspeicher. Instruktionen werden über einen einheitlichen Datenbus vom Hauptspeicher geschrieben und von dort gelesen. Auch die Peripheriegeräte teilen den gleichen Adressraum, die Datenausgabe erfolgt mit den gleichen Instruktionen wie das Speichern von Daten im Hauptspeicher. Erst durch die von Neumann-Architektur wurde es möglich, Programme von einem Peripheriegerät in den Speicher zu laden und dort auszuführen.

### Speicher-Layout eines Prozesses

Selbst in einem modernen Betriebssystem auf einer Maschine mit von Neumann-Architektur kann man innerhalb eines Prozesses unterschiedliche Speicherbereiche ausmachen.

Ein ausführbares Programm in Linux ist in einem File im *Executable and Linkable Format* (ELF) gespeichert. Dieses besteht aus vielen verschiedenen Abschnitten, die beim Laden in den Adressraum unterschiedlich behandelt werden. Diese Segmente können mit Werkzeugen wie `readelf(1)` [50] decodiert werden. Abbildung 10.18 zeigt die Informationen über das ELF-File `/bin/sleep` einer 32 Bit-ARM-Linux-Distribution. Das `.text`-Segment enthält den ausführbaren Code, den der Compiler aus den Programmanweisungen erzeugt hat. Die Ausführbarkeit wird durch das X-Flag in der Flg-Spalte signalisiert. Das `.data`-Segment enthält veränderliche Daten, die bereits initialisiert sind, während sich in `.rodata` Konstanten befinden. Das `.bss`-Segment reserviert nur Speicherplatz für Variablen, die nicht initialisiert werden müssen. Schreibbarkeit wird durch das W-Flag angezeigt.

Wenn das Programm gestartet wird, erzeugt der Kernel eine neue Prozessinstanz mit einem Adressraum, der ebenfalls in verschiedene Speicherbereiche organisiert wird. Abbildung 10.18 zeigt in der unteren Hälfte auch die Memory-Map des Prozesses, die man im `/proc`-Filesystem finden kann. Die ersten drei Zeilen der Memory-Map enthalten drei Speicherblöcke mit unterschiedlichen Berechtigungen, die von den Flags in der zweiten Spalte ablesbar sind. Das Betriebssystem erlaubt wegen des x-Flags die Ausführung von Instruktionen, die aus diesem Bereich gelesen werden, während dies in den nächsten zwei Blöcken nicht erlaubt ist. Das `.text`-Segment wird in diesen Block geladen.

Der zweite Block enthält read-only-Daten, wegen des Fehlens des w-Flags verbietet das Betriebssystem dem Prozess, dorthin zu schreiben. Ein Schreibversuch führt zu einem *segmentation fault*. Der Block wird mit dem Inhalt des `.rodata`-Segments initialisiert.

Der dritte Block ist lesbar und schreibbar, dort werden die Daten aus dem `.data`-Segment geladen und es wird ein Datenblock von der im `.bss`-Segment spezifizierten Größe mit Nullen initialisiert.

Der vierte Block enthält den Heap, in dem der Prozess dynamisch allozierte Objekte ablegen kann. Das Betriebssystem stellt Methoden zur Verfügung, mit der der Heap-Bereich vergrößert werden kann. Bevor Systeme virtuellen Speicher verwalten konnten, wurde dazu der System call `sbrk(2)` [6] verwendet, der die Obergrenze des für den Heap reservierten Speicherbereiches nach oben verschiebt. Mit virtueller Speicherverwaltung kann der Heapbereich dadurch vergrößert werden, dass neue Speicherblöcke in anschließenden Adressbereichen des Adressraumes eingeblendet werden.

Der Prozess verwendet außerdem eine Anzahl Bibliotheken, im Beispiel von Abbil-

```
afm@mi:~$ readelf -S /bin/sleep
There are 28 section headers, starting at offset 0x6284:
```

## Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]	NULL	PROGBITS	00000000	00000000	00000000	00		0	0	0
[ 1]	.interp	PROGBITS	00010154	000154	000019	00	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	00010170	000170	000020	00	A	0	0	4
[ 3]	.note.gnu.build-i	NOTE	00010190	000190	000024	00	A	0	0	4
[ 4]	.gnu.hash	GNU_HASH	000101b4	0001b4	0001bc	04	A	5	0	4
[ 5]	.dynsym	DYNSYM	00010370	000370	000380	10	A	6	1	4
[ 6]	.dynstr	STRTAB	000106f0	0006f0	000261	00	A	0	0	1
[ 7]	.gnu.version	VERSYM	00010952	000952	000070	02	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	000109c4	0009c4	000040	00	A	6	2	4
[ 9]	.rel.dyn	REL	00010a04	000a04	000040	08	A	5	0	4
[10]	.rel.plt	REL	00010a44	000a44	000170	08	AI	5	22	4
[11]	.init	PROGBITS	00010bb4	000bb4	00000c	00	AX	0	0	4
[12]	.plt	PROGBITS	00010bc0	000bc0	00023c	04	AX	0	0	4
[13]	.text	PROGBITS	00010e00	000e00	003b24	00	AX	0	0	8
[14]	.fini	PROGBITS	00014924	004924	000008	00	AX	0	0	4
[15]	.rodata	PROGBITS	0001492c	00492c	0008f4	00	A	0	0	4
[16]	.ARM.exidx	ARM_EXIDX	00015220	005220	000008	00	AL	13	0	4
[17]	.eh_frame	PROGBITS	00015228	005228	000004	00	A	0	0	4
[18]	.init_array	INIT_ARRAY	00025f00	005f00	000004	04	WA	0	0	4
[19]	.fini_array	FINI_ARRAY	00025f04	005f04	000004	04	WA	0	0	4
[20]	.data.rel.ro	PROGBITS	00025f08	005f08	000004	00	WA	0	0	8
[21]	.dynamic	DYNAMIC	00025f0c	005f0c	0000f0	08	WA	6	0	4
[22]	.got	PROGBITS	00026000	006000	0000c8	04	WA	0	0	4
[23]	.data	PROGBITS	000260c8	0060c8	000050	00	WA	0	0	4
[24]	.bss	NOBITS	00026118	006118	000160	00	WA	0	0	8
[25]	.ARM.attributes	ARM_ATTRIBUTES	00000000	006118	00002f	00		0	0	1
[26]	.gnu_debuglink	PROGBITS	00000000	006148	000034	00		0	0	4
[27]	.shstrtab	STRTAB	00000000	00617c	000108	00		0	0	1

## Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  
L (link order), O (extra OS processing required), G (group), T (TLS),  
C (compressed), x (unknown), o (OS specific), E (exclude),  
y (purecode), p (processor specific)

```
afm@mi:~$ cat /proc/18347/maps
```

00010000-00016000 r-xp 00000000 b3:07 1846	/usr/bin/sleep
00025000-00026000 r-p 00005000 b3:07 1846	/usr/bin/sleep
00026000-00027000 rw-p 00006000 b3:07 1846	/usr/bin/sleep
00efbf000-00fc1000 rw-p 00000000 00:00 0	[heap]
b68f0000-b6b6a000 r--p 0006b000 b3:07 200820	/usr/lib/locale/locale-archive
b6b6a000-b6d6a000 r--p 00000000 b3:07 200820	/usr/lib/locale/locale-archive
b6d6a000-b6ea2000 r-xp 00000000 b3:07 133170	/usr/lib/arm-linux-gnueabihf/libc-2.28.so
b6ea2000-b6eb1000 ---p 00138000 b3:07 133170	/usr/lib/arm-linux-gnueabihf/libc-2.28.so
b6eb1000-b6eb3000 r--p 00137000 b3:07 133170	/usr/lib/arm-linux-gnueabihf/libc-2.28.so
b6eb3000-b6eb4000 rw-p 00139000 b3:07 133170	/usr/lib/arm-linux-gnueabihf/libc-2.28.so
b6eb4000-b6eb7000 rw-p 00000000 00:00 0	/usr/lib/arm-linux-gnueabihf/libc-2.28.so
b6ed2000-b6ed6000 r-xp 00000000 b3:07 155832	/usr/lib/arm-linux-gnueabihf/libarmmmem-v7l.so
b6ed6000-b6ee5000 ---p 00004000 b3:07 155832	/usr/lib/arm-linux-gnueabihf/libarmmmem-v7l.so
b6ee5000-b6ee6000 r--p 00003000 b3:07 155832	/usr/lib/arm-linux-gnueabihf/libarmmmem-v7l.so
b6ee6000-b6ee7000 rw-p 00004000 b3:07 155832	/usr/lib/arm-linux-gnueabihf/libarmmmem-v7l.so
b6ee7000-b6f07000 r-xp 00000000 b3:07 132867	/usr/lib/arm-linux-gnueabihf/ld-2.28.so
b6f15000-b6f17000 rw-p 00000000 00:00 0	
b6f17000-b6f18000 r--p 00020000 b3:07 132867	/usr/lib/arm-linux-gnueabihf/ld-2.28.so
b6f18000-b6f19000 rw-p 00021000 b3:07 132867	/usr/lib/arm-linux-gnueabihf/ld-2.28.so
bec9e000-bebf0fff000 rw-p 00000000 00:00 0	[stack]
bee8e000-bee8f000 r-xp 00000000 00:00 0	[sigpage]
bee8f000-bee90000 r--p 00000000 00:00 0	[vvar]
bee90000-bee91000 r-xp 00000000 00:00 0	[vdso]
fffff0000-fffff1000 r-xp 00000000 00:00 0	[vectors]

Abbildung 10.18: Segmente eines ELF-Programm-Files und allozierte Memory-Blöcke zur Laufzeit.

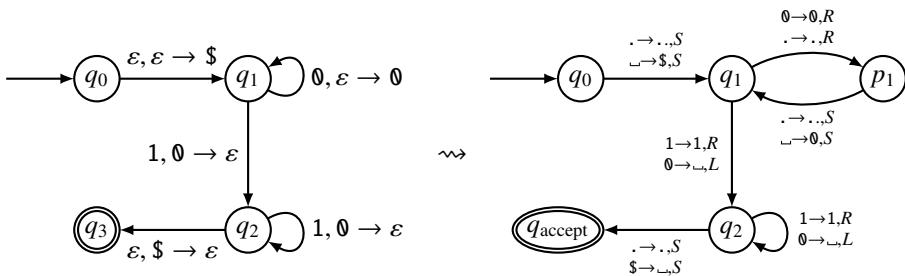


Abbildung 10.19: Umwandlung des Zustandsdiagramms eines Stackautomaten, der die Sprache  $\{0^n 1^n \mid n \geq 0\}$  akzeptiert, in das Zustandsdiagramm einer Turing-Maschine mit zwei Bändern, die den Stackautomaten simuliert. Das obere Band enthält das Inputwort, das untere wird für den Stack verwendet. Die Übergänge der Form  $\cdot \rightarrow \cdot$  bedeuten, dass der Übergang für jedes Bandzeichen möglich ist und dieses nicht verändert.

dung 10.18 die C-Bibliothek `libc`, die Bibliothek `libarmmem-v7l` und den dynamischen Linker, der alle zusätzlichen Bibliotheken lädt und die Adressen in den Text-Segmenten modifiziert, so dass sie auf die Adressen zeigen, auf die die verlangten Objekte geladen worden sind. Jede Bibliothek besteht ebenfalls aus drei Blöcken für Text- und Datensegmente.

Für den Stack des Prozesses wird ein eigener Speicherblock begrenzter Größe angelegt, der in der Abbildung orange hinterlegt ist. Er ist schreib- und lesbar, aber nicht ausführbar. In frühen Betriebssystemen gab es diesen Schutz vor Ausführbarkeit des Stacks nicht, was zu Verwundbarkeiten geführt hat.

Zwischen einigen Blöcken des Prozesses gibt es Adressbereiche, die nicht alloziert sind. Zum Beispiel ist der Adressbereich `0x16000–0x25000` nicht alloziert. Die Bereiche des ausführbaren Codes und der Daten sind also nicht zusammenhängend. Da das Band einer Turing-Maschine einen zusammenhängenden Speicherbereich symbolisiert, kann man die verschiedenen Speicherblöcke als verschiedene Bänder einer Turing-Maschine betrachten. Eine Mehrband-Turing-Maschine ist also ein gutes Modell für die Arbeitsweise eines einzelnen Linux-Prozesses.

### Simulation eines Stackautomaten

Eine Turing-Maschine mit zwei Bändern ist besonders gut dazu geeignet, einen Stackautomaten zu simulieren. Auf dem ersten Band steht das zu verarbeitende Input-Wort, auf dem zweiten Band wird der Stack realisiert. Der Schreib-/Lesekopf des zweiten Bandes zeigt immer auf das oberste Element des Stacks, der nach rechts wächst. Ein Element wird auf den Stack geschrieben, indem der Schreib-/Lesekopf nach rechts bewegt und das neue Zeichen in dieses Feld geschrieben wird. Ein Element wird vom Stack gelesen, indem das aktuelle Feld mit `_` überschrieben und der Schreib-/Lesekopf nach links bewegt wird.

Abbildung 10.19 zeigt links das Zustandsdiagramm eines Stackautomaten, der die Sprache  $L = \{0^n 1^n \mid n \geq 0\}$  akzeptiert. Rechts ist das daraus abgeleitete Zustandsdiagramm für eine Turing-Maschine mit zwei Bändern dargestellt, die den Stackautomaten simuliert. Da der Schreib-/Lesekopf des Stackbandes immer auf das aktuelle Zeichen zeigt,

muss er für die Schreiboperation beim Zustand  $q_1$  erst in einem Übergang zum Hilfszustand  $p_1$  nach rechts bewegt werden, bevor im Übergang von  $p_1$  nach  $q_1$  das neue Zeichen auf das Stackband geschrieben werden kann, ohne dass der Schreib-/Lesekopf bewegt wird.

### 10.2.6 Turing-Maschinen und Programme

Die Diskussion von Abschnitt 10.2 zeigt, dass alle Varianten von Turing-Maschinen die gleichen Probleme lösen können. Sie zeigt außerdem, dass die Varianten ermöglichen, direkte Analogien zwischen Programmen in einem modernen Computer und Turing-Maschinen aufzuzeigen. Es ist daher legitim, sich Turing-Maschinen als einzelne Prozesse auf einem modernen Computer vorzustellen, die ein Programm ausführen.

## 10.3 Turing-erkennbare Sprachen

Turing-Maschinen definieren eine neue Klasse von Sprachen, die Turing-erkennbaren Sprachen.

### 10.3.1 Erkennbarkeit

Damit eine Turing-Maschine ein Wort verarbeiten kann, wird dieses erst auf das ansonsten mit  $\sqcup$  initialisierte Band geschrieben, der Schreib-/Lesekopf wird auf das erste Zeichen des Wortes positioniert. Dann wird die Maschine gestartet. Man sagt auch, die Maschine wir auf dem Input  $w$  gestartet.

**Definition 10.3** (Erkennen eines Wortes). *Eine Turing-Maschine erkennt das Wort  $w \in \Sigma^*$ , wenn die Maschine auf dem Input  $w$  im Zustand  $q_{\text{accept}}$  anhält.*

Ein Wort  $w$  kann aus zwei verschiedenen Gründen von einer Turing-Maschine nicht erkannt werden. Die Turing-Maschine kann auf dem Input  $w$  im Zustand  $q_{\text{reject}}$  anhalten. Dies scheint der einfache Fall zu sein. Schwieriger ist der zweite Fall: Die Turing-Maschine hält auf dem Input  $w$  gar nicht an.

**Definition 10.4** (Erkannte Sprache). *Ist  $M$  eine Turing-Maschine, dann ist*

$$L(M) = \{w \in \Sigma^* \mid M \text{ erkennt } w\}$$

*die von der Turing-Maschine erkannte Sprache. Eine Sprache heißt Turing-erkennbar, wenn es eine Turing-Maschine gibt, die sie erkennt.*

Auf Wörtern in  $\Sigma^* \setminus L(M)$  kann die Maschine im Zustand  $q_{\text{reject}}$  anhalten oder gar nicht anhalten.

Ein deterministischer endlicher Automat kann immer in eine Turing-Maschine umgewandelt werden, wie auf Seite 227 gezeigt wurde. Reguläre Sprachen sind also Turing-erkennbar. Auch ein Stackautomat kann mithilfe einer Mehrband-Turing-Maschine realisiert werden, auch die kontextfreien Sprachen sind daher Turing-erkennbar. Die Turing-erkennbaren Sprachen bilden somit eine Obermenge der regulären und kontextfreien Sprachen wie in Abbildung 10.20 dargestellt.

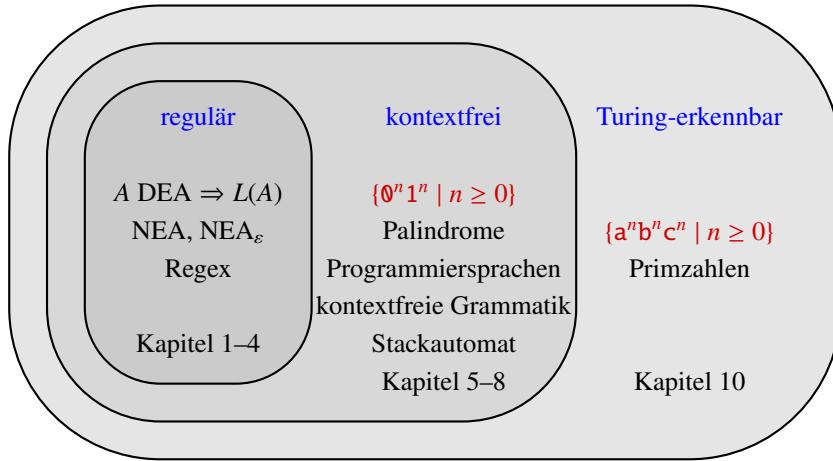


Abbildung 10.20: Die Turing-erkennbaren Sprachen bilden eine Obermenge der regulären und der kontextfreien Sprachen.

## Mengenoperationen

Die regulären und kontextfreien Sprachen waren unter den Mengenoperationen abgeschlossen. Bei den Turing-erkennbaren Sprachen ist die Situation etwas komplizierter.

**Satz 10.5.** *Sind  $L_1$  und  $L_2$  Turing-erkennbare Sprachen, dann sind auch der Durchschnitt  $L_1 \cap L_2$  und die Vereinigungsmenge  $L_1 \cup L_2$  Turing-erkennbar.*

*Beweis.* Turing-erkennbar heißt, dass es Turing-Maschinen gibt, die die Wörter der Sprachen erkennen. Es gibt also Turing-Maschinen  $M_1$  und  $M_2$  mit  $L_1 = L(M_1)$  und  $L_2 = L(M_2)$ . Es sind jetzt Turing-Maschinen zu konstruieren, welche Schnittmenge und Vereinigungsmenge erkennen.

Die gesuchte Turing-Maschine muss das gleiche Wort  $w$  erst mit  $M_1$  und anschließen mit  $M_2$  verarbeiten. Dazu eignet sich eine Zweiband-Turing-Maschine. Für die Schnittmenge erkennt die folgende Turing-Maschine  $M_{\cap}$  die Sprache  $L_1 \cap L_2$ :

1. Kopiere das Inputwort von Band 1 auf Band 2.
2. Lasse die Turing-Maschine  $M_1$  auf dem Band  $L_1$  laufen. Falls  $M_1$  in ihrem Zustand  $q_{\text{reject}}$  anhält, halte im Zustand  $q_{\text{reject}}$  von  $M_{\cap}$  an.
3. Lasse die Turing-Maschine  $M_2$  auf dem Band  $L_2$  laufen. Falls  $M_2$  in ihrem Zustand  $q_{\text{reject}}$  anhält, halte im Zustand  $q_{\text{reject}}$  von  $M_{\cap}$  an.
4. Halte im Zustand  $q_{\text{accept}}$ .

Falls eine der Maschinen nicht anhält, wird das Wort von der entsprechenden Maschine nicht erkannt und ist daher auch nicht in der entsprechenden Menge  $L_k$ .

Für die Vereinigungsmenge  $L_1 \cup L_2$  konstruieren wir die Maschine  $M_{\cup}$ . Sie muss auf dem Input  $w$  genau dann im Zustand  $q_{\text{accept}}$  anhalten, wenn eine der Maschinen  $M_1$  und

$M_2$  das Wort  $w$  erkennt. Es kann aber passieren, dass die Maschine  $M_1$  nicht anhält. Würde man die Maschinen nacheinander laufen lassen, blockiert die Maschine  $M_1$ , dass die Maschine  $M_2$  überhaupt gestartet wird.

Die Schwierigkeit kann umgangen werden, indem die Steuerung der Zweibandmaschine die Verarbeitung auf den beiden Bändern unabhängig voneinander steuert<sup>1</sup>. Auf Band 1 werden die Operationen der Maschine  $M_1$  ausgeführt, auf Band 2 unabhängig davon die Operationen der Maschine  $M_2$ . Wenn eine der Turing-Maschinen in ihrem Zustand  $q_{\text{accept}}$  anhält, halte im Zustand  $q_{\text{accept}}$  an. Falls beide Maschinen in ihrem Zustand  $q_{\text{reject}}$  anhalten, halte im die Maschine  $M_{\cup}$  im Zustand  $q_{\text{reject}}$  an.  $\square$

Für die Konstruktion der Maschine  $M_{\cup}$ , die die Vereinigungsmenge erkennt, mussten die Maschinen parallel ausgeführt werden. Mit moderner Technologie würde man zwei Threads für die Ausführung der beiden Maschinen verwenden.

Über die Mengendifferenz und das Komplement von Turing-erkennbaren Sprachen lässt sich keine Aussage machen. Ist  $M$  eine Turing-Maschine, die die Sprache  $L$  erkennt, dann ist nur bekannt, dass  $M$  auf Inputs  $w \in L$  im Zustand  $q_{\text{accept}}$  anhält. Für Wörter  $w \in \Sigma^* \setminus L$  hält die Maschine  $M$  im Zustand  $q_{\text{reject}}$  oder sie hält gar nicht. Auf der Basis der Maschine  $M$  lässt sich also keine Maschine konstruieren, welche ein Wort  $w \in \Sigma^* \setminus L$  erkennt. Somit kann man nicht schließen, dass das Komplement von  $L$  Turing-erkennbar ist. Es gibt eine Asymmetrie zwischen  $L$  und  $\Sigma^* \setminus L$ .

## Das Collatz-Problem und nicht Turing-erkennbaren Sprachen

Wir betrachten die *Collatz-Funktion*

$$c: \mathbb{N} \rightarrow \mathbb{N} : n \mapsto \begin{cases} 3n + 1 & \text{für } n \text{ ungerade} \\ \frac{n}{2} & \text{für } n \text{ gerade.} \end{cases}$$

Für jede Zahl  $n \in \mathbb{N}$  kann durch wiederholte Anwendung der Funktion  $c$  die *Collatz-Folge*

$$n, c(n), c(c(n)) = c^2(n), \dots, c^k(n), \dots$$

gebildet werden. Durch Experimente stellt man schnell fest, dass für kleine  $n$  die Collatz-Folge immer in  $4 \rightarrow 2 \rightarrow 1$  zu enden scheint. Für den Startwert  $n = 1291$  ergibt sich zum Beispiel die Folge

$$\begin{aligned} 1291 &\rightarrow 3874 \rightarrow 1937 \rightarrow 5812 \rightarrow 2906 \rightarrow 1453 \rightarrow 4360 \rightarrow 2180 \rightarrow 1090 \\ &\rightarrow 545 \rightarrow 1636 \rightarrow 818 \rightarrow 409 \rightarrow 1228 \rightarrow 614 \rightarrow 307 \rightarrow 922 \\ &\rightarrow 461 \rightarrow 1384 \rightarrow 692 \rightarrow 346 \rightarrow 173 \rightarrow 520 \rightarrow 260 \rightarrow 130 \\ &\rightarrow 65 \rightarrow 196 \rightarrow 98 \rightarrow 49 \rightarrow 148 \rightarrow 74 \rightarrow 37 \rightarrow 112 \\ &\rightarrow 56 \rightarrow 14 \rightarrow 7 \rightarrow 22 \rightarrow 11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \\ &\rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \\ &\rightarrow 4 \rightarrow 2 \rightarrow 1. \end{aligned}$$

<sup>1</sup>Die Steuerung kann nach dem Vorbild des Produktautomaten von Abschnitt 1.4 konstruiert werden, der ebenfalls die Übergänge zweier verschiedener Automaten “parallel” ausführen kann.

Die Collatz-Vermutung ist die Aussage, dass die Collatz-Folge tatsächlich für jeden natürlichen Startwert in 1 endet.

Die Sprache

$$L = \{n \in \mathbb{N} \mid \text{die mit } n \text{ beginnende Collatz-Folge endet in } 1\}.$$

Ist Turing-erkennbar, denn man kann ein Programm  $M_{\text{Collatz}}$  schreiben, welches die Berechnung der Collatz-Folge für einen Startwert  $n$  durchführt und im Zustand  $q_{\text{accept}}$  anhält, wenn die Folge den Wert 1 erreicht.

Es ist nicht bekannt, ob  $\mathbb{N} \setminus L \neq \emptyset$  ist. Es ist daher auch nicht bekannt, ob es eine Zahl gibt, auf der die Turingmaschine  $M_{\text{Collatz}}$  im Zustand  $q_{\text{reject}}$  anhalten würde, oder auf dem sie gar nicht anhält. Von der Sprache  $\mathbb{N} \setminus L$  ist daher im Moment nicht bekannt, ob sie Turing-erkennbar ist.

---

*Verständniskontrolle 10.4:* Die berühmte goldbachsche Vermutung besagt, dass jede gerade Zahl  $> 2$  als Summe von zwei Primzahlen geschrieben werden kann. Christian Goldbach hat sie in einem Brief an Leonhard Euler im Jahr 1742 aufgestellt, sie ist bis heute unbewiesen. Ist die Sprache



$$L = \{g \in \mathbb{Z} \mid \text{Es gibt zwei Primzahlen } p_1 \text{ und } p_2 \text{ mit } g = p_1 + p_2\}$$

Turing-erkennbar?

---

### Nicht Turing-erkennbare Sprachen

Turing-Maschinen sind viel mächtiger als alle in früheren Kapiteln studierten Maschinen. Trotzdem gibt es Sprachen, die nicht von einer Turing-Maschine erkannt werden können.

**Satz 10.6.** *Es gibt überabzählbare viele Sprachen  $L \subset \Sigma^*$ , die nicht Turing-erkennbar sind.*

*Beweis.* Zu jeder Turing-erkennbaren Sprache  $L$  gibt es eine Turing-Maschine  $M$ , die die Sprache  $L = L(M)$  erkennt. Eine Turing-Maschine ist aber durch die Festlegung des endlichen Automaten gegeben, der für die Steuerung verantwortlich ist. So ein endlicher Automat kann zum Beispiel in einer Tabellendarstellung als Zeichenkette endlicher Länge dargestellt werden. Es kann also nicht mehr Turing-Maschinen als Zeichenketten endlicher Länge geben. Für ein beliebiges Alphabet  $\Delta$  ist die Menge  $\Delta^*$  der Zeichenketten in diesem Alphabet abzählbar. Es folgt, dass die Menge der Turing-Maschinen abzählbar unendlich ist.

Sprachen sind Teilmengen von  $\Sigma^*$ . Die Menge aller Sprachen ist die Potenzmenge  $P(\Sigma^*)$ , die Menge aller Teilmengen von  $\Sigma^*$ . Nach Satz 9.8 ist  $P(\Sigma^*)$  überabzählbar unendlich. Es gibt also überabzählbar unendlich viele Sprachen, zu denen es keine Turing-Maschine geben kann und die daher nicht Turing-erkennbar sind.  $\square$

Nach Satz 10.6 sind die meisten Sprachen nicht Turing-erkennbar und können daher nicht mit einer Turing-Maschine behandelt werden. Mit der leistungsfähigsten Art von

Maschine, die wir bis jetzt konstruiert haben, die alle bisher identifizierten Klassen von Sprachen behandeln kann, sind wir immer noch nicht in der Lage, die meisten Sprachen zu erkennen. Heißt das, dass wir mit unserer Vorgehensweise auf dem Holzweg sind?

Für die Informatik ist die Menge  $P(\Sigma^*)$  nur von beschränktem Interesse. Die Informatik arbeitet nur mit Sprachen, die in irgend einer Weise spezifiziert werden können. Ein Kunde bestellt bei einem Software-Unternehmen die Programmierung einer Software, welche eine Sprache gemäß der im Vertrag definierten Spezifikation akzeptiert. Der Kunde erwartet nicht, dass die Software mit jeder beliebigen Sprache sinnvoll umgehen kann. Spezifikationen sind jedoch wieder Beschreibungen von Sprachen in Textform, also wieder Elemente von  $\Delta^*$  für ein geeignetes Alphabet  $\Delta$ . Es gibt daher wieder nur abzählbare unendlich viele Spezifikationen und damit nur abzählbar unendlich viele für die Informatik interessante Sprachen.

Mit der Menge der Turing-erkennbaren Sprachen scheinen die Grenzen des für Computer erreichbaren Universums der Sprachen erreicht. Dies schließt aber nicht aus, dass innerhalb der Turing-erkennbaren Sprachen weitere Anforderungen gibt, die gewisse Probleme immer noch zu mit Computern praktisch nicht zu bewältigenden Aufgaben machen. In Kapitel 11 wird darauf eingegangen.

### 10.3.2 Aufzählbare Sprachen

Eine Turing-Maschine verfügt nicht über Ausgabegeräte. Wenn die Turing-Maschine Output produzieren will, dann wird dafür Platz auf dem Band verwendet. In diesem Abschnitt wird untersucht, ob sich die Fähigkeiten einer Turing-Maschine ändern, wenn man sie mit einem Ausgabegerät versieht.

#### Aufzähler

Ein *Aufzähler* ist eine Turing-Maschine mit einem Ausgabegerät, mit dem die Turing-Maschine Wörter ausgeben kann. Wir verzichten auf eine formale Definition, die Vorstellung eines Programmes, welches Wörter auf dem Standard-Output ausgeben kann, soll genügen.

Als Beispiel kann man ein Programm betrachten, welches nacheinander alle natürlichen Zahlen daraufhin testet, ob sie Primzahlen sind, und sie ausgibt. Der Aufzähler erzeugt also nacheinander die Darstellungen der Primzahlen im Zehnersystem.

**Definition 10.7** (aufzählbare Sprache). *Eine Sprache heißt aufzählbar, wenn es einen Aufzähler gibt, der alle Wörter der Sprache aufzählt.*

Die Primzahlen bilden also eine aufzählbare Sprache. Ganz ähnlich sind alle Zahlen, die man in Sammlungen von Resultaten wissenschaftlicher Berechnungen finde, aufzählbare Sprachen. Die Daten und Zeiten der Sonnenfinsternisse der nächsten hundert Jahre bilden eine aufzählbare Sprache.

#### Aufzählbare Sprachen sind Turing-erkennbar

Das Ausgabegerät ändert die Fähigkeiten der Turing-Maschine nicht wesentlich. Sei  $A$  ein Aufzähler für eine Sprache  $L$ . Wir konstruieren eine Turing-Maschine, die die Wörter

von  $L$  erkennt. Dazu verwenden wir eine Turing-Maschine  $M$  mit zwei Bändern. Auf dem ersten Band ist das Inputwort  $w$  gespeichert. Auf dem zweiten Band lassen wir den Aufzähler operieren. Jedes Mal, wenn der Aufzähler ein Wort ausgeben möchte, vergleichen wir die Ausgabe mit dem Inhalt des ersten Bandes. Im Falle einer Übereinstimmung hält die Maschine im Zustand  $q_{\text{accept}}$  an. Andernfalls lässt man den Aufzähler weiter laufen.

Die so konstruierte Turing-Maschine  $M$  hält auf jedem Wort der Sprache  $L$  an, weil der Aufzähler es irgendwann ausgeben wird. Auf allen Wörtern, die nicht in  $L$  sind, wird die Maschine  $M$  unendlich lange laufen.

Mit dem Beispiel eines Primzahllaufzählers kann man auch ein Programm bauen, welches Primzahlen erkennt. Auf einer Zahl, die nicht prim ist, wird dieses Programm aber nicht anhalten.

Aufzählbare Sprachen sind daher immer Turing-erkennbar. Turing-erkennbare Sprachen werden in der Literatur zum Teil auch *rekursiv aufzählbar* genannt.

---

**Verständniskontrolle 10.5:** Eine natürliche Zahl  $n \in \mathbb{N}$  heißt vollkommen, wenn sie die Summe ihrer Teiler ist. Die kleinsten vollkommenen Zahlen sind

$$6 = 1 + 2 + 3$$

$$28 = 1 + 2 + 4 + 7 + 14$$



autospr.ch/v/10.5.pdf

Alle bekannten vollkommenen Zahlen sind gerade, es ist nicht bekannt, ob es ungerade vollkommene Zahlen gibt. Ist die Sprache

$$L = \{n \in \mathbb{N} \mid n \text{ ist eine ungerade vollkommene Zahl}\}$$

aufzählbar?

---

## Entscheider und Aufzähler

Umgekehrt kann man mit einer Turing-Maschine, die eine Sprache erkennt, unter zusätzlichen Voraussetzungen auch einen Aufzähler für die Sprache konstruieren. Sei  $M$  eine Turing-Maschine, die die Sprache  $L$  erkennt, aber zusätzlich die Eigenschaft hat, auf jedem beliebigen Input zu terminieren. Solche Turing-Maschinen heißen *Entscheider*, sie werden in Kapitel 11 genauer untersucht. Mit so einem Entscheider lässt sich jetzt ein Aufzähler für die Sprache  $L$  konstruieren. Dazu wird eine Turing-Maschine mit zwei Bändern verwendet. Auf dem anfänglich leeren zweiten Band werden nach und nach alle Wörter der Länge 1, 2 etc. in lexikographischer Ordnung erzeugt. Jedes Mal, wenn ein neues Wort  $w$  erzeugt worden ist, wird es auf das erste Band kopiert und die Maschine  $M$  darauf gestartet. Falls die Maschine  $M$  das Wort akzeptiert, wird es ausgegeben. Diese Maschine gibt genau die Wörter aus, die  $M$  akzeptiert.

Man beachte, dass eine Maschine  $M$ , die für gewisse Inputwörter nicht anhält, den Aufzählprozess zum Stillstand bringt. Wenn  $M$  auf dem Wort  $w$  nicht anhält, werden die Wörter, die lexikographisch nach  $w$  kommen, gar nicht erst getestet und können daher auch nicht ausgegeben werden.

## 10.4 Die universelle Turing-Maschine

Trotz der Ähnlichkeit der Funktion einer Turing-Maschine mit einem modernen Computer gibt es doch Unterschiede, die in Frage stellen könnten, dass die Turing-Maschine ein gutes Modell für Computer ist.

### 10.4.1 Moderne Computer und Turing-Maschinen

Moderne Computer sind ganz offensichtlich viel komplexer als die sehr primitive Turing-Maschine. Das Konzept der Turing-Maschine ist darauf ausgelegt, die Lösung einer bestimmten Aufgabe durch eine Maschine zu modellieren. Auch wenn die Beschreibung nach einer Hardwarespezifikation klingt, geht es mehr darum, wie sich das System einem Anwendungsprozess präsentiert. Das Betriebssystem spielt dabei eine entscheidende Rolle. Es organisiert den Adressraum, stellt den Speicher bereit, lädt das Programm und vermittelt die Interaktion mit dem Benutzer, mit Files oder mit anderen Prozessen.

Die Kritik, eine Turing-Maschine sei ein unrealistisches Modell für einen Computer, muss daher umformuliert werden. Es ist die Frage zu beantworten, ob das Betriebssystem eines modernen Computers eine Prozessumgebung bereitstellt, die von einer Turing-Maschine so gut modelliert wird, dass die über Turing-Maschinen gewonnenen Erkenntnisse auch für Prozesse gelten.

#### Der unendliche Speicher ist unrealistisch

Die Turing-Maschine verwendet ein beidseitig unbegrenzt ausgedehntes Band. Der Vorteil dieser Konstruktion ist vor allem, dass man sich keine Gedanken darüber machen muss, wie die Maschine reagieren soll, wenn der Schreib-/Lesekopf das Ende des Speichers erreicht. Dazu müssten spezielle Zeichen vorhanden sein, mit denen das Bandende markiert ist. Trifft die Maschine auf ein solches Zeichen, kann sie die Berechnung abbrechen oder andere Maßnahmen treffen, damit die Berechnung weitergeführt werden kann.

In Abbildung 10.18 wurde das Speicher-Layout eines Linux-Prozesses gezeigt. Das eigentliche Programm beginnt bei niedrigen Adressen, gefolgt von Speicherblöcken für die Variablen und schließlich dem Heap, der dynamisch nach oben wachsen kann. Die Startadresse `0xb68f000` des Bereichs der dynamisch geladenen Bibliotheken ist so groß, dass zwischen dem Ende des Heap bei `0x00f1c000` und den Bibliotheken etwa 3 GiB Platz zur Verfügung stehen, in die der Prozess hineinwachsen kann. Der Speicher ist somit zwar begrenzt, aber die obere Grenze ist für alle praktischen Zwecke unerreichbar hoch. In Abschnitt 10.2.3 wurde illustriert, dass ein einseitig unendlicher Speicher einer Turing-Maschine die gleichen Möglichkeiten eröffnet wie ein beidseitig unendlicher Speicher.

Sehr lange Rechnungen oder Programmfehler in der Speicherverwaltung (memory leaks) können natürlich dazu führen, dass einem Prozess der Speicherplatz ausgeht. Auf modernen Maschinen tritt dieser Zustand allerdings meist erst ein, nachdem der Prozess viel mehr als den physisch verfügbaren Hauptspeicher angefordert hat. Der Grund dafür ist, dass die virtuelle Speicherverwaltung des Betriebssystems durch Auslagern von aktuell nicht benötigten Speicherseiten oder Auslagern anderer Prozesse recht lange Speicherplatz beschaffen und in den Adressraum des Prozesses abbilden kann. Erst wenn auch im Swapfile kein Platz mehr ist für Speicherseiten, die ausgelagert werden sollten, wird das

Betriebssystem dem Prozess keinen weiteren Speicher mehr zur Verfügung stellen können. Funktionen zur Speicherallokierung werden Fehler oder Ausnahmen zurückliefern, die der Prozess behandeln müsste, die dann aber meist zum Prozessende führen.

Das iOS Betriebssystem von Apples iPhone und iPad hat einen Mechanismus eingeführt, um einem Prozess vorzeitig bekannt zu geben, dass der Speicher knapp wird und dass Maßnahmen ergriffen werden sollten, den Speicherplatz effizienter zu nutzen. Dazu schickt das Betriebssystem der App eine Speicherwarnung, die der Programmierer im einfachsten Fall durch Implementation der Methode `applicationDidReceiveMemoryWarning:` behandeln kann. Damit werden die Chancen weiter verbessert, dass sich das Programm für den Benutzer verhält, als wäre der Speicher tatsächlich unbegrenzt.

### **Die Turing-Maschine erlaubt keine Interaktion**

Die Definition einer Turing-Maschine enthält keine Elemente, die Interaktion mit einem Benutzer ermöglichen. Natürlich ist Interaktion ein wichtiger Aspekt moderner Computer, doch braucht es dazu auch auf Prozessebene Vorkehrungen, die in einer Turing-Maschine nicht modellierbar sind?

Wenn ein Programm Daten in ein File oder an ein Peripheriegerät ausgeben möchte, dann werden die Daten in einem Speicherbereich vorbereitet und anschließend mit einem *System call* das Betriebssystem dazu aufgefordert, die Daten zu transferieren. Der Prozess wird für die Dauer der Transaktion stillgelegt. Der Kernel kopiert die Daten in einen Bereich unter seiner ausschließlichen Kontrolle und stellt die Ausgabeanforderung in eine Warteschlange. Der Transfer wird also vom Prozess nicht selbst durchgeführt. Vielmehr sind die Daten aus Sicht des Prozesses von einem Moment zum nächsten plötzlich transferiert worden.

Auch bei Eingaben geschieht etwas Ähnliches. Möchte ein Prozess Zeichen von einem Eingabegerät lesen, übergibt er die Kontrolle dem Betriebssystem mit der Aufforderung, die Zeichen in einen bereitgestellten Speicherbereich zu transferieren. Das Betriebssystem kontrolliert dann, ob überhaupt Daten eingetroffen sind, kopiert sie in den Speicherbereich des Prozesses und lässt den Prozess weiter laufen. Sind noch keine Daten eingetroffen, wartet das Betriebssystem damit, den Prozess weiterlaufen zu lassen, bis die Daten eingetroffen sind.

Oft erfolgt ein solcher Datentransfer mit einer Unterbrechung. Ein Interrupt bringt den Prozessor dazu, speziellen Code zur Behandlung der Unterbrechung auszuführen. Treffen Daten auf einer Schnittstelle ein, wird ein Interrupts ausgelöst, der die Daten in einem Kernel-internen Puffer ablegt. Nach Behandlung des Interrupt arbeitet der Prozessor an der Stelle weiter, wo er unterbrochen worden ist. Selbst für den Kernel erscheint es also, als ob sich plötzlich Daten aus dem Nichts im Kernel-Speicher materialisiert hätten.

Die Beispiele illustrieren, dass Interaktion etwas ist, was das Betriebssystem dem Prozess ermöglicht, was aber nicht Teil des Prozessmodells selbst ist. Das Betriebssystem vermittelt dem Anwendungsentwickler die Illusion von Interaktion, während es doch immer nur Modifikation des Speichers des Prozesses ist.

### Turing-Maschinen können immer nur eine Aufgabe bewältigen

Moderne Computer führen viele Prozesse nebeneinander aus. Im einfachsten Fall einer einzelnen CPU wird dies dadurch möglich, dass das Betriebssystem den laufenden Prozess nach kurzer Zeit unterbricht und die Kontrolle einem anderen Prozess übergibt. Mit geeigneten CPUs mit mehreren Kernen oder mehreren CPUs können die Prozesse sogar tatsächlich gleichzeitig laufen. Ein Turing-Maschine kann auch dieses Verhalten abbilden. Ein Turing-Maschine mit mehreren Bändern kann jedem Prozess ein Band zuteilen. Die Steuereinheit kann alle Übergänge in allen Prozessen gleichzeitig durchführen, ähnlich dem Produktautomaten in Abschnitt 1.4.

### Turing-Maschinen werden für eine Aufgabe gebaut

Die früher gezeigten Beispiele von Turing-Maschinen codieren die Logik in der Steuerung der Maschine, die die Übergangsfunktion realisiert. Das Band war immer nur als Speicher für die Daten vorgesehen. Die Übergangsfunktion wird als Teil der Hardware angesehen. Diese Turing-Maschinen sind also für eine ganz spezielle Aufgabe gebaut worden. Soll ein anderes Problem gelöst werden, muss eine neue Maschine konstruiert werden.

Dies entspricht nicht der Vorstellung, die wir mit einem modernen Universalcomputer verbinden. Dessen Hardware interpretiert einen Teil der Daten, die im Speicher abgelegt sind, als Instruktionen, die er auszuführen in der Lage ist. Das Resultat ist die Veränderung eines anderen Teils des Speichers. Um dieses Verhalten abzubilden, muss eine universelle Turing-Maschine konstruiert werden, der man eine beliebige Beschreibung einer konkreten Turing-Maschine wie der Additionsmaschine von Seite 226 zur Ausführung übergeben kann.

Der verlinkte Turing-Maschinensimulator realisiert diese Idee ansatzweise: Das Programm liest eine Turing-Maschinenbeschreibung von einem File und kann sie ausführen. Das immer gleiche Simulatorprogramm kann also beliebige Turing-Maschinen ausführen.



### 10.4.2 Ladbare Programme für Turing-Maschinen

Gibt es eine universelle Turing-Maschine, die jedes beliebige Turing-Maschinenprogramm ausführen kann? Dass sich ein moderner Computer für jede beliebige Aufgabenstellung programmieren lässt, scheint der einzige verbleibende Unterschied zu einer Turing-Maschine zu sein. Tatsächlich hat sich Alan Turing schon in seinem bahnbrechenden Artikel [59] mit dieser Frage auseinandergesetzt. Es gelang ihm in den Abschnitten 6 und 7 seiner Arbeit, eine Turing-Maschine zu konstruieren, die ein auf dem Band vorgegebenes Programm ausführen kann.

Im Lichte unserer früheren Diskussion kann man sich für die Konstruktion einer universellen Maschine mit mehreren Bändern behelfen. Ein Band ist reserviert für den Programmcode, der ausgeführt werden soll. Wir nennen es das Codeband. Die universelle Turing-Maschine wird also eine Harvard-Architektur erhalten. Der Code kann als eine Tabelle formatiert werden, die alle möglichen Übergänge enthält.

Ein weiteres Band wird als Arbeitsband für die simulierte Turing-Maschine verwendet. Der aktuelle Zustand der Maschine kann auf einem dritten Band, dem Zustandsband,

gespeichert sein, es wird mit dem Startzustand der simulierten Maschine initialisiert.

Nach diesen Vorbereitungen kann man den Algorithmus jetzt mit dem folgenden Pseudocode beschreiben:

1. Durchsuche das Codeband, bis der Zustand, der auf dem Zustandsband gespeichert ist, gefunden wird.
2. Verwendet das Zeichen vom Arbeitsband, um mithilfe der Übergangstabelle den nächsten Zustand und das veränderte Zeichen zu ermitteln.
3. Überschreibe das Feld des Arbeitsbandes mit dem neuen Zeichen.
4. Überschreibe den Zustand auf dem Zustandsband mit dem neuen Zustand.
5. Falls der neue Zustand ein Akzeptierzustand ist, halte an.
6. Weiter bei 1.

Damit ist auch die letzte Hürde beim Vergleich zwischen einem Anwendungsprozess und einer Turing-Maschine überwunden. Wir dürfen im Folgenden davon ausgehen, dass eine Turing-Maschine genau das abbildet, was mit einem Prozess in einem modernen Computer gemacht werden kann. Der Prozessor, der den Prozess ausführt, ist eine universelle Turing-Maschine, die die Programmlogik aus dem Speicher liest und in Abhängigkeit von anderen Daten, die ebenfalls im Speicher liegen, eine beliebige Berechnung durchführen kann.

Die universelle Turing-Maschine macht plausibel, dass wir nicht nur in den Begriffen Band und Kopfbewegung nach links oder rechts zu denken brauchen. Wir dürfen statt von Turing-Maschinen auch immer von Programmen sprechen. Die Intuition, die dem Programmierer sagt, was er mit einem Programm machen kann, gilt somit auch dafür, was er mit einer Turing-Maschine machen kann. In Kapitel 14 wird diese Idee mit dem Begriff der Turing-Vollständigkeit nochmals vertieft.

### **10.4.3 Universelle Turing-Maschine in Game of Life**

Die universelle Turing-Maschine zeigt, dass jedes beliebige Programm auf einer universellen Maschine ausgeführt werden kann. Die Konstruktion dieser Maschine scheint sehr speziell zu sein und eine hoch entwickelte Technologie zu ihrer Realisierung vorauszu setzen. Andererseits ist unbestritten, dass biologische Gehirne die Berechnungen einer Turing-Maschine ebenfalls ausführen können. Offenbar hat die Evolution aus einfachsten lebenden Zellen funktionierende Turing-Maschinen hervorgebracht. Wie kann diese Kluft überbrückt werden?

Dieser Abschnitt versucht zu zeigen, dass beliebig komplexe Berechnungen als emergente Eigenschaft eines primitiven zellulären Automaten möglich sind.

#### **Zelluläre Automaten**

In einem Spreadsheet kann ein Benutzer in jedem Feld einer großen Tabelle Zahlen und andere Daten speichern. Das Feld kann aber auch Formeln enthalten, die den Inhalt des

A	B	C	D	E	F	G	H	I	J	K	L
1	Input:	1	0	0	1	1	0	1	1	0	1
2		0	1	2	1	0	1	2	2	1	0
3		reject	reject	reject	accept	reject	reject	reject	reject	reject	accept
4											
5											
6											

Abbildung 10.21: In Open Office implementierter zellulärer Automat, der den deterministischen endlichen Automaten für durch drei teilbare Binärzahlen realisiert.

Feldes berechnen. Wenn sich irgendwo in der Tabelle ein Feldinhalt ändert, werden alle davon abhängenden Zellen neu berechnet.

Ein *zellulärer Automat* ist eine Maschine, die aus einzelnen Zellen in einem ein- oder zweidimensionalen Gitter aufgebaut ist. Jede Zelle kann nur wenige mögliche Zustände annehmen. Den nächsten Zustand berechnet die Zelle aus dem eigenen Zustand und den Nachbarzuständen.

Der in Abbildung 10.21 gezeigte zellulärer Automat mit zwei Zeilen enthält in der ersten Zeile die Zeichen des Wortes  $1001101101$ . Die erste Zelle der zweiten Zeile enthält den Startzustand, jede weitere Zelle der Zeile enthält eine Implementation der Übergangsfunktion des deterministischen endlichen Automaten für durch drei teilbare Binärzahlen. In der dritten Zeile wird angezeigt, ob der in der zweiten Zeile angezeigte Zustand ein Akzeptierzustand ist oder nicht. Der zelluläre Automat ermittelt, dass  $1001101101_2 = 621_{10} = 3^3 \cdot 23$  eine durch drei teilbare Binärzahl ist. Das Beispiel veranschaulicht, dass ein zellulärer Automat auch zu komplexen Berechnungen fähig sein kann. In Abschnitt 14.3.5 wird sogar gezeigt, wie eine Turing-Maschine in einem Spreadsheet realisiert werden kann.

### Game of Life

Game of Life ist ein von John Horton Conway erdachter zellulärer Automat, dessen Zellen zwei mögliche Zustände haben können: tot oder lebendig. Die Zustände einer Zelle ändern sich nach den folgenden Regeln:

1. Eine lebendige Zelle mit weniger als 2 lebendigen Nachbarn oder mit mehr als 3 lebendigen Nachbarn stirbt.
2. Eine lebendige Zelle mit 2 oder 3 lebendigen Nachbarn überlebt
3. Eine tote Zelle mit genau drei Nachbarn wird lebendig.

Diese einfachen Regeln führen auf erstaunlich vielfältige Strukturen. In Abbildung 10.22 sind ein paar Beispiele zusammengestellt.

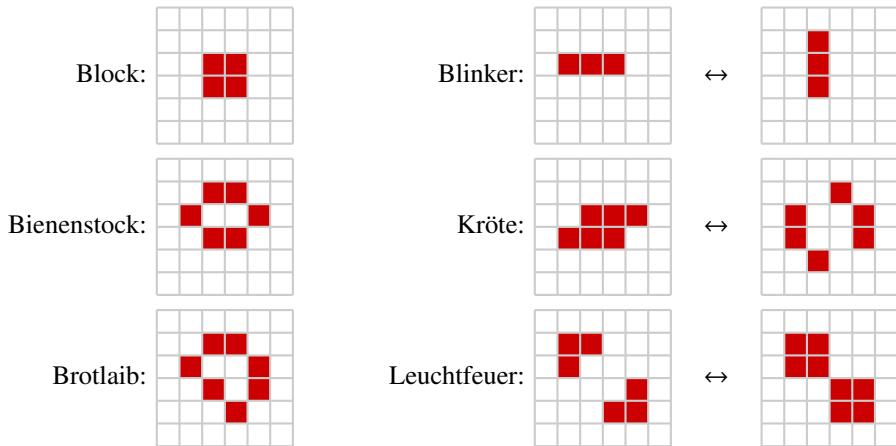


Abbildung 10.22: Beispiele von Strukturen in Game of Life. Die Strukturen ganz links sind stabil, sie verändern sich nicht von Generation zu Generation. die Strukturen in rechts oszillieren zwischen zwei möglichen Zuständen.

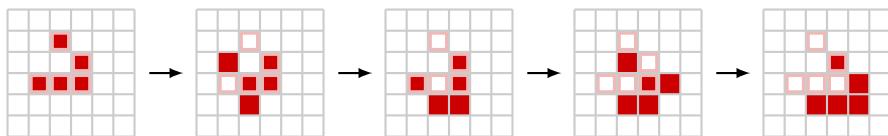


Abbildung 10.23: Gleiter-Struktur, die sich in jeweils vier Phasen über beliebig große Distanzen durch den zellulären Automaten bewegen kann.

Es gibt aber auch Strukturen, die sich über beliebig große Distanzen durch den zellulären Automaten bewegen können. In Abbildung 10.23 ist der Gleiter dargestellt, der nach vier Generationen wieder die gleiche Form hat, jedoch verschoben ist.

Im gleichen Stil ist eine große Vielfalt von Strukturen erfunden worden, die fast beliebige Funktionen übernehmen können. Gleiter können zum Beispiel als Signalboten eingesetzt werden. Es sind Strukturen gefunden worden, mit denen sich logische Operationen zwischen diesen Signalen ausführen lassen, oder die Information speichern können.

## Universelle Turing-Maschine

Paul Rendell ist es in [51] gelungen, eine Turing-Maschine in Game of Life zu realisieren. Die Gleiter werden darin für den Austausch von Signalen verwendet. Die Komponenten einer Turing-Maschine lassen sich im Blockdiagramm in Abbildung 10.24 leicht identifizieren. Besonders interessant ist die Realisierung des Bandes. Dazu werden zwei Stacks verwendet, die Daten in beiden Richtungen austauschen können, genau so, wie wir das Band zu Beginn dieses Kapitels eingeführt haben.



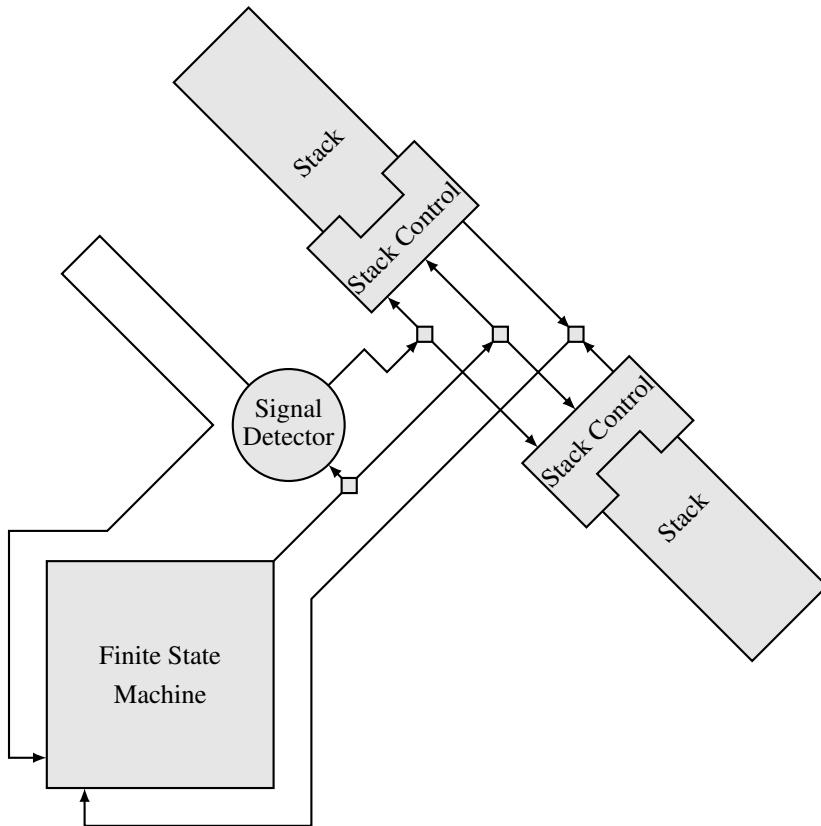


Abbildung 10.24: Turing-Maschine in Game of Life nach Paul Rendell [51], © 2016, Springer International Publishing Switzerland.

## Übungsaufgaben

**10.1.** Beschreiben Sie eine Turing-Maschine, welche eine binär auf dem Band angegebene Zahl um 1 erhöht.

**10.2.** Gegeben ist die Turingmaschine mit dem Zustandsdiagramm von Abbildung 10.25 über dem Alphabet  $\Sigma = \{\emptyset, 1\}$  und dem Bandalphabet  $\Gamma = \{\emptyset, 1, x, \_\}$ .

- Wird das Wort **101** akzeptiert?
- Wird das Wort **0110** akzeptiert?
- Falls das Band zwei durch  $\_$  getrennte Wörter enthält, kann das zweite Wort einen Einfluss darauf haben, ob der Input akzeptiert wird?
- Es wird behauptet, dass die Maschine alle Wörter  $w$  mit  $|w|_0 = |w|_1$  akzeptiert. Ist dies korrekt?

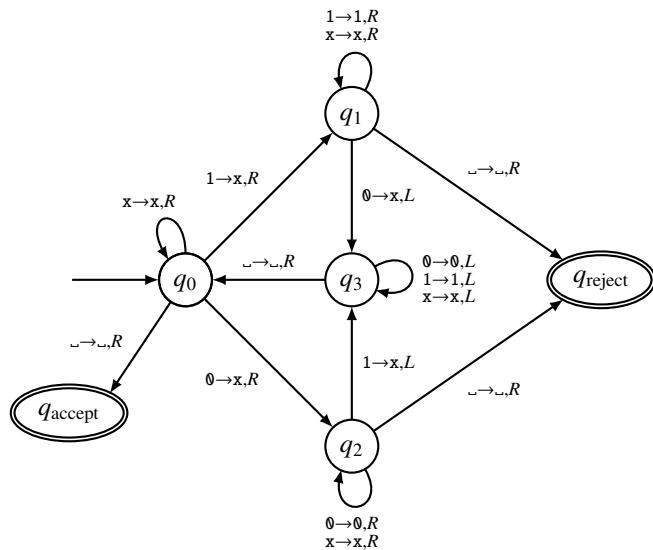


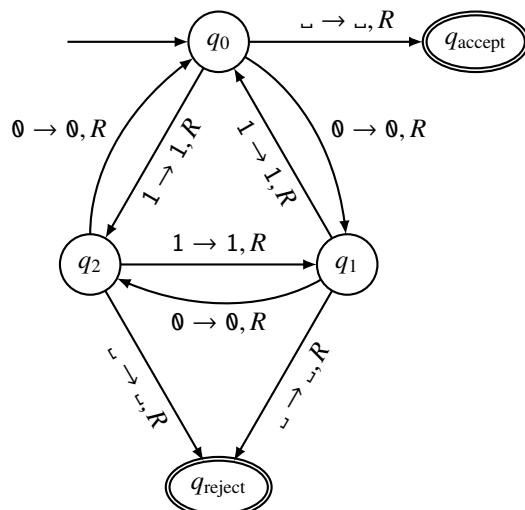
Abbildung 10.25: Zustandsdiagramm für die Turing-Maschine in Aufgabe 10.2.

**10.3.** Sei  $\Sigma = \{\emptyset, 1\}$  und

$$L = \left\{ w \in \Sigma^* \mid \begin{array}{l} w \text{ besteht abwechselnd aus } \emptyset \text{ und } 1 \text{ und hört} \\ \text{mit dem gleichen Zeichen auf wie es beginnt} \end{array} \right\}.$$

Finden Sie das Zustandsdiagramm einer Turing-Maschine, die  $L$  erkennt.

**10.4.** Betrachten Sie die Turing-Maschine  $M$  mit dem Zustandsdiagramm



a) Wie wird das Wort **001011** verarbeitet?

b) Welche der Wörter

**01, 01000, 01000000, 111, 0111, 000111, 000111111, 00010101111**

werden akzeptiert?

c) Welche Sprache wird von  $M$  akzeptiert?

**10.5.** Betrachten Sie folgende Variante einer Turing-Maschine. In dieser Variante ist die Bewegung des Kopfes nach links nicht möglich, der Kopf kann nur nach rechts bewegt werden oder stehenbleiben. Die Übergangsfunktion ist also von der Form

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{S, R\}$$

( $S$  für Stillstand). Welche Art von Sprachen kann mit diesem Typ Maschine erkannt werden?

Lösungen: <https://autospr.ch/uebungen/AutoSpr-110.pdf>



# Kapitel 11

## Entscheidbarkeit

Die sich mit immer neuen Superlativen gegenseitig überbietenden Ankündigungen der IT-Industrie könnten den Eindruck erwecken, dass mit geeignet viel Rechenleistung und Speicherplatz kein Informatikproblem einer Lösung widerstehen könnte. Auch die Sicherheit modernster Kryptographie wird immer wieder damit erklärt, wie lange aktuelle Computer brauchen würden, die Verschlüsselung zu knacken, was natürlich impliziert, dass sie tatsächlich in endlicher Zeit geknackt werden kann. Der Eindruck ist jedoch falsch. Es gibt eine ganze Reihe von nicht trivialen und anwendungsrelevanten Problemen, die von einem Computer prinzipiell nicht gelöst werden können. Einige von ihnen sollen in diesem Kapitel gefunden werden.

### 11.1 Algorithmen und Entscheidbarkeit

Die Entstehung des Begriffs der Entscheidbarkeit lässt sich in die Grundlagenkrise der Mathematik im 19. Jahrhundert zurückverfolgen. Er wird durch den Vortrag von David Hilbert am ersten internationalen Mathematikerkongress 1900 in Paris ins Rampenlicht gerückt.

#### 11.1.1 Hilberts Vortrag am ICM 1900

Am internationalen Mathematikerkongress (International Congress of Mathematicians, ICM) in Paris im Jahre 1900 hielt David Hilbert einen aufsehenerregenden Plenarvortrag, in dem er eine Reihe von Problemen stellte, deren Lösung seiner Meinung nach besonders dazu geeignet sein könnten, die Mathematik weiterzubringen. Hilberts Fragestellungen haben tatsächlich einen großen Einfluss auf die weitere Entwicklung der Mathematik gehabt. Sie sahen sich aber auch mit der Kritik konfrontiert, dass sie von vielen anderen wichtigen Problemen ablenken würden.

Von besonderer Bedeutung für das Thema dieses Buches ist das zehnte hilbertsche Problem.

**Aufgabe 11.1** (Zehntes hilbertsches Problem). *Man gebe ein Verfahren an, welches für eine beliebige diophantische Gleichung entscheidet, ob sie lösbar ist.*

Diophantische Gleichungen sind Gleichungen der Form  $f(x_1, x_2, \dots, x_n) = 0$ , wobei  $f$  ein Polynom mit ganzzahligen Koeffizienten ist. Für die Variablen  $x_1, \dots, x_n$  sind nur ganzzahlige Werte zugelassen. Zum Beispiel ist die Gleichung

$$a^2 + b^2 = c^2 \quad (11.1)$$

die diophantische Gleichung mit dem Polynom  $f(x_1, x_2, x_3) = x_1^2 + x_2^2 - x_3^2$ . Es ist auch bekannt, dass die Gleichung (11.1) von den pythagoräischen Tripeln gelöst wird, zum Beispiel ist das Tripel  $(3, 4, 5)$  eine Lösung, denn  $3^2 + 4^2 = 5^2$ . Andererseits hat die Gleichung

$$3a - 4 = 0$$

keine ganzzahlige Lösung, denn aus der äquivalenten Form  $3a = 4$  ist ablesbar, dass 4 durch 3 teilbar sein müsste.

### 11.1.2 Moderne Formulierung

Zur Zeit von Hilberts Vortrag war noch nicht einmal wirklich klar, welche Form die Antwort auf Hilberts Aufgabenstellung haben müsste. Eine wichtige Entwicklung, die sich aus Hilberts Problemliste ergab, war somit die Klärung der Frage, wonach das zehnte Problem überhaupt fragt.

Aus unserer modernen Perspektive erwarten wir als Antwort ein Computerprogramm, dem man das Polynom als Input übergeben kann und welches für jedes beliebige Polynom in der Lage ist, in endlicher Zeit eine definitive Antwort zu geben.

Ein simpler Algorithmus könnte versuchen, der Reihe nach alle ganzzahligen  $n$ -Tupel  $(x_1, \dots, x_n) \in \mathbb{Z}^n$  in die Gleichung einzusetzen und zu prüfen, ob die Gleichung erfüllt ist. Da die Menge  $\mathbb{Z}^n$  abzählbar unendlich ist (Kapitel 9, Satz 9.5), ist es sicher möglich, alle  $n$ -Tupel aufzuzählen. Wenn die Gleichung eine ganzzahlige Lösung hat, dann wird sie von diesem Algorithmus daher auch gefunden. Schwierigkeiten treten auf mit Gleichungen, die keine ganzzahligen Lösungen haben. Für solche Gleichungen wird das Programm niemals anhalten. In diesem Sinne kann man nicht sagen, dass der Algorithmus das Problem löst.

Von einer Lösung kann man erst sprechen, wenn der Algorithmus sowohl eine positive Antwort gibt, wenn eine Lösung existiert, als auch eine negative, wenn keine Lösung möglich ist. In jedem Fall erwartet man also, dass der Algorithmus irgendwann anhält. Ein solches Programm heißt ein *Entscheider*.

Tatsächlich wurde das zehnte Hilbertsche Problem 1970 von Juri Matijasewitsch gelöst. Er konnte zeigen, dass es kein Programm gibt, welches entscheiden kann, ob eine beliebige diophantische Gleichung eine Lösung hat [29]. Hilberts Traum war immer gewesen, dass sich in der Mathematik alles entscheiden lässt. Auch in anderen Gebieten, zum Beispiel in der Logik, deckten die Erkenntnisse von Gödel und Turing viele nicht entscheidbare Fragestellungen auf. Hilberts Vorstellung, an die er noch 1930 geglaubt hatte, kam so ins Wanken. Es entbehrt nicht einer gewissen Ironie, dass sich schließlich sogar ein Problem, das Hilbert als besonders wichtig ansah, als unlösbar zu gelten hat.

### 11.1.3 Entscheidbarkeit

Das zehnte hilbertsche Problem ist nicht die einzige Fragestellung, für die es keine algorithmische Lösung gibt. Der Begriff der Entscheidbarkeit fasst diese Eigenschaft mathematisch präzis.

#### Der Risch-Pseudoalgorithmus

Ein Computeralgebrasystem (CAS) kann algebraische Umformungen, Ableitungen und symbolische Integration automatisch durchführen. Der Risch-Algorithmus befähigt es, zu entscheiden, ob die Stammfunktion einer Funktion durch elementare Funktionen ausgedrückt werden kann. Elementare Funktionen sind Polynome, Wurzelfunktionen mit beliebigen Potenzen, Exponentialfunktionen und Logarithmen, trigonometrische Funktionen und ihre Umkehrfunktionen und alle Funktionen, die sich daraus durch rationale Operationen, Wurzeln und Komposition erzeugen lassen. Als Teilaufgabe muss im Risch-Algorithmus entschieden werden, ob in der Rechnung auftretende Ausdrücke konstant sind. Für diese Teilaufgabe, die auch das Konstantheitsproblem heißt, ist kein Entscheidungsalgorithmus bekannt. Nimmt man zu den elementaren Funktionen die Betragsfunktion hinzu, kann sogar gezeigt werden, dass das Konstantheitsproblem nicht entscheidbar ist.

Man kann also von einem Computeralgebrasystem grundsätzlich nicht erwarten, dass es entscheiden kann, ob eine elementare Funktion eine elementare Stammfunktion hat. Existierende Computeralgebrasysteme verwenden heuristische Methoden, um die Einschränkung durch das Konstantheitsproblem zu umgehen. Versagen diese Heuristiken, kann das CAS auch keine Antwort geben.

#### Textinput

Den bisher diskutierten Aufgabenstellungen ist gemeinsam, dass ein Entscheidungsprogramm Input in irgendeiner Form entgegennimmt und als Rückgabewert eine ja-/nein-Entscheidung liefert. Aus Sicht der Beschreibung der Arbeitsweise einer Turing-Maschine ist der Input eine Zeichenkette in einem geeigneten Alphabet  $\Sigma$ . Die Wahl des Alphabets hat keinen Einfluss auf die Fähigkeiten einer Turing-Maschine. Zum Beispiel kann man das Polynom  $f(x_1, x_2, x_3) = x_1^2 + x_2^2 - x_3^3$  für das zehnte hilbertsche Problem in der Form

$f: x1^2 + x2^2 - x3^2;$

als Text codieren, die das Computeralgebrasystem Maxima [30] verstehen könnte. Man kann es aber auch in der Form

$f(x\_1, x\_2, x\_3)=x\_1^2+x\_2^2-x\_3^3$

schreiben, welche für die Darstellung in L<sup>A</sup>T<sub>E</sub>X geeignet ist. Da die Zeichen einer Zeichenkette als Folgen von Binärziffern betrachtet werden können, ist auch immer eine Darstellung des Inputs als binäre Zeichenkette möglich.

## Serialisierung und Deserialisierung

Viele Programmiersprachen verfügen über einen Mechanismus, der jedes beliebige Objekt in einen Datenblock zu verwandelt gestattet. Dieser Prozess wird *Serialisierung* genannt. Implementiert eine Java-Klasse das Interface `java.lang.Serializable`, dann können Instanzen davon mit der Methode

```
public final void writeObject(Object o) throws IOException;
```

der Klasse `java.io.ObjectOutputStream` auf einen OutputStream geschrieben werden. Ein `ObjectOutputStream`-Objekt kann mit einem beliebigen anderen OutputStream als Ziel für die serialisierten Daten konstruiert werden.

Ein `java.io.ObjectInputStream` kann mit der Methode

```
public final Object readObject(Object o) throws IOException,
                                         ClassNotFoundException;
```

das Objekt wieder von einem Stream lesen oder wie man sagt *Deserialisieren*. Dieser Mechanismus wird auch von Java RMI (Remote Method Invocation) verwendet.

In Objective-C heißen die Operationen auch *Deaktivieren* und *Aktivieren* einer Objektinstanz.

Verteilte Objekte in einem Netzwerk müssen Daten zwischen möglicherweise verschiedenen Computerarchitekturen austauschen. Dazu verwenden sie ebenfalls eine Art der Serialisierung. Der Open Network Computing Remote Procedure Call ONC RPC ist eine der ältesten noch heute eingesetzten Formen von remote procedure call (RPC), er bildet die Basis des *network file system* (NFS). Er enthält einen Protokoll-Compiler, der Datenstrukturdefinitionen in C-Code-Funktionen zur Serialisierung und Deserialisierung übersetzt. Die Argumente eines RPC werden auf dem Client serialisiert, an den Server übermittelt, dort deserialisiert, die verlangte Serverprozedur wird ausführt, die Resultatdaten werden serialisiert und an den Client zurückschickt, der sie für die aufrufende Anwendung wieder deserialisiert.

SOAP-basierte Webanwendungen verwenden ein XML-basiertes Serialisierungsformat. REST-Anwendungen verwenden meist JSON als Datenformat für die Serialisierung von Objekten, die JavaScript-Funktion `JSON.stringify(objekt)` verwandelt das Argument `objekt` in eine Zeichenkette, die von `JSON.parse()` wieder in ein JavaScript-Objekt verwandelt werden kann. Die Beispiele zeigen auch, dass ein Tupel von Objekten als Input für ein Entscheidungsproblem kein Hindernis ist, da sich ein solches immer als Zeichenkette darstellen lässt.

## Sprachprobleme

**Definition 11.2** (Zeichenkettencodierung). *Sind  $a_1, \dots, a_n$  beliebige mathematische Objekte, dann bezeichnen wir mit  $\langle a_1, \dots, a_n \rangle$  eine beliebige Codierung des Tupels  $(a_1, \dots, a_n)$  als Zeichenkette über einem geeigneten Alphabet.*

Serialisierung ermöglicht, jedes Entscheidungsproblem auf die Verarbeitung einer Zeichenkette zurückzuführen, die entweder akzeptiert oder verworfen wird. Jedes Problem wird auf diese Weise zu einem Sprachproblem.

*Beispiel 11.3.* Die Menge der Primzahlen  $P$  kann als Sprache formuliert werden, indem man die Primzahlen zum Beispiel im Zehnersystem codiert:

$$L = \{w \in \Sigma^* \mid w \text{ ist die Zehnersystemdarstellung einer Primzahl.}\}$$

mit dem Alphabet  $\Sigma = \{\emptyset, 1, \dots, 9\}$ . ○

*Beispiel 11.4.* Das zehnte hilbertsche Problem ist die Sprache

$$H = \left\{ \langle f(x_1, \dots, x_n) \rangle \mid \begin{array}{l} f \in \mathbb{Z}[x_1, \dots, x_n] \text{ ist ein Polynom mit ganzzahligen} \\ \text{Koeffizienten und } f(x_1, \dots, x_n) = 0 \text{ hat ganzzahlige} \\ \text{Lösungen.} \end{array} \right\}.$$

Das Resultat von Matijasewitsch [29] zeigt, dass es keine Turing-Maschine gibt, die auf jedem Input anhält und genau die Wörter von  $H$  akzeptiert. ○

*Verständniskontrolle 11.1:* Formulieren Sie die folgende Fragestellung als Sprachproblem: *Ist ein gegebener endlicher Automat A minimal?*



## Entscheider

Mithilfe einer geeigneten Codierung kann jede Aufgabenstellung als Sprachproblem formuliert werden. In früheren Kapiteln wurden vielfältige Arten von Maschinen vorgestellt, mit denen solche Sprachen akzeptiert werden können. Die Frage, ob eine Zahl durch drei teilbar ist, kann zum Beispiel als das Sprachproblem

$$L = \{w \in \{\emptyset, 1\}^* \mid w \text{ ist eine durch drei teilbare Binärzahl.}\}$$

formuliert werden. Der in Abschnitt 1.3.3 gefundene deterministische endliche Automat akzeptiert die Sprache  $L$ .

Da die Fähigkeiten von endlichen Automaten so beschränkt sind, fragen wir nach einer Turing-Maschine, die die Sprache erkennt. Für das Beispiel 11.3 der Primzahlen kann man den Algorithmus der Probdivision verwenden. Um herauszufinden, ob die Zahl  $p$  eine Primzahl ist, bestimmt man für alle ganzzahligen  $k \leq \sqrt{p}$  den Divisionsrest von  $p$  bei Teilung durch  $k$ . Wird ein Rest 0 gefunden, ist  $p$  keine Primzahl. Dieser Algorithmus liefert also immer eine definitive Antwort in nicht mehr als  $\sqrt{p}$  Probdivisionen.

Der Suchalgorithmus von Seite 258, der nach Lösungen einer diophantischen Gleichung sucht, hält mit Sicherheit an, wenn es eine ganzzahlige Lösung gibt. Wenn die diophantische Gleichung keine Lösung hat, terminiert er nicht. Dieser Fall ist in seinen Auswirkungen nicht von einem Programmierfehler mit einer Endlosschleife zu unterscheiden.

**Definition 11.5** (Entscheider). *Ein Entscheider ist eine Turing-Maschine, welche auf jedem Input anhält.*

Der Softwareentwickler hat normalerweise eine recht klare Vorstellung davon, wie lange eine Funktion, die er geschrieben hat, für ihre Berechnung braucht. Sobald die Funktion deutlich länger arbeitet, vermutet er einen Programmierfehler und wird die Ausführung abbrechen. Er strebt also immer an, dass seine Programme auf jeden Fall terminieren. Gute Programme sind Entscheider!

**Definition 11.6** (Entscheidbare Sprache). *Eine Sprache  $L$  heißt entscheidbar, wenn es einen Entscheider gibt, also eine Turing-Maschine  $M$ , die auf jedem Input  $w \in \Sigma^*$  anhält und genau die Wörter von  $L$  akzeptiert. Man sagt,  $M$  entscheidet die Sprache  $L = L(M)$  oder  $M$  ist ein Entscheider für die Sprache  $L$ .*

### Fleißige Biber

Wie unterscheidet man zwischen einem Programm, welches lange läuft, aber irgend wann einmal anhalten wird, und einem Programm, welches nicht anhält? Anders gefragt: Wie lange muss man eine Maschine beobachten, bis man sicher sein kann, dass sie nicht anhalten wird? Je mehr Zustände eine Turing-Maschine hat, desto mehr verschiedene Situationen können entstehen, die die Maschine durcharbeiten kann, bevor sie schließlich anhält. Offenbar hängt die Frage aber auch vom Inhalt des Bandes ab. Um diesen letzten Einfluss auszugleichen, kann man die Frage wie folgt stellen.



autospr.ch/c/5

**Aufgabe 11.7.** *Wie lange kann eine auf leerem Band gestartete Turing-Maschine mit  $n$  Zuständen höchstens laufen, bis sie anhält?*

Eine Turing-Maschine, die die maximal mögliche Laufzeit zu gegebener Anzahl Zustände erreicht, heißt ein *fleißiger Biber*. Die erreichte Anzahl Turing-Maschinenschritte definiert die *busy beaver*-Funktion  $\text{BB}(n)$ . Die Suche nach fleißigen Bibern begann mit dem Paper [48] von Tibor Radó im Jahr 1962. Erst 2024 wurde bewiesen, dass  $\text{BB}(5) = 47\,176\,870$  ist [62].  $\text{BB}(6)$  ist unbekannt. Die fleißigen Biben für  $n = 2, \dots, 5$  sind auch im Sourcecode des verlinkten Turing-Maschinensimulators als Beispiele enthalten.

Die Möglichkeiten werden noch etwas ausgeweitet, wenn man mehr als zwei Zeichen zulässt. Zum Beispiel ist bereits für mehr als zwei Zeichen nicht bekannt, wie viele Schritte eine Turing-Maschine machen kann. Man weiß aber zum Beispiel, dass eine Maschine mit drei Symbolen und vier Zuständen mehr als  $1.3 \cdot 10^{7036}$  Schritte durchlaufen kann. Diese Zahl ist so riesig, dass selbst die kombinierte Rechenleistung aller je produzierter ARM-cores zusammen in der Zeit seit dem Big Bang erst den  $10^{7000}$ -ten Teil der Arbeit geschafft hätte.

#### 11.1.4 Berechenbarkeit

Entscheidungsprobleme verlangen nur nach einer Ja-/Nein-Antwort, die durch die beiden Akzeptanzzustände  $q_{\text{accept}}$  und  $q_{\text{reject}}$  dargestellt werden kann. Zwar kann jede Aufgabenstellung in die Form eines Sprachproblems gezwungen werden und damit als Entscheidungsproblem behandelt werden, doch wird dies in vielen Fällen der Problemstellung nicht wirklich gerecht. Vielmehr wird von einem Programm oft erwartet, dass es auch Output produzieren kann.

**Definition 11.8** (Berechenbare Funktion). Eine Funktion  $f: \Sigma^* \rightarrow \Sigma^*$  heißt berechenbar, wenn es eine Turing-Maschine  $M$  gibt, die auf jedem Inputwort  $w \in \Sigma^*$  anhält und auf dem Band das Outputwort  $f(w)$  zurücklässt.

Die Arbeit [48] von Radó betrachtet nicht nur die Anzahl Schritte, die ein fleißiger Bieber ausführen kann. Sie enthält auch eine Schranke für die Größe des Funktionswertes, den eine berechenbare Funktion ergeben kann. Radó hat auch gezeigt, dass die Anzahl Schritte  $\text{BB}(n)$  des fleißigen Bibers größer ist als diese Schranke. Daraus folgt, dass die Funktion  $n \mapsto \text{BB}(n)$  nicht berechenbar ist.

Ein Entscheidungsproblem kann wie folgt als ein Berechenbarkeitsproblem formuliert werden. Die Frage, ob die Sprache  $L$  entscheidbar ist, ist gleichbedeutend damit, dass die Funktion

$$f: \Sigma^* \rightarrow \Sigma^*: w \mapsto f(w) = \begin{cases} \text{yes} & w \in L \\ \text{no} & \text{sonst} \end{cases}$$

berechenbar ist, wobei falls nötig die Zeichen für die Antworten dem Alphabet hinzugefügt werden.

## 11.2 Entscheidbare Probleme

Die in den vorangegangenen Kapiteln entwickelte Theorie der regulären Sprachen und endlichen Automaten (Kapitel 1–4) sowie der kontextfreien Grammatiken und Stackautomaten (Kapitel 5–7) stellt auch eine breite Palette von Algorithmen bereit, mit denen sich Fragestellungen über die Sprachen und Automaten beantworten lassen. Alle diese Algorithmen waren Entscheider, sie hielten auf jedem Input an. Die Abschnitte 11.2.1 und 11.2.2 verdeutlichen dies anhand einiger Beispiele. Der nächste Abschnitt 11.3 wird zeigen, dass die Situation für Turing-Maschinen viel komplizierter ist.

### 11.2.1 Entscheidbare Probleme für reguläre Sprachen

Besonders viele verschiedene Algorithmen wurden in den Kapiteln 1–4 für reguläre Sprachen entwickelt. Sie ermöglichen, so ziemlich jede Fragestellung über reguläre Sprachen zu beantworten.

#### Leerheitsproblem

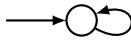
Wie kann man herausfinden, ob eine reguläre Sprache, definiert durch den endlichen Automaten  $A$ , überhaupt ein Wort enthält? Als Sprachproblem formuliert geht es darum, aus einer Zeichenkettencodierung  $\langle A \rangle$  des Automaten abzuleiten, ob es ein Wort  $w \in \Sigma^*$  gibt, welches von  $A$  akzeptiert wird. Diese Sprache wird mit

$$E_{\text{DEA}} = \{\langle A \rangle \mid A \text{ ist ein DEA und } L(A) = \emptyset.\}$$

notiert, mit  $E$  für *emptiness problem*.

Der naive Algorithmus, alle Wörter  $w \in \Sigma^*$  als Input für den Automaten zu verwenden, bis ein Wort gefunden ist, ist kein Entscheider. Denn falls tatsächlich kein Wort von  $A$  akzeptiert wird, wird dieser Algorithmus nicht terminieren.

Wenn ein endlicher Automat keine Wörter akzeptiert, dann ist der zugehörige Minimalautomat



Der folgende Algorithmus entscheidet daher die Sprache  $E_{\text{DEA}}$ :

1. Wandle den endlichen Automaten  $A$  in einen minimalen Automaten  $A'$  um.
2. Akzeptiere, falls der endliche Automat  $A'$  keinen Akzeptierzustand  $F' = \emptyset$  hat.

### Gleichheitsproblem

Akzeptieren zwei Automaten  $A_1$  und  $A_2$  die gleiche Sprache? Als Sprachproblem formuliert geht es um die Sprache

$$EQ_{\text{DEA}} = \{\langle A_1, A_2 \rangle \mid A_k \text{ sind endliche Automaten und } L(A_1) = L(A_2)\}.$$

$EQ$  steht für *equality problem*.

Auch für dieses Problem kann man sofort einen naiven Algorithmus angegeben, der kein Entscheider ist: Man kann die Menge  $\Sigma^*$  nach Wörtern durchsuchen, die vom einen Automaten akzeptiert, vom anderen aber verworfen werden. Gibt es aber kein solches Wort, wird der Algorithmus nicht terminieren, er kann also kein Entscheider sein.

In Abschnitt 1.5.4 wurde dargelegt, wie man mithilfe des Minimalautomaten den verlangten Vergleich durchführen kann. Dazu wandelt man die beiden Automaten zunächst in ihre minimalen DEAs um und schaut dann, ob sich der gleiche Automat ergeben hat.

Es gibt aber noch eine weitere, elegante Möglichkeit, die vom naiven Algorithmus suggeriert wird. Man möchte herausfinden, ob es Wörter gibt, die vom einen Automaten akzeptiert werden, nicht aber vom anderen. Diese Wörter sind in der Menge

$$L(A_1) \Delta L(A_2) = \underbrace{(L(A_1) \setminus L(A_2))}_{\text{von } A_1 \text{ akzeptiert, von } A_2 \text{ verworfen}} \cup \underbrace{(L(A_2) \setminus L(A_1))}_{\text{von } A_2 \text{ akzeptiert, von } A_1 \text{ verworfen}},$$

der bereits in Abschnitt 1.4 eingeführten symmetrischen Differenz. Die symmetrische Differenz  $B \Delta C$  zweier Mengen  $B$  und  $C$  ist genau dann leer, wenn die beiden Mengen  $B = C$  gleich sind<sup>1</sup>.

Mithilfe der Mengenoperationen für deterministische endliche Automaten lässt sich ein DEA  $A$  konstruieren, der die Menge  $L(A_1) \Delta L(A_2)$  akzeptiert (Satz 1.13). Mit dem Entscheider für  $E_{\text{DEA}}$  angewendet auf  $A$  kann jetzt entschieden werden, ob die symmetrische Differenz leer ist. Damit haben wir folgenden Entscheidungsalgorithmus gefunden:

1. Wandle  $A_k$ ,  $k = 1, 2$ , in einen deterministischen endlichen Automaten um.
2. Konstruiere mithilfe des Produktautomaten den Automaten  $A$ , der  $L(A_1) \Delta L(A_2)$  akzeptiert.

<sup>1</sup>Die symmetrische Differenz ist auch im Anhang in Abschnitt A.2.2 definiert.

3. Verwende den Entscheider für  $E_{\text{DEA}}$ . Akzeptiere  $\langle A_1, A_2 \rangle$  genau dann, wenn der Entscheider für  $E_{\text{DEA}}$  den Automaten  $A$  akzeptiert.

*Verständniskontrolle 11.2:* Ist die Sprache

$\{\langle A \rangle \mid A \text{ ist ein minimaler, deterministischer endlicher Automat.}\}$

entscheidbar?



autosp.ch/v/11.2.pdf

### Akzeptanzproblem

Akzeptiert ein endlicher Automat  $A$  ein Wort  $w \in \Sigma^*$ ? In Spezialfällen lässt sich das sehr leicht entscheiden. Das leere Wort  $\varepsilon$  wird von einem deterministischen endlichen Automaten genau dann akzeptiert, wenn der Startzustand ein Akzeptierzustand ist. Für beliebige Wörter ist die Entscheidung etwas schwieriger, es muss dazu der Automat simuliert werden. Diese Standardaufgabe wird von jeder beliebigen Regex-Engine gelöst.

Als Sprachproblem formuliert, handelt es sich bei dieser Frage um die Sprache

$$A_{\text{DEA}} = \{\langle A, w \rangle \mid \text{Der endliche Automat } A \text{ akzeptiert das Wort } w \in \Sigma^*\}.$$

es wird das Akzeptanzproblem oder auch Wortproblem für deterministische endliche Automaten genannt. Der Spezialfall  $w = \varepsilon$  wird auch mit

$$A_{\varepsilon \text{DEA}} = \{\langle A \rangle \mid \text{Der endliche Automat } A \text{ akzeptiert das leere Wort.}\}$$

bezeichnet und heißt das spezielle Akzeptanzproblem für deterministische endliche Automaten.

Für endliche Automaten ist dieses Problem recht einfach und auch für Grammatiken werden wir einen Entscheidungsalgorithmus finden. Für Turing-Maschinen stellt es sich aber heraus, dass dieses Problem nicht allgemein lösbar ist und zudem zu vielen weiteren nicht entscheidbaren Fragestellungen Anlass gibt.

### 11.2.2 Entscheidbare Probleme für kontextfreie Sprachen

Für kontextfreie Sprachen, ihre Grammatiken und Stackautomaten wurden in den Kapiteln 5–8 ebenfalls einige Algorithmen dargestellt, aus denen sich jetzt Lösungen für Entscheidungsprobleme für kontextfreie Sprachen ableiten lassen.

#### Akzeptanzproblem für kontextfreie Grammatiken

Erzeugt eine Grammatik  $G$  ein vorgegebenes Wort  $w \in \Sigma^*$ ? Als Sprachproblem formuliert, geht es um die Sprache

$$A_{\text{CFG}} = \{\langle G, w \rangle \mid \text{Die kontextfreie Grammatik } G \text{ erzeugt das Wort } w \in \Sigma^*\}.$$

Zur Beantwortung der Frage und damit zur Konstruktion eines Entscheiders dient der Cocke-Younger-Kasami-Algorithmus von Abschnitt 6.2. Da der CYK-Algorithmus eine Grammatik in Chomsky-Normalform benötigt, muss die Grammatik  $G$  als erster Schritt in Chomsky-Normalform umgewandelt werden, bevor der CYK-Algorithmus entscheiden kann, ob  $w$  produziert werden kann.

### Das spezielle Akzeptanzproblem $A_{\varepsilon \text{CFG}}$

Für den Spezialfall  $w = \varepsilon$  ist die Entscheidung noch einfacher. Ob eine Grammatik das leere Wort erzeugt, lässt sich besonders leicht in der Chomsky-Normalform (Definition 5.8) sehen. Diese kann höchstens die  $\varepsilon$ -Regel  $S \rightarrow \varepsilon$  enthalten, andere  $\varepsilon$ -Regeln sind nicht zulässig. Offenbar enthält die Chomsky-Normalform die Regel  $S \rightarrow \varepsilon$  genau dann, wenn  $\varepsilon \in L(G)$  ist. Damit kann  $A_{\varepsilon \text{CFG}}$  in folgenden Schritten entschieden werden:

1. Wandle die Grammatik  $G$  in Chomsky-Normalform  $G' = (V', \Sigma, R', S')$  um.
2. Akzeptiere, wenn die Regel  $S' \rightarrow \varepsilon$  eine Regel der Grammatik  $G'$  ist.

### Gleichheitsproblem

Das Gleichheitsproblem für kontextfreie Grammatiken fragt danach, ob zwei Grammatiken  $G_1$  und  $G_2$  die gleiche Sprache erzeugen, ob also  $L(G_1) = L(G_2)$  ist. Als Sprachproblem formuliert ist dies

$$EQ_{\text{CFG}} = \{(G_1, G_2) \mid G_k \text{ sind kontextfreie Grammatiken und } L(G_1) = L(G_2)\}.$$

Ein Entscheider für  $EQ_{\text{CFG}}$  wäre für einen Software-Entwickler, der mit Grammatiken arbeitet, ein sehr nützliches Werkzeug. Er könnte damit überprüfen, ob eine Änderung der Grammatik die Sprache verändert.

Keiner der in den Kapiteln 5–8 vorgestellten Algorithmen ist direkt in der Lage, die Sprache  $EQ_{\text{CFG}}$  zu entscheiden. Tatsächlich stellt sich heraus, dass dieses Problem nicht entscheidbar ist. Einen Beweis dieser erschütternden Tatsache findet man später in Abschnitt 11.6.2.

## 11.3 Das Akzeptanzproblem für Turing-Maschinen

In Abschnitt 11.4 werden wir zeigen, wie man mithilfe der sogenannten Reduktion verschiedene Sprachen bezüglich ihrer Entscheidbarkeit vergleichen kann. Damit lassen sich viele Sprachen als nicht entscheidbar nachweisen, sobald man eine erste, nicht entscheidbare Sprache gefunden hat. In diesem Abschnitt, genauer in Satz 11.9, soll gezeigt werden, dass das Akzeptanzproblem für Turing-Maschinen nicht entscheidbar ist. Es wird später der Ausgangspunkt für alle anderen nicht entscheidbaren Probleme sein.

### 11.3.1 Das Lügner-Paradoxon

Was passiert, wenn Pinocchio sagt “Meine Nase wird jetzt gleich wachsen”? Pinocchios Nase beginnt immer dann zu wachsen, wenn er lügt (Abbildung 11.1). Wenn er also die



Abbildung 11.1: Das Pinocchio-Paradoxon zeigt, wie Selbstbezug und Negation zu Widersprüchen führen können.

Wahrheit sagt, dann hat die Nase keinen Grund zu wachsen, seine Aussage ist also falsch, er hat gelogen. Wenn er aber lügt, dann ist die Aussage falsch, die Nase wächst nicht, obwohl sie dies wegen der Lüge tun sollte. Es gibt also kein widerspruchsfreies Verhalten von Pinocchios Nase.

Dieses Paradoxon ist auch als das Lügner-Paradoxon bekannt. Es tritt immer auf, wenn eine Aussage ihre eigene Negation behauptet. Der Kreter Epimenides soll gesagt haben: "Alle Kreter sind Lügner". Besondere Berühmtheit hat es als das Barbier-Paradoxon, das Bertrand Russell<sup>2</sup> 1918 wie folgt formuliert hat:

You can define the barber as 'one who shaves all those, and those only, who do not shave themselves.' The question is, does the barber shave himself?

In der Mengenlehre tritt das Paradoxon auf, wenn man versucht, die Menge aller Mengen zu konstruieren, die sich nicht selbst als Element enthalten.

<sup>2</sup>Das Leben Russells wird ebenso wie die in diesem Kapitel diskutierten mathematischen Grundlagenfragen im Comic [12] auf unterhaltsame Art porträtiert.

### 11.3.2 Das Akzeptanzproblem $A_{\text{TM}}$ für Turing-Maschinen

Das Akzeptanzproblem für Turing-Maschinen ist die Aufgabe, zu entscheiden, ob eine beliebige Maschine  $M$  ein beliebig vorgegebenes Wort  $w \in \Sigma^*$  akzeptieren wird<sup>3</sup>. Als Sprachproblem formuliert ist dies die Sprache

$$A_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ ist eine Turing-Maschine, die das Wort } w \in \Sigma^* \text{ akzeptiert.}\}.$$

Wieder führt der Versuch, die Maschine  $M$  auf dem Wort  $w$  zu simulieren, immer zum Erfolg, wenn die Maschine das Wort akzeptiert. Wenn die Maschine aber selbst nicht anhält, wartet man vergeblich auf eine Antwort. Diese Vorgehensweise liefert also keinen Entscheider. Der folgende Satz zeigt, dass es einen solchen auch gar nicht geben kann.

**Satz 11.9.**  $A_{\text{TM}}$  ist nicht entscheidbar.

*Beweis.* Um zu zeigen, dass  $A_{\text{TM}}$  nicht entscheidbar ist, führen wir die Annahme, dass es einen Entscheider  $H$  für  $A_{\text{TM}}$  gibt, zu einem Widerspruch. Ein solcher Entscheider ist ein Turing-Maschine, welche in der Lage wäre, eine beliebige Maschine  $M$  und ein beliebiges Wort  $w$  zu analysieren und zuverlässig zu einem Schluss zu kommen, ob  $M$  das Wort  $w$  akzeptiert. Das Problematische an diesem Entscheider ist, dass er auch ein Programm ist.  $H$  kann insbesondere für jedes beliebige Wort  $w$  entscheiden, ob  $H$  das Wort  $w$  akzeptiert. Wir markieren die Antworten von  $H$  im Folgenden **blau**.

Um jetzt einen Widerspruch herbeizuführen, nehmen wir das Muster des Lügner-Paradoxons zum Vorbild und konstruieren ein neues Programm  $D$ , welches die Antwort von  $H$  negiert. Das Programm  $D$  verlangt nach einem Wort  $w$  als Input und führt damit die in Abbildung 11.2 gezeigten Schritte aus. Das Programm  $D$  enthält den Entscheider  $H$  als Unterprogramm. Außerdem wird im orangen Teil die Antwort des Entscheiders negiert. Das Programm  $D$  soll in geeigneter Form dem Entscheider vorgelegt werden, es soll damit eine Pinocchio-Situation wie in Abbildung 11.1 provoziert werden.

Das Inputwort für  $D$  muss von der Form einer Maschinenbeschreibung  $\langle M \rangle$  sein, damit Schritt 1 des Programms sinnvoll ausgeführt werden kann. Die Antworten des Programms werden zur leichteren Verfolgbarkeit **rot** hervorgehoben.

Wir ermitteln jetzt zunächst, was passiert, wenn man dem Programm  $D$  den Input  $\langle D \rangle$  übergibt. Im ersten Schritt wird  $H$  auf dem Input  $\langle D, \langle D \rangle \rangle$  aufgerufen. Wir wissen nicht, was das Resultat sein wird, aber wir wissen, dass  $H$  ein Entscheider und daher immer ein korrektes Resultat zurückgeben wird. Je nach Antwort wird  $D$  im zweiten oder dritten Schritt das genaue Gegenteil der Antwort von  $H$  zurückgeben:

Schritt 1:	Schritt 2/3:
$H$ akzeptiert $\langle D, \langle D \rangle \rangle$	$\Rightarrow D$ verwirft $\langle D \rangle$
$H$ verwirft $\langle D, \langle D \rangle \rangle$	$\Rightarrow D$ akzeptiert $\langle D \rangle$ .

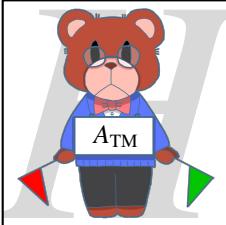
Das Verhalten von  $D$  auf  $\langle D \rangle$  vergleichen wir jetzt dem, was der in Abbildung 11.2 ganz rechts dargestellte Entscheider  $H$  aussagt. Der Entscheider  $H$  von  $A_{\text{TM}}$  sagt für ein

<sup>3</sup>Das Akzeptanzproblem wird manchmal auch das Wortproblem genannt.

Programm  $D$ , Input  $w$

```
/* H ist Teil dieses Programms */
```

Programm  $H$ , Input  $\langle M, v \rangle$



Start mit Eingabewort  $w$ :

0. Verwerfe, falls  $w$  keine TM  $w = \langle M \rangle$  ist.
1. Führe  $H$  auf dem Input  $\langle M, \langle M \rangle \rangle$  aus.
2. Falls  $H$  akzeptiert, verwerfe. 
3. Falls  $H$  verwirft, akzeptiere. 

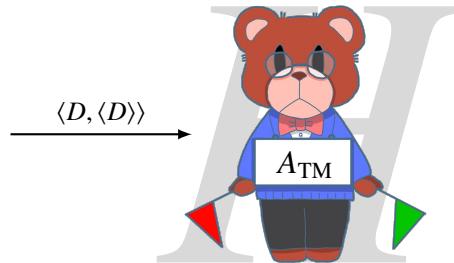


Abbildung 11.2: Definition des Programms  $D$ , dass später in der Form  $\langle D, \langle D \rangle \rangle$  dem Entscheider  $H$  vorgelegt werden soll.

Paar  $\langle M, w \rangle$ , ob  $M$  das Wort  $w$  akzeptiert. Auf dem Input  $\langle D, \langle D \rangle \rangle$  sagt  $H$  also, ob  $D$  das Wort  $\langle D \rangle$  akzeptiert. Diese Interpretation ist grün hervorgehoben:

Resultat von $H$ auf $\langle D, \langle D \rangle \rangle$	Bedeutung als Entscheider für $A_{TM}$
$H$ akzeptiert $\langle D, \langle D \rangle \rangle$	$\Rightarrow D$ akzeptiert $\langle D \rangle$
$H$ verwirft $\langle D, \langle D \rangle \rangle$	$\Rightarrow D$ verwirft $\langle D \rangle$ .

Wir sind also auf zwei verschiedenen Wegen zu einer Antwort gekommen, was  $D$  auf dem Input  $\langle D \rangle$  zurück gibt. Die Resultat sind in der Tabelle

Schritt 1: $H$ auf Input $\langle D, \langle D \rangle \rangle$	Schritt 2/3	Bedeutung der Antwort von $H$
akzeptiert	$D$ verwirft $\langle D \rangle$	$D$ akzeptiert $\langle D \rangle$
verwirft	$D$ akzeptiert $\langle D \rangle$	$D$ verwirft $\langle D \rangle$

zusammengestellt. In der Tabelle ist offensichtlich und in Abbildung 11.3 nochmals visualisiert: Die grünen und roten Antworten in den Spalten 2 und 3 widersprechen sich, die Pinocchio-Situation ist eingetreten. Die Ursache dafür ist die Annahme, dass es den Entscheider  $H$  gibt, sie ist somit falsch.  $\square$

## 11.4 Reduktion

Ausgerüstet mit dem Akzeptanzproblem können jetzt weitere Sprachen als nicht entscheidbar erkannt werden. Die Technik dazu ist die Reduktion, die in Abschnitt 11.4.1 am Bei-

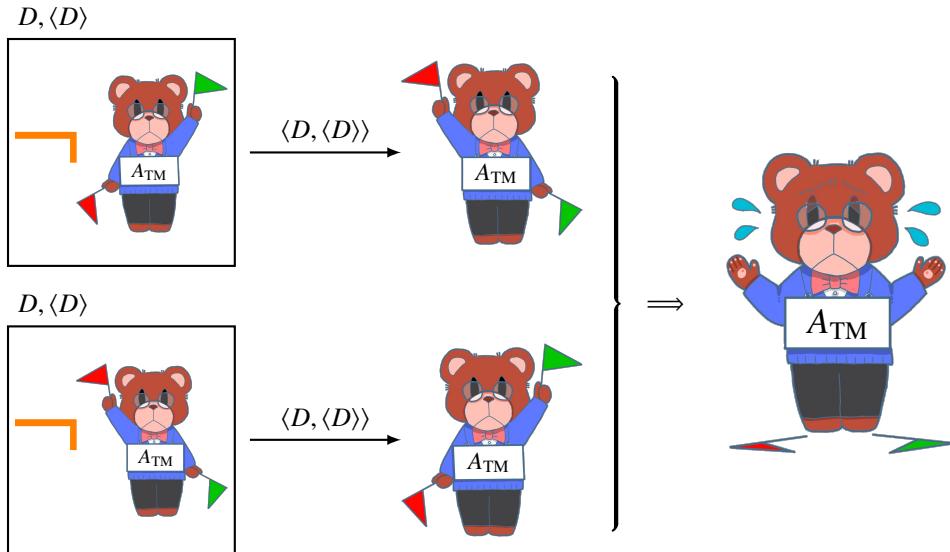


Abbildung 11.3: Versucht man, den Entscheider für  $A_{\text{TM}}$  auf  $\langle D, \langle D \rangle \rangle$  anzuwenden, entsteht eine widersprüchliche Situation. Man muss daher schließen, dass es den Entscheider nicht geben kann.

spiel des Halteproblems motiviert und in Abschnitt 11.4.2 formal entwickelt wird.

### 11.4.1 Das Halteproblem

Das Halteproblem ist die Aufgabe zu entscheiden, ob eine beliebige Turing-Maschine  $M$  auf einem beliebigen Input  $w$  anhalten wird. Als Sprachproblem formuliert ist dies die Sprache

$$\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid \text{die Turing-Maschine } M \text{ hält auf dem Input } w.\}.$$

Das Halteproblem unterscheidet sich offenbar nur wenig vom Akzeptanzproblem  $A_{\text{TM}}$ . Im Akzeptanzproblem geht es darum, ob während der Verarbeitung des Wortes  $w$  durch die Maschine der Zustand  $q_{\text{accept}}$  erreicht wird, während es im Halteproblem darum geht, ob  $q_{\text{accept}}$  oder  $q_{\text{reject}}$  erreicht werden oder nicht.

Eine noch etwas einfachere Variante des Halteproblems ist das spezielle Halteproblem

$$\text{HALT}_{\varepsilon_{\text{TM}}} = \{\langle M \rangle \mid M \text{ ist eine Turing-Maschine, die auf leerem Band anhält.}\}.$$

Offenbar sind  $\text{HALT}_{\varepsilon_{\text{TM}}}$  und  $\text{HALT}_{\text{TM}}$  ebenfalls sehr ähnlich. Die Turing-Maschine  $\langle M \rangle$  ist genau dann in  $\text{HALT}_{\varepsilon_{\text{TM}}}$ , wenn  $\langle M, \varepsilon \rangle \in \text{HALT}_{\text{TM}}$  ist. In beiden Problemen ist "Laufenlassen" keine Option, denn es gibt keine Garantie, dass die Maschine  $M$  anhalten wird.

Wäre  $\text{HALT}_{\varepsilon_{\text{TM}}}$  entscheidbar, könnte man die Ähnlichkeit von  $A_{\text{TM}}$  und  $\text{HALT}_{\varepsilon_{\text{TM}}}$  dazu verwenden, die Frage, ob  $M$  das Wort  $w$  akzeptiert, in eine neue Frage zu übersetzen, ob ein geeignet geschriebenes neues Programm anhält. Mit einem Entscheider für  $\text{HALT}_{\varepsilon_{\text{TM}}}$  könnte man dann die Antwort auf die ursprüngliche Frage geben.

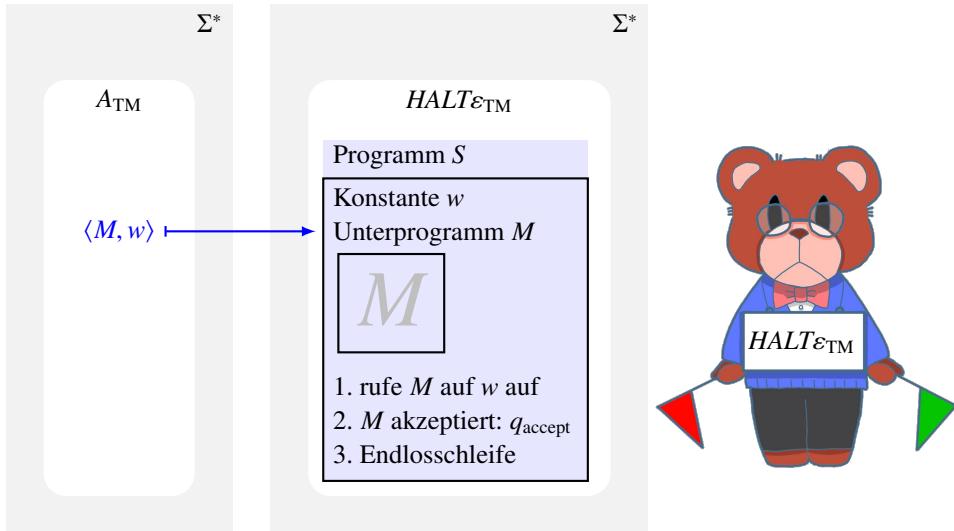


Abbildung 11.4: Übersetzung des Problems  $\langle M, w \rangle \in A_{\text{TM}}$  in das spezielle Halteproblem. Mit der blau hervorgehobenen Übersetzung  $A_{\text{TM}} \rightarrow \text{HALT}_{\mathcal{E}_{\text{TM}}} : \langle M, w \rangle \mapsto \langle S \rangle$  und dem Entscheider für  $\text{HALT}_{\mathcal{E}_{\text{TM}}}$  (rechts) kann das Akzeptanzproblem  $A_{\text{TM}}$  entschieden werden, im Widerspruch zur Nicht-entscheidbarkeit von  $A_{\text{TM}}$ .

Etwas konkreter nehmen wir an, dass es einen Entscheider für  $\text{HALT}_{\mathcal{E}_{\text{TM}}}$  gibt, er ist in Abbildung 11.4 ganz rechts gezeichnet. Außerdem betrachten wir  $\langle M, w \rangle$  und stellen die Frage, ob  $M$  das Wort  $w$  akzeptiert. Wir schreiben jetzt ein neues Programm  $S$ , welches “Akzeptieren” in “Anhalten” übersetzen soll. Die Konstruktion ist in Abbildung 11.4 dargestellt. Das neue Programm ruft zunächst das Unterprogramm  $M$  mit Input  $w$  auf, beide sind Teil des Programms  $S$ . Wenn  $M$  das Wort  $w$  akzeptiert, terminiert  $S$  mit  $q_{\text{accept}}$ . Wenn  $M$  das Wort  $w$  nicht akzeptiert, bleibt  $S$  in einer Endlosschleife. Wir haben also

$$\begin{array}{lcl} \langle M, w \rangle \in A_{\text{TM}} & \Rightarrow & \langle S \rangle \in \text{HALT}_{\mathcal{E}_{\text{TM}}} \\ \langle M, w \rangle \notin A_{\text{TM}} & \Rightarrow & \langle S \rangle \notin \text{HALT}_{\mathcal{E}_{\text{TM}}} \end{array} \quad \Rightarrow \quad \langle M, w \rangle \in A_{\text{TM}} \Leftrightarrow \langle S \rangle \in \text{HALT}_{\mathcal{E}_{\text{TM}}}.$$

Ob  $S$  anhält, kann man aber mithilfe des Entscheiders für  $\text{HALT}_{\mathcal{E}_{\text{TM}}}$  herausfinden. Man legt also  $\langle S \rangle$  dem Entscheider vor. Wenn der Entscheider die grüne Fahne hebt und damit  $\langle S \rangle \in \text{HALT}_{\mathcal{E}_{\text{TM}}}$  anzeigt, dann folgt automatisch auch, dass  $\langle M, w \rangle \in A_{\text{TM}}$ . Wenn der Entscheider die rote Fahne hebt und damit  $\langle S \rangle \notin \text{HALT}_{\mathcal{E}_{\text{TM}}}$  anzeigt, dann folgt automatisch auch, dass  $\langle M, w \rangle \notin A_{\text{TM}}$ .

Die Übersetzung von  $\langle M, w \rangle$  in das Programm  $\langle S \rangle$  kann man sich automatisiert denken. Es gibt sicher ein Programm, mit welchem man  $M$ ,  $w$  und die Programmanweisungen 1 bis 3 in ein neues File kopieren und als neues Programm  $S$  abspeichern kann. Die Übersetzungsfunktion  $\langle M, w \rangle \mapsto \langle S \rangle$  ist also eine berechenbare Funktion im Sinne der Definition 11.8.

Damit ist jetzt der folgende Satz bewiesen.

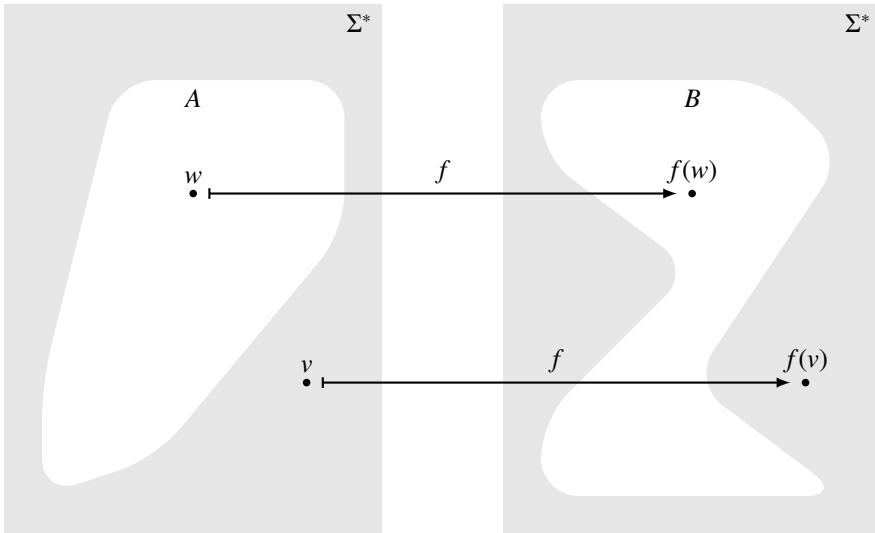


Abbildung 11.5: Reduktion  $f: A \leq B$  mithilfe der berechenbaren Abbildung  $f$ . Es muss sichergestellt sein, dass  $w \in A \Leftrightarrow f(w) \in B$ .

**Satz 11.10** (Spezielles Halteproblem).  *$\text{HALT}_{\text{TM}}$  ist nicht entscheidbar.*

### 11.4.2 Vergleich von Sprachen: Reduktion

Die Vorgehensweise des vorangegangenen Abschnitts kann noch etwas formalisiert werden. Die Konstruktion hat aus einem Wort  $\langle M, w \rangle$  auf berechenbare Art und Weise ein neues Wort  $\langle S \rangle$  gemacht. Außerdem wurden die Elemente von  $A_{\text{TM}}$  in Elemente von  $\text{HALT}_{\text{TM}}$  abgebildet.

**Definition 11.11** (Reduktion). *Eine berechenbare Abbildung  $f: \Sigma^* \rightarrow \Sigma^*$  heißt eine Reduktion der Sprache  $A$  auf die Sprache  $B$ , geschrieben  $f: A \leq B$ , wenn*

$$w \in A \Leftrightarrow f(w) \in B$$

gilt (Abbildung 11.5).  $A \leq B$  wird auch als  $A$  ist leichter entscheidbar als  $B$  gelesen.

Die Konstruktion von Abschnitt 11.4.1 ist eine Reduktion

$$A_{\text{TM}} \leq \text{HALT}_{\text{TM}}$$

im Sinne der Definition 11.11. Der Entscheider für  $\text{HALT}_{\text{TM}}$  konnte dazu verwendet werden, aus einem angenommenen Entscheider für  $\text{HALT}_{\text{TM}}$  einen Entscheider für  $A_{\text{TM}}$  zu konstruieren. Dies ist ein Spezialfall des folgenden Satzes.

**Satz 11.12.** *Seien  $A$  und  $B$  Sprachen über  $\Sigma$  und  $f: A \leq B$ . Wenn  $B$  entscheidbar ist, dann ist auch  $A$  entscheidbar. Wenn  $A$  nicht entscheidbar ist, dann ist auch  $B$  nicht entscheidbar.*

*Beweis.* Ist  $H$  ein Entscheider für  $B$ , dann kann man damit einen Entscheidungsalgorithmus für  $A$  konstruieren. Um zu entscheiden, ob  $w \in A$  ist, berechnet man erst  $f(w)$ , was wegen der Voraussetzung der Berechenbarkeit von  $f$  möglich ist. Anschließend wendet man  $H$  auf  $f(w)$  an. Genau dann, wenn  $H$  das Wort  $f(w)$  akzeptiert, ist  $w \in A$ . Somit folgt, dass  $A$  entscheidbar ist.

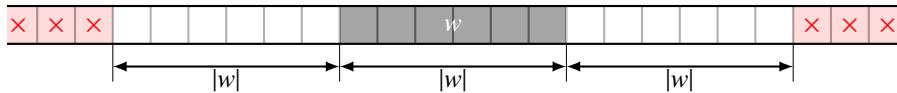
Wenn  $A$  nicht entscheidbar ist, dann führt die Annahme, dass  $H$  ein Entscheider für  $B$  ist, zu einem Widerspruch. Also kann  $B$  nicht entscheidbar sein.  $\square$

---

**Verständniskontrolle 11.3:** Reduzieren Sie das folgende Problem auf das Halteproblem.



Gegeben ist eine Turing-Maschine  $M$  und ein Wort  $w$  auf dem Band. Entscheide, ob die Turing-Maschine während ihrer Laufzeit den Bereich des Bandes verlässt, der  $|w|$  Zeichen vor dem ersten Zeichen von  $w$  beginnt und  $|w|$  Zeichen nach dem letzten Zeichen von  $w$  endet.



### 11.4.3 Implementationseigenschaften

Ist die Sprache

$$\text{USES-STATE}_{\text{TM}} = \{\langle M, q, w \rangle \mid M \text{ braucht für die Verarbeitung von } w \text{ den Zustand } q.\}$$

entscheidbar? Einige Spezialfälle sind leicht entscheidbar. Wenn  $q = q_0$  der Startzustand ist, dann ist  $\langle M, q_0, w \rangle \in \text{USES-STATE}_{\text{TM}}$  aufgrund der Definition, wie eine Turing-Maschine arbeitet. Auch lässt sich die Reduktion

$$f: A_{\text{TM}} \leq \text{USES-STATE}_{\text{TM}} : \langle M, w \rangle \rightarrow \langle M, q_{\text{accept}}, w \rangle$$

konstruieren. Da  $A_{\text{TM}}$  nicht entscheidbar ist, kann auch  $\text{USES-STATE}_{\text{TM}}$  nicht entscheidbar sein.

Für einen Softwareentwickler stellt sich die Fragestellung etwas anders dar. Um herauszufinden, ob ein Programm zur Laufzeit einen bestimmten Zustand erreichen wird, würde er das Programm in den Debugger laden und Breakpoints oder Bedingungen setzen, die das Programm anhalten würden. Die Frage, ob ein Zustand erreicht wird, wird vom Debugger in die Frage übersetzt, ob der Debugger mit geladenem Programm anhält.

Die Frage, ob ein bestimmter Zustand während der Verarbeitung verwendet wird, ist eine Eigenschaft, die nur von der Implementation abhängt. Sie hat keinen Einfluss auf die von einer Maschine akzeptierte Sprache. Die Möglichkeit, mit einem Debugger aus einer Implementationseigenschaft "Anhalten" zu machen, suggeriert, dass Implementationseigenschaften kaum je entscheidbar sind.

Ein Sicherheitsforscher könnte zum Beispiel fragen, ob ein Programm zur Laufzeit eine Netzwerkverbindung aufbauen würde. Auch diese Eigenschaft ließe sich mit einem Debugger testen, indem Breakpoints auf Aufrufen von Netzwerkfunktionen gesetzt werden. Man darf daher vermuten, dass die Frage nicht entscheidbar ist. Tatsächlich kann man leicht eine Reduktion von  $A_{TM}$  konstruieren. Aus  $\langle M, w \rangle$  macht man das Programm, welches erst  $M$  auf dem Wort  $w$  laufen lässt. Wenn  $M$  das Wort akzeptiert, wird eine Netzverbindung aufgebaut. Da  $A_{TM}$  nicht entscheidbar ist, kann auch nicht entscheidbar sein, ob ein Programm eine Netzwerkverbindung aufbauen wird.

## 11.5 Spracheigenschaften und der Satz von Rice

In Abschnitt 11.4.3 wurde angedeutet, dass Implementationseigenschaften von Programmen nicht entscheidbar sind. Im Allgemeinen interessieren die Details einer Implementation nicht, wichtig ist nur, dass ein Programm korrekt arbeitet. Auch für diesen Aspekt lässt sich die Reduktionsidee einsetzen.

### 11.5.1 Spracheigenschaften

In Abschnitt 1.1.4 wurde die Sprache C als diejenigen Zeichenketten definiert, die von einem C-Compiler akzeptiert werden. In der Praxis entsteht jedoch erst eine Spezifikation der Programmiersprache, bevor der Compiler dazu geschrieben wird. Die brennende Frage ist dann, ob der Compiler richtig arbeitet, also genau diejenigen Zeichenketten als zulässigen Sourcecode akzeptiert, die auch von der Spezifikation als korrektes C definiert worden sind.

Der Compiler-Entwickler ist also ständig mit dem Problem konfrontiert zu überprüfen, ob sein Produkt immer noch die Spezifikation erfüllt. Jede Code-Änderung birgt die Gefahr, die akzeptierte Sprache zu verändern und damit die Spezifikation zu verletzen. Ein Softwaretool, welches die Korrektheit des Compilers prüft, wäre hier sehr nützlich. Da es abzählbar unendlich viele mögliche Sourcecodes gibt, ist es der naive Ansatz, den Compiler auf allen korrekten Source-Codes zu testen, nicht ein Entscheider. Gibt es einen Entscheider, der den Sourcecode des Compilers analysiert und ermitteln kann, ob die akzeptierte Sprache des Compilers der Spezifikation entspricht?

Etwas formaler ist ein Entscheider gesucht, der die Sprache

$$C_{TM} = \{ \langle M \rangle \mid \text{Die Turing-Maschine } M \text{ akzeptiert genau korrekten C-Sourcecode.} \}$$

entscheidet. Man beachte die Ähnlichkeit in der Notation mit  $A_{TM}$ . Um zu entscheiden, ob  $\langle M, w \rangle \in A_{TM}$  könnten man auch fragen, ob die akzeptierte Sprache von  $M$  das Wort  $w$  enthält, also ob  $\langle M \rangle$  in

$$A(w)_{TM} := \left\{ \langle M \rangle \mid \begin{array}{l} \text{Die akzeptierte Sprache der Turing-Maschine } M \\ \text{hat die Eigenschaft, dass sie das Wort } w \text{ enthält.} \end{array} \right\} \quad (11.2)$$

liegt. Die Formulierung macht klar, dass es darum geht, Programme  $\langle M \rangle$  daraufhin zu untersuchen, ob die akzeptierte Sprache  $L(M)$  eine bestimmte Eigenschaft  $P$  hat.

**Definition 11.13.** Sei  $P$  eine Eigenschaft von Sprachen über  $\Sigma$ . Dann ist

$$P_{\text{TM}} = \left\{ \langle M \rangle \mid \begin{array}{l} \text{Die akzeptierte Sprache } L(M) \text{ der Turing-} \\ \text{Maschine } M \text{ hat die Eigenschaft } P. \end{array} \right\}$$

die Sprache der Programme, die eine Sprache mit Eigenschaft  $P$  akzeptieren.

Es ist plausibel, dass  $A(w)_{\text{TM}}$  (siehe (11.2)) nicht entscheidbar ist. In den folgenden Abschnitten soll gezeigt werden, dass auch  $P_{\text{TM}}$  für einige naheliegende Eigenschaften nicht entscheidbar ist. Die Beispiele werden eine zusätzliche Voraussetzung an die Eigenschaft  $P$  zutage fördern, bevor allgemein bewiesen werden kann, dass  $P_{\text{TM}}$  nicht entscheidbar ist. Dies ist der Inhalt von Abschnitt 11.5.5.

Zu jeder Eigenschaft  $P$  können wir auch die negierte Eigenschaft  $\bar{P}$  betrachten. Eine Sprache hat die Eigenschaft  $\bar{P}$  genau dann, wenn sie die Eigenschaft  $P$  nicht hat. Kann man  $\bar{P}$  entscheiden, dann kann man offenbar auch  $P$  entscheiden, indem man die Antwort eines Entscheiders für  $\bar{P}$  negiert. Wir sind also frei, jederzeit  $P$  durch  $\bar{P}$  zu ersetzen.

## 11.5.2 Leerheitsproblem

Kann man entscheiden, ob eine Turing-Maschine überhaupt irgendein Wort akzeptiert? Formal handelt es sich hierbei um die Frage, ob die akzeptierte Sprache leer ist. Schreiben wir  $E$  für die Eigenschaft, dass die Sprache leer ist, erhalten wir das *Leerheitsproblem*

$$E_{\text{TM}} = \{ \langle M \rangle \mid M \text{ ist eine Turing-Maschine und } L(M) = \emptyset \}.$$

Der naive Versuch, alle Wörter von  $\Sigma^*$  mit  $M$  zu verarbeiten liefert zwar zuverlässig eine Antwort, wenn die akzeptierte Sprache nicht leer ist. Wenn  $M$  aber keine Wörter akzeptiert, endet dieser Prozess nicht und ist damit kein Entscheider.

Wir vermuten, dass  $E_{\text{TM}}$  nicht entscheidbar ist. Wir formalisieren den Beweis mit der in Abbildung 11.6 dargestellten Reduktion  $A_{\text{TM}} \leq E_{\text{TM}}$ . Sie macht aus  $\langle A, w \rangle$  ein neues Programm  $S$ . Der Entscheider für  $E_{\text{TM}}$  ist in der Lage herauszufinden, ob die von  $S$  akzeptierte Sprache leer ist oder nicht.

Wir müssen ermitteln, welche Sprache  $S$  genau akzeptiert. Dazu hat das Programm auch das Inputwort  $u$ , welches aber im blau hinterlegten Programmtext nicht weiter verwendet wird, alle Wörter werden gleichbehandelt. Die akzeptierte Sprache ist also entweder  $\emptyset$  oder  $\Sigma^*$ . Welcher Fall eintritt, hängt davon ab, was bei der Verarbeitung von  $w$  durch  $M$  im ersten Schritt passiert. Wenn  $M$  akzeptiert, terminiert  $S$  im zweiten Schritt mit  $q_{\text{accept}}$ , d. h. jedes Wort  $u$  wird von  $S$  akzeptiert und  $L(S) = \Sigma^*$ . Wenn  $M$  verwirft oder nicht anhält, dann wird auch das Wort  $u$  von  $S$  nicht akzeptiert, also  $L(S) = \emptyset$ . Es gilt also

$$\langle M, w \rangle \in A_{\text{TM}} \Leftrightarrow L(S) = \Sigma^* \Leftrightarrow L(S) \neq \emptyset \Leftrightarrow \langle S \rangle \in \bar{E}_{\text{TM}}.$$

Damit ist gezeigt, dass die Reduktionsabbildung  $\langle M, w \rangle \mapsto S$  “Akzeptieren” in “nicht leer” übersetzt.

Das Programm  $S$  kann jetzt dem Entscheider für  $E_{\text{TM}}$  vorgelegt werden:

$$\langle M, w \rangle \in A_{\text{TM}} \Leftrightarrow \text{Entscheider für } E_{\text{TM}} \text{ verwirft } \langle S \rangle.$$

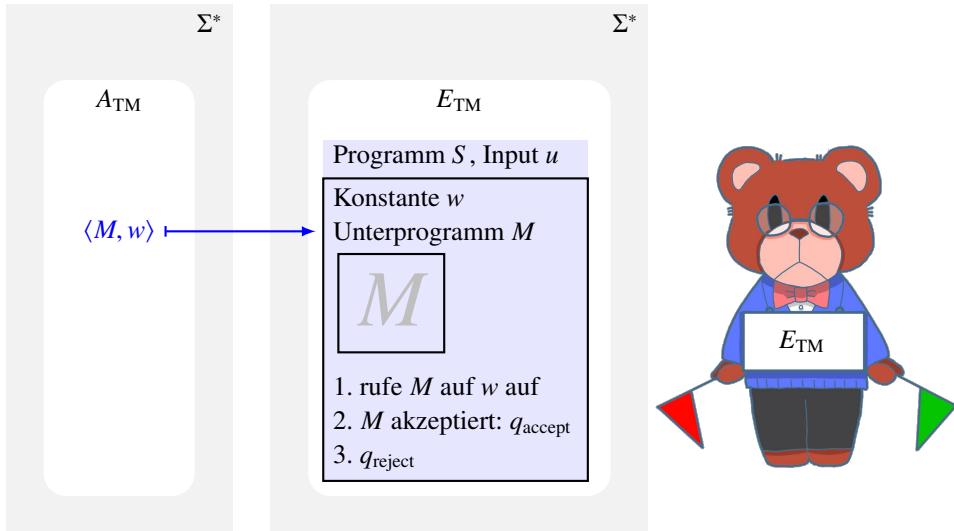


Abbildung 11.6: Übersetzung des Problems  $\langle M, w \rangle \in A_{\text{TM}}$  in das Leerheitsproblem  $E_{\text{TM}}$ . Mit der blau hervorgehobenen Übersetzung  $A_{\text{TM}} \rightarrow E_{\text{TM}} : \langle M, w \rangle \mapsto \langle S \rangle$  und dem Entscheider für  $E_{\text{TM}}$  (rechts) kann das Akzeptanzproblem  $A_{\text{TM}}$  entschieden werden, im Widerspruch zur Nichtentscheidbarkeit von  $A_{\text{TM}}$ .

Mithilfe des Entscheiders für  $E_{\text{TM}}$  wäre es also möglich,  $A_{\text{TM}}$  zu entscheiden. Dies ist ein Widerspruch zur Nichtentscheidbarkeit von  $A_{\text{TM}}$ , also kann es den Entscheider nicht geben.

**Satz 11.14** (Leerheitsproblem). *Das Leerheitsproblem  $E_{\text{TM}}$  ist nicht entscheidbar.*

### 11.5.3 Regularitätsproblem

Kann man entscheiden, ob die von einer Turing-Maschine akzeptierte Sprache regulär ist, oder in anderen Worten, ist die Sprache

$$\text{REGULAR}_{\text{TM}} = \{\langle M \rangle \mid \text{die Turing-Maschine } M \text{ akzeptiert eine reguläre Sprache.}\}$$

entscheidbar? Es geht hier offenbar um die Eigenschaft *REGULAR* einer Sprache, regulär zu sein.

Die Reduktionskonstruktion von Abbildung 11.7 ist sehr ähnlich der Konstruktion für das Leerheitsproblem. Aus  $\langle A, w \rangle$  wird ein neues Programm  $S$  erzeugt, uns interessiert die von  $S$  erkannte Sprache. Diesmal wird jedoch  $u$  im ersten Schritt daraufhin geprüft, ob es von der Form  $0^n 1^n$  ist. Alle Wörter, die nicht von dieser Form sind, werden verworfen. Die erkannte Sprache des Programms  $S$  enthält also höchstens alle Wörter der Form  $0^n 1^n$ . Ob diese Wörter auch wirklich akzeptiert werden, hängt vom Resultat der Verarbeitung von  $w$  durch  $M$  im zweiten Schritt ab. Wenn  $M$  das Wort  $w$  akzeptiert, werden die Wörter  $0^n 1^n$

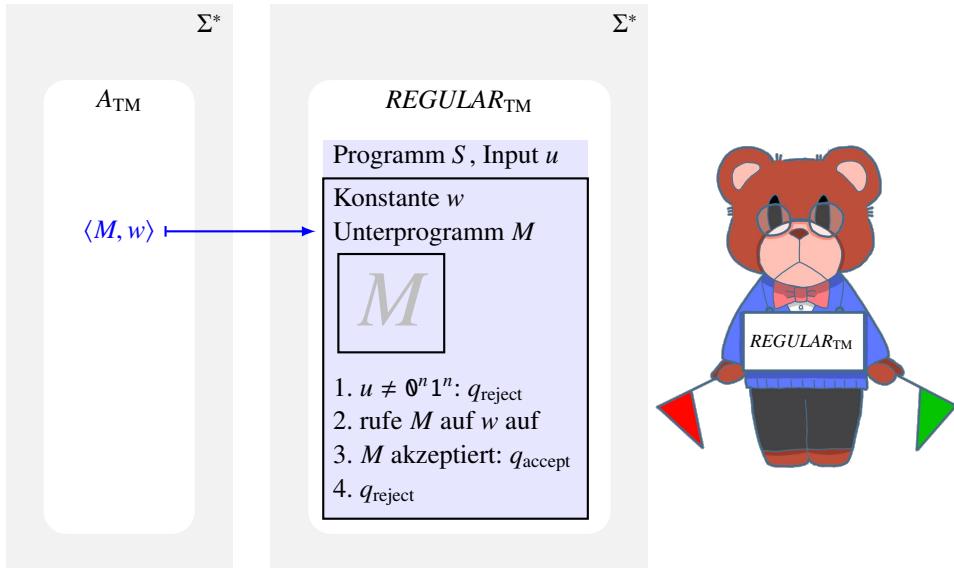


Abbildung 11.7: Übersetzung des Problems  $\langle M, w \rangle \in A_{\text{TM}}$  in das Regularitätsproblem  $REGULAR_{\text{TM}}$ . Mit der blau hervorgehobenen Übersetzung  $A_{\text{TM}} \rightarrow REGULAR_{\text{TM}} : \langle M, w \rangle \mapsto \langle S \rangle$  und dem Entscheider für  $REGULAR_{\text{TM}}$  (rechts) kann das Akzeptanzproblem  $A_{\text{TM}}$  entschieden werden, im Widerspruch zur Nichtentscheidbarkeit von  $A_{\text{TM}}$ .

akzeptiert. Wenn  $M$  nicht akzeptiert oder nicht hält, wird nichts akzeptiert. Somit ist

$$\begin{aligned} \langle M, w \rangle \in A_{\text{TM}} &\Rightarrow L(S) = \{0^n 1^n \mid n \geq 0\} \text{ nicht regulär,} \\ \langle M, w \rangle \notin A_{\text{TM}} &\Rightarrow L(S) = \emptyset \text{ regulär.} \end{aligned} \quad (11.3)$$

Da die Sprache  $\{0^n 1^n \mid n \geq 0\}$  nicht regulär, die Sprache  $\emptyset$  aber regulär ist, kann (11.3) auch als

$$\langle M, w \rangle \in A_{\text{TM}} \Leftrightarrow \langle S \rangle \in \overline{REGULAR}_{\text{TM}}$$

formuliert werden. Die Reduktionsabbildung übersetzt ‘Akzeptieren’ in ‘nicht regulär’. Der Entscheider für  $REGULAR_{\text{TM}}$  kann entscheiden, ob  $\langle S \rangle \in REGULAR_{\text{TM}}$  ist und ermöglicht damit,  $A_{\text{TM}}$  zu entscheiden. Dieser Widerspruch zur Nichtentscheidbarkeit von  $A_{\text{TM}}$  zeigt, dass auch  $REGULAR_{\text{TM}}$  nicht entscheidbar ist.

**Satz 11.15** (Regularitätsproblem für Turing-Maschinen).  $REGULAR_{\text{TM}}$  ist nicht entscheidbar.

*Verständniskontrolle 11.4:* Reduzieren Sie  $A_{\text{TM}}$  auf das Problem

$$CFL_{\text{TM}} = \left\{ \langle M \rangle \mid \begin{array}{l} \text{Die Turing-Maschine } M \text{ akzeptiert} \\ \text{eine kontextfreie Sprache.} \end{array} \right\}$$



### 11.5.4 Nichttriviale Eigenschaften

Die beiden Beispiele  $E_{\text{TM}}$  und  $REGULAR_{\text{TM}}$  zeigen zwei weitere Eigenschaften, die die Eigenschaft  $P$  haben muss, wenn man nach dem gleichen Muster schließen will, dass  $P_{\text{TM}}$  nicht entscheidbar ist.

#### Zwei Beispielsprachen

In der Reduktionsabbildung für  $E_{\text{TM}}$  wurde verwendet, dass  $\emptyset$  die Eigenschaft  $E$  hat, aber  $\Sigma^*$  nicht. In der Reduktionsabbildung für  $REGULAR_{\text{TM}}$  wurde verwendet, dass  $\emptyset$  regulär ist,  $\{\emptyset^n 1^n \mid n \geq 0\}$  aber nicht. Wir haben also immer zwei Sprachen gebraucht, wovon eine die Eigenschaft hatte, die andere nicht.

**Definition 11.16** (Nichttriviale Eigenschaft einer Sprache). *Eine Eigenschaft  $P$  von Sprachen heißt nichttrivial, wenn es zwei Sprachen  $L_1$  und  $L_2$  gibt derart, dass  $L_1$  die Eigenschaft  $P$  hat und  $L_2$  nicht.*

Wenn  $P$  nichttrivial ist mit den Beispielsprachen  $L_1$  und  $L_2$ , dann ist offenbar auch  $\overline{P}$  nichttrivial mit den vertauschten Beispielsprachen  $L_2$  und  $L_1$ . Man kann sogar annehmen, dass eine der Beispielsprachen  $\Sigma^*$  oder  $\emptyset$  ist.

#### Turing-erkennbar

Im Programm  $S$  für  $REGULAR_{\text{TM}}$  wurde im ersten Schritt geprüft, ob das Wort  $u$  von der Form  $\emptyset^n 1^n$  ist. Da dieser Schritt Teil des Programms  $S$  ist, muss sichergestellt werden, dass er mit einer Turing-Maschine tatsächlich durchführbar ist. In Kapitel 10 wurde eine Turing-Maschine für die Sprache  $\emptyset^n 1^n$  mit dem Zustandsdiagramm von Abbildung 10.6 vorgestellt. Für eine beliebige Sprache  $L_1$  oder  $L_2$  ist aber nicht automatisch klar, dass es so eine Turing-Maschine gibt. Es muss daher zusätzlich verlangt werden, dass die beiden Sprachen  $L_1$  und  $L_2$  Turing-erkennbar sind. Dies garantiert die Existenz von Turing-Maschinen  $T_1$  und  $T_2$  mit  $L_1 = L(T_1)$  und  $L_2 = L(T_2)$ .

### 11.5.5 Der Satz von Rice

Für nichttriviale Eigenschaften können die Resultate für  $E_{\text{TM}}$  und  $REGULAR_{\text{TM}}$  auf eine viel größere Klasse verallgemeinert werden.

**Satz 11.17** (Rice). *Ist  $P$  eine nichttriviale Eigenschaft Turing-erkennbarer Sprachen, dann ist  $P_{\text{TM}}$  nicht entscheidbar.*

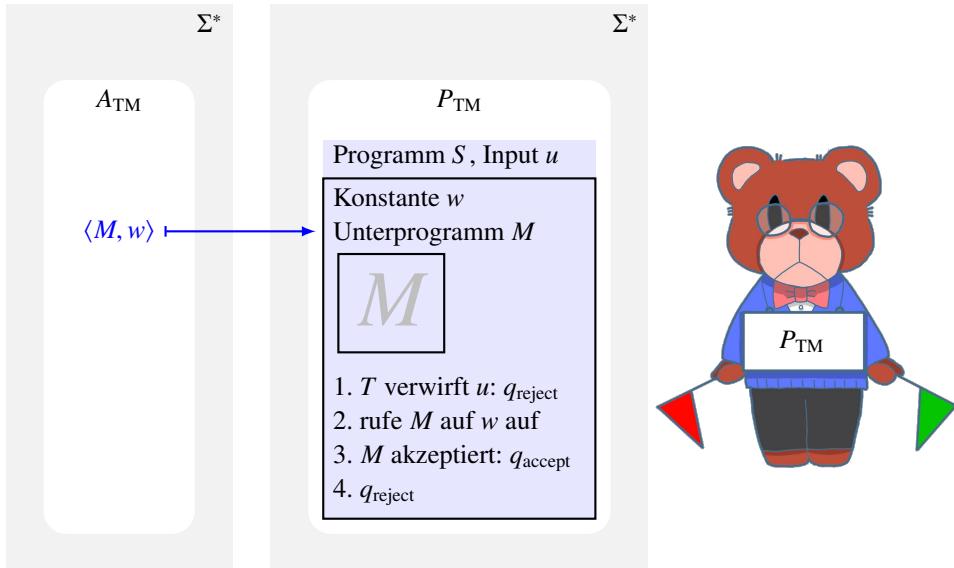


Abbildung 11.8: Übersetzung des Problems  $\langle M, w \rangle \in A_{\text{TM}}$  in das Problem  $P_{\text{TM}}$ . Mit der blau hervorgehobenen Übersetzung  $A_{\text{TM}} \rightarrow P_{\text{TM}} : \langle M, w \rangle \mapsto \langle S \rangle$  und dem Entscheider für  $P_{\text{TM}}$  (rechts) kann das Akzeptanzproblem  $A_{\text{TM}}$  entschieden werden, im Widerspruch zur Nichtentscheidbarkeit von  $A_{\text{TM}}$ .

*Beweis.* Da  $P$  eine nichttriviale Eigenschaft Turing-erkennbarer Sprachen ist, gibt es zwei Beispielsprachen samt Turing-Maschinen, die sie erkennen. Wir dürfen annehmen, dass  $L_1 = \emptyset$  die Eigenschaft  $P$  nicht hat. Weiter nehmen wir an, dass  $L_2$  von der Turing-Maschine  $T$  erkannt wird und die Eigenschaft  $P$  hat. Damit konstruieren wir die in Abbildung 11.8 gezeigte Reduktionsabbildung, die aus  $\langle A, w \rangle$  ein neues Programm  $S$  erzeugt.

Es muss die von  $S$  akzeptierte Sprache bestimmt werden. Im ersten Schritt werden alle Wörter außerhalb der Sprache  $L_2$  verworfen, es ist also  $L(S) \subset L_2$ . Die nachfolgenden Schritte sind für alle  $w$  gleich, im schlimmsten Fall werden alle verbleibenden Wörter verworfen, in diesem Fall ist  $L(S) = \emptyset$ .

Welcher der Fälle eintritt, hängt davon ab, ob im zweiten Schritt des Programms  $M$  das Wort  $w$  akzeptiert. Akzeptiert  $M$  das Wort  $w$ , dann terminiert  $S$  mit  $q_{\text{accept}}$ . Dies ist also der Fall, in dem  $L(S) = L_2$  ist. In allen anderen Fällen bleibt es bei  $L(S) = \emptyset$ . Wir haben also

$$\begin{aligned} \langle A, w \rangle \in A_{\text{TM}} &\Rightarrow L(S) = L_2 \quad \text{hat } P \\ \langle A, w \rangle \notin A_{\text{TM}} &\Rightarrow L(S) = \emptyset \quad \text{hat } P \text{ nicht} \end{aligned} \quad \Rightarrow \quad \langle A, w \rangle \in A_{\text{TM}} \Leftrightarrow \langle S \rangle \in P_{\text{TM}}.$$

Ein Entscheider für  $P_{\text{TM}}$  kann entscheiden, ob  $\langle S \rangle \in P_{\text{TM}}$  und damit ob  $\langle M, w \rangle \in A_{\text{TM}}$  ist, im Widerspruch zur Nichtentscheidbarkeit von  $A_{\text{TM}}$ . Somit kann es keinen Entscheider für  $P_{\text{TM}}$  geben.  $\square$

Der Satz von Rice verlangt nur, dass das Programm  $T$  die Sprache  $L_2$  erkennt. Für

gewisse Wörter in  $\Sigma^* \setminus L_2$  könnte es daher sein, dass das Programm  $S$  den zweiten Schritt gar nicht erreicht. Auf diesen Wörtern hält  $S$  nicht an, also gehören sie auch nicht zur akzeptierten Sprache  $L(S)$ .

Der Satz von Rice besagt, dass fast beliebige Eigenschaften der von einer Turing-Maschine akzeptierten Sprache nicht entscheidbar sind. Es ist also zum Beispiel nicht möglich, ein Softwareprogramm zu schreiben, welches den Sourcecode eines beliebigen C-Compilers analysieren und entscheiden kann, oder er tatsächlich die Sprache C erkennt. Stattdessen werden Validierungssuiten verwendet, die im Wesentlichen eine große Zahl von Testfällen sind, die der Compiler erfolgreich verarbeiten müssen.

---

*Verständniskontrolle 11.5:* Zeigen Sie mithilfe des Satzes von Rice, dass die Frage, ob eine Turing-Maschine eine kontextfreie Sprache akzeptiert, nicht entscheidbar ist.




---

Der Satz 11.17 von Rice hat auch Konsequenzen für das allgemeine Vorgehen beim Testen von Software. Da es keine generische Verifikation geben kann, ob eine Software bestimmte Eigenschaften hat, bleiben nur zwei Möglichkeiten. Man kann versuchen, in einem Einzelfall eine Verifikation der Software durchzuführen. Man konstruiert einen mathematischen Beweis, dass die Software korrekt ist. Dies ist sehr aufwendig und lässt sich nur in Ausnahmefällen rechtfertigen. Als Alternative bleibt somit nur, eine große Zahl von Testfällen zusammenzutragen, die möglichst jeden Aspekt der Software abdecken, beginnend bei den kleinsten sinnvoll testbaren Einheiten. Unit-Testing ist die pragmatische Antwort, sie macht aus der vom Satz von Rice geschaffenen Not eine Tugend.

## 11.6 Nicht entscheidbare Probleme und die Berechnungsgeschichte

Die bisher als nicht entscheidbar erkannten Sprachen hatten alle ganz explizit mit Turing-Maschinen zu tun. Das könnte den Eindruck vermitteln, dass für eine nicht entscheidbare Situation immer eine Turing-Maschine verantwortlich ist, die Teil der Aufgabenstellung ist. Fragestellungen über Zeichenkettenprobleme ohne eine Turing-Maschine, zum Beispiel über kontextfreie Grammatiken, scheinen gegen Nichtentscheidbarkeit immun zu sein. Dieser Eindruck ist falsch.

Eine Turing-Maschine ist eine sehr kompakte Möglichkeit, den Berechnungsprozess für das Akzeptieren eines Wortes zu beschreiben. Die Nichtentscheidbarkeitssätze dieses Kapitels sind in dieser Sichtweise vor allem das Eingeständnis, dass die Komplexität einer solchen Berechnung undurchschaubar ist. Eine andere Beschreibung der Berechnung kann diese Komplexität nicht verstecken. Sobald eine Aufgabenstellung kompliziert genug ist, dass sich darin die Berechnung codieren lässt, ist auch damit zu rechnen, dass diese Aufgabenstellung nicht entscheidbar sein wird.

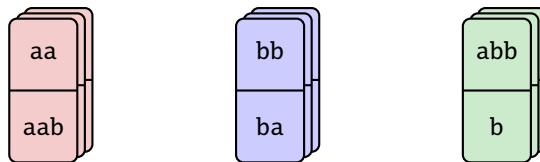
Die Berechnungsgeschichte von Abschnitt 10.1.6 enthält ebenfalls die vollständige Information über den Ablauf der Berechnung auf einem Inputwort  $w$ . Sie besteht aus

einer Folge von Zeichenketten, die sich nur nach sehr einfachen Regeln verändern dürfen. Eine Aufgabenstellung  $L$ , die diese Regeln codieren kann, wird nicht entscheidbar sein. Genauer: Kann man die Frage nach der Existenz einer Berechnungsgeschichte einer Turing-Maschine  $M$  mit dem Input  $w$  als Instanz der Aufgabe  $L$  codieren, dann gibt es eine Reduktion von  $A_{\text{TM}}$  auf  $L$  und  $L$  ist nicht entscheidbar.

In diesem Abschnitt sollen zwei Beispiele von nicht entscheidbaren Problemen dieser Art skizziert werden. Das Post-Korrespondenz-Problem von Abschnitt 11.6.1 zeigt, dass das Phänomen der Nichtentscheidbarkeit sogar in einer Variante des Dominospieles auftreten kann. Abschnitt 11.6.2 zeigt, dass das Gleichheitsproblem für kontextfreie Sprachen nicht entscheidbar ist. Dies hat offensichtliche praktische Konsequenzen für den Softwareentwickler, der mit kontextfreien Grammatiken arbeitet. Es kann kein Softwaretool geben, welches allgemein entscheiden kann, ob eine Umformulierung einer Grammatik die produzierte Sprache ändert.

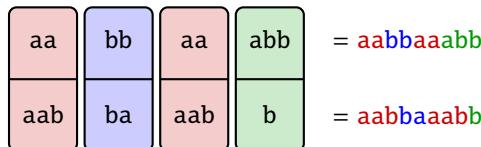
### 11.6.1 Das Post-Korrespondenz-Problem

Dominosteine enthalten jeweils Paare von natürlichen Zahlen. Im Dominispiel müssen die Spieler abwechselnd ihre Steine so aneinanderreihen, dass jeweils gleiche Zahlen aneinander stoßen. Ein ähnliches Spiel für Zeichenketten hat sich Emil Leon Post ausgedacht. Die beiden Hälften der Dominosteine enthalten jetzt aber beliebige Zeichenketten, die auch verschieden lang sein können. Zum Beispiel könnten die Dominosteine



gegeben sein. Wie in der Zeichnung angedeutet, stehen von jedem Stein beliebig viele Kopien zur Verfügung.

Die Aufgabe ist die Folgende: Kann man Steine wählen und nebeneinanderlegen, so dass die zusammengehängten Zeichenketten oben und unten auf den Steinen gleich werden:



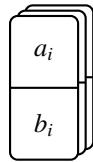
Etwas abstrakter formuliert lautet das Problem wie folgt.

**Problem 11.18** (Post-Korrespondenz-Problem). *Gegeben sind zwei Listen von Zeichenketten  $a_1, \dots, a_n$  und  $b_1, \dots, b_n$ . Gibt es eine Folge von Zahlen  $i_1, \dots, i_k$  derart, dass*

$$a_{i_1} a_{i_2} \dots a_{i_k} = b_{i_1} b_{i_2} \dots b_{i_k}$$

*ist? Unter den Zahlen  $i_1, i_2, \dots, i_k$  darf die gleiche Zahl mehrfach vorkommen.*

Die Zeichenketten  $a_i$  und  $b_i$  soll man sich als den Inhalt des Steins mit der Nummer  $i$  vorstellen:



Doch warum ist das Problem 11.18 nicht entscheidbar? Wie in der Einleitung angedeutet, muss ein Weg gefunden werden, die Frage nach einer akzeptierenden Berechnungsgeschichte in einem Post-Korrespondenz-Problem zu codieren. Aus einem Akzeptanzproblem  $\langle M, w \rangle$  muss eine Liste von Dominosteinen konstruiert werden, mit der sich nur akzeptierende Berechnungsgeschichten zusammensetzen lassen. Lässt sich das Korrespondenzproblem lösen, gibt es eine akzeptierende Berechnungsgeschichte und das Akzeptanzproblem ist gelöst. Ein Entscheider für das Korrespondenzproblem würde also  $A_{TM}$  entscheiden, ein Widerspruch zur Tatsache, dass  $A_{TM}$  nicht entscheidbar ist.

Wir nehmen also an, dass  $M$  das Wort  $w$  akzeptiert. Die Berechnungsgeschichte besteht aus einzelnen Zeilen wie folgt.

$$\begin{array}{cccccc}
 q_0 & 0 & 1 & 0 & 1 \\
 1 & q_1 & 1 & 0 & 1 \\
 1 & 1 & q_3 & 0 & 1 \\
 \vdots & \vdots & \vdots & \vdots & \vdots \\
 1 & q_a & 0 & 0 & 1
 \end{array} \tag{11.4}$$

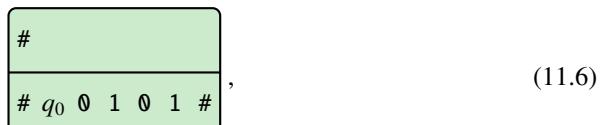
In der Form (11.4) kann sie natürlich im Post-Korrespondenz-Problem nicht auftreten, da dort genau zwei identische Zeilen gefunden werden müssen. Wir wandeln sie daher als erstes in eine einzige Zeile um, indem wir ein neues Trennzeichen # einführen. Damit ist jetzt

$$\# q_0 0 1 0 1 \# 1 q_1 1 0 1 \# 1 1 q_3 0 1 \# \dots \# 1 q_a 0 0 1 \# \tag{11.5}$$

die Berechnungsgeschichte.

Das Korrespondenzproblem muss jetzt aus geeignet gewählten Dominosteinen zwei Kopien der Zeile (11.5) erzeugen. Die beiden Zeilen und ihr Aufbau aus Dominosteinen sind in Abbildung 11.9 zusammengefasst. Im Folgenden werden die verschiedenen, mit Farben codierten Typen von Steinen aus der Definition der Turing-Maschine  $M$  und dem Wort  $w$  konstruiert.

Damit die untere Zeile in Abbildung 11.9 mit  $q_00101$  beginnt, konstruiert man den Startstein



der die Startzeile der Berechnungsgeschichte in der unteren Hälfte enthält.

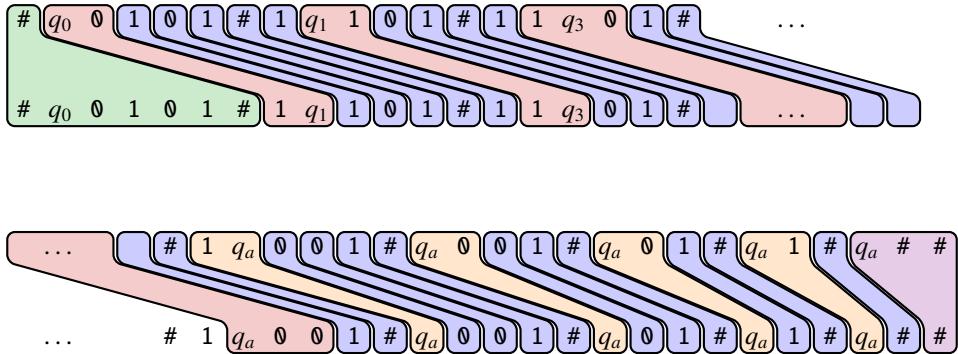
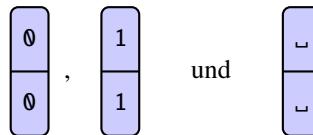


Abbildung 11.9: Zerlegung von zwei identischen, auf eine Zeile aufgereihten Berechnungsgeschichten einer Turing-Maschine  $M$  auf dem Inputwort  $w = 0101$  in Dominosteine für das Post-Korrespondenz-Problem. Die farbigen Dominosteine werden im Text erklärt und lassen sich allein aus der Spezifikation der Turing-Maschine konstruieren. Eine akzeptierende Berechnungsgeschichte gibt es genau dann, wenn das Post-Korrespondenz-Problem mit den farbigen Steinen eine Lösung hat.

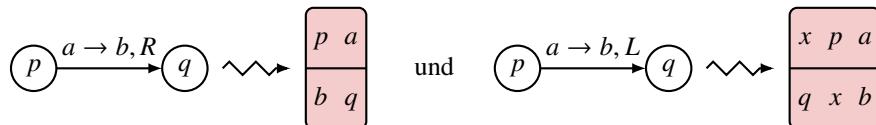
Die nächsten Steine, die rechts an den Startstein (11.6) angesetzt werden, müssen oben die Zeichen der ersten Zeile der Berechnungsgeschichte enthalten und unten die Zeichen der zweiten Zeile. Sie müssen also einen einzelnen Turing-Maschinenübergang beschreiben. Dazu sind nur ganz wenige verschiedene Arten von Steinen nötig. Die meisten Zeichen bleiben ja gleich, nur in unmittelbarer Nähe des Zustandssymbols sind etwas kompliziertere Muster nötig, die die Veränderung des Bandzeichens und die Kopfbewegung abbilden.

Die unveränderten Einzelzeichen können durch blaue Steine der Form



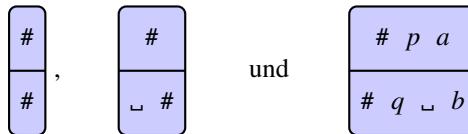
abgebildet werden. Allgemein braucht es für jedes Bandalphabetzeichen einen solchen Stein.

Die Turing-Maschinenübergänge lassen sich in einem schmalen Streifen beobachten, den die beiden roten Dominosteine



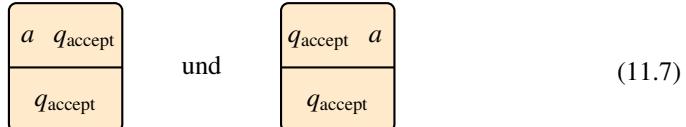
für die angegebenen Übergänge wiedergeben können. Man beachte, dass für einen Übergang mit Kopfbewegung nach links eine zusätzliche Spalte nötig ist, die ein beliebiges Bandalphabetzeichen  $x$  enthalten kann. Es ist ein Stein für jedes mögliche Bandalphabetzeichen  $x$  hinzuzufügen.

Ein weiterer Stein ist nötig, um die Trennzeichen abzubilden. Manchmal kommt es vor, dass am Rand des Wortes ein Leerzeichen gefunden wird. Die blauen Steine



ermöglichen dies.

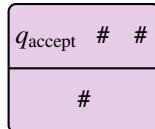
Wenn es eine akzeptierende Berechnungsgeschichte gibt, dann lässt sie sich mit den bisher konstruierten Bausteinen zusammensetzen. So entsteht die obere Hälfte in Abbildung 11.9. Die Steine sind so konstruiert, dass die zweite Zeile immer länger ist. Am Ende der zweiten Zeile steht der Bandinhalt nach Abschluss der Berechnung. Darin stehen außer dem Akzeptierzustand  $q_{\text{accept}}$  noch viele weitere Zeichen, die nicht relevant sind. Durch die Steine



(11.7)

können alle Zeichen, die neben  $q_{\text{accept}}$  stehen, zum Verschwinden gebracht werden. Dieser Prozess findet in der unteren Hälfte von Abbildung 11.9 statt.

Nachdem mit den Steinen (11.7) alle Zeichen außer  $q_{\text{accept}}$  entfernt worden sind, ist die zweite Zeile immer noch länger als die erste. Es ist aber bekannt, was im überstehenden Stück steht, nämlich  $q_{\text{accept}}\#$ . Mit dem violetten Abschlussstein



kann die erste Zeile jetzt mit dem gleichen Ende versehen werden.

Die konstruierten Dominosteine erlauben nur akzeptierende Berechnungsgeschichten herzustellen. Sie codieren den Berechnungsablauf genauso, wie eine Turing-Maschinenbeschreibung dies tut. Das Post-Korrespondenz-Problem lässt sich genau dann lösen, wenn es eine akzeptierende Berechnungsgeschichte gibt.

Die Konstruktion hat noch eine kleine Lücke: Wir sind davon ausgegangen, dass der Startstein (11.6) als erster Stein gewählt wird, dies wird aber durch die Steine allein noch nicht sichergestellt. Durch eine kleine Modifikation der Konstruktion kann dies jedoch auch erreicht werden, siehe zum Beispiel [56, section 5.2].

## 11.6.2 Das Gleichheitsproblem für kontextfreie Sprachen

Das Post-Korrespondenz-Problem illustriert auf anschauliche Art, wie man mithilfe der Berechnungsgeschichte die Komplexität einer Berechnung in eine Aufgabenstellung hineinschmuggeln kann. Einen unmittelbaren Anwendungsbezug hat es jedoch nicht. Die Idee lässt sich aber auf andere Situationen mit direktem Anwendungsbezug übertragen, wo die Codierung der Berechnungsgeschichte etwas komplizierter wird.

### **$ALL_{CFG}$ ist nicht entscheidbar**

Die Sprache

$$ALL_{CFG} = \{\langle G \rangle \mid G \text{ ist eine kontextfreie Grammatik, die } \Sigma^* \text{ produziert.}\}$$

umfasst die kontextfreien Grammatiken, die alle Wörter produzieren können. In diesem Abschnitt soll gezeigt werden, dass diese Sprache nicht entscheidbar ist.

**Satz 11.19.** *Die Sprache  $ALL_{CFG}$  ist nicht entscheidbar.*

Dies ist vielleicht etwas überraschend, denn das Akzeptanzproblem

$$A_{CFG} = \{\langle G, w \rangle \mid \text{die kontextfreie Grammatik } G \text{ produziert } w.\}$$

für kontextfreie Grammatiken ist mit dem Cocke-Younger-Kasami-Algorithmus entscheidbar. Während also für endliche Automaten fast jede Eigenschaft der akzeptierten Sprache entscheidbar ist und nach dem Satz von Rice für Turing-Maschinen keine nichttriviale Eigenschaft, nehmen die kontextfreien Grammatiken eine Mittelstellung ein.

In Kapitel 7 wurde gezeigt, dass es zu jeder kontextfreien Grammatik  $G$  einen Stackautomaten  $P$  mit  $L(G) = L(P)$  gibt und umgekehrt. Ein Entscheidungsproblem über die Eigenschaften der produzierten Sprache einer Grammatik ist also genau dann entscheidbar, wenn auch die Eigenschaft der akzeptierten Sprache eines Stackautomaten entscheidbar ist. Die Entscheidbarkeit von  $ALL_{CFG}$  ist daher mit dem folgenden Satz für Stackautomaten gleichbedeutend.

**Satz 11.20.** *Die Sprache*

$$ALL_{PDA} = \{\langle P \rangle \mid P \text{ ist ein Stackautomat, der } \Sigma^* \text{ akzeptiert.}\}$$

*ist nicht entscheidbar.*

*Beweis.* Um eine Reduktion  $A_{TM} \leq ALL_{PDA}$  zu konstruieren, soll aus  $\langle M, w \rangle$  ein Stackautomat konstruiert werden, der genau dann nicht alle Wörter akzeptiert, wenn  $M$  das Wort  $w$  akzeptiert. Die Idee des Post-Korrespondenz-Problems war, die Berechnungsgeschichte aus geeignet konstruierten Dominosteinen zusammenzusetzen. Es liegt daher nahe, den Stackautomaten so zu konstruieren, dass er die akzeptierenden Berechnungsgeschichten der Turing-Maschine nicht akzeptiert. Genau dann, wenn es keine akzeptierende Berechnungsgeschichte der Turing-Maschine gibt, akzeptiert der Stackautomat alle Wörter.

Der Stackautomat soll alle Wörter akzeptieren, die “fehlerhafte” Berechnungsgeschichten sind. Wir schreiben die Berechnungsgeschichte als Verkettung

$$\# C_1 \# C_2 \# C_3 \# \dots \# C_n \#\#,$$

die  $C_i$  sind die ursprünglichen Zeilen der Berechnungsgeschichte. Eine korrekte, akzeptierende Berechnungsgeschichte zeichnet sich durch folgende Eigenschaften aus:

1. Die Berechnungsgeschichte beginnt mit  $q_0w$ , d. h.  $C_1 = q_0w$ .

2. Die Berechnungsgeschichte endet mit einem Akzeptierzustand  $q_{\text{accept}}$ , d. h.  $C_n$  enthält  $q_{\text{accept}}$ .
3. Zwei aufeinanderfolgende Zeilen  $C_i$  und  $C_{i+1}$  der Berechnungsgeschichte unterscheiden sich nur um einen Turing-Maschinenübergang.

Wenn eine dieser Bedingungen nicht erfüllt ist, wird die Berechnungsgeschichte akzeptiert. Da ein Stackautomat nichtdeterministisch ist, kann er zunächst mit einer  $\epsilon$ -Weiche entscheiden, welche Bedingung geprüft werden soll. Für die einzelnen Bedingungen ist dann folgendes zu tun:

1. Die Berechnungsgeschichte beginnt mit  $q_0 w$ : Der Stackautomat überprüft, ob das erste Segment zwischen  $\#$ -Zeichen  $q_0 w$  ist. Dies ist mit einem regulären Ausdruck möglich, also erst recht mit einem Stackautomaten. Falls das erste Segment nicht  $q_0 w$  ist, wird akzeptiert.
2. Endet mit Akzeptierzustand: Die Sprache der Zeichenketten der Form (11.5), die im zweitletzten Block  $C_n$  zwischen  $\#$  kommt  $q_{\text{accept}}$  kann mit einem regulären Ausdruck beschrieben werden. Somit bilden auch die Wörter, die diese Bedingung nicht erfüllen, eine reguläre Sprache und können von einem endlichen Automaten akzeptiert werden.
3. Damit ein Turing-Maschinenübergang vorliegt, müssen zwei aufeinanderfolgende Zeilen der Berechnungsgeschichte verglichen werden. Zwei Zeilen  $C_i$  und  $C_{i+1}$  unterscheiden sich nur in unmittelbarer Nähe des Zustandssymbols, alle anderen Zeichen sind gleich. Der Stackautomat schreibt also zunächst die Zeile  $C_i$  auf den Stack und vergleicht dann die zweite mit dem, was er auf dem Stack gespeichert hat. Das funktioniert allerdings so nicht, denn zuerst auf dem Stack ist das letzte Zeichen von  $C_i$ , nicht das erste, das gebraucht würde, um den Vergleich mit  $C_{i+1}$  zu beginnen. Daraus schließen wir, dass wir die Berechnungsgeschichte mit alternierender Leserichtung der einzelnen Zeilen zur Verfügung haben müssen. Wir codieren sie daher als

$$\# C_1 \# C'_2 \# C_3 \# C'_4 \# \dots \# C_n \#\#, \quad (11.8)$$

wobei  $C'_i$  das gespiegelte Wort  $C_i$  ist. Jetzt findet der Stackautomat bei der Verarbeitung von  $C_{i+1}$  die Zeichen von  $C_i$  in umgekehrter Reihenfolge auf dem Stack. Sobald er eine Abweichung feststellt, die nicht durch einen Turing-Maschinenübergang gerechtfertigt ist, wird die Berechnungsgeschichte akzeptiert.

Damit ist ein Stackautomat konstruiert, der genau die in der Form (11.8) codierten, fehlerhaften Berechnungsgeschichten akzeptiert und die korrekten Berechnungsgeschichten nicht. Der Stackautomat liefert eine Reduktion  $A_{\text{TM}} \leq \text{ALL}_{\text{PDA}}$ .  $\square$

### Vergleich von Grammatiken: $EQ_{\text{CFG}}$

Mit einer Reduktion  $\text{ALL}_{\text{CFG}} \leq EQ_{\text{CFG}}$  kann man jetzt auch zeigen, dass  $EQ_{\text{CFG}}$  nicht entscheidbar ist. Da  $\Sigma^*$  regulär und damit auch kontextfrei ist, gibt es eine Grammatik  $G_{\Sigma^*}$ , welche alle Wörter zu erzeugen erlaubt (Siehe Verständniskontrolle 11.6). Die Abbildung

$$\langle G \rangle \mapsto \langle G, G_{\Sigma^*} \rangle$$

bildet jetzt eine Instanz von  $ALL_{CFG}$  in eine Instanz von  $EQ_{CFG}$  ab. Damit ist die Reduktion gefunden.

**Satz 11.21.** *Die Sprachen  $EQ_{CFG}$  und  $EQ_{PDA}$  sind nicht entscheidbar.*

**Verständniskontrolle 11.6:** Geben Sie eine Grammatik für die Sprache  $\Sigma^*$  an.



## Übungsaufgaben

**11.1.** Zeigen Sie, dass eine endliche Sprache  $L$  entscheidbar ist.

**11.2.** Seien  $L_1$  und  $L_2$  zwei Turing-entscheidbare Sprachen über dem Alphabet  $\Sigma$ . Zeigen Sie, dass die folgenden Sprachen Turing-entscheidbar sind:

- a)  $L_1 \cup L_2$
- b)  $L_1 L_2$
- c)  $L_1^*$
- d)  $\Sigma^* \setminus L_1$
- e)  $L_1 \cap L_2$

*Hinweis.* Turing-entscheidbar heißt, dass es zwei Funktionen

```
boolean l1(String w);
boolean l2(String w);
```

gibt, welche genau dann `true` zurückgeben, wenn  $w$  in  $L_1$  bzw.  $L_2$  ist. Die Aufgabe besteht dann darin, für jede Sprache  $L$  in den Teilaufgaben eine neue Funktion

```
boolean l(String w) {
    ....
}
```

zu programmieren, welche genau dann `true` zurückgibt, wenn  $w$  in  $L$  ist.

**11.3.** Sei

$$ALL_{DEA} = \{ \langle A \rangle \mid A \text{ ist ein DEA und } L(A) = \Sigma^* \}.$$

Zeigen sie:  $ALL_{DEA}$  ist entscheidbar.

**11.4.** Sei

$$A\varepsilon_{CFG} = \{ \langle G \rangle \mid G \text{ ist eine kontextfreie Grammatik und } \varepsilon \in L(G). \}.$$

Zeigen Sie:  $A\varepsilon_{CFG}$  ist entscheidbar.

**11.5.** Sei

$$INFINITE_{DEA} = \{ \langle A \rangle \mid A \text{ ist ein DEA und } L(A) \text{ ist unendlich.} \}.$$

Zeigen Sie, dass  $INFINITE_{DEA}$  entscheidbar ist.

**11.6.** Sei

$$MIRROR_{DEA} = \left\{ \langle A \rangle \mid \begin{array}{l} \text{Der DEA } A \text{ akzeptiert } w \text{ genau dann, wenn} \\ \text{er das gespiegelte Wort } w^t \text{ akzeptiert.} \end{array} \right\}.$$

Zeigen Sie:  $MIRROR_{DEA}$  ist entscheidbar.

**11.7.** Ein E-Learning-System soll Schülern arithmetische Ausdrücke zur Auswertung geben und die Antworten der Schüler überprüfen. Die Qualitätssicherung verlangt vom Programmierer dieses Moduls, dass er einen unabhängigen Test schreibt, welcher aus dem Source-Code des Moduls ableiten kann, ob je ein inkorrekt arithmetischer Ausdruck als Aufgabe gestellt werden könnte. Kann der Programmierer dieses Problem lösen?

**11.8.** Der chinesische Präsident Xi Jinping hat sich nicht darüber gefreut, dass er auf dem Internet mit Winnie the Pooh verglichen wird. Er hat seine Zensoren angewiesen, Bilder von Winnie the Pooh zu blockieren, sogar in der in China verbreiteten Chat-Anwendung WeChat werden Meldungen, die die Zeichenkette Winnie the Pooh enthalten, mit einer Fehlermeldung quittiert. Trotzdem gibt es natürlich immer noch zahllose Apps, die Winnie the Pooh nicht blockieren und damit die chinesischen Zensoren brüskieren. Gibt es eine Möglichkeit, Apps zu blockieren, die zur Laufzeit die Zeichenkette Winnie the Pooh erzeugen können?

**11.9.** Zeigen Sie, dass die Sprache

$$B_{TM} = \left\{ \langle M \rangle \mid \begin{array}{l} \text{Die Turing-Maschine } M \text{ beschreibt} \\ \text{einen beschränkten Teil des Bandes.} \end{array} \right\}$$

nicht entscheidbar ist.

Lösungen: <https://autospr.ch/uebungen/AutoSpr-111.pdf>



# Kapitel 12

## Komplexität

In Kapitel 11 wurde untersucht, ob ein Problem überhaupt mit einem Computer lösbar ist. Dabei wurde eine Vielfalt von relevanten Fragestellungen gefunden, welche nicht durch ein Computerprogramm beantwortbar sind. Man kann aber auch sagen, dass diese Probleme in der täglichen Programmierpraxis selten auftreten. Viel öfter passiert es, dass ein Problem zwar im Prinzip lösbar ist, aber ein Programm für die Lösung viel zu lange braucht. Dies ist vor allem ein Problem der Skalierung. Kleine Probleme, also solche mit einer kleinen Datenmenge, können fast immer schnell gelöst werden. Schwierigkeiten treten erst auf, wenn die Datenmenge anwächst und die Laufzeit länger wird. Wächst die Laufzeit z. B. exponentiell schnell mit der Problemgröße an, dann erreicht sie schnell Längen, die die menschliche Lebensdauer übersteigt. Solche Probleme sind für alle praktischen Zwecke als unlösbar anzusehen.

Die Komplexitätstheorie, genauer die Theorie der Zeitkomplexität, befasst sich mit der Fragestellung, wie der Zeitaufwand für die Lösung eines Problems von der Inputgröße abhängt. Es stellt sich heraus, dass man Problemstellungen grob in zwei Klassen aufteilen kann, die Klassen P und NP. In diesem Kapitel werden Methoden zur Klassifikation von Aufgabenstellungen entwickelt.

Die Verständniskontrollen in diesem Kapitel werden sich wiederholt auf das folgende Problem beziehen.

---

**Verständniskontrolle 12.1:** Das 8-Damen-Problem verlangt vom Spieler, 8 Damen so auf einem Schachfeld zu platzieren, dass sie sich gegenseitig nicht schlagen können. Eine Dame kann eine andere Figur in der gleichen Zeile, Spalte oder in diagonaler Richtung schlagen.



autospr.ch/v/12.1.pdf

- a) Formulieren Sie das Problem als Sprachproblem.
  - b) Ist es entscheidbar?
-

## 12.1 Laufzeit von Turing-Maschinen

Die Laufzeit eines Programms ist natürlich eine Messgröße, die sehr stark von der verwendeten Hardware abhängt. Der erste zur Gleitkommaarithmetik befähigte Computer war der auf Elektronenröhrentechnologie basierende IBM 704, den IBM 1954 der Öffentlichkeit vorstellt. Er konnte laut Herstellerangaben etwa 4000 Gleitkommaoperationen pro Sekunde durchführen. Diese einmalige Rechenleistung hatte ihren Preis: 2 Mio. USD, oder auf 2024 umgerechnet etwa 22 Mio. USD. Im Jahr 2024 kostet ein Raspberry Pi 5 weniger als 100 USD und kann pro Sekunde über 10 Mia Gleitkommaoperationen durchführen. Der Raspberry Pi ist also etwa 2.5 Mio. mal schneller und über 200000 mal billiger. Doch auch der Speicher ist ein wichtiger Faktor: Der IBM 704 konnte nicht mehr als 144 KiB Hauptspeicher haben, der Raspberry Pi über 5 Mio. mal mehr. Der große Speicher befähigt der Raspberry Pi so, viel größere Probleme in Angriff zu nehmen, die auf einem IBM 704 niemals gelöst werden können. Offensichtlich sind Vergleiche der Laufzeit eines Programms für zwei derart unterschiedliche Maschinen wenig aussagekräftig. Außerdem sagen sie vor allem etwas aus über die Maschine, nicht über die Aufgabenstellung.

In diesem Abschnitt suchen wir daher nach einem hardwareunabhängigen Kriterium, mit dem wir Aufgabenstellungen bezüglich der Skalierungseigenschaften der Laufzeit zu ihrer Lösung klassifizieren können. Wir stellen damit sicher, dass die gefundenen Aussagen unabhängig vom sich schnell wandelnden aktuellen Stand der Technik nützlich sind. Weiter entwickeln wir mit der polynomiellen Reduktion eine Erweiterung der in Abschnitt 11.4 entwickelten Reduktionstechnik, mit der sich die Skalierungseigenschaften verschiedener Problemstellungen miteinander vergleichen lassen.

### 12.1.1 Maßzahl für die Laufzeit

Moderne Maschinen werden immer schneller, sie können in der gleichen Zeit eine viel größere Zahl von Schritten ausführen. Für unsere Zwecke ist daher nicht die tatsächliche Laufzeit von Interesse, sondern nur die Anzahl der Schritte, die die Maschine ausführen muss.

**Definition 12.1** (Laufzeit). *Die Laufzeit einer Turing-Maschine  $M$  bei der Verarbeitung eines Wortes  $w \in \Sigma^*$  ist die Anzahl  $t(w)$  der Schritte, die die Turing-Maschine ausführt, bis sie anhält.*

In vielen Aufgabenstellungen gibt es Fälle, die besonders leicht zu behandeln sind. Um diese Fälle zu eliminieren, interessiert uns nur der schlimmste Fall.

**Definition 12.2** (Worst case Laufzeit). *Die worst case Laufzeit  $t(n)$ , die die Maschine zur Verarbeitung von Inputs der Länge  $n$  braucht, ist*

$$t(n) = \max\{t(w) \mid |w| \leq n\}.$$

Die Laufzeit  $t(w)$  wird unendlich, wenn die Maschine auf dem Input  $w$  nicht anhält. Damit wird auch  $t(n)$  unendlich, wenn es Inputwörter  $w$  mit Länge  $|w| \leq n$  gibt, auf denen die Maschine nicht anhält. Wir fordern daher im Folgenden immer, dass die betrachteten Maschinen Entscheider sind. Die entwickelte Theorie ist daher nur auf entscheidbare Probleme anwendbar.

### 12.1.2 Polynomielle Laufzeit klassischer Maschinen

In Abschnitt 10.2 sind verschiedene Varianten von Turing-Maschinen vorgestellt worden. Die verschiedenen Spielarten wurden dort auch mit technologischen Fortschritten und Architekturänderungen motiviert. Wir erwarten unterschiedliche, aber ineinander umrechenbare Laufzeiten.

**Bandalphabet:** Eine Turing-Maschine  $M_1$  mit einem größeren Bandalphabet kann auf einer Turing-Maschine  $M_2$  mit Bandalphabet  $\{\emptyset, 1, \sqcup\}$  simuliert werden. Um einen Schritt der größeren Turing-Maschine mit der einfacheren zu simulieren, muss das binär codierte Bandzeichen in mehreren Schritten vom Band gelesen werden und der neue Wert in gleich vielen Schritten wieder auf das Band geschrieben werden, so wie das in Abbildung 10.11 dargestellt wurde. Für jeden Schritt der größeren Maschine ist also die gleiche Anzahl  $N$  Schritte der einfacheren Maschine nötig. Die Laufzeit  $t_2$  auf der einfachen Maschine ist daher  $N$  mal größer als die Laufzeit  $t_1$  der größeren Maschine:  $t_2(w) = N \cdot t_1(w)$ . Da uns nur die Größenordnung interessiert, können wir  $t_2(n) = O(t_1(n))$  schreiben<sup>1</sup>.

**Mehrspurmaschine:** Eine Mehrspurmaschine  $M_1$  kann als eine Maschine mit größerem Bandalphabet als die Vergleichsmaschine  $M_2$  betrachtet werden. Es gilt also auch hier, dass sich die Abhängigkeit der Laufzeit  $t_1(n) = O(t_2(n))$  von  $n$  nicht ändert.

**Mehrbandmaschine:** Um eine Mehrbandmaschine  $M_1$  auf einer Maschine  $M_2$  mit nur einem mehrspurigen Band zu simulieren, muss für jeden Schritt der Mehrbandmaschine das Band der Mehrspurmaschine nach den Marken für die Kopfposition durchsucht werden (Abbildung 10.17). Dazu sind so viele Schritte nötig, wie der beschriebene Teil des Bandes lang ist. Im schlimmsten Fall ist dies die bisherige Laufzeit des Programms. Für jeden Schritt der Mehrbandmaschine sind also  $O(t_1(n))$  Schritte der Mehrspurmaschine nötig. Die Laufzeit der Simulation ist somit  $t_2(n) = t_1(n) \cdot O(t_1(n)) = O(t_1(n)^2)$ .

Die Konstruktionen können geschachtelt werden, wenn mit mehreren Speichern arbeitende Programme andere solche Programme aufrufen. Dann wird die Laufzeit bei der Simulation mehrmals quadriert, es gilt dann  $t_2(n) = O(t_1(n)^{2^k})$ . Da  $k$  endlich ist, kann sich die Laufzeit in allen Fällen höchstens um eine Potenz ändern.

Da die verschiedenen Maschinenvarianten Hardwareeigenschaften reflektieren und nichts mit dem Schwierigkeitsgrad des Problems zu tun haben, müssen wir Laufzeiten als gleich behandeln, wenn Sie sich nur um eine Potenz unterscheiden. Als hardwareunabhängiges Kriterium bietet sich daher die Eigenschaft eines Algorithmus, polynomielle Laufzeit gemäß der folgenden Definition zu haben, an.

**Definition 12.3** (polynomielle Laufzeit). *Ein Algorithmus hat polynomielle Laufzeit, wenn seine worst case Laufzeit  $O(n^k)$  ist mit  $k \in \mathbb{N}$ .*

Umgekehrt suggerieren diese Überlegungen auch, dass man durch Hardwaremaßnahmen die Rechenzeit bestenfalls um Potenzen verbessern kann, zum Beispiel von  $t(n)$  auf

<sup>1</sup>Die  $O$ -Notation wird in Anhang A.5 aufgefrischt.

$O(t(n)^{1/l})$ , wobei  $l > 1$  sein kann. In Kursen über Algorithmen lernt man, dass Laufzeiten von der Größenordnung von  $O(n)$  oder  $O(n^r)$  für kleine  $r$  akzeptabel sind. Für eine beliebige polynomielle Laufzeit  $O(n^k)$  kann man also im Idealfall mit Hardwaremaßnahmen die Laufzeit  $O(n^{k/l})$  erreichen, also “akzeptable” Laufzeiten. Wächst die Laufzeit aber schneller als ein Potenz von  $n$ , sind alle Hardwaremaßnahmen vergebens.

**Verständniskontrolle 12.2:** Überlegen Sie sich einen einfachen Algorithmus, mit dem Sie Lösungen für das  $n$ -Damen-Problem finden können, und bestimmen sie die Laufzeit in Abhängigkeit von  $n$ .



autospr.ch/v/12.2.pdf

### 12.1.3 Polynomielle Reduktion

Nur selten muss ein Programmierer eine Problemlösung von Grund auf neu programmieren. Meistens verwendet er bereits existierende Bibliotheken und APIs, die er neu kombiniert. Er verlässt sich dabei darauf, dass eine breit eingesetzte Bibliothek zuverlässiger und schneller ist als ein *quick hack*. In einigen Fällen geben die Bibliotheksautoren sogar Garantien über die Laufzeit ab, zum Beispiel verspricht die Funktion `std::sort` der C++-Standardlibrary Laufzeiten zwischen  $O(n \log n)$  und  $O(n^2)$  für das Sortieren. Die Laufzeit und die Skalierung hängt dann sowohl von der Rechenzeit ab, die zur Vorbereitung der Bibliotheksaufrufe notwendig sind, vor allem aber auch von der Qualität der in der Bibliothek implementierten Algorithmen.

Etwas formaler geht es um das folgende Problem. Der Programmierer muss ein Programm schreiben, welches die Sprache  $A$  entscheidet. Er erkennt eine Ähnlichkeit zur Sprache  $B$ , für die es bereits einen Entscheider gibt. Er konstruiert daher nur eine Funktion  $f: \Sigma^* \rightarrow \Sigma^*$ , die  $A$  auf  $B$  reduziert, also  $f: A \leq B$ , wie in Definition 11.11. Die Problemlösung erfolgt jetzt dadurch, dass zunächst die Abbildung  $f$  auf dem Input  $w$  aufgerufen wird, anschließend wird der Entscheider für  $B$  auf dem Input  $f(w)$  aufgerufen.

Die Laufzeit des Verfahrens setzt sich jetzt zusammen aus der Laufzeit für die Funktion  $f$  auf dem Input  $w$  und der Laufzeit für den Entscheider für  $B$  auf dem Input  $f(w)$ . Ein schneller Entscheider für  $B$  bringt nichts, wenn die Berechnung der Funktion  $f$  deutlich schlechter skaliert. Da das Ziel immer ist, polynomielle Algorithmen zu erhalten, muss man fordern, dass die Laufzeit von  $f$  polynomiell von der Länge des Inputwortes  $w$  abhängt.

Die Laufzeit des Entscheiders für  $B$  wird durch  $f$  ebenfalls beeinflusst. Das Wort  $f(w)$ , welches der Entscheider von  $B$  verarbeiten muss, könnte deutlich länger sein als der ursprüngliche Input  $w$ . Die Länge von  $f(w)$  kann aber nicht größer sein als die Laufzeit, denn in jedem Schritt kann ja höchstens ein Zeichen von  $f(w)$  geschrieben werden. Da die Laufzeit für  $f$  polynomiell ist, ist auch die Länge von  $f(w)$  polynomiell in  $|w|$ .

Ist die Laufzeit des Entscheiders für  $B$  von der Ordnung  $O(n^k)$  und die Laufzeit von  $f$  von der Ordnung  $O(n^l)$ , dann wird die Laufzeit der Zusammensetzung die Ordnung  $O(|f(w)|^k) \leq O((n^l)^k) = O(n^{lk})$  haben. Somit ist die Laufzeit des Entscheiders ebenfalls polynomiell.

**Definition 12.4** (Polynomielle Reduktion). Eine Reduktionsabbildung  $f: A \leq B$  heißt polynomielle Reduktion, wenn sie in polynomieller Zeit berechenbar ist. Die polynomielle Reduktion wird auch  $A \leq_P B$  geschrieben und kann als  $A$  ist polynomiell leichter zu entscheiden als  $B$  gelesen werden.

Die Argumentation, die die Definition der polynomiellen Reduktion motiviert hat, zeigt auch, dass beliebige endliche Kompositionen von berechenbaren Funktionen mit polynomieller Laufzeit wieder polynomielle Laufzeit haben.

### 12.1.4 Vertex-Coloring und Stundenplan

In diesem Abschnitt werden zwei interessante Probleme aus der Praxis vorgestellt und es wird gezeigt, wie sich das eine auf das andere reduzieren lässt.

#### Stundenplan

Ein Stundenplan legt fest, zu welchen Zeiten Lehrveranstaltungen durchgeführt werden. Er berücksichtigt dabei vor allem, dass zwei Lehrveranstaltungen nicht gleichzeitig stattfinden können, wenn sich Studierende für beide angemeldet haben. Die Lektionen müssen in ein vorgegebenes Zeitraster, welches in vielen Schulen durch eine laute Glocke erzwungen wird, geplant werden. Es steht also eine feste Anzahl  $k$  von Zeitfenstern zur Verfügung, die nicht verändert werden kann. Natürlich werden in der Praxis noch viele weitere Parameter einbezogen, auf die für die vorliegende Diskussion nicht eingegangen werden soll.

Der Stundenplan in dieser Form ist kein Entscheidungsproblem. Wenn wir aber danach fragen, ob überhaupt ein Stundenplan mit den  $k$  offenen Zeitfenstern möglich ist, dann wird die Aufgabenstellung zu einem Entscheidungsproblem, welches wir *STUNDENPLAN* nennen.

#### Färbeproblem

Gegeben ist ein Graph<sup>2</sup> mit Knotenmenge  $V$  und Kantenmenge  $E$ . Ist es möglich, die Knoten des Graphen mit  $k$  verschiedenen Farben so einzufärben, dass benachbarte Knoten immer verschiedene Farben haben? Dies ist das  $k$ -Färbeproblem oder  $k$ -VERTEX-COLORING. Es ist ein Entscheidungsproblem, welches wir auch als das Sprachproblem

$$k\text{-VERTEX-COLORING} = \left\{ \langle G \rangle \mid \begin{array}{l} G \text{ ist ein Graph, dessen Knoten mit } k \text{ Farben so} \\ \text{eingefärbt werden können, dass benachbarte} \\ \text{Knoten verschiedene Farben haben.} \end{array} \right\}$$

schreiben können.

Abbildung 12.1 zeigt den sogenannten Petersen-Graphen  $K_{10}$ , der sich mit drei Farben einfärben lässt. Die Einfärbung mit nur zwei Farben ist nicht möglich, da sich schon das äußere Fünfeck nicht mit zwei Farben einfärben lässt.

Ein vollständig verbundener Graph, bei dem jeder Knoten mit jedem anderen Knoten verbunden ist, benötigt immer soviele verschiedene Farben, wie der Graph Knoten hat. Ein Graph ohne Kanten kann mit einer einzigen Farbe eingefärbt werden. Die minimal nötige

<sup>2</sup>In Anhang A.3 werden ein paar Grundbegriffe zur Graphentheorie zusammengestellt.

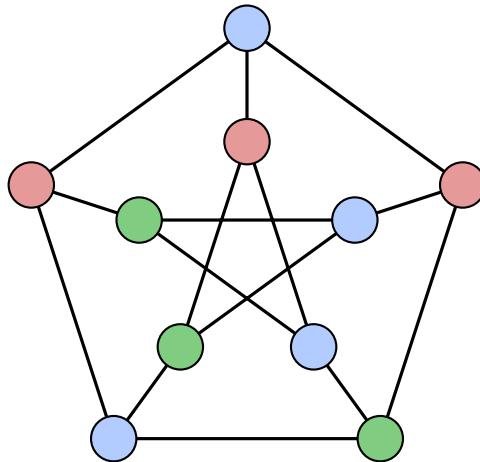


Abbildung 12.1: Der Petersen-Graph kann mit drei Farben eingefärbt werden, mit zwei Farben ist dies allerdings nicht möglich.

Anzahl Farben, die sogenannte *chromatische Zahl* des Graphen, sagt also etwas über die Anzahl der vorhandenen Verbindungen aus.

### Politische Karten

Eine politische Karte zeigt keine Topographie, sondern nur die politische Gliederung. In Abbildung 12.2 sind die Kantone der Schweiz mit vier Farben so eingefärbt, dass benachbarte Kantone immer verschiedene Farben haben. Auf den beiden Seiten einer Grenze findet man also immer verschiedene Farben. Dies ist das  $k$ -Vertex-Coloring-Problem für einen Graphen, der einen Knoten für jeden Kanton hat und eine Kante für jedes Paar von Kantonen, die eine gemeinsame Grenze haben.

Die Graphen, die politischen Karten zugrunde liegen, haben die zusätzliche Eigenschaft, dass sie sich in der Ebene realisieren lassen. Solche Graphen heißen *planar*. Es war lange Zeit ein offenes Problem, wie viele Farben für die Einfärbung einer Karte benötigt werden, bis Kenneth Appel und Wolfgang Haken 1976 unter massivem Computereinsatz den Vier-Farben-Satz beweisen konnten, der garantiert, dass die Färbung planarer Graphen immer mit vier Farben möglich ist.

### Die Reduktion $STUNDENPLAN \leq_P k\text{-VERTEX-COLORING}$

Das Stundenplanproblem lässt sich polynomiell auf das Färbeproblem reduzieren. Im Stundenplan muss man jeder Lehrveranstaltung ein Zeitfenster zuordnen, im Färbeproblem jedem Knoten eine Farbe. Damit ist klar, dass die Reduktionsabbildung  $f$  aus jeder Lehrveranstaltung einen Knoten des Graphen machen muss und aus jedem Zeitfenster eine Farbe. Zwei Lehrveranstaltung dürfen nicht ins gleiche Zeitfenster geplant werden, wenn sich Studierende für beide angemeldet haben. Dies entspricht der Bedingung, dass benachbarte Knoten des Graphen nicht die gleiche Farbe haben dürfen.  $f$  bildet daher Paare von

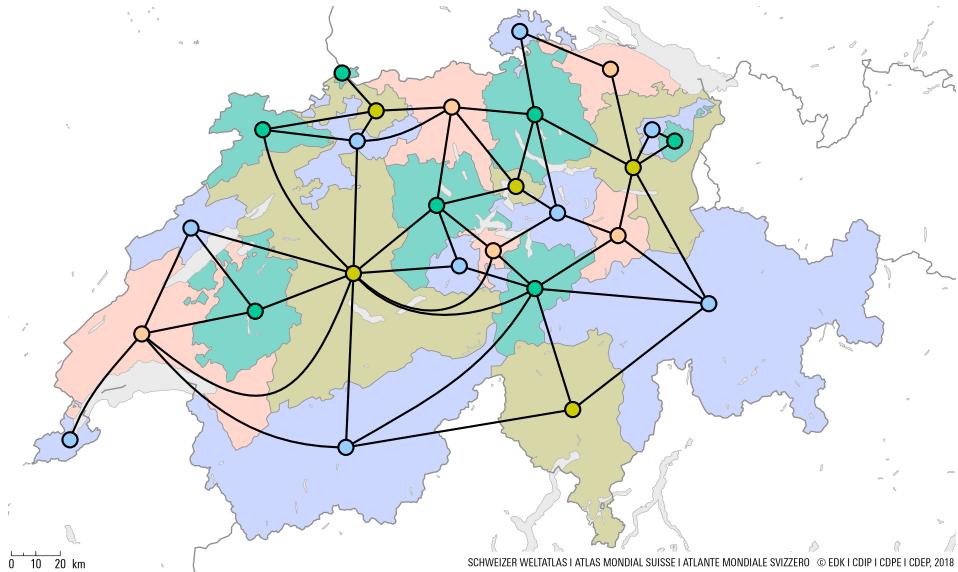


Abbildung 12.2: Politische Karte der Schweiz. Die Kantone sind mit vier Farben so eingefärbt, dass benachbarte Kantone verschiedene Farben haben. Dies ist das  $k$ -Vertex-Coloring-Problem für  $k = 4$  und den Nachbarschaftsgraphen der Kantone. Einige Kantone sind nicht zusammenhängend, nämlich Genf, Waadt, Freiburg, Solothurn, Obwalden, Schaffhausen und Appenzell Innerrhoden. Besonders die komplizierten Verhältnisse der Kantone Obwalden (blau, Mitte) und Nidwalden (rot, teilt Obwalden in zwei Teile) führen dazu, dass der Graph nicht planar ist, trotzdem ist er mit vier Farben einfärbbar. (Kartengrundlage: [52], © EDK, 2024.)

Lehrveranstaltungen, auf die sich Studierende angemeldet haben, auf Kanten des Graphen ab. Das Stundenplanproblem hat genau dann eine Lösung, wenn das Färbeproblem eine Lösung hat.

Die Reduktion bildet die Objekte des Stundenplanproblems eins zu eins auf die Objekte des Färbeproblems ab. Die Reduktion ist daher umkehrbar: Jedes Färbeproblem kann auch als Stundenplanproblem formuliert werden.

**Definition 12.5** (polynomiell äquivalent). *Zwei Sprachen  $A$  und  $B$  heißen polynomiell äquivalent, geschrieben  $A \equiv_P B$ , wenn  $A \leq_P B$  und  $B \leq_P A$ .*

**Satz 12.6.** *Das Stundenplanproblem und das Färbeproblem sind polynomiell äquivalent.*

*Beweis.* Die oben beschriebene Reduktion ist eine Eins-zu-eins-Reduktion, sie ist in beide Richtungen möglich. Somit gilt

$$\begin{aligned} STUNDENPLAN &\leq_P k\text{-VERTEX-COLORING} \\ \text{und} \quad k\text{-VERTEX-COLORING} &\leq_P STUNDENPLAN. \end{aligned}$$

Es ist noch zu überprüfen, ob die Reduktionsabbildung polynomielle Laufzeit hat. Bei der Reduktion wird jedem Objekt im Stundenplan eineindeutig ein Objekt im Färbe-

blem zugeordnet. Die Laufzeit ist damit direkt proportional zur Größe des Stundenplanproblems.  $\square$

Leider vereinfacht die Reduktion das Finden eines Stundenplans nicht wesentlich. Wir werden später in Abschnitt 13.3.2 sehen, dass beide Probleme NP-vollständig sind, was nach aktuellem Wissen bedeutet, dass kein polynomieller Algorithmus zu ihrer Lösung bekannt ist.

### ***k-CLIQUE***

Die Theorie der Graphen ist ein sehr ergiebiges Reservoir für schwierige, abstrakte Fragestellungen mit oft vielfältigen Anwendungen. Neben dem bereits vorgestellten Färbeproblem ist das  $k$ -Cliquenproblem ein weiteres nützliches Beispiel.

**Definition 12.7 ( $k$ -Clique).** Eine  $k$ -Clique in einem Graphen  $G$  ist eine Menge  $C$  von  $k$  Knoten des Graphen,  $C \subset V$ , die alle miteinander verbunden sind, also  $\forall a, e \in C (a \neq e \Rightarrow \{a, e\} \in E)$ .

*Beispiel 12.8.* Die Mitglieder des sozialen Netzwerks Facebook können Bekanntschaftsbeziehungen eingehen. Diese bilden einen ungerichteten Graphen mit den Mitgliedern als Knoten und einer Kante zwischen zwei Mitgliedern, die befreundet sind. Eine  $k$ -Clique auf Facebook sind  $k$  Mitglieder, die alle miteinander befreundet sind.  $\circ$

Das  $k$ -Cliquenproblem ist die Sprache

$$k\text{-CLIQUE} = \{\langle G \rangle \mid \text{Der Graph } G \text{ hat eine } k\text{-Clique.}\}. \quad (12.1)$$

Wie das Färbeproblem ist auch das  $k$ -Cliquenproblem in einem großen Graphen nicht einfach zu lösen. Es benötigt typischerweise exponentielle Zeit.

---

*Verständniskontrolle 12.3:* Formulieren Sie das  $n$ -Damen-Problem als  $k$ -CLIQUE-Problem.




---

## 12.2 Nichtdeterminismus

In Kapitel 3 wurde Nichtdeterminismus im Zusammenhang mit endlichen Automaten eingeführt und die Stackautomaten von Kapitel 7 waren grundsätzlich immer nichtdeterministisch. In diesem Abschnitt wird untersucht, wie sich Nichtdeterminismus in einer Turing-Maschine manifestieren kann und wie er die Laufzeit beeinflusst.

### 12.2.1 Nichtdeterministische Turing-Maschinen

Nichtdeterministische endliche Automaten und Stackautomaten haben eine Übergangsfunktion  $\delta$ , nicht eindeutig bestimmte Werte sondern Mengen von Werten annimmt. Mindestens einer der Werte führt zum Erfolg. Für eine Implementation ist die Schwierigkeit, einen solchen Wert zu finden. Die Implementation kann alle Fälle durchprobieren, sie kann probabilistische Argumente verwenden oder sie kann ein Orakel konsultieren, welches die richtige Wahl des Wertes der Übergangsfunktion verraten kann. Eine nichtdeterministische Turing-Maschine kann auf die gleiche Art definiert werden.

**Definition 12.9** (Nichtdeterministische Turing-Maschine). *Eine nichtdeterministische Turing-Maschine ist ein 7-Tupel*

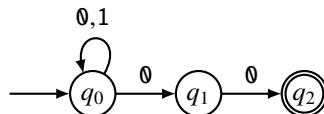
$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

in dem alle Komponenten die gleiche Bedeutung haben wie bei einer Turing-Maschine (Definition 10.1) außer die Übergangsfunktion, die eine Funktion

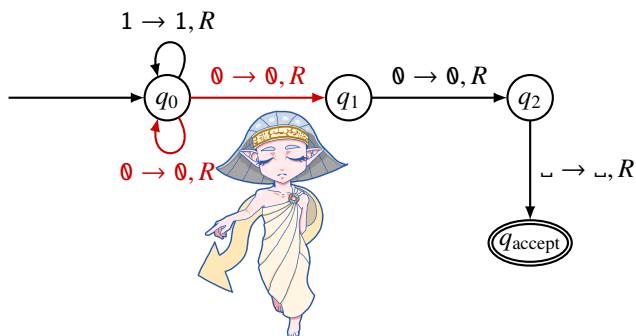
$$\delta: Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$$

mit Werten in der Potenzmenge  $P(Q \times \Gamma \times \{L, R\})$  ist.

*Beispiel 12.10.* Der nichtdeterministische endliche Automat



akzeptiert die Sprache  $L = \{w \in \Sigma^* \mid w \text{ endet mit } 00\}$ . Daraus lässt sich jetzt eine nichtdeterministische Turing-Maschine mit dem Zustandsdiagramm



machen. Die beiden rot eingezeichneten Übergänge beim Zustand  $q_0$  haben die gleichen Voraussetzungen, sie sind also nicht deterministisch. An dieser Stelle muss das Orakel konsultiert werden.  $\circ$

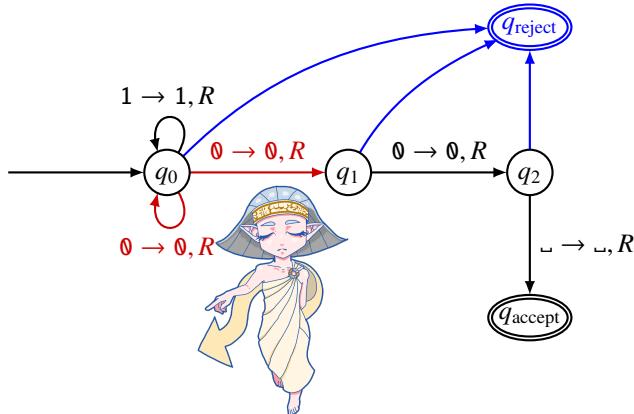


Abbildung 12.3: Vervollständigung einer nichtdeterministischen Turing-Maschine. Die blauen Übergänge gelten für Bandzeichen, für die die Übergangsfunktion beim entsprechenden Zustand keinen Übergang vorsieht. Sie garantieren, dass für jeden Zustand und jedes Bandzeichen der Übergang nach  $q_{\text{reject}}$  zur Verfügung steht, wenn die Übergangsfunktion keinen anderen Übergang erlaubt.

### Berechnungsgeschichte

Eine Berechnungsgeschichte einer nichtdeterministischen Turing-Maschine sieht genau gleich aus wie die Berechnungsgeschichte einer gewöhnlichen Turing-Maschine. Da es an einigen Stellen verschiedene Möglichkeiten für einen Übergang gibt, gibt es auch verschiedene Berechnungsgeschichten.

*Beispiel 12.11.* Die nichtdeterministische Turing-Maschine von Beispiel 12.10 hat gleich zu Beginn einen nichtdeterministischen Übergang. Daraus ergeben sich die folgenden Berechnungsgeschichten für das Wort 1000:

$\vdash q_0 \ 1 \ 0 \ 0 \ \vdash$	$\vdash q_0 \ 1 \ 0 \ 0 \ \vdash$	$\vdash q_0 \ 1 \ 0 \ 0 \ \vdash$
$\vdash 1 \ q_0 \ 0 \ 0 \ \vdash$	$\vdash 1 \ q_0 \ 0 \ 0 \ \vdash$	$\vdash 1 \ q_0 \ 0 \ 0 \ \vdash$
$\vdash 1 \ 0 \ q_1 \ 0 \ \vdash$	$\vdash 1 \ 0 \ q_0 \ 0 \ \vdash$	$\vdash 1 \ 0 \ q_0 \ 0 \ \vdash$
$\vdash 1 \ 0 \ 0 \ q_2 \ \vdash$	$\vdash 1 \ 0 \ 0 \ q_1 \ \vdash$	$\vdash 1 \ 0 \ 0 \ q_0 \ \vdash$
$\vdash 1 \ 0 \ 0 \ \vdash q_a$	Abbruch	Abbruch

Die nicht deterministischen Übergänge sind rot hervorgehoben. Es gibt eine Berechnungsgeschichte, die im Akzeptierzustand  $q_{\text{accept}} = q_a$  terminiert. In den anderen beiden Berechnungsgeschichten entsteht eine Situation, in der die Übergangsfunktion keinen weiteren Übergang ermöglicht, die Berechnung kann nicht weitergeführt werden, weil sich die Turing-Maschine sozusagen “verklemmt” hat.  $\circ$

Die unglückliche Situation des Beispiels kann vermieden werden, indem die Übergangsfunktion  $\delta$  so modifiziert wird, dass  $\delta(q, a)$  nie leer ist:

$$\delta': Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\}) : (q, a) \mapsto \begin{cases} \delta(q, a) & \text{falls } \delta(q, a) \neq \emptyset \\ \{(q_{\text{reject}}, a, R)\} & \text{falls } \delta(q, a) = \emptyset. \end{cases}$$

Abbildung 12.3 zeigt die auf diese Weise vervollständigte nichtdeterministische Turing-Maschine. Statt zu “verklemmen”, hält die Maschine im Zustand  $q_{\text{reject}}$ .

### Akzeptieren mit einer nichtdeterministischen Turing-Maschine

Bei einem nichtdeterministischen endlichen Automaten kann es für ein Wort  $w$  viele mögliche Wege durch das Zustandsdiagramm geben, die nicht alle in einem Akzeptierzustand enden müssen. Es genügt, wenn es einen Weg gibt, der in einem Akzeptierzustand ankommt. Daher definieren wir analog eine nichtdeterministische Turing-Maschine folgendermaßen.

**Definition 12.12** (Akzeptieren). Eine nichtdeterministische Turing-Maschine  $M$  akzeptiert das Wort  $w \in \Sigma^*$ , wenn es eine Berechnungsgeschichte gibt, die im Zustand  $q_{\text{accept}}$  endet.

Die Turing-Maschine von Beispiel 12.10 akzeptiert das Wort 100. Für das Wort 010 ergeben sich mit der vervollständigten Turing-Maschine von Abbildung 12.3 die Berechnungsgeschichten

$$\begin{array}{lll} \sqsubset q_0 \ 0 \ 1 \ 0 \ \sqsubset & \sqsubset q_0 \ 0 \ 1 \ 0 \ \sqsubset & \sqsubset q_0 \ 0 \ 1 \ 0 \ \sqsubset \\ \sqsubset 0 \ q_1 \ 1 \ 0 \ \sqsubset & \sqsubset 0 \ q_0 \ 1 \ 0 \ \sqsubset & \sqsubset 0 \ q_0 \ 1 \ 0 \ \sqsubset \\ \sqsubset 0 \ 1 \ q_r \ 0 \ \sqsubset & \sqsubset 0 \ 1 \ q_0 \ 0 \ \sqsubset & \sqsubset 0 \ 1 \ q_0 \ 0 \ \sqsubset \\ & \sqsubset 0 \ 1 \ 0 \ q_0 \ \sqsubset & \sqsubset 0 \ 1 \ 0 \ q_1 \ \sqsubset \\ & \sqsubset 0 \ 1 \ 0 \ q_r \ \sqsubset & \sqsubset 0 \ 1 \ 0 \ q_r \ \sqsubset \end{array}$$

Die nichtdeterministischen Übergänge sind rot, die vervollständigten Übergänge blau gekennzeichnet. Keine dieser Berechnungsgeschichten endet in  $q_{\text{accept}}$ , also wird das Wort 010 nicht akzeptiert.

### Laufzeit einer nichtdeterministischen Turing-Maschine

Die Verarbeitung eines Wortes mit einer nichtdeterministischen Turing-Maschine kann mit mehreren verschiedenen Berechnungsgeschichten möglich sein. Die in Abbildung 12.3 dargestellte Vervollständigung der Übergangsfunktion kann auch sicherstellen, dass sich die Turing-Maschine nicht “verklemmt”. Damit ist aber noch nicht garantiert, dass es Berechnungsgeschichten gibt, die nicht terminieren. Bei den deterministischen Turing-Maschinen wurde diese für die Definition der Laufzeit ungeeignete Situation dadurch ausgeschlossen, dass nur Entscheider zugelassen wurden.

**Definition 12.13** (Nichtdeterministischer Entscheider). Eine nichtdeterministische Turing-Maschine heißt (nichtdeterministischer) Entscheider, wenn jede Berechnungsgeschichte terminiert.

Da für einen Entscheider jede Berechnungsgeschichte terminiert, kann jetzt auch die Laufzeit definiert werden.

			3	7	6			
		6			9			
	8					4		
	9					1		
6						9		
3					4			
7				8				
1			9					
	2	5	4					

9	5	4	1	3	7	6	8	2
2	7	3	6	8	4	1	9	5
1	6	8	2	9	5	7	3	4
4	9	5	7	2	8	3	6	1
6	8	1	4	5	3	2	7	9
3	2	7	9	6	1	5	4	8
7	4	9	3	1	2	8	5	6
5	1	6	8	7	9	4	2	3
8	3	2	5	4	6	9	1	7

Abbildung 12.4: In einem Sudoku-Rätsel sind in einem  $9 \times 9$ -Feld einige schwarze Ziffern vorgegeben. Die leeren Felder sind mit den Ziffern  $1, \dots, 9$  so auszufüllen, dass jede Ziffer in jeder Zeile, Spalte und in jedem  $3 \times 3$ -Unterquadrat genau einmal vorkommt. Die Lösung ist grau eingetragen.

**Definition 12.14** (Laufzeit eines nichtdeterministischen Entscheiders). *Sei  $M$  eine nichtdeterministische Turing-Maschine und  $w \in \Sigma^*$ . Die Laufzeit  $t(w)$  von  $M$  auf dem Input  $w$  ist die Länge der längsten möglichen Berechnungsgeschichte. Die worst case Laufzeit von  $M$  für ein Input der Länge höchstens  $n$  ist*

$$t(n) = \max\{t(w) \mid w \in \Sigma^* \wedge |w| \leq n\}.$$

Eine nichtdeterministische Turing-Maschine  $M$  muss im schlimmsten Fall in jedem Schritt zwischen verschiedenen möglichen Übergängen wählen. In Frage kommen alle Zustände  $Q$ , alle neuen Bandsymbole  $\Gamma$  und die beiden Kopfbewegungsrichtungen  $L$  und  $R$ . Es gibt also in jedem Schritt höchstens  $C = |Q| \cdot |\Gamma| \cdot 2$  Möglichkeiten. Um eine nichtdeterministische Turing-Maschine mit einer gewöhnlichen Turing-Maschine  $M'$  zu simulieren, muss jeder der  $C^{t(n)}$  möglichen Fälle durchgerechnet werden, die worst case Laufzeit ist daher

$$t'(n) = O(C^{t(n)}) = O(2^{t(n)\log 2}) = 2^{O(t(n)\log 2)} = 2^{O(t(n))}.$$

Die worst case Laufzeit wird somit exponentiell langsamer. Dieses Phänomen hat sich auch schon bei der Verwendung nichtdeterministischer Automaten für reguläre Ausdrücke in Abschnitt 4.3 gezeigt.

## Backtracking

Die Abschätzung der Laufzeit für die Simulation einer nichtdeterministischen Turing-Maschine ging davon aus, dass im schlimmsten Fall alle Berechnungsgeschichten vollständig durchgerechnet werden müssen. In der Praxis wird in solchen Fällen oft eine effizientere Vorgehensweise gewählt.

Bei einem Sudoku-Rätsel sind in einem  $9 \times 9$ -Spielfeld einige Felder bereits mit Ziffern  $1, \dots, 9$  ausgefüllt (Abbildung 12.4). Der Spieler muss die noch leeren Felder so mit

Ziffern ausfüllen, dass in jeder Zeile, Spalte und in jedem  $3 \times 3$ -Unterquadrat jede Ziffer genau einmal vorkommt. Beim Versuch, das Rätsel zu lösen, trifft der Spieler immer wieder auf Situationen, wo die Regeln den Inhalt für keines der noch nicht ausgefüllten Felder eindeutig festlegen, es bleiben mehrere Möglichkeiten. Ganz ähnlich wie bei einer nichtdeterministischen Turing-Maschine muss er alle Fälle durchprobieren.

Der Spieler optimiert seine Vorgehensweise aber mittels Backtracking. Sobald er an einen Punkt kommt, wo die Regeln ein Feld nicht mehr eindeutig festlegen, wählt er eine der verbleibenden Möglichkeiten und versucht damit, das Rätsel zu Ende zu lösen. Ist er erfolgreich, war die Wahl richtig. Gelingt es nicht, macht er alles rückgängig (Backtracking) und versucht die nächste Möglichkeit. Natürlich kann beim Versuch, das Rätsel fertig zu lösen, erneut die Situation eintreten, dass die Regeln keinen weiteren Feldinhalt eindeutig bestimmen. Es bietet sich daher eine rekursive Vorgehensweise an, wie sie auch im Algorithmus von Abbildung 3.1 zur Realisierung eines NEA vorgeschlagen wurde. Trotz Backtracking muss im schlimmsten Fall immer noch mit exponentieller Laufzeit gerechnet werden.

### 12.2.2 Verifizierer

Das Sudoku-Rätsel von Abbildung 12.4 illustriert auch noch ein anderes vertrautes Phänomen. Es ist viel einfacher, die Korrektheit der Lösung eines Sudoku-Rätsels zu verifizieren, als das Rätsel selbst zu lösen.

**Definition 12.15.** Sei  $A$  eine entscheidbare Sprache. Ein Verifizierer für  $A$  ist eine Turing-Maschine, so dass es für jedes Wort  $w \in \Sigma^*$  ein Wort  $c \in \Sigma^*$  gibt, das sogenannte Zertifikat, so dass  $w$  genau dann in  $A$  ist, wenn  $V$  das Paar  $\langle w, c \rangle$  akzeptiert:

$$w \in A \Leftrightarrow \langle w, c \rangle \in L(V).$$

Die Definition sagt nichts darüber aus, was das Lösungszertifikat beinhalten soll oder darf. Die Spezifikation des Verifizierers für die Sprache  $A$  muss auch festlegen, was im Zertifikat  $c$  an Information geliefert werden muss, damit der Verifizierer seine Arbeit machen kann.

*Beispiel 12.16.* Wir formalisieren das Sudoku-Rätsel zum Sprachproblem

$$SUDOKU = \left\{ \langle S \rangle \mid S \text{ ist ein teilweise ausgefülltes Sudoku-Rätsel, welches sich lösen, d. h. regelkonform vollständig ausfüllen lässt.} \right\}.$$

Die Sprache ist sicher entscheidbar, denn man kann alle  $9^{81}$  Möglichkeiten, das Spielfeld auszufüllen, durchprobieren. Sollte sich darunter keine Lösung befinden, ist das Rätsel nicht lösbar. Dieser Algorithmus ist natürlich nicht effizient, es geht aber auch nur darum, die Entscheidbarkeit festzustellen.

Ein Verifizierer braucht als zusätzliche Information das zu einem Sudoku-Rätsel  $w$  passende Lösungszertifikat  $c$ . Am einfachsten ist, das vollständig ausgefüllte Spielfeld als Zertifikat anzufordern. Dann muss der Verifikationsalgorithmus folgende Schritte ausführen:

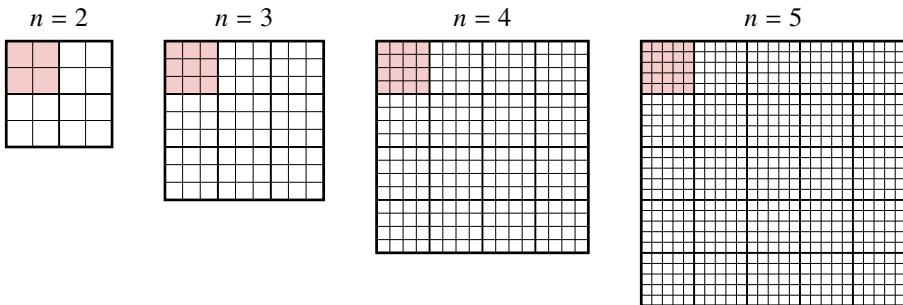


Abbildung 12.5: Skalierung des klassischen Sudoku-Rätsels auf einem  $n^2 \times n^2$ -Spielfeld mit  $n \times n$  Unterquadrate mit je  $n \times n$  Feldern.

1.  $w$  beschreibt ein Sudoku-Feld und  $c$  beschreibt eine Lösung dafür, d. h. die vorgegebenen Felder sind auch in  $c$  mit den gleichen Ziffern belegt.
2. Für jedes Feld des Rätsels kontrollieren, ob die dortige Ziffer in der gleichen Zeile nicht mehr vorkommt.
3. Für jedes Feld des Rätsels kontrollieren, ob die dortige Ziffer in der gleichen Spalte nicht mehr vorkommt.
4. Für jedes Feld des Rätsels kontrollieren, ob die dortige Ziffer im gleichen Unterquadrat nicht mehr vorkommt.  $\circlearrowright$

Die Definition des Verifizierers bildet die eingangs hervorgehobene Beobachtung, dass Verifizieren einfacher ist als Lösen, noch nicht ab. Unsere Betrachtungsweise misst den Schwierigkeitsgrad mit der Abhängigkeit der Laufzeit von der Problemgröße  $|w|$ . Alle Sudoku-Rätsel sind gleich groß, das Spielfeld ist  $9 \times 9$  Felder groß und kann mit 81 Zeichen spezifiziert werden.

Sudoku-Varianten können in beliebiger Größe formuliert werden. Statt von einem Spielfeld aus  $3 \times 3$  Unterquadrate von jeweils  $3 \times 3$  Feldern könnte man auch von einem Spielfeld aus  $n \times n$  Unterquadrate mit jeweils  $n \times n$  Feldern ausgehen, wie es in Abbildung 12.5 dargestellt ist. Ein solches  $n$ -Sudoku hätte  $n^4$  Felder, in die jeweils eines von  $n^2$  verschiedenen Symbolen einzutragen ist. Das bekannte Sudoku ist der Fall  $n = 3$ , in dem Ziffern zwischen 1 und  $n^2 = 3^2 = 9$  in den einzelnen Feldern Platz finden.

**Definition 12.17** (Polynomieller Verifizierer). *Ein polynomieller Verifizierer für die entscheidbare Sprache A ist ein Verifizierer V für A, dessen Laufzeit polynomiell in |w| ist.*

Man beachte, dass die Größe des Zertifikats keinen Einfluss auf die Laufzeit haben darf.

**Beispiel 12.18.** Das  $n$ -Sudoku lässt sich polynomiell verifizieren. Der Verifikationsalgorithmus von Beispiel 12.16 ist unverändert anwendbar, es muss nur noch der Rechenaufwand geschätzt werden.

Der Aufwand zur Überprüfung der Korrektheit der Aufgabenstellung ist proportional zur Größe des Rätsels, also  $O(n^4)$ . In den Schritten 2–4 muss für jedes Feld die gleiche Arbeit geleistet werden, es müssen  $n^2$  Felder geprüft werden. Für alle  $n^4$  Felder zusammen bedeutet dies einen Aufwand von  $O(n^6)$ .

Der Aufwand wird in Abhängigkeit von der Inputgröße abgeschätzt. Ein  $n$ -Sudoku hat Größe  $m = n^4$ , also ist der Rechenaufwand von der Größenordnung  $O(n^6) = O(n^{4+1.5}) = O(m^{1.5})$ , also mit Sicherheit polynomiell.  $\circlearrowright$

---

*Verständniskontrolle 12.4:* Konstruieren Sie einen polynomiellen Verifizierer für das  $n$ -Damen-Problem von Verständniskontrolle 12.1.



Die Rechnung am Schluss des Beispiels trifft man immer wieder an. Nehmen wir an, es gibt einen Parameter  $n$ , von dem sowohl die Inputgröße  $i(n)$  als auch die Rechenzeit  $\tau(n)$  abhängt. Im Beispiel ist die Inputgröße  $i(n) = n^4$  und die Rechenzeit  $\tau(n) = n^6$ . Wenn Inputgröße und Rechenzeit polynomiell in  $n$  sind, dann gibt es Exponenten  $k_i$  und  $k_\tau$  mit  $i(n) \approx n^{k_i}$  und

$$\tau(n) = O(n^{k_\tau}) = O(n^{k_i \cdot (k_\tau/k_i)}) = O(i(n)^{(k_\tau/k_i)}) \quad \Rightarrow \quad t(m) = O(m^{k_\tau/k_i}).$$

Die Rechenzeit ist also polynomiell mit dem Exponenten  $k_\tau/k_i$ . Es erübrigt sich somit, die Laufzeit explizit durch die Inputgröße auszudrücken, solange uns nur interessiert, ob die Laufzeit polynomiell ist.

*Beispiel 12.19.* Wir zeigen, dass das Färbeproblem polynomiell verifizierbar ist. Sei  $G$  ein Graph mit  $n$  Knoten, dann kann  $G$  höchstens  $n^2$  Kanten haben. Wir können also davon ausgehen, dass die Größe der Aufgabenstellung polynomiell in  $n$  ist. Jetzt soll ermittelt werden, ob  $G$  mit  $k$  Farben eingefärbt werden kann. Das Problem ist offenbar entscheidbar, man kann alle  $k^n$  Farbbelegungen der Knoten von  $G$  durchprobieren.

Ein Verifizierer verlangt als Zertifikat die Farben, die den Knoten zugeordnet sind. Damit wird der folgende Algorithmus durchgeführt:

Schritt	Verifikation	Laufzeit
1	Kontrolliere die Korrektheit der Graphdefinition.	$O(n^2)$
2	Für jede Kante, kontrolliere, ob die Farben, die den Endpunkten zugeordnet sind, verschieden sind.	$O(n^2)$
	Total	$O(n^2)$

Die Laufzeit ist polynomiell in  $n$ , also auch in der Inputlänge.  $\circlearrowright$

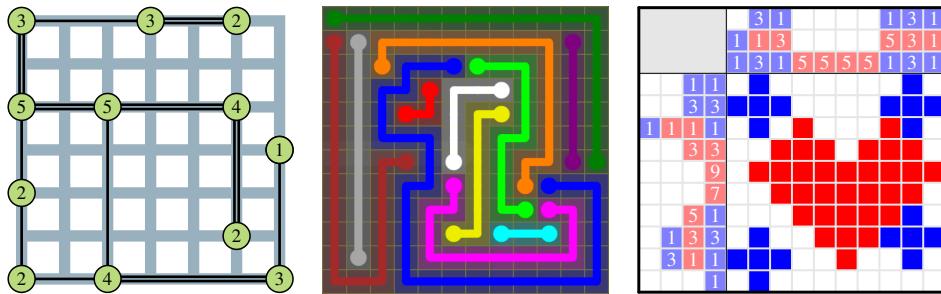


Abbildung 12.6: Viele populäre Rätselspiele stellen die Aufgabe, eine Spielfeld derart mit Zeichen zu füllen, dass gewisse Regeln eingehalten werden. Wenn die Regeln sich in polynomieller Zeit prüfen lassen, entsteht ein *polynomielles Ausfüllrätsel*. Von links nach rechts: Hashiwokakero (橋をかけろ), Flow und Nonogramm.

### 12.2.3 Polynomiale Ausfüllrätsel

Die Aufgabe des Sudoku-Rätsels besteht darin, eine Tabelle mit gewissen Zeichen nach vorgegebenen Regeln auszufüllen. Viele weitere solche Rätsel werden in Unterhaltungsspalten von Zeitungen abgedruckt und können auf Rätselwebsites gefunden werden. Abbildung 12.6 zeigt drei Beispiele. Die Erfahrung zeigt, dass diese Rätsel nur dann herausfordernd sind, wenn die Lösung exponentielle Laufzeit braucht. Rätsel mit polynomieller Laufzeit machen einen mechanischen Eindruck und langweilen den Rätsellöser.

Zu den bei Normalanwendern beliebtesten Anwendungsprogrammen gehören Tabellenkalkulationsprogramme wie Excel, OpenOffice Calc oder Numbers. Die Aufgabe dabei ist, eine Tabelle nach gewissen Regeln mit Elementen eines Alphabets zu füllen und so irgendein Problem zu lösen. Matt Parker vom YouTube-Kanal Stand-up Maths [45] hat verschiedene ernsthafte, aber auch komische Anwendungen solcher Spreadsheets in Videoform vorgestellt. Problemlösung mithilfe einer Tabelle scheint ein naheliegender und intuitiver Lösungsansatz zu sein.

Auch Problemstellungen, die auf den ersten Blick nichts mit einer Tabelle zu tun haben, können oft in diese Form gebracht werden. Sei  $G = (V, E)$  ein gerichteter Graph mit Knoten  $V$  und Kanten  $E$ . Der Graph kann durch eine quadratische Tabelle wie in Abbildung 12.7 dargestellt werden, die für jeden Knoten von  $V$  eine Zeile und eine Spalte enthält. Gibt es eine Kante von  $v_1$  nach  $v_2$ , dann bleibt das Feld  $(v_1, v_2)$  weiß, andernfalls wird es grau. Aus dieser Tabelle lässt sich der Graph jederzeit wieder rekonstruieren.

Ein Pfad in diesem Graphen ist eine Auswahl von Kanten. Wenn der Pfad kein Ende haben soll, muss jeder Knoten, der im Pfad vorkommt, sowohl als Anfangspunkt wie auch als Endpunkt einer Kante vorkommen. In der Tabellenformulierung können diese ausgewählten Kanten durch rote Punkte in den weißen Feldern dargestellt werden. Ein Pfad ohne Anfangs- oder Endpunkte kann dadurch charakterisiert werden, dass wenn Zeile  $v \in V$  einen roten Punkt enthält, dann muss auch Spalte  $v$  einen roten Punkt enthalten.

Die Tabelle ausfüllen heißt, nach gewissen Regeln für jedes Tabellenfeld  $(i, k)$  ein Zeichen  $z_{ik} \in \Sigma$  zu finden. Da es nur endlich viele mögliche Werte für Variablen  $z_{ik}$  gibt, müssen sich die Regeln durch logische Formeln in Ausdrücken der Form  $(z_{ik} = c)$  formu-

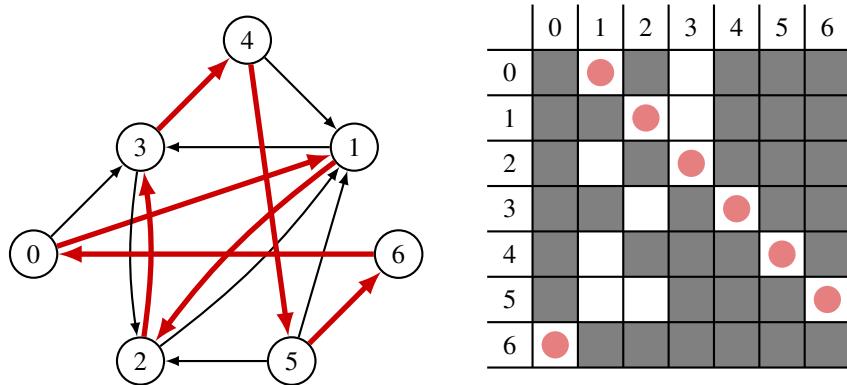


Abbildung 12.7: Das Problem *HAMCIRCUIT*, in einem gerichteten Graphen einen hamiltonschen Kreis zu finden, kann umformuliert werden in die Aufgabe, eine Tabelle nach gewissen Regeln mit roten Markierungen auszufüllen. Damit der Pfad jeden Punkt erreicht, muss zum Beispiel in jeder Zeile und Spalte ein roter Punkt vorhanden sein. Dies allein garantiert aber noch nicht, dass ein geschlossener Pfad entsteht.

lieren lassen. Ein polynomielles Problem liegt aber erst vor, wenn die Auswertung dieser Formeln ebenfalls in polynomieller Zeit möglich ist.

**Definition 12.20** (Polynomielles Ausfüllrätsel). *Ein polynomielles Ausfüllrätsel ist eine Aufgabe, bei der Zeichen eines vorgegebenen Alphabets  $\Sigma$  nach gewissen Regeln in ein  $n \times m$ -Spielfeld eingetragen werden müssen. Die Regeln lassen sich durch logische Formeln ausdrücken, die in einer Zeit ausgewertet werden können, die polynomiell in  $n \cdot m$  ist.*

*Beispiel 12.21.*  $n^2 \times n^2$ -Sudoku kann als polynomielles Ausfüllrätsel aufgefasst werden. Die Variablen  $z_{ik}$  können die Werte  $1, \dots, n^2$  annehmen. Die Regeln sagen, dass die Felder der gleichen Zeile verschiedene Werte haben sollen. Das kann man als

$$z_{ik} \neq z_{il} \quad \forall l \neq k$$

ausdrücken. Die Beschreibung der Formeln vor der Definition 12.20 verlangt aber eine etwas eingeschränktere Form, nämlich eine logische Formel, in der nur Terme der Form  $z_{uv} = c$  vorkommen. Die Sudoku-Regeln lassen sich aber in diese Form bringen. Ist  $c$  der Inhalt des Feldes  $z_{ik}$ , dann sagt die Formel

$$(z_{ik} = c) \Rightarrow \bigwedge_{l \neq k} (z_{il} \neq c),$$

dass die anderen Felder einen anderen Inhalt haben müssen. Dies muss für alle möglichen Werte  $c$  gelten, also muss die Formel

$$\varphi_{ik, \text{Zeile}} = \bigwedge_{c \in \Sigma} \left( (z_{ik} = c) \Rightarrow \bigwedge_{l \neq k} (z_{il} \neq c) \right)$$

wahr sein. Ähnliche Formeln lassen sich für die Spalten- und die Unterquadratbedingung aufstellen. Die Größe der Formeln ist polynomiell durch  $n$  beschränkt. Somit liegt tatsächlich ein polynomielles Ausfüllrätsel vor.  $\circ$

**Satz 12.22.** *Polynomielle Ausfüllrätsel lassen sich in polynomieller Zeit verifizieren.*

*Beweis.* Der polynomielle Verifizierer braucht als Zertifikat die Lösung des Ausfüllrätsels. Der Verifikationsalgorithmus überprüft die Regeln, was nach Definition eines polynomiellen Ausfüllrätsels in polynomieller Zeit möglich ist. □

---

**Verständniskontrolle 12.5:** Formulieren Sie das  $n$ -Damen-Problem von Verständniskontrolle 12.1 als polynomielles Ausfüllrätsel.



## 12.3 Komplexitätsklassen P und NP

Wir können jetzt die Laufzeit für die Lösung eines Problems als Klassifikationskriterium für entscheidbare Sprachen verwenden.

### 12.3.1 Polynomielle Probleme

In Abschnitt 12.1.2 wurde gezeigt, dass die Eigenschaft der Laufzeit, polynomiell von der Größe des Inputs abzuhängen, unabhängig von der Art der Hardware einer deterministischen Turing-Maschine ist. Die Sprachen, die sich in polynomieller Zeit entscheiden lassen, bilden daher eine Klasse von Sprachen, die unabhängig von der zur Verfügung stehenden Hardware ist.

**Definition 12.23.** *Eine entscheidbare Sprache gehört zur Klasse P, wenn sie sich in polynomieller Zeit entscheiden lässt.*

In der Klasse P findet man die meisten Aufgabenstellungen, für die ein Kurs über Algorithmen Lösungen lehrt:

- Ein lineares Gleichungssystem mit  $n$  Gleichungen für  $n$  Unbekannte ist ein Problem, dessen Größe durch die  $n^2$  Koeffizienten bestimmt ist. Der Gauß-Algorithmus braucht  $O(n^3)$  Operationen zur Lösung. Er ist also ein Algorithmus in P.
- Die schnelle Fourier-Transformation (FFT) eines Vektors von  $n$  Zahlen lässt sich in  $O(n \log n)$ -Operationen bestimmen. Wegen  $n \log n \leq n^2$  ist die Laufzeit polynomiell und die schnelle Fourier-Transformation ist in P.
- Sortieralgorithmen zum Sortieren eines Array von  $n$  Elementen brauchen typischerweise zwischen  $O(n \log n)$  und  $O(n^2)$  Vergleichsoperationen. Sortieren ist eine Aufgabenstellung in P.

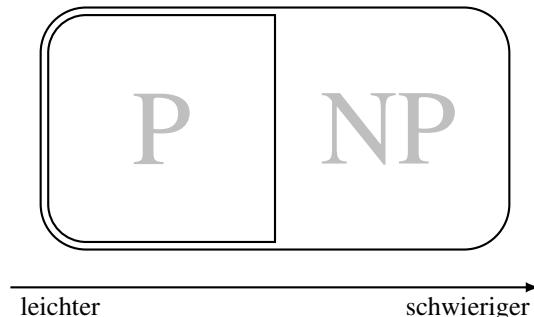


Abbildung 12.8: Die Klassen P und NP.

### 12.3.2 Die Klasse NP

Eine nichtdeterministische Turing-Maschine ist leistungsfähiger als eine gewöhnliche Turing-Maschine, sie hat Nichtdeterminismus als zusätzliche Geheimwaffe bei der Berechnung einer Lösung. Eine gewöhnliche Turing-Maschine ist eine nichtdeterministische Turing-Maschine, die diese spezielle Fähigkeit nicht nutzt.

**Definition 12.24.** Eine entscheidbare Sprache gehört zur Klasse NP, wenn sie sich von einer nichtdeterministischen Turing-Maschine in polynomieller Zeit entscheiden lässt.

Die Sprachen der Klasse NP bilden also eine Obermenge der Klasse P, wie dies Abbildung 12.8 zeigt.

Das Arbeiten mit nichtdeterministischen Turing-Maschinen ist nicht sehr intuitiv. Die zur Verfügung stehenden Computer sind deterministisch, Programmierer sind nicht gewohnt, nichtdeterministisch zu denken. Daher hilft es sehr, dass die Klasse NP auch mithilfe der Verifizierbarkeit definiert werden kann.

**Satz 12.25.** Eine entscheidbare Sprache gehört genau dann zur Klasse NP, wenn sie sich in polynomieller Zeit verifizieren lässt.

*Beweis.* Es sind zwei Dinge zu beweisen. Erstens muss gezeigt werden, wie zu einer Sprache in der Klasse NP ein polynomieller Verifizierer konstruiert werden kann. Sei also  $M$  eine nichtdeterministische Turing-Maschine, die  $A$  in polynomieller Zeit entscheidet. Wenn ein Wort  $w$  von  $A$  akzeptiert wird, dann gibt es eine Berechnungsgeschichte, die zu  $q_{\text{accept}}$  führt. In dieser Berechnungsgeschichte kann es einige Stellen geben, wo verschiedene Übergänge möglich wären. Sobald man weiß, welche Übergänge gewählt werden müssen, ist es einfach nachzuprüfen, ob die Berechnungsgeschichte tatsächlich im Zustand  $q_{\text{accept}}$  endet. Als Zertifikat werden daher die zu wählenden nichtdeterministischen Übergänge verlangt, die die Berechnung in den Zustand  $q_{\text{accept}}$  führen. Der Verifikationsalgorithmus verwendet das Wort  $w$  und die nichtdeterministischen Übergänge in  $c$  und verifiziert, ob die Berechnung tatsächlich in  $q_{\text{accept}}$  endet. Da die Länge der Berechnungsgeschichte polynomiell ist, ist die Laufzeit des Verifizierers polynomiell.

Zweitens muss gezeigt werden, dass wir zu einer Aufgabenstellung, die in polynomieller Zeit verifiziert werden kann, eine nichtdeterministische Turing-Maschine konstruieren

können, die die Lösung in polynomieller Zeit finden kann. Sei also  $A$  eine Sprache, die in polynomieller Zeit verifiziert werden kann und  $V$  der Verifizierer. Sei weiter  $w$  ein Wort. Der folgende Algorithmus ermittelt, ob  $w \in A$  ist:

1. Konsultiere das Orakel und erfrage das Lösungszertifikat für das Wort  $w$ . Dieser Schritt verwendet den Nondeterminismus.
2. Lasse den Verifizierer  $V$  auf  $\langle w, c \rangle$  laufen. Es ist  $w \in A$  genau dann, wenn der Verifizierer  $\langle w, c \rangle$  akzeptiert.

Der erste Schritt braucht keine Zeit, der zweite braucht polynomielle Zeit. Damit ist ein nondeterministischer Entscheidungsalgorithmus mit polynomieller Laufzeit gefunden.

□

Die Algorithmen der Klasse NP sind also jene, für die möglicherweise keine Lösung in polynomieller Zeit bekannt ist. Es ist aber immer möglich, einen Lösungskandidaten in polynomieller Zeit zu verifizieren. Das Problem *SUDOKU* gehört in diese Kategorie, alle bekannten Lösungsalgorithmen brauchen exponentielle Zeit, aber die Verifikation ist in polynomieller Zeit möglich.

Die Sicherheit digitaler Signatur-Algorithmen beruht darauf, dass es ohne zusätzliches Wissen sehr schwierig ist und typischerweise exponentielle Zeit verlangt, eine Signatur zu erzeugen, während die Verifikation einer Unterschrift leicht in polynomieller Zeit möglich ist.

In der Praxis versucht man Aufgabenstellungen in der Klasse NP zu vermeiden, sie erfordern typischerweise exponentielle Zeit und skalieren daher nicht.

Für Probleme außerhalb von NP lässt sich eine Lösung in polynomieller Zeit nicht einmal verifizieren. Damit ist die Entwicklung von Software für solche Probleme erheblich erschwert, denn nichttrivialer Testcode braucht ebenfalls exponentielle Laufzeit. Es sind daher nur sehr simple statische Tests möglich. Man kann daher sagen, dass sich der größte Teil der Informatik innerhalb von P abspielt, mit gelegentlichen Abstechern nach NP.

### 12.3.3 Das P vs. NP-Problem

Wie in Abbildung 12.8 dargestellt ist  $P \subset NP$ . Es ist unbekannt, ob die Klassen P und NP verschieden sind. Für eine große Zahl von Problemstellungen in NP sind keine Algorithmen mit polynomieller Laufzeit bekannt. Sie sind Kandidaten für Sprachen in  $NP \setminus P$ . Kapitel 13 befasst sich mit NP-vollständigen Problemen, einer interessanten Menge besonders schwieriger Aufgabenstellungen, die sicher in  $NP \setminus P$  sind, vorausgesetzt, P und NP sind verschieden. Für keines dieser Probleme sind polynomielle Algorithmen bekannt, was darauf hinweist, dass P und NP tatsächlich verschieden sein könnten.

$P = NP$  hätte dramatische Konsequenzen. Es hätte zur Folge, dass es für jedes Problem in NP auch einen polynomiellen Algorithmus gäbe. Kryptographische Algorithmen, deren Sicherheit auf der langen Laufzeit basieren, wären damit nicht mehr sicher. Aus diesem Grund hat das Clay Research Institute im Jahre 2000 nach dem Vorbild der hilbertschen Probleme einen Preis von 1 Mio. USD für die Klärung dieser und weiterer Fragen ausgeschrieben. Es glaubt, dass diese Millennium-Probleme die Mathematik des 21. Jahrhunderts prägen könnten [24].

		1	4
2	4		

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

Abbildung 12.9:  $2^2 \times 2^2$ -Sudoku zu Aufgabe 12.4, links die Problemstellung, rechts die Lösung.

## Übungsaufgaben

**12.1.** Konstruieren Sie ein binäres Additionsprogramm für eine geeignet ausgebauten Turing-Maschine, welches Laufzeit  $O(n)$  hat, wobei  $n$  die Anzahl Stellen der Summanden ist. Warum ist Ihre Maschine so viel schneller als die im auf Seite 226 verlinkten Video gezeigte Maschine?

**12.2.** Zwei Graphen  $G$  und  $H$  heißen isomorph, wenn die Knoten von  $G$  so umgeordnet werden können, dass die beiden Graphen übereinstimmen. Zeigen Sie, dass

$$ISO = \{(G, H) \mid G \text{ und } H \text{ sind isomorph.}\}$$

in NP ist.

**12.3.** Seien  $A_1$  und  $A_2$  Sprachen, die von einer nichtdeterministischen Turing-Maschine in polynomieller Zeit entscheidbar sind. Zeigen Sie:

- a) Die Schnittmenge  $A_1 \cap A_2$  ist in NP.
- b) Die Verkettung  $A_1 A_2$  ist in NP.

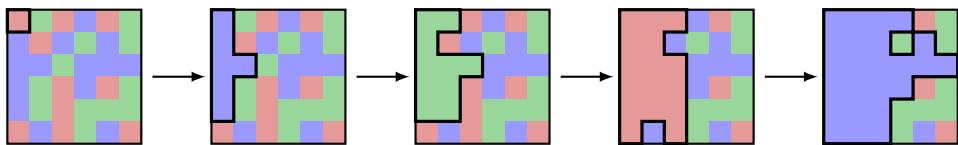
**12.4.** Konstruieren Sie eine Reduktion  $SUDOKU \leq_P VERTEX-COLORING$ .

*Hinweis.* Es reicht, die Reduktion für den Fall  $n = 2$  durchzuführen, also für das  $2^2 \times 2^2$ -Sudoku, wenn daraus klar wird, wie die Verallgemeinerung für größere  $n$  zu handhaben ist. Sie können ganz konkret das Beispiel aus Abbildung 12.9 für Ihre Konstruktion verwenden.

**12.5.** Beim Spiel Flood-It sind die Felder eines  $n \times m$ -Spielfeld mit mindestens drei verschiedenen Farben eingefärbt. Man sagt, zwei Felder sind verbunden, wenn sie die gleiche Farbe haben und über eine Kante benachbart sind. Ein Gebiet besteht aus allen miteinander verbundenen Feldern. Der Spieler kann jetzt jeweils die Farbe des Feldes in der linken

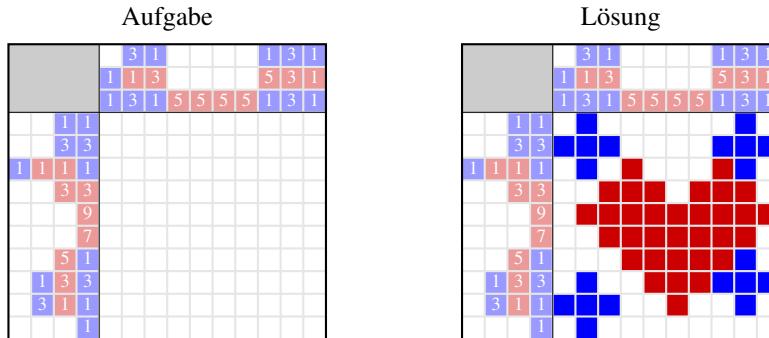
oberen Ecke ändern, dabei wird auch die Farbe des ganzen zugehörigen Gebietes geändert. Da an das Gebiet Felder der neuen Farbe anstoßen können, kann das Gebiet durch den Farbwechsel größer werden. Ziel ist, das Gebiet der linken oberen Ecke in möglichst wenigen Farbwechseln so anwachsen zu lassen, dass es das ganze Spielfeld überdeckt.

Im folgenden Bild sind vier Schritte des Spiels dargestellt, das Gebiet des Feldes in der linken oberen Ecke ist jeweils fett ausgezogen.



Kann eine nichtdeterministische Turing-Maschine in polynomieller Zeit herausfinden, ob zu einer vorgegeben Zahl  $k$  ein Flood-It Rätsel in höchstens  $k$  Schritten gelöst werden kann?

**12.6.** Die japanische Designerin Non Ishida hat im Jahre 1986 eine Art von Logikrätsel erfunden, die ihr zu Ehren Nonogramme genannt werden. In einem  $n \times m$  Spielfeld sind die einzelnen Felder mit einer von mehreren möglichen Farben zu füllen. Am Rand des Feldes wird angegeben, wie lange Folgen benachbarter gleichfarbiger Felder in einer Zeile oder Spalte jeweils zu finden sind:



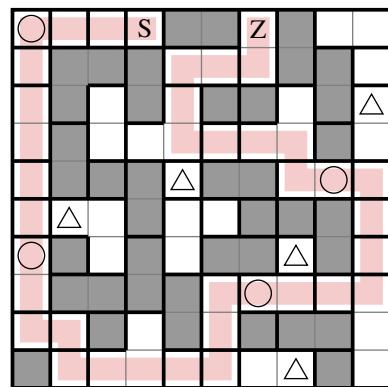
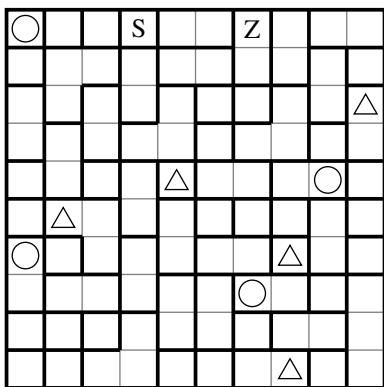
Die Vorgabe blau 1, rot 3, blau 3 in der drittuntersten Zeile bedeutet, dass in dieser Zeile zunächst ein einzelnes blaues Feld vorkommen muss, dann drei benachbarte rote Felder gefolgt von drei benachbarten blauen Feldern.

Das Problem *NONOGRAMM* ist also die Aufgabe, zu entscheiden, ob ein Nonogramm-Rätsel überhaupt lösbar ist.

- Ist *NONOGRAMM* entscheidbar?
- Kann eine nichtdeterministische Turing-Maschine in polynomieller Zeit entscheiden, ob ein Nonogramm gelöst werden kann?

**12.7. Nurimeizu** (塗り メイズ, von 塗り Malerei, und メイズ Labyrinth) ist ein japanisches Logikrätsel, welches auf einem  $n \times m$ -Feld gespielt wird. Das Feld wird von fetten Linien in kleine Gebiete unterteilt. Der Spieler muss nun Gebiete grau einfärben, die keines der Zeichen  $S$ ,  $Z$ , Dreieck oder Kreis enthalten. Die verbleibenden weißen Felder bilden ein Labyrinth. Weder die weißen noch die grauen Felder dürfen irgendwo einen Bereich der Größe  $2 \times 2$  enthalten. Die weißen Felder müssen einen zusammenhängenden Bereich bilden. Sie bilden ein Wegnetz im Labyrinth, welches keinen Rundweg enthalten darf. Es muss einen Weg von  $S$  nach  $Z$  geben, der alle mit Kreis markierten Felder enthält, aber keines der mit Dreieck markierten Felder.

Das folgende Beispiel zeigt links das Rätsel und rechts die Lösung. Der gesuchte Weg ist rosa eingezzeichnet.



Kann eine nichtdeterministische Turing-Maschine in polynomieller Zeit entscheiden, ob ein Nurimeizu-Rätsel lösbar ist?

Lösungen: <https://autospr.ch/uebungen/AutoSpr-112.pdf>

# Kapitel 13

## NP-Vollständigkeit

### 13.1 NP-vollständige Sprachen

Wie groß ist die Klasse NP? In Abbildung 12.8 scheint es eine obere Schranke für den Schwierigkeitsgrad für die Sprachen in NP zu geben. Das muss nicht so sein. Die Definition der Klasse NP besagt ja, dass eine nichtdeterministische Turing-Maschine ein Problem in dieser Klasse in polynomieller Zeit lösen kann. So ein Problem kann also nicht “beliebig schwierig” sein, doch ist nicht klar, wie einschränkend die Bedingung ist. Dies soll in diesem Abschnitt geklärt werden.

#### 13.1.1 Die schwierigsten Probleme in der Klasse NP

Gibt es ein schwierigstes Problem in der Klasse NP? In diesem Fall könnte man die Klasse NP eher wie in Abbildung 13.1 zeichnen. Natürlich ist noch nicht klar, ob es nur ein solches maximal schwieriges Problem gibt. Der rechte Rand der Klasse NP könnte auch

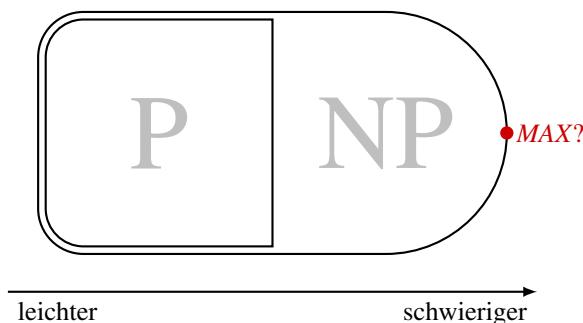


Abbildung 13.1: Gibt es in der Klasse NP ein schwierigstes Problem, hier als MAX bezeichnet?

mehrere Ausbuchungen nach rechts mit jeweils eigenen schwierigsten Problemen haben. Es ist sogar denkbar, dass solche Probleme nicht miteinander vergleichbar sind, also bereits die Verwendung der horizontalen Schwierigkeitsskala irreführend ist.

Das Problem *MAX* in dieser Darstellung scheint eine Art universelles Problem zu sein. Alle anderen Probleme in NP liegen weiter links. Nach der Definition der horizontalen Schwierigkeitsskala mithilfe der polynomiellen Reduktion müsste man dies so interpretieren, dass sich jedes NP-Problem auf das Problem *MAX* polynomiell reduzieren lässt. Auf den ersten Blick scheint es lächerlich anzunehmen, dass sich jedes Problem in NP in eine Formulierung übersetzen lässt, die anschließend mit einem universellen Algorithmus gelöst werden kann, der mindestens auf einer nichtdeterministischen Maschine sogar in polynomieller Zeit laufen kann. Was hat ein Problem wie *SUDOKU* oder *UHAMCIR-CUIT* mit dem Stundenplanproblem gemeinsam?

Bei näherer Betrachtung stellt sich jedoch heraus, dass die Probleme in NP eine wichtige Gemeinsamkeit haben. Sie alle lassen sich zwar nicht immer in polynomieller Zeit lösen, aber nach Satz 12.25 in polynomieller Zeit verifizieren. Die Gemeinsamkeiten werden noch ausgeprägter bei den polynomiellen Ausfüllrätseln von Abschnitt 12.2.3. Alle diese Probleme haben die Struktur einer Tabelle, die mit geeigneten Zeichen ausgefüllt werden muss. Ob eine Lösung vorliegt, wird durch die Regeln bestimmt, die für jedes Feld gelten und die sich in polynomieller Zeit evaluieren lassen. Die Regeln lassen sich als Formeln mit einem booleschen Wert formulieren. Eine Lösung liegt vor, wenn alle Formeln einen wahren Wert ergeben. Den polynomiellen Ausfüllrätseln ist also gemeinsam, dass sie sich auf die Frage reduzieren lassen, ob eine Formel den Wahrheitswert *true* annehmen kann.

Die polynomiellen Ausfüllrätsel bilden also bereits eine größere Familie von Problemen, die ein gemeinsames Lösungsverfahren mit nichtdeterministischen Maschinen haben. Es ist daher nicht abwegig zu vermuten, dass es tatsächlich ein genügend verallgemeinertes Problem geben könnte, welches die Rolle eines “schwierigsten” Problems wahrnehmen kann.

### 13.1.2 Formale Definition

Die Überlegungen des vorangegangenen Abschnitts haben gezeigt, dass es nicht ganz weit hergeholt ist zu vermuten, dass es so etwas wie “ein schwierigstes” Problem in der Klasse NP gibt. Die folgende Definition formalisiert diese Idee, legt sich aber nicht darauf fest, dass es nur ein solches Problem geben kann.

**Definition 13.1** (NP-vollständig). *Eine Sprache B in NP heißt NP-vollständig, wenn sich jede andere Sprache A in NP polynomiell auf B reduzieren lässt:  $A \leq_P B$ .*

Wir werden später sehen, dass es NP-vollständige Probleme in großer Zahl gibt. Es gibt also nicht ein maximal schwieriges Problem wie in Abbildung 13.1 suggeriert, sondern sehr viele extrem schwierige Probleme, die wie in Abbildung 13.2 die Klasse NP am rechten Rand abschließen oder vervollständigen, was auch den Namen rechtfertigt.

Hat man ein erstes NP-vollständiges Problem gefunden, lassen sich weitere mithilfe geeigneter Reduktionen finden.

**Satz 13.2.** *Ist A NP-vollständig und  $A \leq_P B$ , B in NP, dann ist B NP-vollständig.*

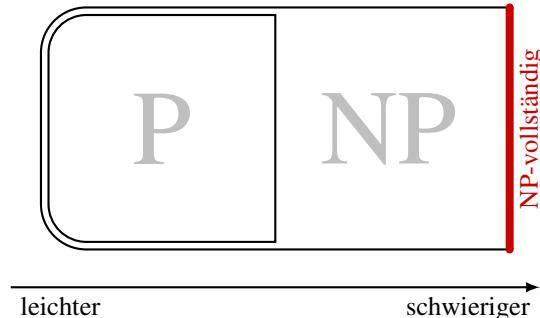


Abbildung 13.2: Die NP-vollständigen Probleme (rot) bilden den Abschluss der Klasse NP nach oben, es gibt keine “schwierigeren” Probleme in der Klasse NP.

*Beweis.* Man muss zeigen, dass sich jedes andere Problem  $C$  in NP polynomiell auf  $B$  reduzieren lässt. Da aber  $A$  NP-vollständig ist, lässt sich  $C$  polynomiell auf  $A$  reduzieren. Durch einen weiteren polynomiellen Reduktionsschritt auf  $B$ , der wegen  $A \leq_P B$  möglich ist, lässt sich auch eine Reduktion auf  $B$  erreichen.  $\square$

Wir haben in Abschnitt 12.1.4 bereits eine umkehrbare Reduktion des Stundenplanproblems auf  $k$ -VERTEX-COLORING konstruiert. Später werden wir zeigen, dass  $k$ -VERTEX-COLORING tatsächlich NP-vollständig ist. Wegen der Reduktion

$$k\text{-VERTEX-COLORING} \leq_P STUNDENPLAN$$

folgt jetzt mit Satz 13.2, dass auch *STUNDENPLAN* ein NP-vollständiges Problem ist.

In Abschnitt 12.3.3 wurde darauf hingewiesen, dass aktuell nicht bekannt ist, ob die Klassen P und NP verschieden sind. Wäre aber ein NP-vollständiges Problem  $B$  in P, wäre es also in polynomieller Zeit lösbar, dann hätte das zur Folge, dass jedes andere Problem in NP in polynomieller Zeit gelöst werden kann. Dazu gehören insbesondere auch die kryptographischen Algorithmen, die auf der Faktorisierung von großen Primzahlprodukten oder auf dem diskreten Logarithmus basieren. Ihre Sicherheit verdanken diese Algorithmen der Tatsache, dass sich zum Beispiel ein Schlüssel nach aktuellem Wissen nur in exponentieller Zeit finden lässt. Ist der Schlüssel jedoch bekannt, sind alle nötigen Berechnungen in polynomieller Zeit möglich. Aus dieser Perspektive sind die NP-vollständigen Probleme diejenigen, für die es am lohnendsten wäre, einen polynomiellen Algorithmus zu finden und damit die Frage, ob P und NP verschieden sind, endgültig negativ zu entscheiden.

### 13.1.3 Alle NP-vollständigen Probleme sind äquivalent

Die Definition 13.1 hat eine interessante Konsequenz. Sind die beiden Sprachen  $A$  und  $B$  beide NP-vollständig, dann sind beide Sprachen in NP. Da  $B$  NP-vollständig ist, lässt sich  $A$  auf  $B$  reduzieren, d. h.  $A \leq_P B$ . Ebenso ist  $B \leq_P A$ , weil  $A$  NP-vollständig ist. Somit sind alle NP-vollständigen Probleme polynomiell äquivalent, sie lassen sich immer aufeinander reduzieren.

**Satz 13.3.** *Sind  $A$  und  $B$  NP-vollständige Sprachen, dann ist  $A \leq_P B$  und  $B \leq_P A$  und somit  $A \equiv_P B$ .*

Der Satz 13.3 besagt, dass alle NP-vollständigen Probleme polynomiell äquivalent sind.

### 13.1.4 Der Umgang mit NP-Vollständigkeit in der Praxis

Die Abschnitte 13.3 und 13.4 werden zeigen, dass NP-vollständige Probleme in der Softwareentwicklungspraxis regelmäßig auftauchen. Für NP-vollständige Probleme ist nach aktuellem Wissen kein Algorithmus bekannt, der polynomiell skaliert. Wir erhoffen daher von der Theorie zusätzliche Hilfestellung bei der Entwicklung performanter Anwendungen. Die folgenden Abschnitte stellen ein paar Ideen zusammen, wie man mit Problemstellungen umgehen kann, für die kein polynomieller Algorithmus bekannt ist.

#### Spezialfälle ausnützen

In der Praxis tritt ein Problem selten in seiner reinsten Form isoliert auf. Vielmehr ist es eingebettet in einen größeren Zusammenhang. Es ist also nicht ein allgemeines Problem, wie man es in einem Katalog NP-vollständiger Probleme findet, sondern es ist fast immer auf die eine oder andere Art ein Spezialfall. Durch Ausnützen der speziellen Rahmenbedingungen können sich manchmal Wege ergeben, die die Komplexität der Problemlösung senken können.

#### Das Problem modifizieren

Wird das Problem rechtzeitig erkannt, kann man versuchen, die Aufgabenstellung abzuändern und damit hoffentlich die Problemlösung exponentiell zu beschleunigen. Da Verifizieren in polynomieller Zeit möglich ist, kann dies bedeuten, dass Lösungen oder Teillösungen gespeichert werden, so dass sie nur noch abgerufen werden müssen. Da nur noch kontrolliert werden muss, ob die zusammengesetzten Lösungen korrekt sind, ist eine besser skalierende Lösung denkbar.

Ein schönes Beispiel, wo zusätzliche Daten eine exponentielle Beschleunigung ermöglichen, ist aus der Datenbanktechnik bekannt. Ohne Hilfe dauert die lineare Suche in einer Tabelle einer Datenbank nach einem Datensatz im schlimmsten Fall so lange, wie das Lesen der ganzen Tabelle braucht. Hat die Tabelle  $n$  Zeilen, dann ist die Laufzeit  $O(n)$ . Verwendet man aber einen Index, der typischerweise als binärer Baum organisiert wird, dann kann die Laufzeit auf  $O(\log_2 n)$  verbessert werden. Dies ist eine exponentielle Beschleunigung.

#### Approximation

Bei einem NP-vollständigen Problem kann man oft nicht erwarten, eine exakte Lösung in vernünftiger Zeit zu erhalten. Man wird sich daher mit einer Approximation begnügen müssen, die sich in polynomieller Zeit finden lässt. Die Approximation kann aber nicht beliebig genau sein. Wäre es nämlich möglich, beliebig genaue Resultate in polynomieller

Zeit zu erhalten, dann könnte man die Genauigkeitsanforderungen so einstellen, dass es keinen praktischen Unterschied mehr gibt zwischen der exakten Lösung und der Approximation. Damit hätte man eine polynomiale Lösung gefunden. Approximationslösungen in polynomieller Zeit sind also grundsätzlich von eingeschränkter Genauigkeit. Dieser Zusammenhang wird in [17] ausführlich untersucht.

## Zufall

In Abschnitt 3.4 wurde gezeigt, wie nichtdeterministische Entscheidungen eines endlichen Automaten durch Zufallsprozesse ersetzt werden können. Dies ist selbstverständlich auch für Turing-Maschinen möglich. Abschnitt 15.3.1 zeigt, wie sich daraus die neue Komplexitätsklasse BPP ergibt, in der gewisse NP-Probleme eine polynomiale Lösung haben, allerdings mit der Einschränkung, dass mit einer nicht verschwindenden Fehlerwahrscheinlichkeit zu rechnen ist.

## Heuristik

Die exponentielle Laufzeit von NP-Algorithmen kommt daher, dass es viele Auswahlmöglichkeiten gibt. Statt eine Auswahl rein zufällig zu treffen, kann man versuchen, durch heuristische Regeln die Anzahl der Auswahlmöglichkeiten einzuschränken. Man kann Heuristiken als eine spezielle Art eines Orakels betrachten. Ein Fachmann, der aus seiner Erfahrung heraus eine Problemlösung auf einem direkteren Weg finden kann, wird für einen Nichtfachmann nicht anders aussehen als ein Orakel, welches den richtigen Weg einfach so “kennt”.

Selbst mit dem Einsatz von Heuristiken mag das Problem immer noch exponentiell skalieren. Da aber weniger Möglichkeiten zu untersuchen sind, sind Approximationen oder Zufallsalgorithmen erfolgversprechender. Diesen Ansatz verfolgt das automatische Prüfungsplansystem der OST Ostschweizer Fachhochschule. In Abschnitt 13.3.2 wird gezeigt, dass das Stundenplanproblem NP-vollständig ist. Die Anzahl der Prüfungen und Anmeldungen wäre viel zu groß, um alle Möglichkeiten durchzuprobieren. Daher werden zusätzliche Regeln verwendet, um die Menge der zu untersuchenden Lösungen einzuschränken. Einige Regeln sind mit einer “Strafe” versehen, so dass jeder Lösungsversuch auch eine Bewertung erhält, die dazu verwendet werden kann, die Suche nach der besten Lösung zu priorisieren.

## Genetische Algorithmen

Genetische Algorithmen verwenden Zufall und Heuristik auf eine sehr spezielle Art, um effizient gute approximative Lösungen zu finden. Zu diesem Zweck werden Lösungen aus einer Menge von “Genen” konstruiert, die man auch als Parametrisierung der Lösung betrachten kann. Durch Neukombination von Genen guter Lösung kann man die Wahrscheinlichkeit erhöhen, noch bessere Lösungen zu finden.

## Artificial Intelligence

In neuerer Zeit sind AI-Modelle leistungsfähig genug geworden, um die Funktion des Orakels mindestens approximativ zu übernehmen. Im Prinzip ist dies ein Spezialfall der Idee, den Zufall zu Hilfe zu rufen. Ein AI-Modell lernt aus vielen guten Beispielen ein stochastisches Modell, welches für jede Vorgabe eine Antwort geben kann. Die vom AI-Modell gelernte Antwort muss nicht die beste sein, aber sie kann als Ansatz für die Suche nach einer besseren Lösung dienen.

## 13.2 Der Satz von Cook-Levin

Bis jetzt wissen wir nicht mit Sicherheit, dass es überhaupt ein NP-vollständiges Problem gibt. Wir haben nur plausible Argumente, warum es so etwas geben könnte und wir haben mithilfe der polynomiellen Ausfüllrätsel auch Kandidaten dafür identifiziert. Es ist das Verdienst von Stephen A. Cook, 1971 bewiesen zu haben, dass das Problem *SAT* tatsächlich NP-vollständig ist. Leonid Levin hat 1973 unabhängig von Cook ein vergleichbares Resultat bewiesen. In diesem Abschnitt soll dieser Satz von Cook und Levin mit der Methode der polynomiellen Ausfüllrätsel nach [39] bewiesen werden.

### 13.2.1 SAT: Erfüllbarkeit für logische Formeln

Ob eine vorgeschlagene Lösung eines Sudoku-Rätsels tatsächlich eine Lösung ist, kann durch Evaluation logischer Formeln überprüft werden. Dies hat Sudoku mit polynomiellen Ausfüllrätseln gemeinsam. Lösungen zu polynomiellen Ausfüllrätseln können durch Evaluation logischer Formeln verifiziert werden. Ein *constraint solver* ist eine Software, die zu einer gegebenen logischen Formel  $\varphi(x_1, \dots, x_n)$  geeignete Wahrheitswerte der Variablen  $x_1, \dots, x_n$  findet, die die Formel wahr machen. Ein constraint solver kann also sowohl Sudoku als auch beliebige Ausfüllrätsel lösen. Die logischen Formeln, die sich bei Sudoku oder einem polynomiellen Ausfüllrätsel ergeben, sind aber ziemlich speziell, ein guter constraint solver kann viel allgemeinere Probleme lösen. Damit haben wir den Kandidaten für das universelle schwierigste Problem gefunden.

**Problem 13.4 (SAT). Die Sprache**

$$SAT = \{\varphi \mid \varphi \text{ ist eine erfüllbare logische Formel.}\}$$

heißt das Erfüllbarkeitsproblem oder Satisfiability.

**Satz 13.5 (Cook-Levin).** *SAT ist NP-vollständig.*

Der Satz besagt, dass sich jedes beliebige Problem in NP polynomiell auf *SAT* reduzieren lässt. Sei  $A$  eine Sprache in NP, dann muss zu jedem Wort  $w \in \Sigma^*$  eine logische Formel  $\varphi$  konstruiert werden, die genau dann erfüllbar ist, wenn  $w \in A$  ist.

Das Einzige, was alle Sprachen in NP gemeinsam haben, ist die Tatsache, dass es für jede Sprache eine nichtdeterministische Turing-Maschine gibt, die sie entscheiden kann. Diese Eigenschaft muss verwendet werden, um die gesuchte Formel zu konstruieren. Ein

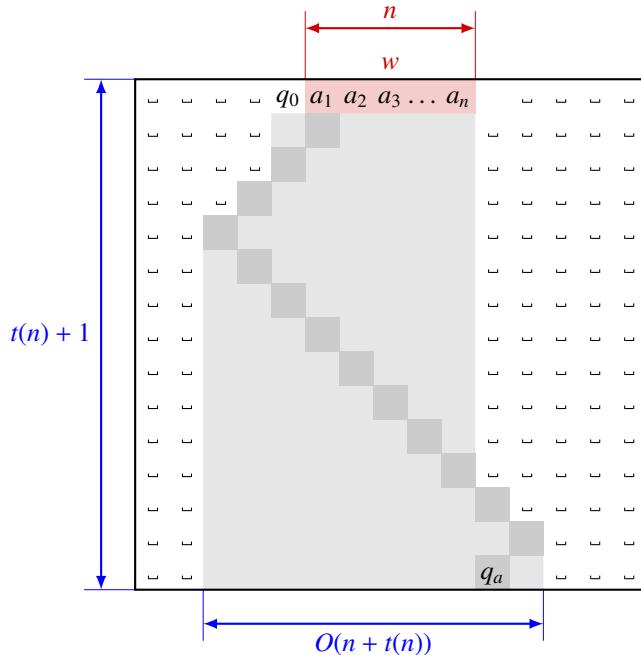


Abbildung 13.3: Die Berechnungsgeschichte, die das Wort  $w = a_1 a_2 \dots a_n$  akzeptiert, kann als Ausfüllrätsel betrachtet werden. Da die Laufzeit kleiner als  $t(n)$  ist und der Schreib-/Lesekopf nicht weiter als  $t(n)$  Felder nach links oder rechts von der Startposition gelangen kann, ist das Feld für die Berechnungsgeschichte von der Größe  $O(n + t(n)) \times (t(n) + 1)$ . Die erste Zeile enthält als Vorgabe das Wort  $w$  und den Startzustand, die letzte Zeile muss den Akzeptierzustand  $q_{\text{accept}}$  enthalten. Die Regeln des polynomiellen Ausfüllrätsels stellen sicher, dass aufeinanderfolgende Zeilen mögliche Übergänge der Turing-Maschine sind.

Wort  $w \in A$  hat eine Berechnungsgeschichte, die im Zustand  $q_{\text{accept}}$  endet. Für ein Wort außerhalb  $A$  gibt es keine solche Berechnungsgeschichte. Es liegt daher nahe, die Formel so zu konstruieren, dass sie genau dann erfüllbar ist, wenn es eine akzeptierende Berechnungsgeschichte gibt. Dazu gehen wir wie folgt vor:

1. Reduktion der Sprache  $A$  auf ein polynomielles Ausfüllrätsel (Abschnitt 13.2.2).
2. Reduktion des polynomiellen Ausfüllrätsels auf  $SAT$  (Abschnitt 13.2.3).

In Abschnitt 13.2.4 werden wir zeigen, dass die Reduktion auch noch einen Schritt weiter auf  $3SAT$  gehen kann, so dass auch gleich gezeigt ist, dass  $3SAT$  NP-vollständig ist.

### 13.2.2 Reduktion auf ein polynomielles Ausfüllrätsel

Die Berechnungsgeschichte (siehe Abbildung 13.3) besteht aus einzelnen Zeilen, die Zeichen des Bandalphabets  $\Gamma$  der Turing-Maschine und Zustandssymbole aus  $Q$  enthalten.

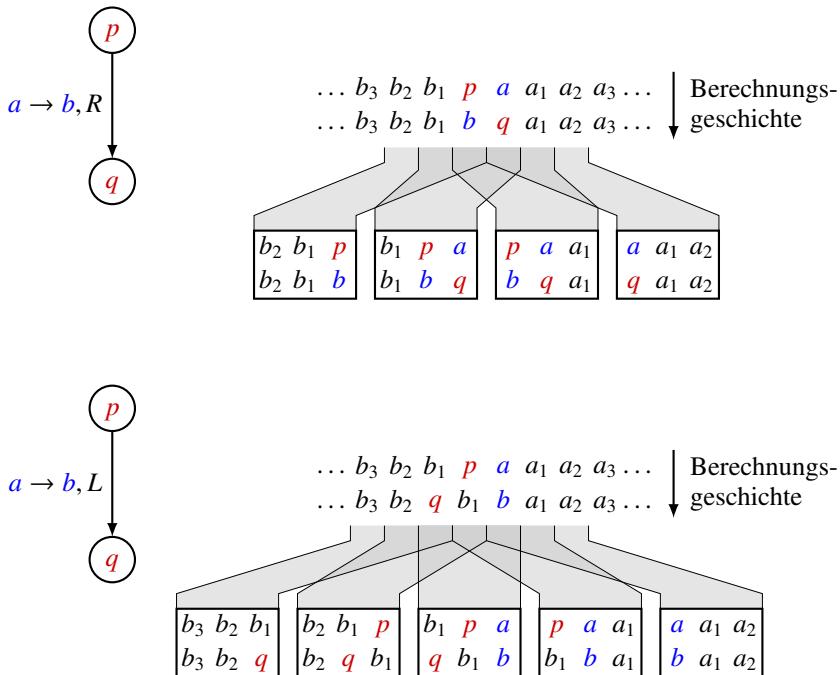


Abbildung 13.4: Die Regeln für das Ausfüllrätsel werden festgelegt, indem Fenster der Größe  $3 \times 2$  daraufhin geprüft werden, ob darin die Zeichen eines Turing-Maschinenübergangs sichtbar sind.

Die erste Zeile der Berechnungsgeschichte ist ebenfalls bekannt. Sie enthält das Symbol für den Startzustand  $q_0$ , gefolgt von den Zeichen des zu verarbeitenden Wortes  $w = a_1 a_2 \dots a_n$ . Am Ende der Berechnung erwarten wir den Zustand  $q_{\text{accept}}$  irgendwo auf der letzten Zeile der Berechnungsgeschichte. Die Berechnungsgeschichte entsteht also dadurch, dass man eine geeignete Tabelle mit Zeichen aus  $\Gamma \cup Q$  füllt, wobei die erste Zeile bereits gegeben ist und in der letzten Zeile irgendwo  $q_{\text{accept}}$  stehen muss.

Die Größe der Tabelle ist ebenfalls bekannt. Sie kann nicht höher sein als die Laufzeit  $t(n) + 1$  der Turingmaschine. Die Breite der Tabelle ist mindestens  $n$ , der Schreib-/Lesekopf kann aber nach links oder rechts über das Anfangswort hinausfahren und damit den beschriebenen Teil des Bandes vergrößern. Weiter als  $t(n)$  nach links oder rechts kann der Schreib-/Lesekopf aber nicht gelangen, so dass die Tabelle nicht breiter sein kann als  $n + 2t(n)$  oder  $O(n + t(n))$ . Da sowohl die Höhe  $t(n) + 1$  als auch die Breite  $n + 2t(n)$  polynomiell in  $n$  sind, ist die Größe der Tabelle polynomiell in  $n$ .

Um ein polynomielles Ausfüllrätsel zu erhalten, müssen jetzt noch die Regeln festgelegt werden, die für jedes Feld der Tabelle einzuhalten sind. Zwei aufeinanderfolgende Zeilen müssen immer durch einen Turing-Maschinenübergang auseinander hervorgehen. Bei einem solchen ändert sich nur in der unmittelbaren Umgebung des Zustandssymbols etwas. Das Symbol selbst kann durch ein anderes aus  $Q$  ersetzt werden, und das Zeichen unmittelbar rechts davon kann mit einem anderen Symbol aus  $\Gamma$  überschrieben werden. Alle Änderungen finden also vollständig in einem kleinen Fenster von nur  $3 \times 2$  Feldern

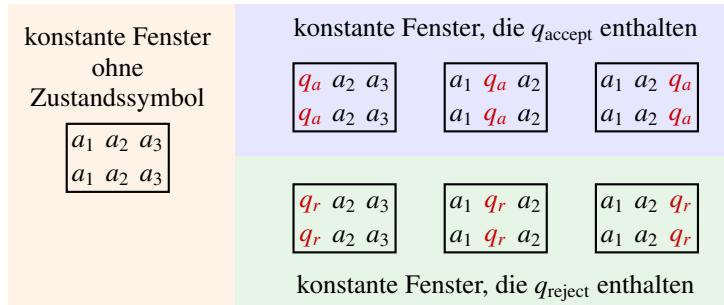


Abbildung 13.5: Konstante Fenster enthalten zwei identische Zeilen. Dieser Fall tritt ein, wenn der Schreib-/Lesekopf sich außerhalb des Fensters befindet oder wenn auf der ersten Zeile ein Akzeptierzustand gefunden wird. Da die Maschine in den Akzeptierzuständen anhält, muss die zweite Zeile identisch zur ersten sein.

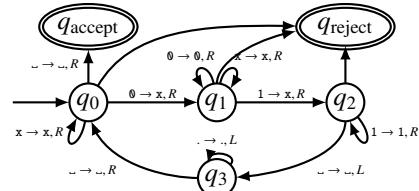
statt, wie in Abbildung 13.4 dargestellt.

Zu den genannten Fenstern kommen weitere hinzu, die beschreiben, dass sich zwischen den beiden Zeilen der Berechnungsgeschichte nichts ändert. Dieser in Abbildung 13.5 illustrierte Fall tritt ein, wenn sich der Schreib-/Lesekopf außerhalb des Rechtecks befindet oder wenn die Maschine einen Akzeptierzustand erreicht. In einem Akzeptierzustand hält die Maschine an und die zweite Zeile des Rechtecks muss identisch zur ersten Zeile sein.

*Verständniskontrolle 13.2:* Das untenstehende Zustandsdiagramm gehört zu einer Turingmaschine, die die Sprache  $\{\emptyset^n 1^n\}$  entscheidet. Gehören die nachstehenden  $3 \times 2$ -Ausschnitte zu einer korrekten Berechnungsgeschichte? Wenn ja, können Sie den möglicherweise nicht sichtbaren Zustand ermitteln?



- a) 
$$\begin{array}{|c|c|c|} \hline \emptyset & q_1 & 1 \\ \hline \emptyset & x & q_2 \\ \hline \end{array}$$
- b) 
$$\begin{array}{|c|c|c|} \hline 1 & 1 & \sqcup \\ \hline q_2 & 1 & \sqcup \\ \hline \end{array}$$
- c) 
$$\begin{array}{|c|c|c|} \hline q_3 & x & x \\ \hline \sqcup & x & x \\ \hline \end{array}$$
- d) 
$$\begin{array}{|c|c|c|} \hline x & x & q_3 \\ \hline x & x & \sqcup \\ \hline \end{array}$$



Die Regeln für das polynomielle Ausfüllrätsel besagen also, dass jedes  $3 \times 2$ -Fenster eines der von Turing-Maschinenübergängen erlaubten Fenster (Abbildung 13.4) oder ein konstantes Fenster (Abbildung 13.5) sein muss. Es gibt  $O(|\Gamma \cup Q|^6)$  solche Fenster, somit können die Regeln polynomiell durch Formeln ausgedrückt werden, die sich aus Gleichungen der Form  $z_{ik} = c$  aufbauen lassen, wobei  $z_{ik}$  das Zeichen ist, welches in das Feld in Zeile  $i$  und Spalte  $k$  eingetragen wird. Insbesondere ist die Größe der Formeln polynomiell

in der Länge  $n$  des Inputwortes. Damit ist gezeigt, dass sich das Finden einer akzeptierenden Berechnungsgeschichte auf ein polynomielles Ausfüllrätsel reduzieren lässt.

### 13.2.3 Reduktion auf SAT

Die Formeln  $\varphi_{ik}$ , die für jedes einzelne Feld  $(i, k)$  wahr werden müssen, müssen auch alle zusammen wahr werden. So entsteht eine Formel

$$\varphi = \bigwedge_{i,k} \varphi_{ik},$$

die genau dann erfüllbar ist, wenn  $w \in A$  ist. Falls die  $\varphi_{ik}$  in konjunktiver Normalform sind, dann ist auch  $\varphi$  in konjunktiver Normalform.

Das Problem *SAT* spricht von logischen Formeln, aber  $\varphi_{ik}$  ist eine Formel, die sich aus Termen der Form  $z_{uv} = c$  oder  $z_{uv} \neq c$  zusammensetzt, wobei  $c \in C = \Gamma \cup Q$ . Die Variablen  $z_{uv}$  müssen durch die logischen Variablen  $x_{uvc}$  ersetzt werden, die durch

$$x_{uvc} = (z_{uv} = c) \quad \text{und} \quad \bar{x}_{uvc} = (z_{uv} \neq c)$$

definiert sind. Damit können die Formeln durch logische Formeln in den Variablen  $x_{uvc}$  ausgedrückt werden.

Die Variablen  $x_{uvc}$  können nicht beliebig gewählt werden. Für ein Paar  $(u, v)$  darf immer nur genau eine der Variablen  $x_{uvc}$ ,  $c \in C$ , wahr sein. Wenn  $x_{uvc}$  wahr ist, müssen alle anderen falsch sein, es muss also

$$\psi_{uvc} = x_{uvc} \wedge \bigwedge_{d \in C \setminus \{c\}} \neg x_{uvd} \tag{13.1}$$

wahr sein.

Da das Feld  $(u, v)$  jeden Wert in  $C$  haben kann, muss eine der Formeln  $\psi_{uvc}$  von (13.1) wahr sein, es muss als

$$\psi_{uv} = \bigvee_{c \in C} \psi_{uvc} = \bigvee_{c \in C} \left( x_{uvc} \wedge \bigwedge_{d \in C \setminus \{c\}} \neg x_{uvd} \right)$$

wahr sein.

Die Größe der Formeln  $\psi_{uv}$  ist wieder polynomiell in  $n$ . Wenn man alle diese Formeln mit einer UND-Verknüpfung hinzunimmt, bleibt die Reduktion polynomiell. Falls die Formel bereits in konjunktiver Normalform war, sollten die hinzugefügten Formeln  $\psi_{uv}$  erst in konjunktive Normalform gebracht werden, um diese Eigenschaft zu erhalten. Damit ist der Satz von Cook-Levin bewiesen.

### 13.2.4 3SAT ist NP-vollständig

Die Reduktion einer beliebigen Sprache  $A$  auf *SAT* hat eine Formel ergeben, deren Größe polynomiell von  $n$  abhängt. Durch Äquivalenzumformung können wir diese Formeln sogar in konjunktive Normalform bringen, deren Länge ebenfalls polynomiell von  $n$  abhängen wird. Die Reduktion ist damit immer noch polynomiell in  $n$ . Wir können aber noch einen Schritt weiter gehen und sogar konjunktive Normalform mit genau drei Literalen pro Klausel erreichen. Dieses Problem ist bekannt als *3SAT*.

**Problem 13.6.** *Die Sprache*

$$3SAT = \left\{ \varphi \mid \begin{array}{l} \varphi \text{ ist eine erfüllbare logische Formel in konjunktiver} \\ \text{Normalform mit drei Literalen pro Klausel.} \end{array} \right\}$$

wird mit  $3SAT$  bezeichnet.

**Satz 13.7.**  $3SAT$  ist NP-vollständig.

*Beweis.* Wir müssen die Reduktion, die wir für  $SAT$  durchgeführt haben, noch einen Schritt weiter führen. Die Reduktion hat bis jetzt aus einem Wort  $w \in \Sigma^*$  eine Formel  $\varphi_0$  in konjunktiver Normalform geliefert, die genau dann erfüllbar ist, wenn  $w \in A$  war. Wir müssen jetzt  $\varphi_0$  in konjunktive Normalform mit drei Literalen pro Klauseln bringen, ohne an der Erfüllbarkeit etwas zu ändern.

Für Klauseln mit nur einem oder zwei Literalen ist dies nicht schwierig, wir wiederholen einfach ein Literal genügend oft:

$$(x_i) = (x_i \vee x_i \vee x_i) \quad \text{und} \quad (x_i \vee x_k) = (x_i \vee x_k \vee x_k).$$

Für Klauseln mit vier Literalen können wir eine zusätzliche logische Variable  $z$  hinzufügen und dann

$$(x_i \vee x_j \vee x_k \vee x_l) \quad \text{in} \quad (x_i \vee x_j \vee z) \wedge (\bar{z} \vee x_k \vee x_l) \quad (13.2)$$

umwandeln. Die beiden Formeln in (13.2) sind nicht äquivalent, aber sie sind wie man sagt *erfüllungsäquivalent*. Die linke Formel kann genau dann wahr gemacht werden, wenn die rechte wahr gemacht werden kann. Die linke Formel wird wahr, wenn mindestens einer der Literale wahr wird. Dieses wahre Literal kommt in einer der beiden Klammern der rechten Formel vor und macht sie wahr. Die andere Klammer ist möglicherweise noch nicht wahr, kann aber mit der Variable  $z$  oder  $\bar{z}$  wahr gemacht werden.

Für noch längere Klauseln kann der Trick von (13.2) wiederholt angewendet werden:

$$\begin{aligned} (x_{i_1} \vee \dots \vee x_{i_n}) &\mapsto (x_{i_1} \vee x_{i_2} \vee z_1) \\ &\quad \wedge (\bar{z}_1 \vee x_{i_3} \vee z_2) \\ &\quad \wedge (\bar{z}_2 \vee x_{i_4} \vee z_3) \\ &\quad \wedge \dots \\ &\quad \wedge (\bar{z}_{n-4} \vee x_{i_{n-2}} \vee z_{n-3}) \\ &\quad \wedge (\bar{z}_{n-3} \vee x_{i_{n-1}} \vee x_{i_n}). \end{aligned}$$

Die rechte Formel ist wieder erfüllungsäquivalent. Dies zeigt, dass sich jede Klausel einer Formel in konjunktiver Normalform erfüllungsäquivalent in konjunktive Normalform mit drei Literalen pro Klausel umformen lässt.  $\square$

*Verständniskontrolle 13.3:* Verwandeln Sie die Formel

$$(x_1 \vee x_2 \vee \bar{x}_3 \vee \bar{x}_4 \vee x_5) \wedge (x_6 \vee x_7 \vee \bar{x}_8 \vee x_9) \wedge (x_{10} \vee \bar{x}_{11}) \wedge x_{12}$$

in eine erfüllungsäquivalente Formel in konjunktiver Normalform mit drei Literalen pro Klausel.



13.3  
autosp.ch/v/13.3.pdf

## 13.3 Weitere NP-vollständige Probleme

*SAT* und *3SAT* sind nicht die einzigen NP-vollständigen Probleme. Im Gegenteil scheinen NP-vollständige Sprachen allgegenwärtig zu sein. Die Vorgehensweise, mit der weitere NP-vollständige Problem gefunden werden können, ist in Abschnitt 13.1 vorgezeichnet. Für jedes Problem muss eine Reduktionsabbildung von einem bereits bekannten NP-vollständigen Problem konstruiert werden. In diesem Abschnitt sollen ein paar interessante NP-vollständige Probleme genauer studiert werden.

### 13.3.1 3SAT und *k-CLIQUE*

Graphen sind besonders gut geeignet, abstrakte Beziehungen zwischen Objekten darzustellen. Zum Beispiel wurde in Abschnitt 12.1.4 gezeigt, wie sich das Stundenplanproblem auf das Färbeproblem für Graphen reduzieren lässt. Die Kanten des Graphen haben dort eine Ausschlussbedingung wiedergegeben, verbundene Knoten durften *nicht* die gleiche Farbe bekommen.

Auch wenn Kanten Gemeinsamkeiten zwischen Knoten ausdrücken, können interessante NP-vollständige Probleme entstehen. Der Begriff der *k*-Clique ist in Definition 12.7 und das *k*-Cliquenproblem in (12.1) bereits eingeführt worden. Ziel dieses Abschnitts ist zu zeigen, dass auch das Cliquenproblem NP-vollständig ist.

Da in einem vollständigen Graphen ohnehin jeder Knoten mit jedem anderen Knoten verbunden ist, ist in so einem Graphen das Cliquenproblem immer lösbar. In diesem Abschnitt soll gezeigt werden, dass es jedoch schwierig ist, eine große Clique in einem beliebigen, großen Graphen zu finden.

**Satz 13.8.** *k-CLIQUE* ist NP-vollständig.

*Beweis.* Es muss eine Reduktionsabbildung von einem bekannten NP-vollständigen Problem auf *k-CLIQUE* konstruiert werden. Da noch nicht viele solche Probleme zur Auswahl stehen, versuchen wir es mit *3SAT*. Es muss also zu einer Formel

$$\varphi = c_1 \wedge c_2 \wedge \cdots \wedge c_n$$

in konjunktiver Normalform mit drei Literalen pro Klausel ein Graph gefunden werden, der genau dann eine Clique einer bestimmten Größe hat, wenn die Formel  $\varphi$  wahr gemacht werden kann. Jede Klausel ist von der Form

$$c_l = x_{l1} \vee x_{l2} \vee x_{l3}, \quad \text{mit} \quad x_{li} = \begin{cases} x_j & \text{oder} \\ \bar{x}_j. \end{cases}$$

Jeder Term ist also eine der Variablen  $x_j$  oder der negierten Variablen  $\bar{x}_j$ .

Die Reduktionsabbildung ist für eine Beispieldformel in Abbildung 13.6 dargestellt. Die einzelnen Elemente der Formel werden wie folgt abgebildet:

**Literale und Klauseln:** Zu jedem Literal in einer Klausel konstruieren wir einen Knoten, zu jeder Klausel also die drei Knoten  $v_{k1}$ ,  $v_{k2}$  und  $v_{k3}$ , die wir mit dem jeweiligen Literal anschreiben, somit

$$c_k = (\textcolor{blue}{x_{k1}} \vee \textcolor{blue}{x_{k2}} \vee \textcolor{green}{x_{k3}}) \quad \mapsto \quad \textcolor{blue}{\circlearrowleft} \textcolor{blue}{x_{k1}} \quad \textcolor{blue}{\circlearrowleft} \textcolor{blue}{x_{k2}} \quad \textcolor{green}{\circlearrowleft} \textcolor{green}{x_{k3}} \quad (13.3)$$

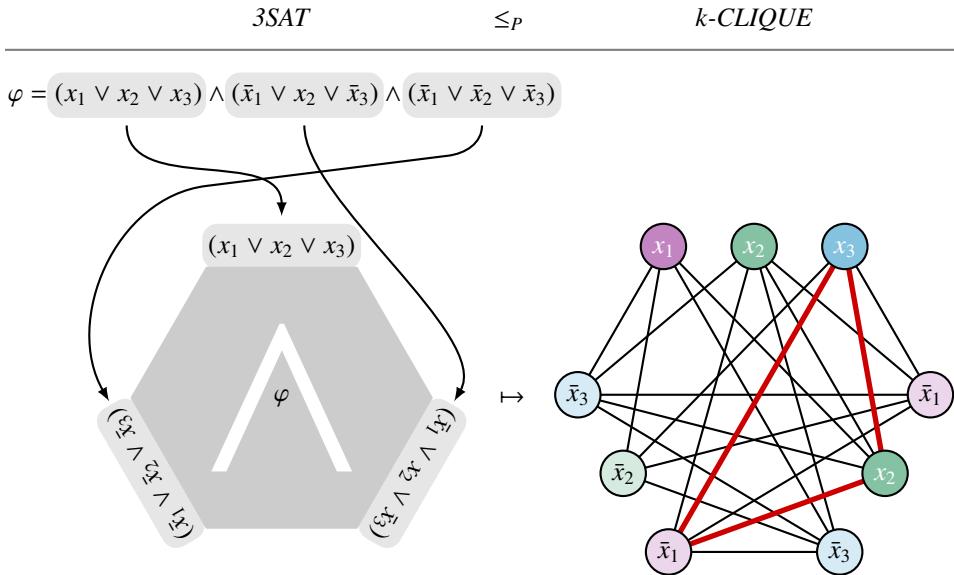


Abbildung 13.6: Reduktionsabbildung  $3SAT \leq_P k\text{-CLIQUE}$  zeigt, dass  $k\text{-CLIQUE}$  NP-vollständig ist. Jeder Klausel wird eine Dreiergruppe von Knoten zugeordnet, die einzelnen Knoten stehen für die Terme der Klausel. Zwei Knoten sind mit einer Kante verbunden, wenn sie gleichzeitig wahr sein können. Die Formel ist erfüllbar, wenn der Graph eine 3-Clique hat (rot).

**Kanten:** Der konstruierte Graph muss ausdrücken können, ob die Formel  $\varphi$  wahr werden kann. Dazu muss in jeder Klausel ein Literal wahr sein. Die Literale der verschiedenen Klauseln sind jedoch nicht unabhängig voneinander. Damit  $\varphi$  wahr wird, muss es gelingen, in jeder Klausel mindestens ein Literal wahr zu machen. Ob dies möglich ist kann man dadurch visualisieren, dass man diejenigen Knoten miteinander verbindet, die gleichzeitig wahr gemacht werden können. Die Knoten  $x_i$  und  $x_k$  können wie in



verbunden werden. Dabei wird angenommen, dass  $x_i \neq \bar{x}_k$  ist.  $x_i$  und  $\bar{x}_i$  bzw.  $x_k$  und  $\bar{x}_k$  sind nicht verbunden, da sie nicht gleichzeitig wahr sein können. Die Kanten des konstruierten Graphen verbinden also Knoten, die zu verschiedenen Klauseln gehören und gleichzeitig wahr gemacht werden können.

**Erfüllbarkeit:** Die Formel  $\varphi$  ist genau dann erfüllbar, wenn jede Klausel wahr gemacht werden kann, was gleichbedeutend damit ist, dass in jeder Klausel mindestens ein Literal wahr ist. Da alle diese Literale gleichzeitig wahr sein können, sind sie im konstruierten Graphen alle miteinander verbunden, sie bilden also eine Clique mit  $n$

Knoten.

**Die Zahl  $k$ :** Die Zahl  $k$ , die in der Problemstellung  $k$ -CLIQUE vorkommt, ist die Größe der Clique. Somit muss die Reduktion für  $k$  die Anzahl  $k = n$  der Klauseln gewählt werden.

Damit ist eine polynomielle Reduktion von 3SAT auf  $k$ -CLIQUE konstruiert und es folgt, dass  $k$ -CLIQUE NP-vollständig ist.  $\square$

### 13.3.2 Das Färbeproblem und das Stundenplanproblem

Das Graphenfärbeproblem wurde bereits in Beispiel 12.19 informell vorgestellt, hier die formelle Definition.

**Problem 13.9 ( $k$ -VERTEX-COLORING).** Gegeben ist ein Graph  $G = (V, E)$  mit Knoten  $V$  und Kanten  $E$  und eine Zahl  $k$ . Ist es möglich, die Knoten des Graphen mit  $k$  Farben so einzufärben, dass durch Kanten verbundene Knoten verschiedene Farben haben? Als Sprachproblem:

$$k\text{-VERTEX-COLORING} = \left\{ \langle G, k \rangle \mid \begin{array}{l} \text{Es gibt eine Einfärbung der Knoten des Graphen} \\ \text{mit } k \text{ Farben derart, dass zwei durch eine Kante} \\ \text{verbundenen Knoten verschiedene Farben haben.} \end{array} \right\}.$$

**Satz 13.10.**  $k$ -VERTEX-COLORING ist NP-vollständig.

Natürlich ist nicht jedes Färbeproblem schwierig zu entscheiden. Falls es mehr Farben als Knoten gibt, wenn also  $k \geq |V|$  ist, lässt sich immer eine Einfärbung finden. Für viele weitere spezielle Fälle wurden polynomielle Algorithmen gefunden, eine Übersicht findet man in [17].

#### Beweis der NP-Vollständigkeit von $k$ -VERTEX-COLORING

Die NP-Vollständigkeit von  $k$ -VERTEX-COLORING kann mit einer Reduktion von 3SAT gezeigt werden. Die folgende Konstruktion, schematisch dargestellt in Abbildung 13.7, ist inspiriert von [25], die dortige Konstruktion ist jedoch fehlerhaft.

Sei also

$$\varphi = c_1 \wedge \cdots \wedge c_m$$

eine Formel in konjunktiver Normalform. Die Klauseln  $c_1, \dots, c_m$  enthalten nur die Variablen  $x_1, \dots, x_n$  oder die negierten Variablen  $\bar{x}_1, \dots, \bar{x}_n$ . Wir konstruieren jetzt einen Graphen, der sich genau dann mit  $n + 1$  Farben einfärben lässt, wenn die Formel  $\varphi$  erfüllbar ist.

Wir beginnen damit, aus den Knoten  $v_0, v_1, \dots, v_n$  einen vollständigen Graphen zu konstruieren, jeder der Knoten  $v_i$  ist mit jedem anderen verbunden. Dieser Teilgraph lässt sich nur einfärben, wenn jeder Knoten eine eigene Farbe erhält. Wir wählen für den Knoten  $v_0$  die Farbe schwarz, die Farben der Knoten  $v_1, \dots, v_n$  nennen wir farbig.

Jetzt konstruieren wir für jede logische Variable  $x_i$  zwei Knoten, die wir mit  $x_i$  und  $\bar{x}_i$  bezeichnen. Wir verbinden sowohl  $x_i$  als auch  $\bar{x}_i$  mit allen Knoten  $v_k$  mit  $k = 1, \dots, n$ .

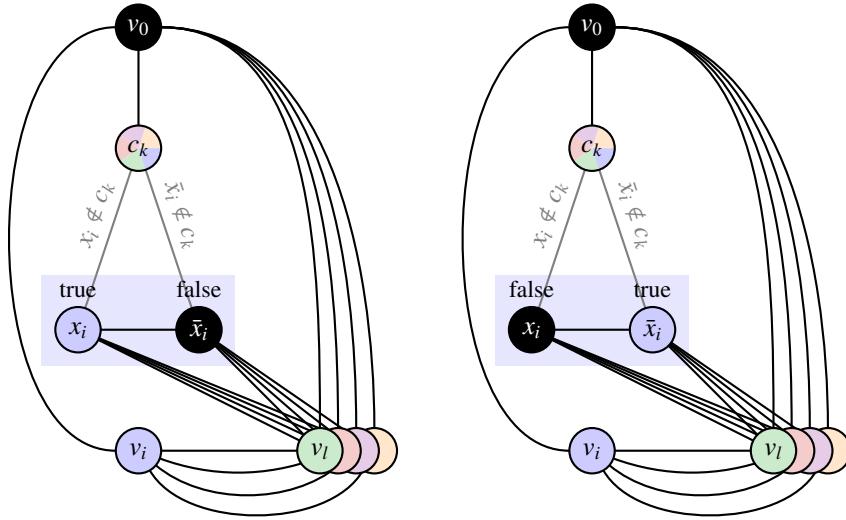


Abbildung 13.7: Konstruktion der polynomiellen Reduktion  $3SAT \leq_P k\text{-VERTEX-COLORING}$ . Die Knoten  $v_i, v_l, \dots$  bilden einen vollständigen Graphen und bekommen daher verschiedene Farben. Die Farbe  $v_0$  ist als schwarz definiert und bedeutet den logischen Wert *false*. Die Knoten  $x_i$  und  $\bar{x}_i$  sind mit allen  $v_l$  verbunden außer mit  $v_i$  und  $v_0$ , sie können daher nur die Farbe von  $v_i$  oder die Farbe schwarz annehmen. Es sind daher nur die beiden Färbungen links und rechts möglich, die für  $x_i = \text{true}$  bzw.  $\bar{x}_i = \text{true}$  stehen.

und  $i \neq k$ . Die Knoten werden also nur mit  $v_0$  und  $v_i$  nicht verbunden. Eine Lösung des Färbeproblems gibt also jeweils einem der beiden Knoten  $x_i$  und  $\bar{x}_i$  die Farbe von  $v_i$ , der andere wird schwarz (Abbildung 13.8).

Die Farben, die die Knoten  $x_i$  erhalten, interpretieren wir als logische Werte der Variablen  $x_i$ . Wird  $x_i$  farbig, also mit der gleichen Farbe wie  $v_i$ , dann soll dies bedeuten, dass  $x_i$  wahr wird. Wird  $x_i$  schwarz, bedeutet dies, dass  $x_i$  falsch ist und  $\bar{x}_i$  wahr.

Jetzt muss noch für jede Klausel ein Knoten konstruiert werden, der mit  $c_k$  bezeichnet wird. Damit die Formel  $\varphi$  wahr wird, muss jede Klausel wahr werden. Da Farbe mit wahr gleichgesetzt werden soll, müssen alle Knoten  $c_k$  mit  $v_0$  verbunden werden. Eine Färbung des Graphen ist nur möglich, wenn den Knoten  $c_k$  Farben gegeben werden können, wobei schwarz ausgeschlossen ist.

Die Farbe, die der Knoten  $c_k$  bekommen kann, soll die Farbe einer Variablen  $x_i$  oder  $\bar{x}_i$  sein, die den Knoten wahr machen kann. Kommt  $x_i$  in  $c_k$  vor, dann kann  $x_i$  die Klausel wahr machen. Die Kanten dürfen also nicht verhindern, dass  $c_k$  die Farbe von  $x_i$  bekommen kann.  $c_k$  darf also nur genau dann mit  $x_i$  verbunden werden, wenn  $x_i$  nicht in  $c_k$  vorkommt. Kommt  $\bar{x}_i$  in  $c_k$  vor, dann kann  $\bar{x}_i$  die Klausel wahr machen. Die Kanten dürfen also nicht verhindern, dass  $c_k$  die Farbe von  $\bar{x}_i$  bekommen kann.  $c_k$  darf also nur genau dann mit  $\bar{x}_i$  verbunden werden, wenn  $\bar{x}_i$  nicht in  $c_k$  vorkommt. Die Konstruktion ist in Abbildung 13.8 für eine Beispielformel durchgeführt.

Die Konstruktion der Verbindungen der  $c_k$  mit den  $x_i$  und  $\bar{x}_i$  stellt sicher, dass  $c_k$  die Farbe von  $v_i$  annehmen kann, wenn  $x_i$  oder  $\bar{x}_i$  die Klausel wahr machen. Ist die Formel

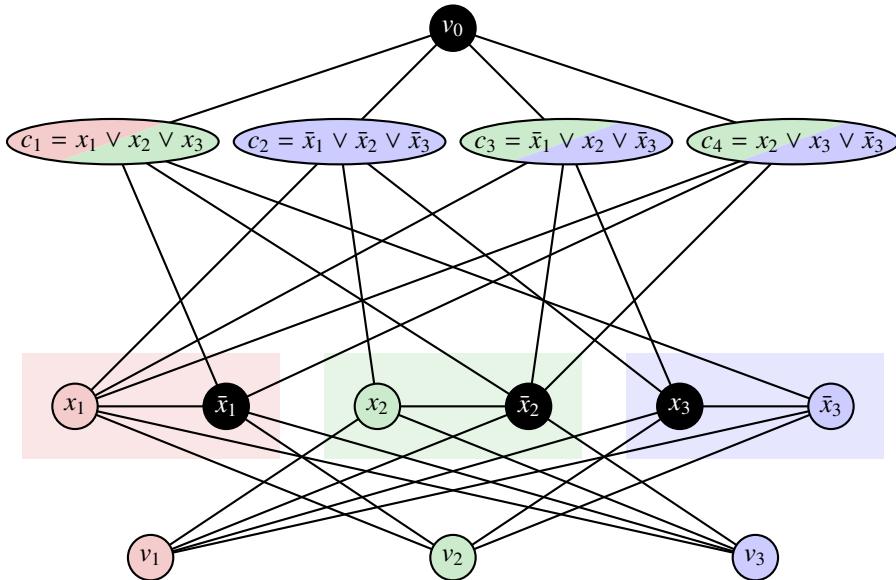


Abbildung 13.8: Polynomielle Reduktion der Formel  $\varphi = c_1 \wedge c_2 \wedge c_3 \wedge c_4$  auf ein 4-VERTEX-COLORING-Problem im Beweis von Satz 13.10. Die Knoten  $v_0, \dots, v_3$  bilden einen vollständigen Graphen, diese Verbindungen sind nicht eingezeichnet. Sie erzwingen, dass die Knoten  $v_i$  verschiedene Farben haben müssen. Die Verbindungen zu den  $x_k$  und  $\bar{x}_k$  erzwingen, dass in jedem farbigen Rechteck in Knoten die Farbe des Rechtecks haben muss und einer schwarz sein muss. Die Formel ist genau dann erfüllbar, wenn die Knoten für die Klauseln nicht schwarz sein können. In den Ellipsen sind alle Farben bezeugt, die für diesen Knoten möglich sind.

$\varphi$  erfüllbar, dann ist das Färbeproblem lösbar. Man wählt dazu für die Knoten  $c_k$  eine der Farben der Variablen, die  $c_k$  wahr machen können.

Hat umgekehrt das Färbeproblem eine Lösung, dann hat in jedem Paar  $x_i$  und  $\bar{x}_i$  einer der Knoten die Farbe von  $v_i$ , der andere ist schwarz. Ist  $x_i$  farbig, muss die logische Variable  $x_i$  wahr gewählt werden, andernfalls falsch. Mit dieser Wahl der Wahrheitswerte wird  $\varphi$  wahr und  $\varphi$  ist erfüllbar. Damit ist die Reduktion vollständig und der Satz 13.10 bewiesen.

## Das Stundenplanproblem

Das in Abschnitt 12.1.4 informell eingeführte Stundenplanproblem ist das folgende Entscheidungsproblem.

**Problem 13.11 (STUNDENPLAN).** Sei  $F$  eine Menge von Fächern und  $A$  eine Menge von zweielementigen Mengen  $\{f_1, f_2\}$  von Fächern, die Anmeldungen von Studierenden auf die beiden verschiedenen Fächer  $f_1$  und  $f_2$  darstellen. Das Stundenplanproblem ist auch das

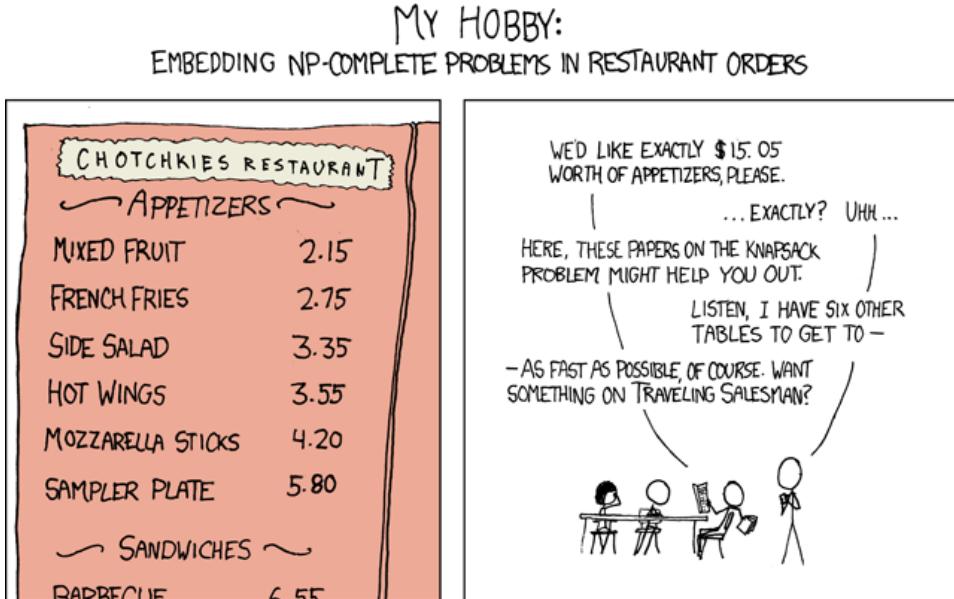


Abbildung 13.9: Der XKCD-Comic #287 vom 9. Juli 2007 illustriert das *SUBSET-SUM* Problem als Teil einer Restaurantbestellung [35]. Statt Produkte von der Speisekarte zu bestellen, gibt der Guest die Summe vor, die die Produkte kosten sollen, und überlässt es der Servierpersonen, das zugehörige *SUBSET-SUM*-Problem zu lösen. © 2007 xkcd.com

### Sprachproblem

$$STUNDENPLAN = \left\{ \langle F, A, k \rangle \mid \begin{array}{l} \text{Es gibt eine Planung der Fächer } f \in F \text{ auf } k \\ \text{Zeitfenster derart, dass es keine Anmeldung} \\ \{f_1, f_2\} \in A \text{ gibt derart, dass } f_1 \text{ und } f_2 \text{ auf das} \\ \text{gleiche Zeitfenster geplant sind.} \end{array} \right\}.$$

In dieser Formulierung des Stundenplanproblems wird klar, dass die Abbildung

$$STUNDENPLAN \leq_P k\text{-VERTEX-COLORING} : \langle F, A, k \rangle \mapsto \underbrace{\langle F, A \rangle}_{= G}, k$$

eine polynomielle Äquivalenz des Stundenplanproblems und des Färbeproblems ist. Da das Färbeproblem als NP-vollständig erkannt wurde, gilt dasselbe auch für das Stundenplanproblem.

### 13.3.3 SUBSET-SUM und Kryptographie

Der XKCD-Comic in Abbildung 13.9 illustriert das Rucksackproblem oder *SUBSET-SUM*. Es ist auch eine wesentliche Komponente des Hellman-Merkle-Kryptosystems, welches seine Sicherheit daraus bezieht, dass das Rucksackproblem NP-vollständig ist. In diesem

Abschnitt soll dies gezeigt werden. Etwas formaler ist *SUBSET-SUM* das folgende Problem.

**Problem 13.12 (*SUBSET-SUM*).** Gegeben ist eine Liste

$$S = [s_i \mid 1 \leq i \leq n]$$

von natürlichen Zahlen  $s_i \in \mathbb{N}$  und eine natürliche Zahl  $t \in \mathbb{N}$ . Sie heißt auch der Rucksack. Ist es möglich, aus  $S$  eine Teilliste  $T \subset S$  auszuwählen derart, dass die Elemente von  $T$  die Summe

$$t = \sum_{s \in T} s$$

haben?

Man beachte, dass  $S$  nicht eine Menge ist, dass also Zahlen mehrfach in der Liste vorkommen können. Dies führt dazu, dass gewisse Rucksackprobleme sehr einfach lösbar sind.

*Beispiel 13.13.* Das Rucksackproblem für den Rucksack

$$S = [\underbrace{1, 1, \dots, 1}_n]$$

ist für die Summe  $t$  genau dann lösbar, wenn  $t \leq n$  ist.  $\circlearrowright$

*Beispiel 13.14.* Das Rucksackproblem für den Rucksack

$$S = [1, 2, 4, 8, \dots, 2^k, \dots, 2^{n-1}]$$

ist für die Summe  $t$  genau dann lösbar, wenn  $t \leq 2^n - 1$  ist.

Da die Summe der Zahlen von  $S$

$$\sum_{x \in S} x = 1 + 2 + \dots + 2^{n-1} = \sum_{k=0}^{n-1} 2^k = 2^n - 1$$

ist, ist ein Rucksackproblem für eine größere Summe nicht lösbar. Ist  $t \leq 2^n - 1$ , dann kann  $t$  im Zweiersystem als

$$t = b_{n-1}b_{n-2}\dots b_2b_1b_0 = \sum_{k=0}^{n-1} b_k 2^k$$

mit  $b_k \in \{0, 1\}$  geschrieben werden. Dies bedeutet, dass

$$T = [2^k \mid b_k = 1] \subset S$$

das Rucksackproblem löst.  $\circlearrowright$

**Satz 13.15.** Sei  $b \in \mathbb{N}$ ,  $b \geq 2$ . Das Rucksackproblem für den Rucksack

$$S = [\underbrace{1, \dots, 1}_{b-1}, \underbrace{b, \dots, b}_{b-1}, \underbrace{b^2, \dots, b^2}_{b-1}, \dots, \underbrace{b^{n-1}, \dots, b^{n-1}}_{b-1}]$$

und die Summe  $t$  ist genau dann lösbar ist, wenn  $t \leq b^n - 1$ .

Es gibt also durchaus Rucksäcke mit speziellen Eigenschaften, in denen sich ein Rucksackproblem sehr effizient entscheiden lässt. NP-Vollständigkeit besagt nur, dass es keinen für beliebige Rucksäcke funktionierenden polynomiellen Lösungsalgorithmus gibt.

### Reduktion von 3SAT

Um zu zeigen, dass *SUBSET-SUM* NP-vollständig ist, muss eine Reduktionsabbildung von einem bekannteren NP-vollständigen Problem auf *SUBSET-SUM* konstruiert werden. Die folgende Reduktion beginnt mit *3SAT*, d. h. es muss zu jeder logischen Formel

$$\varphi = c_1 \wedge c_2 \wedge \dots \wedge c_n = (x_{i_1} \vee x_{i_2} \vee x_{i_3}) \wedge (x_{i_4} \vee x_{i_5} \vee x_{i_6}) \wedge \dots \wedge (x_{i_{3n-2}} \vee x_{i_{3n-1}} \vee x_{i_{3n}})$$

in konjunktiver Normalform mit genau drei Literalen pro Klausel ein Rucksackproblem konstruiert werden. Wir illustrieren die Vorgehensweise an der Formel

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3). \quad (13.5)$$

Aus  $\varphi$  muss jetzt eine Liste von Zahlen und eine Summe konstruiert werden, so dass die Formel  $\varphi$  genau dann erfüllbar ist, wenn das Rucksackproblem lösbar ist.

**Variablen und Klauseln auseinanderhalten:** Die logische Formel unterscheidet die logischen Variablen  $x_i$  und die Klauseln  $c_j$ . Das Rucksackproblem spricht nur von Zahlen, die addiert werden können. Die Reduktion muss also sicherstellen, dass sich die Bedingungen an die Variablen und Klauseln immer noch in den Zahlen erkennen lassen. Dazu eignet sich die Darstellung der Zahlen im Zehnersystem. Wir reservieren die vordersten  $m$  Stellen für Variablen und die hinteren  $n$  Stellen für die Klauseln:

$x_1$	$x_2$	$\dots$	$x_m$	$c_1$	$c_2$	$\dots$	$c_n$
-------	-------	---------	-------	-------	-------	---------	-------

(13.6)

Für die Formel (13.5) werden also  $3+3=6$ -stellige Zahlen erzeugt, deren erste drei Stellen für die Variablen stehen und die letzten drei für die Klauseln.

**Logische Variablen:** In *3SAT* müssen logische Variablen als wahr oder falsch gewählt werden, während im Rucksackproblem Zahlen aus einem Rucksack ausgewählt werden. Daher muss die Reduktion jeder logischen Variable Zahlen eines Rucksacks zuordnen. Da in den Klauseln auch negierte Variablen vorkommen können, werden für jede logische Variable  $x_i$ ,  $i = 1, \dots, m$ , zwei Zahlen  $y_i$  und  $z_i$  erzeugt, die an der Stelle  $x_i$  eine 1 haben und an allen anderen Stellen Nullen:

	$x_1$	$\dots$	$x_i$	$\dots$	$x_m$	$c_1$	$\dots$	$c_n$	
$y_i =$	0	...	1	...	0				$\rightarrow x_i$ wahr
$z_i =$	0	...	1	...	0				$\rightarrow x_i$ falsch.

(13.7)

Die Stellen für die Klauseln (grau) werden später festgelegt. Ist  $x_i$  wahr, soll dies bedeuten, dass  $y_i$  aus dem Rucksack gewählt werden muss, andernfalls soll  $z_i$  gewählt werden müssen. Damit genau eine von den Variablen gewählt wird, muss die Summe  $t$  geeignet gewählt werden. Sie muss an der entsprechenden Stelle eine 1 haben (siehe Fußzeile von Tabelle 13.1).

**Klauseln:** Eine Klausel wird wahr, wenn einer der Literale wahr ist. Wenn  $x_i$  in der Klausel  $c_k$  vorkommt, dann wird die Stelle von  $c_k$  von  $y_i$  auf 1 gesetzt. Wenn  $\bar{x}_i$  in der

Klausel  $c_k$  vorkommt, wird die entsprechende Stelle von  $z_i$  auf 1 gesetzt. Die Klausel  $c_k$  wird also genau dann wahr, wenn in der Summe der gewählten Zahlen  $y_i$  und  $z_i$  in der Spalte von  $c_k$  keine 0 steht:

	$x_{i_1}$	$\dots$	$x_{i_2}$	$\dots$	$x_{i_3}$	$c_k = (x_{i_1} \vee \bar{x}_{i_2} \vee x_{i_3})$	
$y_{i_1} =$	1		0		0	1 0	$i_1$
$z_{i_1} =$	1		0		0		
$\vdots$							
$y_{i_2} =$	0		1		0	0 1	$i_2$
$z_{i_2} =$	0		1		0		
$\vdots$							
$y_{i_3} =$	0		0		1	1 0	$i_3$
$z_{i_3} =$	0		0		1		

**Füllvariablen:** Wenn die Formel erfüllt ist, dann wird die Summe der aus  $y_i$  und  $z_i$  ausgewählten Zahlen an den Stellen  $c_1, \dots, c_n$  mindestens eine 1 und höchstens 3 (weil die Klausel nur 3 Literale hat). Ein Rucksackproblem gibt aber einen festen Wert vor, der erreicht werden muss. Um diese Variationsbreite auszugleichen, fügen wir noch für jede Klausel zwei Zahlen

	$x_1$	$\dots$	$x_m$	$c_1$	$\dots$	$c_k$	$\dots$	$c_n$	
$g_k =$	0	$\dots$	0	0	$\dots$	1	$\dots$	0	
$h_k =$	0	$\dots$	0	0	$\dots$	1	$\dots$	0	

hinzu. Mit diesen Zahlen lässt sich an jeder Stelle einer Klausel, wo die Summe der ausgewählten  $y_i$  und  $z_i$  bereits mindestens den Wert 1 erreicht, die Summe zu 3 ergänzen (siehe Fußzeile von Tabelle 13.1).

**Summe:** Die zu erreichende Summe muss 1 haben an den Stellen, die zu den Variablen  $x_i$  gehören, und 3 an den Stellen, die für Klauseln  $c_k$  stehen:

$$t = \underbrace{1 \dots 1}_m \underbrace{3 \dots 3}_n .$$

Die vollständige Übersetzung der Beispielformel 13.5 in das Rucksackproblem mit dem Rucksack

$$S = [y_i, z_i, g_k, h_k \mid 1 \leq i \leq m \wedge 1 \leq k \leq n]$$

ist in Tabelle 13.1 zusammengestellt.

Damit ist die Reduktion  $3SAT \leq_P SUBSET-SUM$  vollständig und wir haben den folgenden Satz bewiesen.

**Satz 13.16.** Das Problem 13.12 SUBSET-SUM ist NP-vollständig.

Zahl	$x_1$	$x_2$	$x_3$	$c_1$	$c_2$	$c_3$
$y_1$	1	0	0	1	0	0
$z_1$	1	0	0	0	1	1
$y_2$	0	1	0	1	1	0
$z_2$	0	1	0	0	0	1
$y_3$	0	0	1	1	0	0
$z_3$	0	0	1	0	1	1
$g_1$	0	0	0	1	0	0
$h_1$	0	0	0	1	0	0
$g_2$	0	0	0	0	1	0
$h_2$	0	0	0	0	1	0
$g_3$	0	0	0	0	0	1
$h_3$	0	0	0	0	0	1
$t$	1	1	1	3	3	3

Tabelle 13.1: Rucksackproblem für die Beispielformel (13.5).

*Verständniskontrolle 13.4:* Konstruieren Sie die Zahlen  $y_i, z_i, g_i, h_i$  und  $t$ , die sich bei der Reduktion des 3SAT-Problems für die Formel

$$\varphi = (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \quad (13.10)$$

auf ein SUBSET-SUM-Problem ergibt. Lösen Sie anschließend das SUBSET-SUM-Problem und leiten Sie aus der Lösung eine Belegung der Variablen  $x_i$  mit Wahrheitswerten ab, die die Formel  $\varphi$  erfüllen.



autospqr.ch/v/13.4.pdf

### Ein einfaches Rucksackproblem

Nicht jedes Rucksackproblem ist schwierig. Die Liste

$$A = [1, 2, 4, 9, 20, 38] \quad (13.11)$$

hat die besondere Eigenschaft, dass jede Zahl größer ist als die Summe aller kleineren Zahlen. Für einen solchen Rucksack lässt sich das Rucksackproblem sehr effizient entscheiden.

**Definition 13.17.** Ein stark wachsender Rucksack ist eine Liste  $[a_1, \dots, a_n]$  von natürlichen Zahlen derart, dass jeder Wert größer ist als die Summe

$$\sum_{a_i < a_k} a_i < a_k$$

aller kleineren Werte.

**Beispiel 13.18.** Als Beispiel nehmen wir als Summe den Wert  $t = 26$  an. Er soll durch Summanden aus dem Rucksack (13.11) erreicht werden. Der Summand 38 ist offensichtlich zu groß. Der nächstkleinere Summand 20 kann nicht weggelassen werden, denn die

Summe der kleineren Summanden ergibt nur 16. Die Summe  $t = 26$  lässt sich ohne den Summanden 20 nicht mehr erreichen. Wenn das Problem lösbar ist, dann nur unter Verwendung des Summanden 20. Damit bleibt jetzt noch die Summe  $t_1 = t - 20 = 6$  mit dem kleineren Rucksack  $A_1 = [1, 2, 4, 9]$ . Dafür ist  $t_1 = 6 = 2 + 4$  eine Lösung.  $\circ$

Mit einem stark wachsenden Rucksack lässt sich die Vorgehensweise des Beispiels ganz allgemein durchführen.

**Satz 13.19.** *Das Rucksackproblem für einen stark wachsenden Rucksack mit  $n$  Elementen lässt sich in Zeit  $O(n)$  lösen.*

*Beweis.* Der folgende geizige Algorithmus entscheidet das Rucksackproblem.

1. Sei  $A_0 = A$  der Startrucksack und  $t_0 = t$  die zu erreichende Summe.
2. Setze  $k = 0$ .
3. Falls  $t_k = 0$  ist, ist das Rucksackproblem gelöst und der Algorithmus endet.
4. Falls  $A_k = \emptyset$  stehen keine weiteren Summanden mehr zur Verfügung, der Algorithmus endet, das Rucksackproblem ist nicht lösbar.
5. Finde den größten Summanden  $a_{i_k} \in A_k$  derart, dass  $a_{i_k} \leq t_k$  ist.
6. Bilde den neuen Rucksack

$$A_{k+1} = [a_i \in A_k \mid a_i < a_{i_k}]$$

und die neue Summe

$$t_{k+1} = t_k - a_{i_k}.$$

7. Inkrementiere  $k$  und fahre weiter bei 3.

Im Schritt 5 des Algorithmus wird das nächste zu verwendende Element des Rucksacks bestimmt. Da der Rucksack stark wachsend ist, muss das Element  $a_{i_k}$  verwendet werden, da andernfalls die Summe nicht erreicht werden kann. Im Schritt 6 wird das Rucksackproblem dann auf einen kleineren Rucksack und für die verbleibende Restsumme reduziert.

Der Algorithmus braucht höchstens so viele Iterationen, wie der Rucksack Elemente hat, also ist die Laufzeit  $O(n)$ . Der Algorithmus lässt sich iterativ oder rekursiv implementieren.  $\square$

## Kryptographie mit dem Rucksackproblem

Ein einfaches Rucksackproblem mit einem schnell wachsenden Rucksack kann durch einen Trick in ein schwieriges Rucksackproblem verwandelt werden. Wenn dies gelingt, kann man versuchen, geheim zu haltende Daten als ein Rucksackproblem, also als einen Rucksack und eine Zahl zu codieren. Die Entschlüsselung läuft dann darauf hinaus, das Rucksackproblem zu lösen. Die Transformation muss also die algebraischen Eigenschaften des Rucksackproblems erhalten.

Die Elemente eines Rucksacks  $A$  können als Vektor  $a$  mit den Komponenten  $a_1, \dots, a_n$  betrachtet werden. Die Auswahl kann durch einen binären Vektor dargestellt werden, der für jedes gewählte Element eine 1 und für jedes nicht gewählten Element eine 0 enthält. Die Elemente 2, 4 und 20 aus dem Rucksack (13.11) können durch den Vektor

$$d = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

gewählt werden. Die Berechnung der Summe der gewählten Elemente ist dann einfach das Skalarprodukt

$$t = 0 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 + 0 \cdot 9 + 1 \cdot 20 + 0 \cdot 38 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 4 \\ 9 \\ 20 \\ 38 \end{pmatrix} = \sum_{i=1}^n d_i a_i = d \cdot a$$

mit dem Wert  $t = 26$ . Der Rucksack  $a$  verwandelt also einen binären Vektor in eine Summe. Da der Rucksack schnell wachsend ist, ist das Umkehrproblem ebenfalls effizient lösbar.

Wir suchen nach einer Transformation des Rucksackproblems, die die algebraischen Eigenschaften nicht zerstört, aber aus dem einfachen Problem mit dem schnell wachsenden Rucksack ein schwieriges Problem mit einem allgemeinen Rucksack macht. Damit die linearen Eigenschaften nicht zerstört werden, müssen wir nach einer linearen Transformation suchen, die sich aber mit dem Skalarprodukt mit dem Datenvektor  $d$  verträgt. Vielfache der Einheitsmatrix sind die einzigen Transformationsmatrizen, die diese Bedingung erfüllen.

Wir wählen also einen Multiplikator  $m \in \mathbb{N}$  und erzeugen aus dem Rucksack  $A$  einen neuen Rucksack

$$B = [ma_1, \dots, ma_n] \quad \text{oder vektoriell} \quad b = ma.$$

Das Skalarprodukt mit  $d$  wird jetzt

$$c = d \cdot b = d \cdot (ma) = m(d \cdot a) = mt.$$

Es entsteht also ein Rucksackproblem für den Rucksack  $B$  mit der Summe  $mt$ .

Ist  $A$  ein stark wachsender Rucksack, dann sind auch die Zahlen  $ma_i$  stark wachsend, das Rucksackproblem mit dem Rucksack  $B$  und der Summe  $mt$  ist also genau gleich einfach zu lösen. Ursache dafür ist die Tatsache, dass in den ganzen Zahlen die Ordnungsrelation mit den arithmetischen Operationen verträglich ist. Die Ordnungseigenschaft geht in einem endlichen Körper verloren. Wir verwenden daher zusätzlich eine Primzahl  $p$  und

führen alle Operationen nicht in  $\mathbb{Z}$ , sondern im Körper  $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$  der Reste modulo  $p$  aus<sup>1</sup>.

Damit alle möglichen Summen von Elementen des Rucksacks in  $\mathbb{F}_p$  unterscheidbar bleiben, muss  $p$  größer sein als die Summe

$$\sum_{i=1}^n a_i < p$$

aller Elemente des Rucksacks. Für  $p = 107$  und den Multiplikator  $m = 31$  erhalten wir zum Beispiel den neuen Rucksack

$$B = [31, 62, 17, 65, 85, 1].$$

Die besondere Wachstumseigenschaft des Rucksacks  $A$  ist verschwunden.

Die Multiplikationsoperation mit  $m$  muss rückgängig gemacht werden können. Dazu wird durch  $m$  dividiert, was in  $\mathbb{F}_p$  genau dann möglich ist, wenn  $m \in \mathbb{F}_p$  oder  $0 < m < p$  ist. Sei also  $r$  die multiplikative Inverse von  $m$  in  $\mathbb{F}_p$ . Für  $p = 107$  und  $m = 31$  ist  $r = 38$ , da  $31 \cdot 38 = 1178 = 11 \cdot 107 + 1$ .

Das Rucksackproblem mit dem Rucksack  $A$  und der Summe  $s$  wird durch Multiplikation mit  $m$  modulo  $p = 107$  zu einem Rucksackproblem mit dem Rucksack  $B$  und der Summe  $c = 57$ , das jetzt jedoch in  $\mathbb{F}_{107}$  gelöst werden muss, nicht in  $\mathbb{Z}$ . Da der Rucksack  $B$  keine besonderen Eigenschaften mehr hat, die die Lösung des Rucksackproblems vereinfachen würden, ist dies ein schwieriges Problem. Selbst wenn man den Rucksack  $B$  und die Summe  $c$  öffentlich bekannt gibt, wird es ohne zusätzliches Wissen schwierig, den Vektor  $d$  zu rekonstruieren.

Für jemanden, der außer  $B$  und  $c$  auch den Multiplikator  $m$  und den Modulus  $p$  kennt, ist es jedoch einfach. Er bestimmt zunächst die multiplikative Inverse  $r$  von  $m$  und multipliziert dann  $B$  und  $c$  mit  $r$  in  $\mathbb{F}_p$ . Es ergibt sich wieder der stark wachsende Rucksack  $A$  und die Summe  $mc = s$ . Der Vektor  $d$  kann mit dem geizigen Algorithmus von Satz 13.19 effizient gefunden werden.

Es ist nicht unbedingt nötig, dass  $p$  eine Primzahl ist, solange die Multiplikation mit  $m$  modulo  $p$  immer noch invertierbar ist. Für Paare  $(m, p)$  von teilerfremden Zahlen ist dies immer der Fall.

**Satz 13.20** (Merkle-Hellman). *Sei  $A = [a_i \mid 1 \leq i \leq n]$  ein stark wachsender Rucksack,  $p$  ein Modulus und  $m$  ein Multiplikator mit*

$$p > \sum_{i=1}^n a_i$$

*und  $m < p$  teilerfremd zu  $p$ .*

*Der öffentliche Schlüssel ist der Rucksack*

$$B = [ma_i \in \mathbb{Z}/p\mathbb{Z} \mid 1 \leq i \leq n].$$

*Ein  $n$ -dimensionaler Bitvektor  $d$  wird als das Skalarprodukt  $c = d \cdot b \in \mathbb{Z}/p\mathbb{Z}$  verschlüsselt.*

---

<sup>1</sup>Eine Einführung in die Arithmetik modulo einer Primzahl kann in [38] gefunden werden.

*Der private Schlüssel besteht aus den Zahlen  $m$  und  $p$ . Die Entschlüsselung erfolgt durch Multiplikation von  $c$  mit  $r$ , der Vektor  $d$  ist die Lösung des Rucksackproblems  $d \cdot a = rc$  mit dem Rucksack  $A$ .*

*Beispiel 13.21.* Für das Beispiel war  $m = 31$  und  $p = 107$  gewählt worden, es ergab sich die Summe  $c = 57$ , für die das Rucksackproblem nicht einfach zu lösen ist. Multipliziert man aber mit  $r = 38$ , ergibt sich wieder

$$t = rc = 38 \cdot 57 = 2166 \equiv 26 \pmod{107}.$$

Damit ist das ursprüngliche Rucksackproblem für den Rucksack  $A$  und die Summe  $t = 26$  wiedergewonnen, welches effizient lösbar ist.  $\circ$

Dieses Verfahren wurde 1978 von Ralph Merkle und Martin Hellman vorgeschlagen [31]. Seine Sicherheit beruht auf der Vermutung, dass das durch modulare Multiplikation erzeugte Rucksackproblem allgemein genug ist, dass es nicht in polynomieller Zeit gelöst werden kann. Dies ist leider nicht wahr, wie Adi Shamir 1983 in [53] gezeigt hat. Das Verfahren illustriert zwar schön den Grundsatz, dass ein NP-vollständiges Problem dazu genutzt werden kann, kryptographische Sicherheit durch Laufzeitkomplexität zu erreichen, kann aber heutzutage nicht mehr als sicher gelten.

### 13.3.4 CIRCUIT und MINESWEEPER-CONSISTENCY

Im Spiel Minesweeper (siehe auch Abbildung 13.10) kann der Spieler beliebige Felder des Spielfeldes betreten. Unter den Feldern können sich allerdings Bomben verstecken. Betritt der Spieler eine Bombe, ist das Spiel zu Ende. Betritt er ein Feld ohne Bombe, wird ihm mit farbigen Zahlen angezeigt, unter wie vielen Nachbarfeldern sich Bomben befinden. Daraus kann der Spieler Schlüsse ziehen, wo sich Bomben befinden. Viele Implementierungen des Spieles ermöglichen ihm, solche Felder zu markieren, damit er sie nicht versehentlich betrifft.

#### Das NP-Problem MINESWEEPER-CONSISTENCY

Das Minesweeper-Spiel ist natürlich kein Entscheidungsproblem und damit der Behandlung durch die in diesem Kapitel entwickelte Komplexitätstheorie noch nicht zugänglich. Die für den Spieler spannende Frage ist ja, wie die Bombenverteilung zu den bekannten Zahlen aussieht. Es ist aber zunächst nicht klar, ob es zu gegebenen Zahlen auch eine Bombenverteilung gibt, die diese Zahlen hervorbringt. Die Frage, ob es eine mit den Zahlen konsistente Bombenverteilung gibt, ist ein Entscheidungsproblem.

**Problem 13.22 (MINESWEEPER-CONSISTENCY).** *Gibt es zu einer Zahlenverteilung auf den aufgedeckten Feldern eines Minesweeper-Spielfeldes eine Bombenverteilung auf den nicht aufgedeckten Feldern, so dass die Zahl auf einem Feld die Anzahl der Bomben unter den Nachbarfeldern ergibt?*

Der Reiz des Spieles kommt daher, dass die Antwort dieser Entscheidungsfrage nicht ganz einfach ist. Es ist zwar sicher ein NP-Problem, denn das Finden einer konsistenten Bombenverteilung ist ein polynomielles Ausfüllrätsel. Könnte das Problem sogar NP-vollständig sein? Die Fernsehserie Numb3rs hat diese Frage zum Thema der vierten Folge



Abbildung 13.10: Im Spiel *Minesweeper* erfährt der Spieler aus den angezeigten farbigen Zahlen, unter wie vielen der Nachbarfelder sich Bomben befinden. Daraus soll er erschließen, wo sich Bomben befinden, und weitere ungefährliche Felder betreten und damit neue Information sichtbar machen. Tritt er auf einer Bombe, wie im vorliegenden Fall auf das orange Feld in Zeile 5 und Spalte 10, ist das Spiel zu Ende. Tritt er auf ein Feld ohne Bombe, werden die Zahlen auch für dieses neue Feld angezeigt.

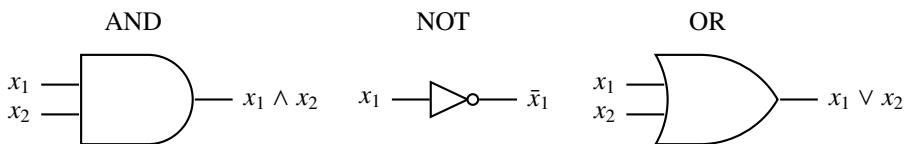


Abbildung 13.11: Logische Gatter, aus denen beliebige logische Schaltungen aufgebaut werden können.

ihrer ersten Staffel [41] gemacht. Tatsächlich ist *MINESWEEPER-CONSISTENCY* NP-vollständig. Die Hoffnung, dass man damit der Frage, ob P = NP ist, näherkommen könnte, hat sich allerdings wieder verflüchtigt.

### Reduktion $SAT \leq_P CIRCUIT$

Die logischen Operationen, die der Programmierer in praktisch jeder Programmiersprache zur Verfügung hat, werden im Prozessor von der sogenannten *arithmetic logic unit* (ALU) ausgeführt, die dazu spezielle, Gatter genannte logische Schaltkreise verwendet. Abbildung 13.11 zeigt Gatter für logisch UND, ODER und NICHT. Diese realisieren die logischen Operationen mit elektrischen Signalen.

Jede logische Formel in den Variablen  $x_1, \dots, x_n$  kann als eine Schaltung von Gattern mit den Eingängen  $x_1, \dots, x_n$  und einem einzigen Ausgang realisiert werden. Abbil-

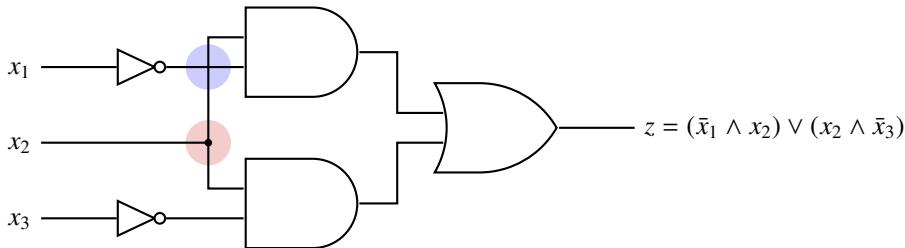


Abbildung 13.12: Realisierung der logischen Formel  $(\bar{x}_1 \wedge x_2) \vee (x_2 \wedge \bar{x}_3)$  als Gatterschaltung. Zusätzlich zu den Gattern und Verbindungsdrähten sind Kreuzungsstellen (blau) und Verzweigungen (rot) nötig.

dung 13.12 zeigt als Beispiel die logische Formel  $(\bar{x}_1 \wedge x_2) \vee (x_2 \wedge \bar{x}_3)$  realisiert mit einer Gatterschaltung. Eine Formel ist genau dann erfüllbar, wenn es möglich ist, die Eingänge der Schaltung so mit logischen Signalen anzusteuern, dass der Ausgang wahr wird. Damit ist gezeigt, dass das folgende Entscheidungsproblem NP-vollständig ist.

**Problem 13.23 (CIRCUIT).** Ist es möglich, die Eingänge aus UND-, ODER- und NICHT-Gattern aufgebauten logischen Schaltung so mit Signalen zu belegen, dass der Ausgang wahr wird?

### Reduktion $CIRCUIT \leq_P MINESWEEPER\text{-}CONSISTENCY$

Der zweite Reduktionsschritt muss einer Gatterschaltung ein *MINESWEEPER-CONSISTENCY*-Problem zuordnen, welches genau dann lösbar ist, wenn die Gatterschaltung am Ausgang den Wert wahr erreichen kann. Wenn es gelingt, den Schaltplan einer Gatterschaltung auf ein Minesweeper-Spielfeld abzubilden, ist die Reduktion gelungen. Wie in Abbildung 13.12 farbig hervorgehoben, enthält ein Schaltplan außer den Gattern noch drei weitere wichtige Elemente: Drähte, die die Gatter miteinander verbinden, Verzweigungen, die Signale an mehr als ein Nachfolgegatter weitergeben können und Kreuzungen. Diese müssen ebenfalls als Minesweeper-Spielpläne realisiert werden.

Für logische Signale müssen zwei mögliche Zustände codiert und weitergeleitet werden können. Die Wire-Struktur in Abbildung 13.13 wurde mit der verlinkten Software visualisiert. Sie enthält jeweils  $2 \times 1$ -Blöcke von nicht aufgedeckten Feldern, umgeben von Einsen. Die einzige Möglichkeit einer konsistenten Bombenbelegung hat eine Bombe in einem der beiden Felder. Außerdem müssen benachbarte Zweierblöcke die gleiche Struktur haben. Wir interpretieren *Bombe im ersten Feld in Laufrichtung des Signals* als logische 0 oder *falsch*, eine Bombe im zweiten Feld als logische 1 oder *wahr*.



autospr.ch/c/6

Die in Abbildung 13.14 gezeigte Splitter-Struktur transportiert ein von links kommendes Signal in die beiden Arme nach oben und unten. Nach rechts wird die Negation des Signals weitergegeben. Die Splitter-Struktur implementiert also sowohl die Verzweigung wie auch die Negation.

Die Kreuzung zweier Signale muss Werte  $x_1$  und  $x_2$  ohne Änderung auf die andere Seite der Kreuzung bringen. Die Vorgaben von Abbildung 13.15 stellen genau dies sicher.

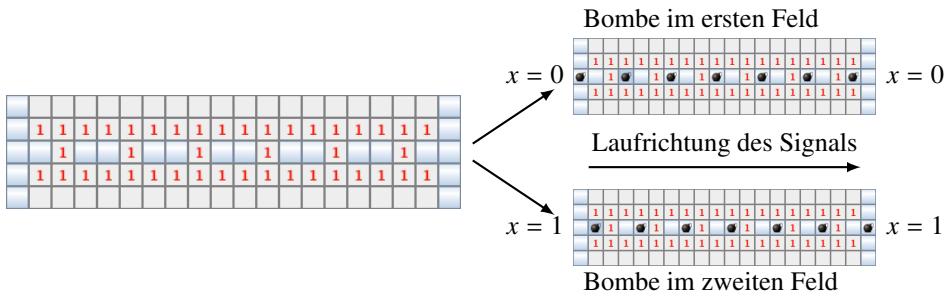


Abbildung 13.13: Logische Signale werden durch die Platzierung einer Bombe in einem von zwei nicht aufgedeckten Feldern codiert. Eine Bombe im ersten Feld in Laufrichtung des Signals bedeutet logisch 0, Bombe im zweiten Feld bedeutet logisch 1. Die Wire-Struktur ist in der Lage, das Signal weiter zu transportieren.

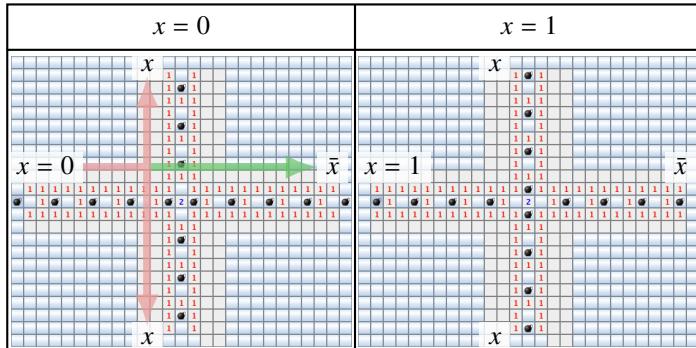


Abbildung 13.14: Die Splitterstruktur transportiert ein von links kommendes logisches Signal sowohl nach oben und unten (rote Pfeile), leitet aber auch die Negation davon nach rechts weiter (grüner Pfeil).

Schließlich muss die UND-Verknüpfung realisiert werden. Diese hat zwei Eingänge, die die Form der Wire-Struktur von Abbildung 13.13 haben und einen ebensolchen Ausgang. Die Zustandstabelle von Abbildung 13.16 zeigt, dass die dort abgebildete Struktur tatsächlich die UND-Operation realisiert.

Damit ist die Reduktion abgeschlossen. Jedes Detail eines logischen Schaltplans lässt sich auf einen Block eines MINESWEEPER-CONSISTENCY-Problems abbilden. Außerdem kann man durch Platzierung einer Bombe erzwingen, dass der Ausgang der Schaltung den Wert *wahr* haben muss. Aus der Lösung des MINESWEEPER-CONSISTENCY-Problems lässt sich dann ablesen, welche Wahrheitswerte die Eingangsvariablen haben müssen, damit die Formel *wahr* wird.

$x_2$	0	1
$x_1$		
0	$x_2 = 0$	$x_2 = 1$
	$x_1 = 0$	$x_1 = 1$
1	$x_2 = 0$	$x_2 = 1$
	$x_1 = 1$	$x_1 = 0$

Abbildung 13.15: Realisation der Kreuzung zweier signalführender Strukturen für das *MINESWEEPER-CONSISTENCY*-Problem.

### Andere NP-vollständige Ausfüllrätsel

Die am Beispiel von *MINESWEEPER-CONSISTENCY* vorgeführte Reduktion von *SAT* via *CIRCUIT* auf ein Problem, dessen NP-Vollständigkeit nachgewiesen werden soll, ist auch in einigen anderen interessanten Fällen erfolgreich angewendet worden.

Für Game of Life von Conway (siehe Abschnitt 10.4.3) wurden Strukturen gefunden, die in die Struktur einfliegende Gleiter wie logische Signale behandeln können und logische Operationen realisieren können. Damit lassen sich auch in Game of Life beliebige Schaltpläne als dynamische Strukturen realisieren.

Der japanische Rätselverlag Nikoli hat eine große Zahl von Rätseln publiziert, die sich alle als polynomiale Ausfüllrätsel betrachten lassen. Wohl das berühmteste Produkt des Verlages ist Sudoku. Für das Spiel *Spiral Galaxies* hat Erich Friedman in [16] mit einer Reduktion von CIRCUIT die NP-Vollständigkeit gezeigt.

Für das Rätselspiel Tatamibari (タタミバリ) wurde in [1] von Adler et al eine Technik verwendet, die jede Formel in konjunktiver Normalform mit drei Literalen pro Klauseln auf ein Tatamibari-Rätsel reduziert. Da 3SAT NP-vollständig ist, ist auch das Entscheidungsproblem, ob ein Tatamibari-Rätsel eine Lösung hat, ein NP-vollständiges Problem.

$x_1 \setminus x_2$	0	1
0	<p><math>x_1 = 0</math></p> <p><math>x_1 \wedge x_2 = 0</math></p> <p><math>x_2 = 0</math></p>	<p><math>x_1 = 0</math></p> <p><math>x_1 \wedge x_2</math></p> <p><math>= 0</math></p>
1	<p><math>x_1 = 1</math></p> <p><math>x_1 \wedge x_2 = 0</math></p> <p><math>x_2 = 0</math></p>	<p><math>x_1 = 1</math></p> <p><math>x_1 \wedge x_2</math></p> <p><math>= 1</math></p>

Abbildung 13.16: Realisation der UND-Verknüpfung als *MINESWEEPER-CONSISTENCY*-Problem.

## 13.4 Ein Katalog von NP-vollständigen Problemen

Bereits ein Jahr nach der Veröffentlichung der Arbeit von Cook stellte Richard Karp in [25] eine Sammlung von bekanntermaßen schwierigen Problemen zusammen und zeigte, dass sie alle NP-vollständig sind. Um zu zeigen, dass ein Problem  $A$  NP-vollständig ist, muss man eine Reduktion  $SAT \leq_P A$  oder von einem anderen bereits als NP-vollständig bekannten Problem auf  $A$  finden. Mit jedem neu gefundenen NP-vollständigen Problem wird es leichter, solche Reduktionen zu konstruieren. Mit der Zeit sind auf diese Weise große Sammlungen von NP-vollständigen Problemen wie das Buch [17] entstanden.

Die für die Reduktion nötigen Konstruktionen sind zum Teil sehr phantasievoll. Für den Praktiker ist es wichtig zu verstehen, ob er mit einem NP-vollständigen Problem zu tun hat, weil ihm dies zu erwartende Skalierungsprobleme offenbart. Er wird aber kaum die Zeit und Energie aufwenden können, eine solche komplexe Reduktion durchzuführen und damit zu beweisen, dass das Problem NP-vollständig ist. Die Chancen stehen aber gut, dass das Problem bereits in einer Sammlung NP-vollständiger Problem gefunden werden kann. Zum Beispiel hat sich ja herausgestellt, dass das Stundenplanproblem in der Form

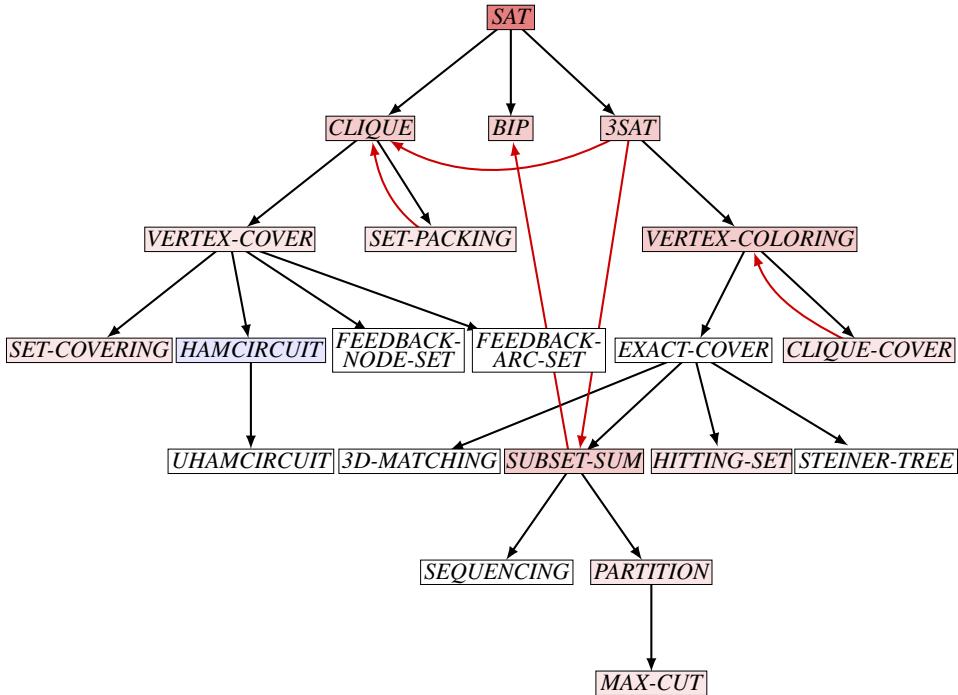


Abbildung 13.17: Reduktionsbaum des Katalogs NP-vollständiger Probleme nach Richard Karp. Die NP-Vollständigkeit ist in diesem Buch für die rot hinterlegten Probleme bewiesen worden, zum Teil unter Verwendung anderer Reduktionen als in der Arbeit [25] von Karp (rote Pfeile). Hellrote Probleme werden nicht vollständig als NP-vollständig bewiesen. Das Problem *HAMCIRCUIT* wurde bereits in Abschnitt 12.2.3 erwähnt.

des Färbeproblems bereits als NP-vollständig bekannt ist. Für diese Erkenntnis muss er nur in der Lage sein, eine Eins-zu-eins-Reduktion durchzuführen, in der jedem Begriff des Aufgabenproblems ein Begriff des Vergleichsproblems zugeordnet werden muss.

### 13.4.1 Der Reduktionsbaum von Karp

Der Katalog [25] NP-vollständiger Problem von Richard Karp ist ein guter Startpunkt für den Anwender. Das Studium der Problembeschreibungen fördert die Intuition, in welch vielfältigen Formen sich NP-Vollständigkeit manifestieren kann, und liefert eine solide Basis für Eins-zu-eins-Vergleiche. Lässt sich in Karps Katalog kein Vergleichsproblem finden, können größere Sammlungen wie [17] oder [28] oder Spezialliteratur herangezogen werden.

An der Spitze steht natürlich das Problem *SAT*, für das der Satz 13.5 von Cook-Levin zeigt, dass es NP-vollständig ist. Alle anderen Probleme wurden durch polynomiale Reduktion ausgehend von *SAT* oder einem anderen bereits analysierten Problem als NP-vollständig erkannt.

Auf der zweitobersten Ebene des Baumes sind die Probleme *k-CLIQUE* und *3SAT* zu

finden, für die bereits gezeigt wurde, dass sie NP-vollständig sind. Das Problem *BIP* wird weiter unten in Abschnitt 13.4.2 untersucht.

### Die 21 NP-vollständigen Probleme von Karp

Die noch nicht diskutierten Probleme des Katalogs 21 NP-vollständigen Problemen von Richard Karp sollen in den folgenden Abschnitten skizziert werden. Wir richten uns grob nach der Struktur des in Abbildung 13.17 wiedergegebenen Ableitungsbaumes von Karp. Nicht in allen Fällen wird auch die Reduktion gezeigt, die die NP-Vollständigkeit begründet. In den Fällen, wo eine Reduktion gezeigt wird, wird nicht immer die Reduktion dargestellt, die Karp in [25] verwendet hat. Es werden jedoch keine so komplizierten und phantasievollen Reduktionen gezeigt, wie sie zum Beispiel für  $3SAT \leq_P CLIQUE$  nötig war. Dafür wird dem Leser die Spezialliteratur [25, 17, 56] empfohlen. Die hier präsentierten, weniger komplexen Umformungen sollen die Vertrautheit mit ein paar Tricks fördern, die für einfache Reduktionen immer wieder hilfreich sein können.

Die folgende Zusammenstellung orientiert sich weniger an der hierarchischen Struktur des Baumes, sondern stellt jeweils Probleme gegenüber, die von ähnlichen Sachverhalten sprechen und leicht verwechselt werden können. Dadurch können die Unterschiede etwas besser hervorgehoben werden, was letztendlich auch klarer hervorhebt, wo die Schwierigkeit der Problemlösung herkommt.

#### 13.4.2 *BIP*: Binary Integer Programming

Das Problem *BIP* heißt *binary integer programming* und ist das folgende Entscheidungsproblem.

**Problem 13.24.** Gegeben eine ganzzahlige Matrix  $C$  und ein ganzzahliges Vektor  $d$ , gibt es einen binären Vektor  $x$  derart, dass  $Cx = d$ ? Als Sprachproblem ist

$$BIP = \left\{ \langle C, d \rangle \mid \begin{array}{l} \text{Zu } C \in M_{n \times m}(\mathbb{Z}) \text{ und } d \in \mathbb{Z}^n \text{ gibt es einen binären Vektor} \\ x \in \{0, 1\}^m \text{ derart, dass } Cx = d. \end{array} \right\}$$

als binary integer programming oder auch 0-1 integer programming bekannt.

Die Bezeichnungen und auch die Formulierung des Problems werden in der Literatur nicht einheitlich verwendet. Die Notation in Problem 13.24 stützt sich auf [25].

**Satz 13.25.** *BIP* ist NP-vollständig.

Ist die Matrix  $C$  nur eine Zeile, ist also  $n = 1$  und damit  $d$  nur eine Zahl, klingt die Problemstellung 13.24 sehr ähnlich wie *SUBSET-SUM*. Die Ähnlichkeit ist ausreichend, dass es für einen Beweis von Satz 13.25 reicht.

*Beweis.* Es ist klar, dass *BIP* in NP ist. Es genügt daher, eine Reduktion

$$SUBSET-SUM \leq_P BIP$$

zu konstruieren.

Dazu muss zu einer Menge  $S$  von ganzen Zahlen und einer Summe  $t$  eine Matrix und ein Vektor konstruiert werden. Als Matrix nehmen wir eine Matrix mit einer Zeile, in der die Elemente von  $S$  stehen. Als Vektor  $d$  nehmen wir den Vektor mit den einen Komponenten  $t$ . Einen binären Vektor  $x$  mit  $Cx = d$  finden ist jetzt gleichbedeutend damit, eine Auswahl von Zahlen in  $S$  zu finden, deren Summe  $t$  ist. Also ist  $SUBSET-SUM \leq_P BIP$ .  $\square$

### 13.4.3 NP-vollständige Probleme zu Graph-Überdeckungen

Die beiden Probleme *VERTEX-COVER* und *CLIQUE-COVER* handeln davon, dass ein gegebener Graph irgendwie von einer Menge beschränkter Größe mit einer besonderen Eigenschaft aus erreicht werden kann. Bei *CLIQUE-COVER* ist es die besondere Eigenschaft, dass alle Knoten in einer vorgegebenen Anzahl Cliques liegen sollen, bei *VERTEX-COVER* sind es Knoten, von denen aus alle Kanten des Graphen unmittelbar zugänglich sein sollen.

**Problem 13.26 (CLIQUE-COVER und EXACT-CLIQUE-COVER).** Gegeben ein Graph  $G$  und eine Zahl  $k$ , gibt es  $k$  Cliques im Graphen  $G$ , die alle Knoten des Graphen enthalten? Können die Cliques zudem disjunkt gewählt werden? Als Sprachprobleme:

$$\text{CLIQUE-COVER} = \left\{ \langle G, k \rangle \mid \begin{array}{l} \text{Es gibt } k \text{ Cliques} \\ G_1 = (V_1, E_1), \dots, G_k = (V_k, E_k), \text{ die zusammen} \\ \text{alle Knoten von } G \text{ abdecken, also } V = \bigcup_{i=1}^k V_i. \end{array} \right\}$$

bzw.

$$\text{EXACT-CLIQUE-COVER} = \left\{ \langle G, k \rangle \mid \begin{array}{l} \text{Es gibt } k \text{ disjunkte Cliques} \\ G_1 = (V_1, E_1), \dots, G_k = (V_k, E_k), V_i \cap V_j = \emptyset \\ \text{für } i \neq j, \text{ die zusammen alle Knoten von } G \\ \text{abdecken, also } V = \bigcup_{i=1}^k V_i. \end{array} \right\}.$$

Wir schreiben auch  $k$ -CLIQUE-COVER, wenn die Abhängigkeit von  $k$  deutlich gemacht werden soll.

**Satz 13.27.** CLIQUE-COVER und EXACT-CLIQUE-COVER sind NP-vollständig.

Der folgende Beweis behandelt nur das EXACT-CLIQUE-COVER-Problem.

*Beweis.* Das EXACT-CLIQUE-COVER-Problem ist eine Umformulierung des Färbeproblems, wie die folgende polynomiale Reduktion zeigt. Sei  $G = (V, E)$  ein Graph, dann bilden wir den komplementären Graphen  $G' = (V, E')$ , wobei die Menge der Kanten

$$E' = \{\{a, e\} \subset V \mid a \neq e \wedge (a, e) \notin E\}$$

genau aus den Kanten besteht, die nicht in  $E$  sind. Abbildung 13.18 zeigt die Konstruktion für den Peterson-Graphen.

Falls sich der Graph mit  $k$  Farben einfärben lässt, dann haben gleichfarbige Knoten keine Verbindungen. Somit sind alle gleichfarbigen Knoten im Graphen  $G'$  miteinander

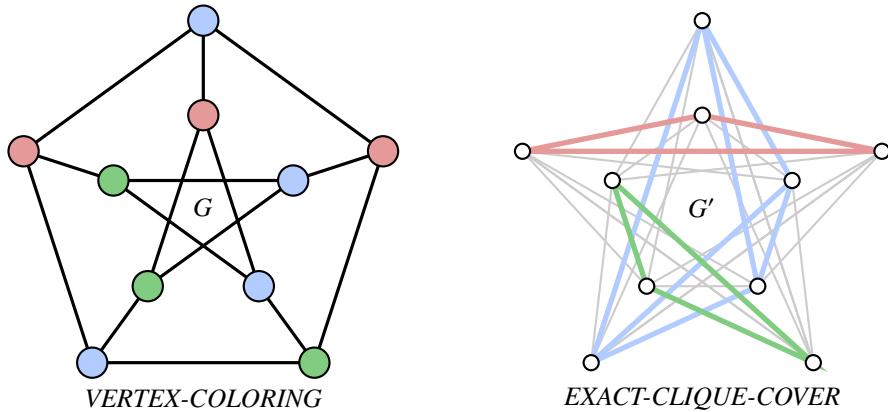


Abbildung 13.18: Das Färbeproblem (links) geht in ein *EXACT-CLIQUE-COVER*-Problem über, indem man den Graphen  $G = (V, E)$  durch den komplementären Graphen  $G' = (V, E')$  ersetzt, dessen Kanten die Kanten sind, die in  $G$  fehlen. Die farbig eingezeichneten Cliques enthalten alle Knoten des Graphen.

verbunden, sie bilden eine Clique. Zu jeder Farbe in  $G$  gibt es eine Clique in  $G'$ , diese Cliques decken alle Knoten ab.

Ist umgekehrt  $G_1, \dots, G_k$  eine Überdeckung von  $G'$  mit Cliques, dann haben die Knoten von  $G_i$  in  $G$  keine Verbindung, sie dürfen also die gleiche Farbe haben. Wählt man für jede Clique  $G_i$  eine Farbe und färbt damit die Knoten von  $G_i$ , hat man eine Lösung des Färbeproblems mit  $k$  Farben für den Graphen  $G$  gefunden.  $\square$

### **VERTEX-COVER**

Beim Problem *VERTEX-COVER* geht es um die Frage, ob man von einer beschränkten Zahl  $k$  von Knoten eines Graphen direkten Zugang zu allen Kanten erhält, ob also die mit diesen Knoten verbundenen Kanten alle Kanten des Graphen abdecken.

**Problem 13.28 (VERTEX-COVER).** Gegeben ein Graph  $G$  und eine Zahl  $k$ , gibt es eine Menge  $W$  von  $k$  Knoten derart, dass jede Kante von  $G$  einen Endpunkt in  $W$  hat? Als Sprachproblem:

$$\text{VERTEX-COVER} = \left\{ \langle G = (V, E), k \rangle \mid \begin{array}{l} \text{Es gibt eine } k\text{-elementige Teilmenge } W \subset V \\ \text{mit } W \cap e \neq \emptyset \text{ für alle } e \in E. \end{array} \right\}.$$

Wir schreiben auch  $k$ -VERTEX-COVER, wenn die Abhängigkeit von  $k$  deutlich gemacht werden soll.

In einem vollständigen Graphen ist jeder Knoten mit jedem anderen verbunden, die einzige Lösung des *VERTEX-COVER*-Problems ist also die Menge aller Knoten.

Auch für die Reduktion von *CLIQUE* verwenden wir den bereits im Beweis von Satz 13.27 verwendeten komplementären Graphen  $G' = (V, E')$  mit den Kanten, die im Graphen  $G = (V, E)$  fehlen. Zwischen den Knoten einer  $k$ -Clique gibt es in  $G'$  keine Ver-

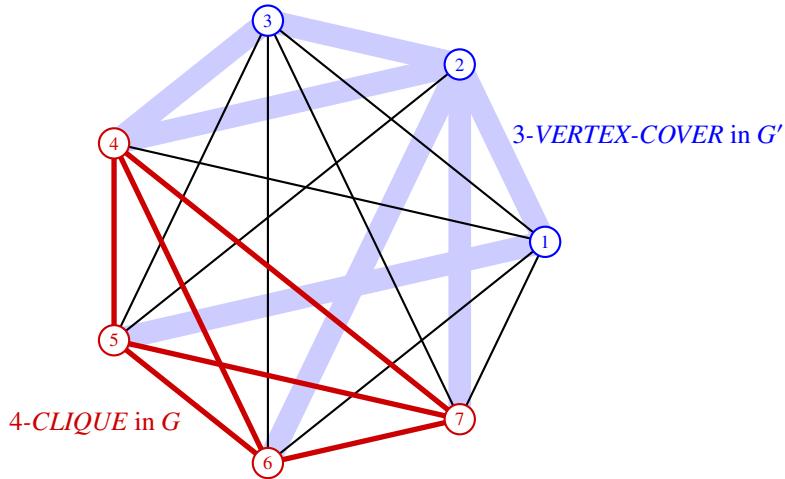


Abbildung 13.19: Reduktion von  $k$ -CLIQUE auf  $k'$ -VERTEX-COVER durch Übergang zum komplementären Graphen  $G'$  und  $k' = |V| - k$ . Die Kanten des Graphen  $G'$  fehlen in  $G$  und sind breit hellblau dargestellt.

bindungen. Die Verbindungen in  $G'$  müssen daher in den Knoten des Komplements der  $k$ -Clique enden. Das Komplement einer  $k$ -Clique ist also eine Lösung des  $k'$ -VERTEX-COVER-Problems für den Graphen  $G'$  mit  $k' = |V| - k$ .

**Satz 13.29.** *Die Probleme CLIQUE und VERTEX-COVER sind polynomiell äquivalent. VERTEX-COVER ist NP-vollständig.*

### 13.4.4 NP-vollständige Probleme über Familien von Mengen

Die Probleme SET-PACKING, SET-COVERING und EXACT-COVER handeln alle von einer Familie  $(S_i)_{i \in I}$  von nichtleeren endlichen Mengen. Wir bezeichnen die Vereinigung der Mengen mit

$$U = \bigcup_{i \in I} S_i.$$

In allen Problemen geht es um die Existenz einer Teilfamilie mit bestimmten Eigenschaften. Eine solche Teilfamilie wird durch die Auswahl einer Teilmenge  $J \subset I$  der Indexmenge  $I$  festgelegt.

#### SET-PACKING

Wie viele der Mengen  $S_i$  kann man in  $U$  hineinpacken, ohne dass sie sich überlappen?

**Problem 13.30 (SET-PACKING).** *Gegeben die Familie  $(S_i)_{i \in I}$  und eine Zahl  $k$ , gibt es eine  $k$ -elementige Teilfamilie  $(S_i)_{i \in J}$ ,  $J \subset I$ ,  $|J| = k$  von paarweise disjunkten Teilmengen? Als*

*Sprachproblem:*

$$\text{SET-PACKING} = \left\{ \langle (S_i)_{i \in I}, k \rangle \mid \begin{array}{l} \text{Es gibt eine } k\text{-elementige Teilstamme} \\ J \subset I, |J| = k \text{ von paarweise disjunkten} \\ \text{Mengen } S_k \cap S_l = \emptyset \forall k, l \in J. \end{array} \right\}.$$

Zur Verdeutlichung der Abhängigkeit von  $k$  kann dies auch  $k$ -SET-PACKING geschrieben werden.

Je größer die Zahl  $k$  ist, desto schwieriger wird es, genügend viele Teilmengen auszuwählen, die sich nicht schneiden. Daher ist es nicht überraschend, dass dieses Problem NP-vollständig ist.

**Satz 13.31.** SET-PACKING ist NP-vollständig.

Im folgenden Beweis zeigen wir sogar etwas mehr, nämlich dass die SET-PACKING und CLIQUE polynomiell äquivalent sind. Der Beweis ist in Abbildung 13.20 visualisiert.

*Beweis.* Beim SET-PACKING-Problem geht es nur um die kombinatorische Eigenschaft, ob sich Mengen der Familie schneiden oder nicht. Diese Eigenschaft lässt sich durch einen Graphen ausdrücken. Die Knoten des Graphen sind die Mengen der Familie, identifiziert durch die Elemente von  $I$ . Kanten werden hinzugefügt, wenn zwei Mengen sich nicht schneiden. Die Suche nach einer Teilstamme von  $k$  sich nicht schneidenden Mengen ist daher gleichbedeutend mit der Suche nach einer  $k$ -Clique im Graphen (Abbildung 13.20 rechts).

Umgekehrt kann man zu einem Graphen  $G = (V, E)$  eine Familie von Mengen  $(S_v)_{v \in V}$  bilden, wobei die Menge

$$S_v = \{e \in E \mid v \text{ ist kein Endpunkt von } e\}$$

aus den Kanten besteht, die nicht mit  $v \in V$  verbunden sind. Die Menge  $S_v$  besteht also aus den von  $v$  ausgehenden Kanten, die im Graphen fehlen (Abbildung 13.20 links). Wenn sich zwei solche Mengen  $S_v$  und  $S_w$  nicht schneiden, heißt das, dass es keine Kante mit  $v$  und  $w$  als Endpunkten gibt, die im Graphen fehlt, die Knoten  $v$  und  $w$  sind also verbunden. Findet man  $k$  solche Mengen, dann sind sie alle miteinander verbunden und bilden somit eine Clique.  $\square$

### SET-COVERING

In einem gewissen Sinne komplementär ist das Problem, die ganze Vereinigung  $U$  mit einer beschränkten Anzahl  $i$  von Teilmengen zu überdecken. Anstelle der Bedingung, dass sich die Mengen nicht schneiden dürfen, tritt die Bedingung, dass die Mengen zusammen die Menge  $U$  ergeben.

**Problem 13.32 (SET-COVERING).** Gegeben die Familie  $(S_i)_{i \in I}$  und eine natürliche Zahl  $k$ , gibt es eine Teilstamme  $J \subset I$  von  $k$  Mengen,  $|J| = k$ , die die gleiche Vereinigung

$$\bigcup_{j \in J} S_j = U = \bigcup_{i \in I} S_i$$

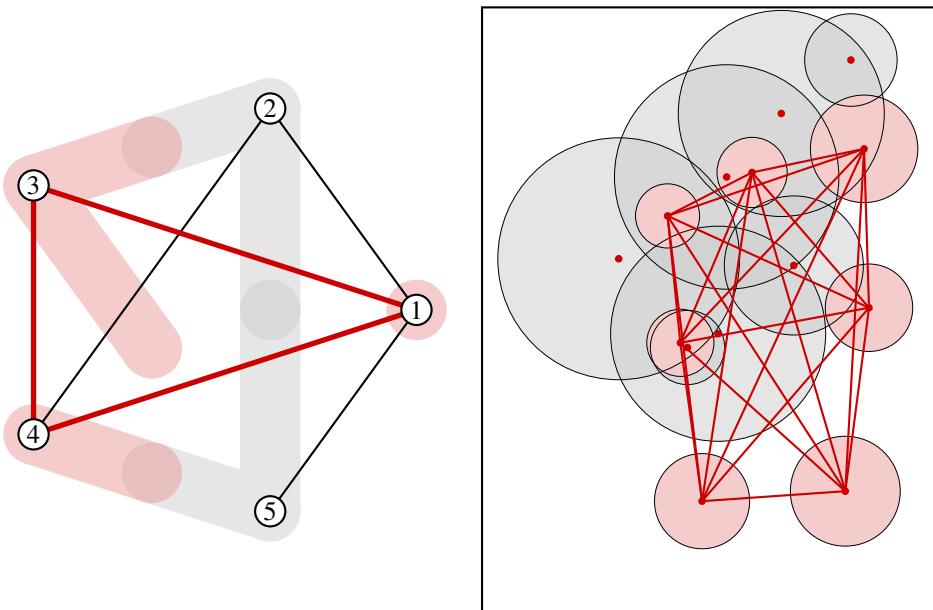


Abbildung 13.20: Polynomielle Äquivalenz von *CLIQUE* und *SET-PACKING*. Links die Reduktion eines *CLIQUE*-Problems auf das *SET-PACKING*-Problem für die Mengen der fehlenden Kanten, die als hell farbige Kantenstummel eingezeichnet sind. Die roten Mengen schneiden sich nicht, die zugehörigen Knoten bilden eine Clique. Rechts das *SET-PACKING*-Problem für eine Familie von 12 Kreisflächen in einem Rechteck und  $k = 7$ . Gefunden werden müssen sieben Mengen, die sich nicht schneiden. Die Mengen bilden eine  $k$ -Clique in dem Graphen, der als Knoten die Mengen hat und eine Kante für jedes Paar von Mengen, die sich nicht schneiden.

haben? Als Sprachproblem:

$$\text{SET-COVERING} = \left\{ \langle (S_i)_{i \in I}, k \rangle \mid \begin{array}{l} \text{Es gibt eine } k\text{-elementige Teilstammel} \\ \text{lie } J \subset I, |J| = k, \text{ mit } \bigcup_{j \in J} S_j = U. \end{array} \right\}.$$

Je kleiner die Zahl  $k$  ist, desto schwieriger ist es, genügend große Mengen zu finden. Sind zum Beispiel die Mengen  $(K_i)_{1 \leq i \leq 26}$  die inneren Punkte der 26 Kantone der Schweiz, wie sie auf der politischen Karte in Abbildung 12.2 farbig ausgefüllt sind, dann sind diese Mengen disjunkt. Nur wenn  $k = 26$ , hat das *SET-COVERING*-Problem eine Lösung. Für kleinere Werte von  $k$  gibt es keine Lösung. Das *SET-PACKING*-Problem für diese Familie ist dagegen für jeden kleineren Wert von  $k$  lösbar.

**Satz 13.33.** *SET-COVERING ist NP-vollständig.*

*Beweis.* Eine Reduktion von *VERTEX-COVER* kann wie folgt konstruiert werden. Sei  $G = (V, E)$  ein Graph. Zu jedem Knoten  $v \in V$  sei

$$S_v = \{e \in E \mid v \text{ ist ein Endpunkt von } e\}$$

die Menge der von  $v$  ausgehenden Kanten. Da jede Kante in einem Knoten endet, ist

$$\bigcup_{v \in V} S_v = E.$$

Eine Lösung des *SET-COVERING*-Problems für  $(S_v)_{v \in V}$  ist eine Menge von Knoten  $W \subset V$  derart, dass

$$\bigcup_{w \in W} S_w = \bigcup_{v \in V} S_v = V$$

ist. Mit anderen Worten: Jede Kante des Graphen hat einen der Knoten von  $W$  als Endpunkt. Damit ist  $W$  auch eine Lösung des *VERTEX-COVER*-Problems.  $\square$

### ***EXACT-COVER***

Kann man die Bedingungen von *SET-PACKING* und *SET-COVERING* kombinieren? Natürlich nicht, ohne dafür die Bedingung an die Anzahl der verwendeten Mengen wegzulassen.

**Problem 13.34 (*EXACT-COVER*).** Gegeben die Familie  $(S_i)_{i \in I}$ , gibt es eine Teilfamilie  $J \subset I$  von disjunkten Mengen  $S_k \cap S_l = \emptyset \forall k, l \in J, k \neq l$  mit der gleichen Vereinigung

$$\bigcup_{j \in J} S_j = U = \bigcup_{i \in I} S_i?$$

Als Sprachproblem:

$$\text{EXACT-COVER} = \left\{ (S_i)_{i \in I} \mid \begin{array}{l} \text{Es gibt eine Teilfamilie } J \subset I \text{ paarweise} \\ \text{disjunkter Mengen mit der gleichen Vereinigung.} \end{array} \right\}.$$

Ein erster Versuch einer Reduktion von *VERTEX-COLORING* auf *EXACT-COVER* könnte darin bestehen, zu einem Graphen  $G$  die Menge  $\mathcal{S}$  der Teilmengen  $S \in V$  von Knoten zu bilden, die die gleiche Farbe haben können. Es ist klar, dass die Vereinigung all dieser Mengen  $V$  ist. Eine exakte Überdeckung von  $V$  durch nur  $k$  Mengen  $S_1, \dots, S_k \in \mathcal{S}$  ergibt eine Einfärbung des Graphen, indem man jeder Menge  $S_i$  eine Farbe zuteilt. Leider ist dieser Reduktionsversuch keine polynomielle Reduktion. Hat nämlich der Graph  $G$  gar keine Kanten, dann ist  $\mathcal{S} = P(V)$  die Potenzmenge der Knotenmenge, d. h. das abgeleitete *EXACT-COVER*-Problem wächst exponentiell mit der Größe von  $V$ . Daher ist eine polynomielle Reduktion etwas komplizierter, man findet sie in [25, p. 99]. Wir verzichten auf einen vollständigen Beweis und konstatieren nur den folgenden Satz.

**Satz 13.35.** *EXACT-COVER* ist NP-vollständig.

### ***HITTING-SET***

Das Problem *HITTING-SET* sucht zu einer Familie von Mengen eine Menge von Punkten (Elemente der Vereinigung dieser Mengen), die jede Menge der Familie in genau einem Punkt treffen. Man beachte, dass es keine Forderung über die Anzahl der Punkte gibt.

**Problem 13.36 (HITTING-SET).** Gibt es eine Menge  $W \subset U$  derart, dass  $W$  mit jeder Menge  $S_i$  nur ein Element gemeinsam hat? Als Sprachproblem

$$\text{HITTING-SET} = \left\{ \langle (S_i)_{i \in I} \rangle \mid \begin{array}{l} \text{Es gibt eine Teilmenge } W \subset U = \bigcup_{i \in I} S_i \\ \text{derart, dass } |W \cap S_i| = 1 \text{ für alle } i \in I. \end{array} \right\}.$$

**Satz 13.37.** HITTING-SET ist NP-vollständig.

Zum Beweis kann man eine polynomiale Reduktion von EXACT-COVER konstruieren.

*Beweis.* Sei  $(S_i)_{i \in I}$  ein Familie endlicher Mengen eines EXACT-COVER-Problems. Eine Lösung des Problems besteht aus Mengen  $(S_j)_{j \in J}$ ,  $J \subset I$  derart, dass jedes Element  $x \in U$  genau eine der Mengen schneidet. Das klingt wie das HITTING-SET-Problem, aber mit vertauschten Rollen von Mengen und Elementen.

Daher konstruieren wir zu jedem  $x \in U$  die Menge

$$U_x = \{j \in I \mid x \in S_j\}$$

der Indizes von Mengen  $S_j$ , die das Element  $x$  enthalten. Statt der Menge von Punkten von  $U$  betrachten wir jetzt also die Menge von Indizes von Mengen, die den Punkte enthalten. Damit vollziehen wir den Rollentausch von Mengen und Elementen.

Da jede Menge  $S_j$  mindestens ein Element  $x \in S_j$  enthält, enthält  $U_x$  auch  $j$ , somit ist die Vereinigung der  $U_x$

$$\bigcup_{x \in U} U_x = I.$$

Wenn  $W$  eine Lösung des HITTING-SET-Problems für die Familie  $(U_x)_{x \in U}$  ist, dann ist  $|U_x \cap W| = 1$  für alle  $x \in U$ . Für jedes  $x \in U$  gibt es also genau einen Index  $j \in W$  derart, dass  $x \in S_j$ , d. h. die Familie  $(S_j)_{j \in W}$  ist eine Lösung des EXACT-COVER-Problems.  $\square$

*Verständniskontrolle 13.5:* Finden Sie ein NP-vollständiges Problem, welches sich eins-zu-eins auf das folgende Problem TOURIST reduzieren lässt.



autospr.ch/v/13.5.pdf

Von jedem Aussichtspunkt aus kann man eine endliche Anzahl Sehenswürdigkeiten erkennen. Ein Tourist, der es sehr eilig hat, möchte mit dem Besuch von nur  $k$  Aussichtspunkten alles sehen, was man an Sehenswürdigkeiten von allen Aussichtspunkten aus sehen könnte. Warum ist es schwierig, für den Touristen eine Auswahl zu treffen?

### 13.4.5 FEEDBACK-NODE-SET und FEEDBACK-ARC-SET

In den Problemen *FEEDBACK-NODE-SET* und *FEEDBACK-ARC-SET* geht es um einen gerichteten Graphen  $G = (V, E)$ . Die Kantenmenge  $E$  besteht aus gerichteten Kanten  $(a, e)$  mit  $a, e \in V$ . In beiden Problemen geht es zudem um die Eigenschaften von Zyklen in diesem Graphen. Ein *gerichteter Zyklus* in  $G$  ist eine Folge von Knoten  $a_0, a_1, a_2, \dots, a_n = a_0$ , die jeweils Enden von Kanten sind, also  $(a_i, a_{i+1}) \in E$ ,  $i = 0, \dots, n - 1$ .

**Problem 13.38 (FEEDBACK-NODE-SET).** *Gibt es zu einem gerichteten Graphen  $G$  und einer Zahl  $k$  eine Menge  $W \subset V$  von  $k$  Knoten derart, dass jeder geschlossene Zyklus durch einen dieser Knoten geht?*

**Problem 13.39 (FEEDBACK-ARC-SET).** *Gibt es zu einem gerichteten Graphen  $G$  und einer Zahl  $k$  eine Menge  $F \subset E$  von  $k$  Kanten derart, dass jeder geschlossene Zyklus eine dieser Kanten enthält?*

Beide Probleme werden einfach, wenn die Zahl  $k$  genügend groß ist. Ist zum Beispiel  $k = |V|$ , dann kann man  $W = V$  wählen und damit das *FEEDBACK-NODE-SET*-Problem lösen. Wählt man  $k = |E|$ , dann kann man mit der Wahl  $W = F$  das *FEEDBACK-ARC-SET*-Problem lösen.

### 13.4.6 NP-vollständige Probleme über Pfade in Graphen

Ein *hamiltonscher Pfad* in einem Graphen  $G = (V, E)$  ist ein Pfad, der jeden Knoten des Graphen genau einmal besucht. Ein solcher Pfad definiert eine Reihenfolge  $a_1, \dots, a_n$  von Knoten, so dass jede Kante von  $a_i$  nach  $a_{i+1}$  in  $E$  ist. Für einen gerichteten Graphen sind dies die Kanten  $(a_i, a_{i+1}) \in E$ , für einen ungerichteten Graphen gilt  $\{a_i, a_{i+1}\} \in E$ . Ein *hamiltonscher Zyklus* ist ein geschlossener hamiltonscher Pfad.

Einen hamiltonschen Pfad oder Zyklus in einem gerichteten oder ungerichteten Graphen zu finden, ist ein schwieriges Problem.

**Problem 13.40 (HAMPATH und HAMCIRCUIT).** *Gibt es einen gerichteten hamiltonschen Zyklus im gerichteten Graphen  $G$ ? Als Sprachproblem:*

$$\text{HAMPATH} = \left\{ \langle G \rangle \mid \begin{array}{l} \text{Im gerichteten Graphen } G \text{ gibt es} \\ \text{einen gerichteten hamiltonschen Pfad.} \end{array} \right\}.$$

*Das Problem HAMCIRCUIT verlangt zusätzlich, dass der Pfad geschlossen ist.*

**Problem 13.41 (UHAMPATH und UHAMCIRCUIT).** *Gibt es einen hamiltonschen Zyklus im ungerichteten Graphen  $G$ ? Als Sprachproblem:*

$$\text{UHAMPATH} = \left\{ \langle G \rangle \mid \begin{array}{l} \text{Im Graphen } G \text{ gibt es einen} \\ \text{hamiltonschen Pfad.} \end{array} \right\}.$$

*Das Problem UHAMCIRCUIT verlangt zusätzlich, dass der Pfad geschlossen ist.*

**Satz 13.42.** *Alle vier Probleme HAMPATH, UHAMPATH, HAMCIRCUIT und UHAMCIRCUIT sind NP-vollständig.*

Beweise für *HAMCIRCUIT* und *UHAMCIRCUIT* kann man in [25] finden. Das Buch [56] enthält eine polynomielle Reduktion von *3SAT* auf *HAMPATH*.

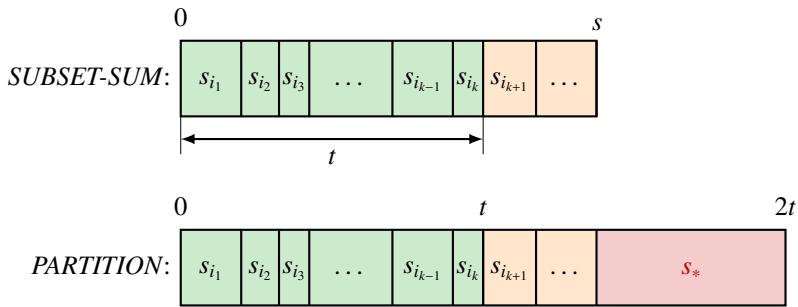


Abbildung 13.21: Polynomielle Reduktion von *SUBSET-SUM* auf *PARTITION* für den Fall  $t \leq s < 2t$ . Durch Hinzufügen der Zahl  $s_*$  zur Liste  $S$  entsteht ein *PARTITION*-Problem, welches genau dann lösbar ist, wenn das *SUBSET-SUM*-Problem lösbar ist.

### 13.4.7 Probleme mit einer Aufteilung in zwei Teilmengen

Das Problem *SUBSET-SUM* spricht von einer Auswahl aus einer Liste von Zahlen, die gewisse Eigenschaften erfüllen soll. Man kann das aber auch interpretieren als eine Aufteilung einer Liste von Zahlen in zwei Teillisten, nämlich der Liste der gewählten und der nicht gewählten Elemente. Aus dieser Sicht ergeben sich zwei neue, verwandte Probleme: *PARTITION* und *MAX-CUT*.

#### *PARTITION*

**Problem 13.43 (PARTITION).** *Gibt es eine Unterteilung einer Liste von ganzen Zahlen in zwei disjunkte Listen mit gleicher Summe? Als Sprachproblem:*

$$\text{PARTITION} = \left\{ S = [s_1, \dots, s_n] \left| \begin{array}{l} \text{Es gibt eine Teilliste } T = [s_{i_1}, \dots, s_{i_k}] \subset S, \text{ deren} \\ \text{Elemente die gleiche Summe haben wie die} \\ \text{Elemente der Liste } \bar{T} = S \setminus T. \end{array} \right. \right\}.$$

**Satz 13.44.** *PARTITION ist NP-vollständig.*

*Beweis.* Es kann eine polynomielle Reduktion von *SUBSET-SUM* konstruiert werden. Sei also  $S = [s_1, \dots, s_n]$  eine Liste von natürlichen Zahlen und  $t \in \mathbb{N}$  eine Summe. Es muss ein *PARTITION*-Problem konstruiert werden, welches genau dann lösbar ist, wenn das *SUBSET-SUM*-Problem mit  $S$  und  $t$  lösbar ist.

Sei  $s$  die Summe aller Elemente in der Liste  $S$ , also

$$s = \sum_{i=1}^n s_i.$$

Wir können annehmen, dass  $t \leq s$  ist, denn andernfalls ist das Problem ohnehin nicht lösbar.

Wir betrachten erst den Fall, dass  $2t > s \geq t$  ist, er ist in Abbildung 13.21 dargestellt. Durch Hinzufügen der zusätzlichen, in der Abbildung rot hervorgehobenen Zahl  $s_* = 2t - s$

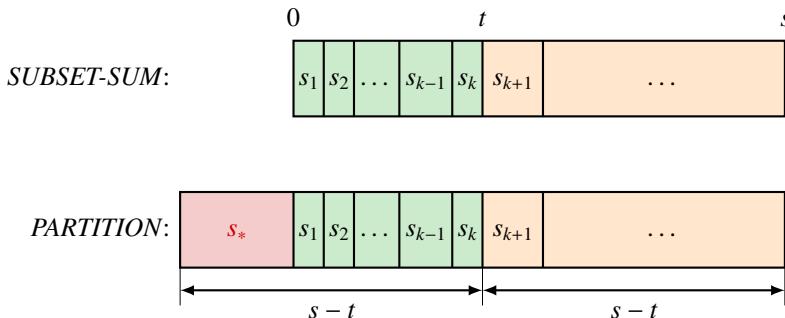


Abbildung 13.22: Polynomiale Reduktion von *SUBSET-SUM* auf *PARTITION* für den Fall  $t \leq 2t < s$ . Durch Hinzufügen der Zahl  $s_*$  zur Liste  $S$  entsteht ein *PARTITION*-Problem, welches genau dann lösbar ist, wenn das *SUBSET-SUM*-Problem lösbar ist.

zu  $S$  entsteht eine Liste  $\tilde{S} = S \cup [s_*] = [s_1, \dots, s_n, s_*]$ . Wenn das *PARTITION*-Problem für  $\tilde{S}$  lösbar ist, dann gibt es eine Aufteilung von  $\tilde{S}$  derart, dass jeder Teil als Summe der Elemente den Wert  $t$  hat. Die Teilliste, die  $s_*$  nicht enthält, ist eine Teilliste von  $S$  mit Summe  $t$ , ist also eine Lösung des *SUBSET-SUM*-Problems.

Falls die in Abbildung 13.22 dargestellte Situation  $2t < s$  eingetreten ist, fügt man der Liste  $S$  das in der Abbildung rote Element  $s_* = s - 2t$  hinzu und erhält  $\tilde{S} = S \cup [s_*] = [s_1, \dots, s_n, s_*]$ . Falls das *PARTITION*-Problem für  $\tilde{S}$  lösbar ist, dann gibt es eine Teilliste  $T$  von  $\tilde{S}$ , welche als Summe  $s - t$  hat und außerdem das Element  $s_*$  enthält. Die Elemente  $T \setminus [s_*]$  summieren sich daher zu  $s - t - s_* = s - t - (s - 2t) = t$ , sie bilden also eine Lösung des *SUBSET-SUM*-Problems. Da auch die Umkehrung funktioniert, ist eine polynomiale Reduktion  $SUBSET-SUM \leq_P PARTITION$  gefunden.  $\square$

### MAX-CUT

Auch das *MAX-CUT*-Problem spricht von einer Unterteilung in zwei Teilmengen.

**Problem 13.45 (MAX-CUT).** Gegeben ein Graph  $G = (V, E)$  mit einer Gewichtsfunktion  $\varphi: E \rightarrow \mathbb{Z}$  der Kanten und einer ganzen Zahl  $w \in \mathbb{Z}$ , gibt es eine Aufteilung der Knotenmenge  $V = V_1 \cup V_2$  in zwei disjunkte Teilmengen, so dass das Gewicht der Kanten, die  $V_1$  und  $V_2$  verbinden, mindestens  $w$  ist:

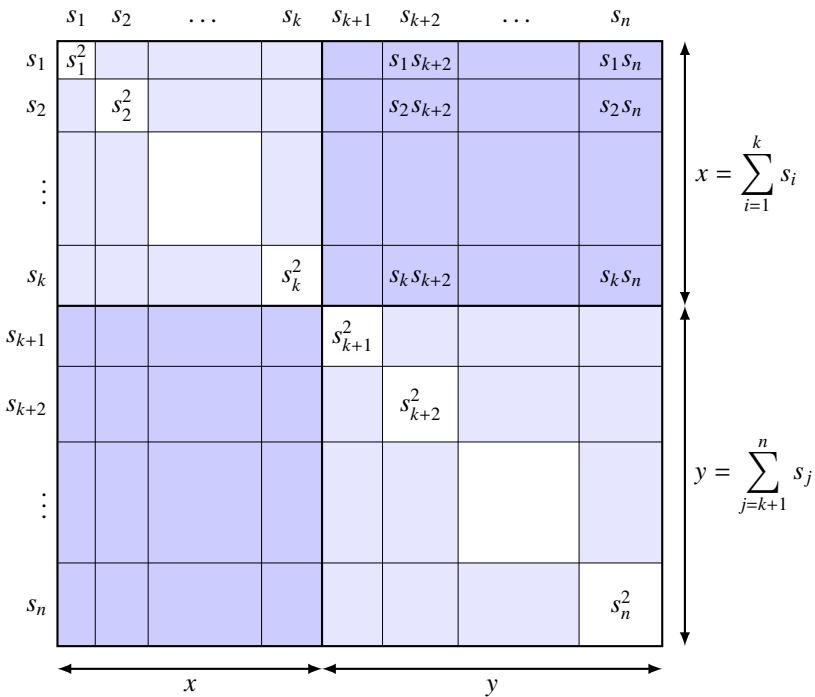
$$\varphi(V_1, V_2) = \sum_{v_1 \in V_1, v_2 \in V_2, e=\{v_1, v_2\} \in E} \varphi(e) \geq w?$$

**Satz 13.46.** *MAX-CUT* ist NP-vollständig.

*Beweis.* Das *PARTITION*-Problem kann wie folgt auf *MAX-CUT* reduziert werden. Aus den Zahlen  $s_1, \dots, s_n$  wird der vollständige Graph mit den Knoten  $V = \{1, \dots, n\}$  und den Kantengewichten

$$\varphi(\{i, j\}) = s_i \cdot s_j$$

konstruiert. Eine Lösung des *PARTITION*-Problems zerlegt die Liste der Zahlen in zwei Teillisten. Wir dürfen ohne Beschränkung der Allgemeinheit annehmen, dass die Zahlen

Abbildung 13.23: Reduktion von *PARTITION* auf *MAX-CUT*.

so nummeriert sind, dass die beiden Teillisten die ersten  $k$  Zahlen und die letzten  $n - k$  Zahlen sind. Die zugehörige Bewertung ist die Summe

$$d = \sum_{i=1}^k \sum_{j=k+1}^n \varphi(\{i, j\}) = \sum_{i=1}^k \sum_{j=k+1}^n s_i s_j,$$

die sich gemäß Abbildung 13.23 als halber Flächeninhalt der dunkelblau gefärbten Flächen

$$= \underbrace{\left( \sum_{i=1}^k s_i \right)}_{=x} \underbrace{\left( \sum_{j=k+1}^n s_j \right)}_{=y} = xy$$

berechnen lässt. Außerdem ist  $x + y = c$ . Das Maximum von  $x(c - x)$  wird erreicht im Intervallmittelpunkt bei  $x = c$  und erreicht dort den Wert  $\frac{1}{4}c^2$ . Wenn man also verlangt, dass  $d \geq \frac{1}{4}c^2$  sein soll, dann gibt es nur dann eine Lösung für das *MAX-CUT*-Problem, wenn es auch eine Lösung für das *PARTITION*-Problem gibt. Da  $d$  eine ganze Zahl ist, ist dies gleichbedeutend mit der Forderung, dass

$$d \geq w = \left\lceil \frac{1}{4}c^2 \right\rceil = \left\lceil \frac{1}{4} \left( \sum_{i=1}^n s_i \right)^2 \right\rceil.$$

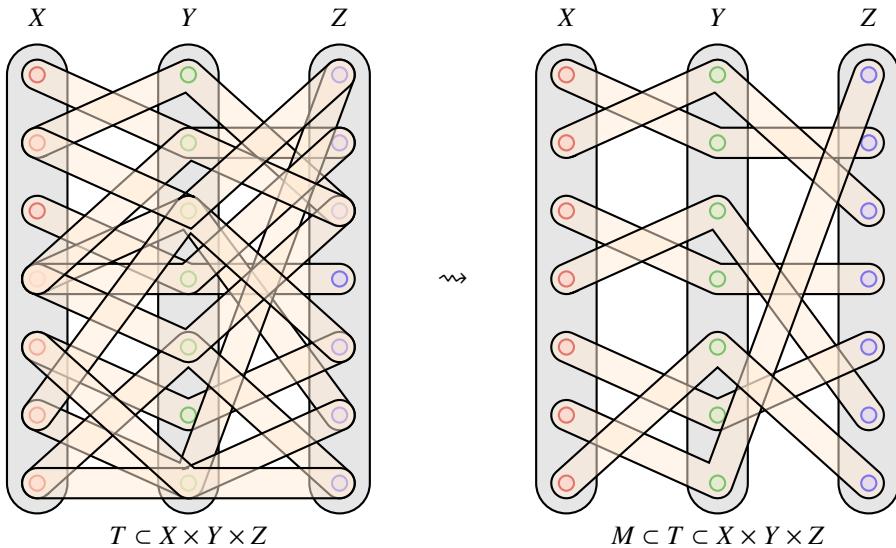


Abbildung 13.24: *3D-MATCHING* ist das Problem aus einer Menge von Tripeln eine Auswahl zu treffen, so dass jeder Punkt der drei Mengen  $X$ ,  $Y$  und  $Z$  in genau einem Tripel vorkommt. Nimmt man an, dass die drei Mengen wie dargestellt disjunkt sind, kann man das *3D-MATCHING*-Problem auch als ein *EXACT-COVER*-Problem betrachten, wobei die Tripel als dreielementige Teilmengen der disjunkten Vereinigung von  $X$ ,  $Y$  und  $Z$  interpretiert werden können.

Mit diesem Wert  $w$  ist die Reduktion auf *MAX-CUT* vollständig. □

### 13.4.8 3D-MATCHING

Im klassischen Heiratsproblem sind  $n$  Frauen  $f_i$  und  $n$  Männer  $m_k$  gegeben sowie eine Menge von Paaren  $(f_i, m_k)$ , für die eine erfolgreiche Heirat erwartet werden kann. Gesucht ist eine Auswahl der Paarungen, so dass jede Frau und damit auch jeder Mann verheiratet sind. Für dieses Problem existieren effiziente Algorithmen.

3D-Matching ist eine Erweiterung des klassischen Heiratsproblems. Neben den  $n$  Frauen und  $n$  Männern gibt es jetzt auch noch  $n$  Wohnungen  $w_l$ , in denen die Paare leben könnten. Statt einer Menge von Paaren ist jetzt eine Menge von Tripeln  $(f_i, m_k, w_l)$  von erfolgversprechenden Paaren und einer dem Paar zusagenden Wohnung gegeben. Gefragt wird jetzt, ob es eine Auswahl der Tripel gibt, so dass jede Frau und jeder Mann verheiratet wird und jede Wohnung mit genau einem Paar bewohnt wird, dem die Wohnung auch gefällt.

Die folgende, abstrakte Formulierung ist auch in Abbildung 13.24 visualisiert.

**Problem 13.47 (3D-MATCHING).** Gegeben sind drei endliche Mengen  $X$ ,  $Y$  und  $Z$  mit gleich vielen  $n = |X| = |Y| = |Z|$  Elementen und  $T \subset X \times Y \times Z$ . Gibt es eine  $n$ -elementige Teilmenge  $M \subset T$  derart, dass keine zwei Tripel von  $M$  in einer Koordinate übereinstimmen?

Die Abbildung 13.24 zeigt links die Menge der Tripel  $T$ , einige der Punkte in  $X$ ,  $Y$  und  $Z$  kommen in mehr als einem Tripel vor. Rechts ist nur noch die Menge  $M$  dargestellt, jeder Punkt kommt jetzt nur noch in genau einem Tripel vor.

Die Abbildung stellt die Mengen  $X$ ,  $Y$  und  $Z$  als disjunkte Mengen dar. Ein Tripel  $(x, y, z) \in T$  kann daher auch als Menge  $\{x, y, z\} \subset X \cup Y \cup Z$  betrachtet werden. Ein Match  $M$  liegt vor, wenn jeder Punkt genau in einer der dreielementigen Mengen liegt. Das 3D-MATCHING-Problem ist also ein EXACT-COVER Problem. Tatsächlich konstruiert [25] eine Reduktion von EXACT-COVER auf 3D-MATCHING.

### 13.4.9 Kombinatorische Optimierungsprobleme

Die Probleme STEINER-TREE und SEQUENCING können als Beispiele für die große Klasse sogenannter kombinatorischer Optimierungsprobleme dienen, in denen NP-Vollständigkeit sehr häufig auftritt. Das berühmte Problem des Handelsreisenden gehört auch dazu. Der Handelsreisende soll eine große Zahl von Punkten besuchen und dabei eine möglichst kleine Strecke zurücklegen. Solche Probleme treten in vielen praktischen Anwendungen auf. In der Elektronik muss in einer Leiterplatte eine große Zahl von Löchern gebohrt werden. Die Bohrmaschine arbeitet am produktivsten, wenn die Zeit für die Bewegung des Bohrkopfes von Loch zu Loch möglichst klein gehalten werden kann. Dies ist eine Variante des Problems des Handelsreisenden. Vielfältige Methoden sind zur Lösung derartiger Probleme entwickelt worden [26].

In Optimierungsproblemen, die man mit den Methoden der Analysis zu lösen lernt, sind reelle Variablen stetig zu variieren, bis das Maximum oder Minimum einer Zielfunktion erreicht wird. Die Ableitung stellt sich als sehr effizientes Werkzeug zur Lösung solcher Aufgaben heraus. In kombinatorischen Optimierungsproblemen sind es jedoch diskrete Variablen, ganze Zahlen, Permutationen und weitere, die verändert werden können, um eine Zielfunktion zu maximieren oder zu minimieren. Die Anzahl der möglichen Einstellungen dieser Variablen wächst, wie in kombinatorischen Problemen nicht unerwartet, exponentiell mit der Größe des Inputs. Es stellt sich heraus, dass viele kombinatorische Optimierungsprobleme NP-vollständig sind, wenn man sie als Entscheidungsprobleme formuliert.

#### STEINER-TREE

**Problem 13.48 (STEINER-TREE).** Gibt es zu einem Graphen  $G = (V, E)$  mit einer Gewichtsfunktion  $w: E \rightarrow \mathbb{Z}$ , einer Teilmenge  $R \subset V$  der Knoten und einer Zahl  $k$  einen Teilbaum  $T = (V_1, E_1) \subset G$  des Graphen, der alle Knoten von  $R$  enthält und dessen Kantengewicht

$$\sum_{e \in E_1} w(e) \leq k$$

ist?

Dieses Problem hat eine gewisse Bedeutung in der Netzwerktechnik. Man kann sich die Gewichtsfunktion als die Kosten oder den Zeitaufwand für den Transport von Netzwerkpaketen entlang einzelner Teilstrecken eines Kommunikationsnetzes vorstellen. Ein

Baum wie in 13.48 transportiert die Pakete auf eindeutig bestimmten Wegen und zu geringen Kosten.

Die Aufgabenstellung ist nicht als Frage nach dem “optimalen Baum” formuliert. Sie fragt danach, ob es möglich ist, einen Wert  $k$  mit der Zielfunktion zu unterschreiten. Dies ist auf die Einschränkung zurückzuführen, dass wir im Rahmen der entwickelten Theorie nur über Entscheidungsprobleme sprechen können. Kann man die Frage effizient beantworten, dann kann man durch wiederholte Lösung des Entscheidungsproblems auch herausfinden, ob eine Lösung in einem Intervall möglich ist. Durch fortgesetzte Intervallhalbierung kann man dann auch den exakten Wert des Minimums bestimmen. Wäre das Entscheidungsproblem in polynomieller Zeit lösbar, dann wäre auch die Optimierungsaufgabe in polynomieller Zeit lösbar. Wir verlieren als nichts, wenn wir uns nur auf die Frage konzentrieren, ob das Entscheidungsproblem NP-vollständig ist.

## **SEQUENCING**

Viele Planungsaufgaben, wie sie zum Beispiel auch beim Scheduling von Prozessen in der Datenverarbeitung gelöst werden müssen, führen ebenfalls auf kombinatorische Optimierungsprobleme.

**Problem 13.49 (SEQUENCING).** *Die Komponenten der drei Vektoren*

$$\text{Laufzeiten: } T = (T_1, \dots, T_p) \in \mathbb{Z}^p$$

$$\text{Deadlines: } D = (D_1, \dots, D_p) \in \mathbb{Z}^p$$

$$\text{Strafen: } P = (P_1, \dots, P_p) \in \mathbb{Z}^p$$

beschreiben je einen Job. Job  $i$  hat Laufzeit  $T_i$ , sollte zur Zeit  $D_i$  abgeschlossen sein und kostet die Strafe  $P_i$ , wenn er nicht rechtzeitig fertig ist. Die Jobs sollen nacheinander laufen. Sobald ein Job fertig ist, wird der nächste gestartet. Ist es möglich, die Reihenfolge der Ausführung der Jobs so zu planen, dass die entstehenden Strafen  $\leq k$  sind?

**Satz 13.50.** STEINER-TREE und SEQUENCING sind NP-vollständig.

## **Übungsaufgaben**

**13.1.** Ein Verkehrsnetz soll regelmäßig durch Mitarbeiter kontrolliert werden, die ihre Basis an einzelnen Knotenpunkten des Netzes haben. Kann man auf effiziente Art und Weise herausfinden, an welchen Knotenpunkten man Kontrolleure stationieren muss, damit jede Strecke in einem Knoten mit Kontrolleur endet?

**13.2.** Für eine Gruppenarbeit sollen  $k$  Gruppen gebildet werden. Um die Zeit für das gegenseitige Kennenlernen möglichst kurz zu halten, sollen sich die Leute einer Gruppe bereits gegenseitig kennen. Alle Leute sollen beschäftigt sein. Können Sie einen effizienten Algorithmus formulieren, mit dem eine solche Gruppeneinteilung auch bei einer großen Teilnehmerzahl gefunden werden kann?

**13.3.** Ein aufstrebendes Film-Festival ist derart gewachsen, dass der Vorführsaal nicht mehr reicht. Daher müssen jetzt zwei gleich große Säle verwendet werden, und trotzdem

ist das Festival wieder ausverkauft, und zwar in einem Maße, dass überhaupt nur Stars und Prominente samt ihrer Entourage eingelassen werden können. Für einzelne Besucher gibt es keine Plätze.

Doch die Stars stören sich daran, dass sie möglicherweise nicht ihre ganze Entourage im gleichen Saal haben können. Daher muss kurzfristig eine Aufteilung der Festival-Gäste gefunden werden, so dass die beiden Säle so gefüllt werden können, dass jede Entourage vollständig in einem der Säle Platz nimmt.

Der Festival-Direktor ist jedoch sehr überrascht, dass die Bestimmung einer solchen Aufteilung so lange dauert. Warum sind Sie nicht überrascht?

**13.4.** In einem Entwicklungsland sollen die aus dem Ausland erhaltenen Unterstützungsmittel dazu verwendet werden, endlich alle Ortschaften mit mindestens 100 Einwohnern ans Stromnetz anzuschließen. Der Bau von Leitungen zwischen einzelnen Ortschaften ist je nach Gelände unterschiedlich teuer, zum Teil auch schlicht unmöglich. Es wird entschieden, dass man in einer ersten Phase auf Redundanz des neu zu erstellenden Netzes verzichten will. Der Minister möchte endlich wissen, ob das vorhandene Geld für das Projekt ausreicht und ist sehr ungedhalten darüber, dass die Verwaltung so lange braucht, diese Frage zu beantworten. Kann man dies erklären?

**13.5.** Für eine medizinische Studie ist eine große Zahl von Probanden rekrutiert worden. Sie sind bereits auf Allergien getestet worden, man weiß also von jedem Probanden, auf welche Allergene (Pollen, Katzenhaare, Hausstaub, Lactose,...) er allergisch reagiert. Die Untersuchung soll sich auf eine Teilmenge von  $k = 17$  oder noch mehr ausgewählten Allergenen beschränken, die so beschaffen ist, dass kein Proband auf mehr als eines der ausgewählten Allergene reagiert. Es stellt sich als schwierig heraus, eine solche Teilmenge zu finden. Warum?

**13.6.** Die Prüfungsvorbereitungszeit ist intensiv, manchmal reicht die Zeit nicht, alle Prüfungen vorzubereiten, bis die Prüfungssession beginnt. Es ist daher unumgänglich, während der Prüfungssession weiter zu lernen. Nehmen wir an, dass für die Vorbereitung der  $p$  Fächer  $1, \dots, p$  die Vorbereitungszeiten  $t_1, \dots, t_p$  notwendig sind. Ebenfalls bekannt ist, dass die Prüfungen zu den Zeiten  $d_1, \dots, d_p$  stattfinden, dann muss die Vorbereitung abgeschlossen sein, weil die Prüfung sonst nicht zu bestehen ist. Es stellt sich daher die Frage, in welcher Reihenfolge die Vorbereitungen durchgeführt werden sollen, damit höchstens  $k$  Prüfungen nicht bestanden werden. Ein Student, der *Automaten und Sprachen* nicht studiert hat, wendet viel Zeit dafür auf herauszufinden, ob es überhaupt eine Reihenfolge dieser Art gibt, versteht aber nicht, warum dies so aufwendig ist. Können Sie dies erklären?

Lösungen: <https://autospr.ch/uebungen/AutoSpr-113.pdf>



## Kapitel 14

# Programmiersprachen und Turing-Vollständigkeit

Turing-Maschinen sind unser bevorzugtes Modell für Berechnungen, die mit Computern angestellt werden können. Programmierer arbeiten jedoch nicht mit Turing-Maschinen und ihren Zustandsdiagrammen, sondern mit höheren Programmiersprachen. Diese sind prozessorunabhängig und verbergen die Eigenheiten der Hardware und des Betriebssystems weitgehend. Der Programmierer sieht eine einheitliche Schnittstelle und kann sich auf die Lösung des Anwendungsproblems konzentrieren. Es stellt sich jedoch die Frage, ob durch diese Abstraktion auch Fähigkeiten oder Eigenschaften einer Turing-Maschine verloren gehen. Gibt es Probleme, die eine Turing-Maschine lösen kann, die aber in einer höheren Programmiersprache nicht oder nicht effizient gelöst werden können?

### 14.1 Turing-Vollständigkeit

Turing-Vollständigkeit ist die Eigenschaft, die garantiert, dass sich in einer Programmiersprache alles machen lässt, was man auch mit einer Turing-Maschine machen kann. Dieser Abschnitt erklärt, was eine Programmiersprache ist und stellt die Verbindung zu Turing-Maschinen her.

#### 14.1.1 Programmiersprachen

Moderne Computer sind Allzweckmaschinen. In Kapitel 10 wurde argumentiert, dass alles, was sich berechnen lässt, auch mit einer Turing-Maschine berechnet werden kann. In der informatischen Praxis werden aber nicht Turing-Maschine verwendet. Programme werden in höheren Programmiersprachen geschrieben. Um die Fähigkeiten einer Programmiersprache mit den Fähigkeiten einer Turing-Maschine zu vergleichen, muss erst definiert werden, was eine Programmiersprache ist.

Wir nehmen dazu an, dass die zugrundeliegende Hardware eine universelle Turing-Maschine ist. Zu dieser Annahme sind wir durch den Erfolg moderner Prozessoren, denen keine Aufgabe zu schwierig zu sein scheint, durchaus berechtigt.

Eine Programmiersprache wird nach Abschnitt 1.1.4 durch den Compiler definiert, der die Sprache akzeptiert. Zusätzlich erzeugt der Compiler aber auch den ausführbaren Maschinencode oder im Falle von Java den Bytecode.

**Definition 14.1** (Programmiersprache und Compiler). *Eine Programmiersprache ist eine Turing-erkennbare Sprache  $L \subset \Sigma^*$  und eine berechenbare Abbildung  $c: L \rightarrow \Sigma^*$  so, dass  $c(w)$  ein Programm für eine universelle Turing-Maschine ist. Die Abbildung  $c$  heißt der Compiler für die Programmiersprache.*

Die Benennung von  $c$  als der Compiler für die Sprache schließt nicht aus, dass die Sprache interpretiert wird. Auch ein Interpreter für die Sprache wählt für jede interpretierte Anweisung laufend den Maschinencode, der die Anweisung realisiert. Der Interpreter tut dies im Unterschied zum Compiler erst zur Laufzeit.

Einige Sprachen erzeugen nicht direkt Maschinencode, sondern Code für eine virtuelle Maschine. Die erste solche Sprache war BCPL, deren Compiler im ersten Schritt sogenannten O-code erzeugte, eine Zwischensprache, die dann im zweiten Schritt in Maschinencode übersetzt wurde. Da O-code viel einfacher zu kompilieren ist, wurde die Aufgabe, den Compiler auf eine neue Plattform zu portieren, wesentlich vereinfacht.

UCSD-Pascal und Java gehen noch einen Schritt weiter. Der Compiler erzeugt p-Code bzw. Java-Bytecode, der von einer virtuellen Maschine interpretiert wird. Dies ermöglicht, dass sogar die kompilierten Programme auf verschiedenen Architekturen laufen können, eine Eigenschaft die Sun Microsystems mit dem Slogan “Write once, run everywhere” vermarktet hat. p-Code oder Bytecode-Programme können auch kompakter sein. Vor allem in kurzen Schleifen und oft wiederholten Programmteilen kann sich der zusätzliche Aufwand der Interpretation des Codes merklich auf die Laufzeit auswirken. Ein Just-in-Time-Compiler (JIT-Compiler) übersetzt Teile des Bytecodes während der Ausführung in Maschinencode und optimiert so die Laufzeit. JIT-Compiler werden zum Beispiel auch bei Python und JavaScript eingesetzt.

In allen Fällen wird aus dem ursprünglichen Programmtext möglicherweise in mehreren Stufen, auf verschiedenen Wegen und zu verschiedenen Zeiten Code erzeugt oder ausgewählt, der der Maschine zur Ausführung übergeben werden kann. Die berechenbare Abbildung  $c$  von Definition 14.1 fasst alle diese Schritte zu einem Schritt zusammen.

## 14.1.2 Turing-Vollständigkeit

Die Programmiersprache  $L$  ermöglicht, Programme für die universelle Turing-Maschine zu schreiben. Es ist aber unklar, ob sich in der Programmiersprache eine Problemlösung für jedes beliebige, mit einer Turing-Maschine lösbarer Problem programmieren lässt.

**Definition 14.2** (Turing-vollständig). *Die Programmiersprache  $L$  heißt Turing-vollständig, wenn jede Sprache, die von einer Turing-Maschine entschieden werden kann, auch von einem in der Sprache  $L$  geschriebenen Programm entschieden werden kann.*

Statt eine neue Theorie über Programmiersprachen aufzubauen, reicht es, in der Programmiersprache einen Turing-Maschinen-Simulator zu schreiben. Da ein solcher Simulator eine beliebige Turing-Maschinen-Beschreibung lesen und ausführen kann, folgt aus seiner Existenz sofort, dass die Sprache Turing-vollständig ist.

**Satz 14.3.** *Falls in der Sprache L ein Turing-Maschinen-Simulator geschrieben werden kann, ist die Sprache L Turing-vollständig.*

Ein Hello-World-Programm mag also interessant sein, um einen ersten Eindruck von einer Programmiersprache zu erhalten. Viel mehr Einsicht in eine Programmiersprache würde man allerdings durch einen Turing-Maschinen-Simulator als erstes Programm in der neuen Sprache erhalten.

### 14.1.3 Moderne Programmiersprachen

Es ist nicht überraschend, dass die modernen Programmiersprachen Turing-vollständig sind. Zum Nachweis müsste allerdings in jedem Fall ein Turing-Maschinen-Simulator in der betreffenden Sprache geschrieben werden. Manchmal findet sich jedoch eine Abkürzung, wie die folgenden Beispiele zeigen.

#### C und C++

Der bereits im Kapitel 10 vorgestellte Turing-Maschinen-Simulator ist in der Programmiersprache C geschrieben. Mit dem Simulator lässt sich jedes beliebige Turing-Maschinenprogramm ausführen. Damit ist die Bedingung von Satz 14.3 erfüllt und es folgt, dass die Sprache C Turing-vollständig ist. Da die Sprache C++ eine Erweiterung von C ist, folgt auch, dass C++ Turing-vollständig ist.



#### JavaScript

Fabrice Bellard stellt auf [5] einen x86-Emulator in JavaScript bereit, der in der Lage ist, ein Linux-System zu booten. Mit der eingebauten File-Transferfunktion kann man den C-Sourcecode des Turing-Maschinensimulators hochladen und den Simulator kompilieren. Damit lässt sich jedes beliebige Turing-Maschinenprogramm innerhalb einer JavaScript-Engine ausführen, wie auch Abbildung 14.1 zeigt. Damit folgt, dass auch JavaScript Turing-vollständig ist.



#### XSLT

Die *extensible stylesheet language for transformations* (XSLT) ist eine Programmiersprache zur Transformation von XML-Dokumenten. Sie ermöglicht sogenannte Templates zu definieren, die wie Funktionen aufgerufen werden können. Sie ermöglichen funktionale Programmierung und es stellt sich die Frage, ob sie Turing-vollständig ist. Die Arbeit [44] von Onder und Bayram schlägt ein XSL-Stylesheet vor, welches eine Turing-Maschinen-Beschreibung als XML-Dokument in die Berechnungsgeschichte transformiert. Damit ist auch XSLT als Turing-vollständig nachgewiesen.

```

Loading...

Welcome to JS/Linux (x86)

Use 'vflogin username' to connect to your account.
You can create a new account at https://vfsync.org/signup .
Use 'export_file filename' to export a file to your computer.
Imported files are written to the home directory.

[root@localhost ~]# zcat turing-1.0.tar.gz | tar xf -
[root@localhost ~]# cd turing-1.0/
[root@localhost turing-1.0]# make
gcc -o turing -g -Wall -O -std=c99 turing.c
turing.c: In function 'tm_run':
turing.c:484:5: warning: implicit declaration of function 'usleep'; did you mean
'sleep'? [-Wimplicit-function-declaration]
    usleep(delay);
    ^
sleep
[root@localhost turing-1.0]# ls -l
total 72
-rw-r--r-- 1 1000 20 1250 Apr 25 15:09 Makefile
-rw-r--r-- 1 1000 20 685 Apr 25 15:09 README
drwxr-xr-x 2 1000 20 165 Apr 25 15:09 data
drwxr-xr-x 2 1000 20 154 Apr 25 15:09 pgm
-rwxr-xr-x 1 root root 39408 Apr 25 15:13 turing
-rw-r--r-- 1 1000 20 15026 Apr 25 15:09 turing.c
[root@localhost turing-1.0]#

```

Paste Here

© 2011-2019 Fabrice Bellard - [News](#) - [VM list](#) - [FAQ](#) - [Technical notes](#)

Abbildung 14.1: Der von Fabrice Bellard auf [5] bereitgestellte x86-Emulator in JavaScript ist in der Lage, den Turing-Maschinensimulator in C auszuführen. Dies beweist, dass JavaScript Turing-vollständig ist.

## LATEX

Das Textformatierungssystem TEX, mit dem auch dieses Buch gesetzt worden ist, verwendet eine Turing-vollständige Makrosprache. Der Beweis der Turing-Vollständigkeit kann auf verschiedene Weise geführt werden. Die Webseite [60] definiert eine Reihe von LATEX-Makros, die eine Turing-Maschinendefinition in die als PDF-Dokument formatierte Berechnungsgeschichte umwandeln.

Die Website [4] implementiert einen Emulator für den AVR Mikrokontroller von Atmel als TEX-Makros. Damit eröffnet sich eine weitere Möglichkeit, eine beliebige Turing-Maschine in TEX auszuführen. Da es für den AVR-Prozessor einen C-Compiler gibt, dürfen wir annehmen, dass er eine universelle Turing-Maschine implementiert. Mit dem C-Compiler kann man den Turing-Maschinen-Simulator in C in AVR-Maschinencode umwandeln und schließlich mithilfe von [4] vom TEX-Prozessor ausführen lassen. Damit folgt

wieder, dass T<sub>E</sub>X Turing-vollständig ist.

## 14.2 Die Sprache LOOP

Die modernen Programmiersprachen haben sich in Abschnitt 14.1.3 alle als Turing-vollständig herausgestellt. Allerdings sind moderne Programmiersprachen auch vollgepackt mit Features, die es dem Programmierer möglichst leicht machen sollen, seine Algorithmen zu formulieren. Welche Eigenschaften einer Programmiersprache sind dafür verantwortlich, dass die Sprache Turing-vollständig ist? In diesem Abschnitt wird eine Programmiersprache konstruiert, die über eine vollständige Arithmetik, einen unendlich großen Speicher und eine Schleifenkonstruktion verfügt, aber trotzdem nicht Turing-vollständig ist. Es werden auch Beispiele von Situationen gezeigt, wo diese Eigenschaft nutzbringend angewendet worden ist.

### 14.2.1 Grundelemente

Die konstruierte Sprache soll natürliche Zahlen verarbeiten können. Zu diesem Zweck gibt es Variablen  $x_0, x_1, x_2 \dots$ , denen mit der Zuweisungsoperation  $x_k := c$  beliebige konstante Werte  $c \in \mathbb{N}$  zugewiesen werden können. Es ist aber nicht vorgesehen, dass der Wert einer Variable einer anderen Variablen zugewiesen werden kann. Es gibt also keine Zuweisungsanweisung  $x_k := x_l$ , welche den Wert von  $x_l$  in  $x_k$  speichert.

Die Sprache soll Berechnungen mit natürlichen Zahlen ausdrücken können. Dazu steht die Additionsanweisung  $x_k := x_l + c$  für beliebige Konstanten  $c \in \mathbb{N}$  zur Verfügung, die den Wert von  $x_l$  um  $c$  vermehrt und in  $x_k$  abspeichert. Die fehlende Variablenzuweisung kann dadurch emuliert werden, dass die Konstante 0 addiert wird:  $x_k := x_l + 0$ . Weiter kann mit  $x_k := x_l - c$  ein konstanter Wert subtrahiert werden. Falls  $c > x_l$  ist, wird der Variablen  $x_k$  der Wert 0 zugewiesen, damit das Resultat in  $\mathbb{N}$  bleibt.

Ein Programm in der bisher definierten Sprache ist eine Folge von Anweisungen, die der Reihe nach ausgeführt werden. Das Programm

$$\begin{aligned} x_0 &:= 2 \\ x_0 &:= x_0 + 3 \\ x_0 &:= x_0 + 5 \\ x_0 &:= x_0 + 7 \\ x_0 &:= x_0 + 11 \\ x_0 &:= x_0 - 100 \end{aligned} \tag{14.1}$$

berechnet die Summe der ersten fünf Primzahlen und subtrahiert 100 vom Resultat. Wegen des Unterlaufes wird  $x_0$  am Ende den Wert 0 haben.

Es sind keine weiteren arithmetischen Operationen und bis jetzt auch keine Kontrollstrukturen vorgegeben. Mit den unendlich vielen Variablen  $x_i$  steht der Sprache zwar ein im Prinzip unendlich großer Speicher zur Verfügung, der die Rolle des Bandes einer Turing-Maschine übernehmen könnte, aber es fehlt die Fähigkeit, Instruktionen bedingt und immer wieder auszuführen. Die Grundelemente der Sprache sind also noch nicht Turing-vollständig.

### 14.2.2 Die LOOP-Kontrollstruktur

Turing-Vollständigkeit verlangt eine Schleifenstruktur, die ermöglicht, Code immer wieder auszuführen, bis eine geeignete Abbruchbedingung erfüllt ist. Daher wird jetzt die folgende LOOP-Kontrollstruktur hinzugefügt. Der Code

```
LOOP  $x_i$  DO
    P
END
```

führt das Teilprogramm  $P$  genauso oft aus, wie die Variable  $x_i$  beim Eintritt in die Schleife angibt. Der Wert von  $x_i$  wird vor der ersten Ausführung von  $P$  ermittelt. Eine Veränderung des Wertes von  $x_i$  innerhalb von  $P$  hat keinen Einfluss auf die Anzahl der Ausführungen von  $P$ . Die entstehende Sprache heißt LOOP.

Trotz der Einschränkungen lassen sich viele neue Funktionen aus LOOP ableiten. Die folgenden Beispiele sind zum Teil alles andere als effizient. Es geht aber auch nicht primär um Effizienz, sondern um die Frage, ob eine bestimmte Funktion mit der Sprache überhaupt realisierbar ist.

#### Addition

Eine allgemeine Additionsoperation für zwei Variablen wie in  $x_i := x_k + x_l$  gibt es nicht. Sie kann aber mit der LOOP-Iteration durch den Code

```
 $x_i := x_k$ 
LOOP  $x_l$  DO
     $x_i := x_i + 1$ 
END
```

(14.2)

nachgebildet werden. Der Code (14.2) addiert  $x_l$ -mal eine Eins zum Initialwert  $x_k$  von  $x_i$ , so dass am Ende des Programms in der Variablen  $x_i$  der Wert der Summe  $x_k + x_l$  steht.

Auf ähnliche Weise kann auch eine Subtraktionsfunktion  $x_i := x_k - x_l$  konstruiert werden.

#### Multiplikation

Die Multiplikation ist wiederholte Addition und sollte daher mit der Additionsoperation des vorangegangenen Abschnitts realisierbar sein. Tatsächlich berechnet der Code

```
 $x_i := 0$ 
LOOP  $x_k$  DO
     $x_i := x_i + x_l$ 
END
```

$\Rightarrow$

```
 $x_i := 0$ 
LOOP  $x_k$  DO
    LOOP  $x_l$  DO
         $x_i := x_i + 1$ 
    END
END
```

(14.3)

das Produkt  $x_i := x_k * x_l$  als  $x_k$ -fach wiederholte Addition von  $x_l$  zu  $x_i$  (links). Die Addition muss aber ihrerseits gemäß (14.2) in eine wiederholte Addition von Einsen aufgelöst werden (rechts).

## Bedingte Ausführung

Eine IF-Anweisung macht die Ausführung eines Teilprogramms von einer Bedingung abhängig. So möchte man, dass in

```
IF  $x_i$  THEN
    P
END
```

der Programmteil  $P$  nur dann ausgeführt wird, wenn  $x_i > 0$  ist. Mit der LOOP-Operation kann dies als

```
y := 0
LOOP  $x_i$  DO
    y := 1
END
LOOP y DO
    P
END
```

realisiert werden. Die erste LOOP-Schleife setzt  $y$  auf 1, wenn  $x_i > 0$  ist, sogar mehrfach, wenn  $x_i > 1$  ist. Die zweite Schleife führt dann  $P$  genau  $y$  mal aus.

Es ist plausibel, dass sich mithilfe dieser Operationen auch die übrigen arithmetischen Operationen wie die ganzzahlige Division mit Rest realisieren lässt. Trotzdem ist die Sprache LOOP noch sehr beschränkt, wie Abschnitt 14.2.3 zeigen wird.

**Verständniskontrolle 14.1:** In der Sprache LOOP können verschiedene einfache Operationen realisiert werden. Zum Beispiel wurde eine Subtraktion  $x_a := x_b - x_c$  realisiert, welche allerdings negative Resultate immer in 0 umwandelt. Lösen Sie im gleichen Sinne die folgenden Aufgaben:



autosp.ch/v/14.1.pdf

- Schreiben Sie ein LOOP-Programm, welches die Werte der Variablen  $x_i$  und  $x_k$  vergleicht und ein Programm  $P$  ausführt, wenn  $x_i \leq x_k$  ist.
- Schreiben Sie ein LOOP-Programm, welches den Rest der Division der Zahlenwerte in den Registern  $x_i$  und  $x_k$  berechnet.

## Unentgeltliche Anweisungen

Die Resultate dieses Abschnitts haben gezeigt, dass mit der LOOP-Kontrollstruktur eine ganze Menge neuer Operationen einhergehen, die wir in Zukunft jeweils stillschweigend voraussetzen werden. Um die Programme etwas kompakter schreiben zu können, werden wir in Zukunft auch erlauben, mehrere Anweisungen durch Strichpunkt getrennt auf einer Zeile zu schreiben. Es folgt eine Zusammenstellung von solchen “unentgeltlichen” Operationen

- Addition und Subtraktion von beliebigen Variablenwerten, realisiert wie in (14.2).
- Multiplikation von beliebigen Variablenwerten, ganzzahlige Division mit Rest, realisiert wie in (14.3) und Verständniskontrolle 14.1.

3. Bedingte Anweisungen, die genau dann ausgeführt werden, wenn eine Bedingung der Form  $x_i \leq x_k$  erfüllt ist (Verständniskontrolle 14.1, a)). Durch geschachtelte Prüfung, ob auch  $x_k \leq x_i$  ist, lässt sich daraus eine Bedingung der Form  $x_i = x_k$  konstruieren. Durch vorgängige Zuweisung einer Konstanten  $c$  zu einer Hilfsvariablen, lassen sich auch bedingte Anweisungen für  $x_i = c$ ,  $x_i \leq c$  oder  $x_i \geq c$  ableiten.
4. Negation bedingter Anweisungen: Mit einer Hilfsvariablen  $y$ , die beim Eintreten der Bedingung auf 0 gesetzt wird, lässt sich die Negation realisieren:

$$\left. \begin{array}{c} \text{IF } \neg \text{Bedingung DO} \\ \quad P \\ \quad \text{END} \end{array} \right\} \quad \text{realisieren als} \quad \left\{ \begin{array}{l} y := 1 \\ \text{IF Bedingung DO} \\ \quad y := 0 \\ \quad \text{END} \\ \text{LOOP } y \text{ DO } P \text{ END} \end{array} \right.$$

5. Auch eine IF-THEN-ELSE-Anweisung kann mit zwei Hilfsvariablen wie folgt umgesetzt werden:

$$\left. \begin{array}{c} \text{IF Bedingung DO} \\ \quad P \\ \quad \text{ELSE} \\ \quad Q \\ \quad \text{END} \end{array} \right\} \quad \text{realisieren als} \quad \left\{ \begin{array}{l} \text{IF Bedingung DO} \\ \quad i := 0; \quad e := 1 \\ \quad \text{END} \\ \text{IF } \neg \text{Bedingung DO} \\ \quad i := 1; \quad e := 0 \\ \quad \text{END} \\ \text{LOOP } i \text{ DO } P \text{ END} \\ \text{LOOP } e \text{ DO } Q \text{ END} \end{array} \right.$$

Die Lösung scheint auf den ersten Blick vielleicht kompliziert, man könnte doch  $P$  gleich innerhalb der ersten IF-Struktur ausführen. Dies stimmt aber nicht, denn  $P$  könnte ändern, ob die Bedingung zutrifft. Es könnte daher passieren, dass sowohl  $P$  als auch  $Q$  ausgeführt werden.

6. Aus der IF-THEN-ELSE-Anweisung erwächst durch Schachtelung auch eine Mehrfachverzweigung, die wir als SWITCH-CASE-Anweisung schreiben:

$$\left. \begin{array}{c} \text{SWITCH } x_i \\ \quad \text{CASE } c_1 : P_1 \text{ END} \\ \quad \text{CASE } c_2 : P_2 \text{ END} \\ \quad \text{CASE } c_3 : P_3 \text{ END} \\ \quad \text{END} \end{array} \right\} \quad \text{realisieren als} \quad \left\{ \begin{array}{l} \text{IF } x_i := c_1 \text{ DO} \\ \quad P_1 \\ \quad \text{ELSE} \\ \quad \quad \text{IF } x_i = c_2 \text{ DO} \\ \quad \quad \quad P_2 \\ \quad \quad \quad \text{ELSE} \\ \quad \quad \quad \quad \text{IF } x_i = c_3 \text{ DO } P_3 \text{ END} \\ \quad \quad \quad \quad \text{END} \\ \quad \quad \quad \text{END} \end{array} \right.$$

Die LOOP-Sprache scheint fast alles zu haben, was man in einer typischen Programmiersprache erwartet. Es fehlt aber eine unbeschränkte Schleifenkonstruktion wie die while-Schleife in C oder JavaScript.

### 14.2.3 Das Halteproblem für LOOP

Für Turing-Maschinen gilt, dass die Frage, ob eine Turing-Maschine auf einem gegebenen Inputwort anhalten wird, nicht entscheidbar ist. Da in einer Turing-vollständigen Sprache jedes beliebige Turing-Maschinenprogramm simuliert werden kann, kann auch nicht entschieden werden, ob ein in dieser Sprache geschriebenes Programm anhalten wird. Für die LOOP-Sprache finden wir aber, dass das Halteproblem entscheidbar ist.

**Satz 14.4.** *Jedes LOOP-Programm terminiert.*

*Beweis.* Ein Programm LOOP besteht aus endlich vielen Anweisungen, es kann also nur dann nicht anhalten, wenn der gleiche Code immer und immer wieder ausgeführt wird. Einem LOOP-Programm steht dazu einzig die LOOP-Kontrollstruktur zur Verfügung. Eine einzelne LOOP-Struktur reicht aber sicher nicht, denn nach Konstruktion ist die Anzahl Schleifendurchläufe bereits zu Beginn der LOOP-Struktur festgelegt und kann nicht mehr verändert werden.

Wenn es also möglich sein sollte, dass ein LOOP-Programm nicht anhält, dann müsste dazu eine komplexe Schachtelung von LOOP-Konstruktionen verwendet werden. Es ist zu zeigen, dass dies trotzdem nicht reicht. Genauer zeigen wir, dass jede endliche Schachtelung von LOOPS terminiert.

Wir verwenden vollständige Induktion nach der Schachtelungstiefe  $n$  der LOOP-Konstruktionen.

- Induktionsverankerung  $n = 0$ : Ein Programm ohne LOOP-Strukturen ist eine lineare Abfolge von Instruktionen wie in (14.1), die einmalig ausgeführt werden. Am Ende der Liste von Instruktionen hält das Programm an.
- Induktionsannahme: Es wird angenommen, dass die Behauptung für Programme mit höchstens  $n$  geschachtelten LOOP-Strukturen bereits bewiesen ist.
- Induktionsschritt: Sei jetzt  $P$  ein LOOP-Programm, welches genau  $n+1$  geschachtelte LOOP-Strukturen enthält. Jede solche Struktur besteht aus einer LOOP-Anweisung der Form

$$\begin{array}{c} \text{LOOP } x_i \text{ DO} \\ \quad Q \\ \text{END} \end{array} \tag{14.4}$$

Das Teilprogramm  $Q$  ist ein LOOP-Programm mit höchstens  $n$  geschachtelten LOOP-Strukturen. Nach Induktionsannahme terminiert  $Q$  immer. Der Wert von  $x_i$  legt die Anzahl der Schleifendurchläufe unabhängig von Änderungen durch  $Q$  fest. Nach Induktionsannahme terminiert jede Ausführung von  $Q$ . Somit terminiert auch die äußere LOOP-Struktur (14.4) mit Schachtelungstiefe  $n+1$ .

Kommen im Programm  $P$  mehrere LOOP-Strukturen wie in (14.4) nacheinander vor, terminiert ebenfalls jede einzelne. Somit terminiert auch das Programm  $P$ . Damit ist der Induktionsschritt vollzogen.

Nach dem Prinzip der vollständigen Induktion folgt die Behauptung des Satzes. □

```

int list_search(list_t *e, int search) {
    while (e) {
        if (e->val == search)
            return;
        e = e->fd;
    }
    return 0;
}

int list_search(list_t *e, int search) {
    int i = 0;
    while ((e) && (i++ < MAX_ITER)) {
        if (e->val == search)
            return;
        e = e->fd;
    }
    return 0;
}

```

Abbildung 14.2: Modifikation einer Schleife, die in einer verketteten Liste nach einem Wert sucht, um der Forderung der zweiten Regel von *The Power of 10* Rechnung zu tragen. Durch den Zähler mit der Variablen *i* wird die Zahl der Iterationen auf *MAX\_ITER* limitiert.

Da in der LOOP-Sprache das Halteproblem gelöst ist, kann die LOOP-Sprache nicht Turing-vollständig sein. Der Beweis des Satzes 14.4 zeigt insbesondere, dass es in LOOP nicht möglich ist, eine Endlosschleife zu programmieren.

#### 14.2.4 Anwendungen

Die beschränkte Schleifenkonstruktion LOOP der LOOP-Sprache ist nicht mächtig genug, eine Turing-vollständige Sprache zu erzeugen. Programme in einer Programmiersprache, die nur über diese Art von Kontrollstruktur verfügt, terminieren immer. Man kann dies dazu nutzen, die Vorhersagbarkeit des Verhaltens eines Programms zu verbessern. Dieser Abschnitt zeigt ein paar Beispiele.

##### The Power of 10

*The Power of 10* ist eine Zusammenstellung von Codierstandards für sicherheitskritischen Code vor allem für die Programmiersprache C. Einige der Regeln sind auch auf andere Sprachen übertragbar. The Power of 10 wurde von Gerard J. Holzmann von NASA/JPL entwickelt [57]. Im Kontext der Diskussion über Kontrollstrukturen ist vor allem die zweite Regel wichtig:

2. All loops must have fixed bounds. This prevents runaway code.

Die Regel verbietet auf den ersten Blick die Verwendung von Kontrollstrukturen wie *for* oder *while*, mit denen sich in C Endlosschleifen programmieren lassen. Die Empfehlung wird daher oft dahingehend präzisiert, dass bei jeder Schleife eine explizite obere Schranke für die Anzahl der Iterationen programmiert werden muss. Abbildung 14.2 zeigt an einem Beispiel, wie dies realisiert werden kann. Die Suchschleife, die den Wert *search* in einer verketteten Liste sucht, ist grundsätzlich nicht beschränkt. Durch den expliziten Zähler mit der Variablen *i*, der im rechten Codefragment in Abbildung 14.2 hinzugefügt worden ist, wird die Anzahl der Iterationen limitiert.

Es ist natürlich möglich, mit unbeschränkt tiefer Rekursion den gleichen Effekt wie mit einer unbeschränkten Schleifenkonstruktion zu erreichen, daher enthält *The Power of 10* als erste Regel

1. Avoid complex flow constructs, such as goto and recursion.

Natürlich gibt es auch Situationen, in denen eine Schleife nicht terminieren darf. Eine Event-Loop oder ein Scheduler sollen zum Beispiel niemals anhalten. Die Regel zielt aber vor allem darauf ab, dass der Code so gestaltet sein muss, dass man für jede Schleife im Sinne des Satzes 14.4 beweisen kann, ob sie anhält oder nicht.

## Iteratoren

Ein Iterator ist ein Objekt, mit dem die Elemente einer Menge von Objekten durchlaufen werden können. In der C++ Standard Library werden Iteratoren für den Zugriff auf die Elemente von Arrays, Listen, Queues und weiteren Containern verwendet. Der Algorithmus `std::for_each` profitiert von dem gemeinsamen API, das die Iteratoren für alle Container bieten. Die Funktion

```
void std::for_each(InputIterator first, InputIterator last, Function f);
```

wendet die Funktion `f` auf jedes Element im Container zwischen `first` und `last` an, ganz unabhängig davon, welche Art von Container vorliegt.

Da die Anzahl der Elemente im Container beim Aufruf von `std::for_each` feststeht, ist auch bekannt, wie viele Aufrufe von `f` stattfinden werden. `std::for_each` ist also der LOOP-Konstruktion ähnlicher als der `for`-Schleife von C, allerdings nur solange der Containerinhalt sich während der Traversierung nicht ändert. Viele der Container der C++ Standard Library gestatten keine Veränderungen des Containerinhaltes, Veränderungen invalidieren aktive Iteratoren. Das Argument `f` kann auch ein Funktorobjekt sein, welches eine Referenz auf den Container hält und versucht, den Inhalt des Containers zu verändern. Solche Programme sind je nach verwendetem Container inkorrekt, sie können sogar zu Abstürzen führen. Da der Compiler von `f` möglicherweise nur das Interface kennt, kann die Schwierigkeit zur Compilezeit nicht erkannt werden.

In C++11 wurde eine neue Syntax für den `for`-Operator eingeführt. Mit dem Code

```
for (auto& x : items) {  
    ...  
}
```

wird ebenfalls der Code in den geschweiften Klammern auf alle Elemente im Container `items` angewendet. Auch diese Konstruktion verbietet in vielen Fällen, dass der Container verändert werden darf. Die folgende `for`-Schleife soll alle Elemente des Containers anzeigen und falls ein Element mit Wert 11 gefunden wird, soll ein neues Element in den Container eingefügt werden:

```
std::vector<int> l = { 2, 3, 5, 7, 11, 13, 17, 23 };  
for (auto& x : l) {  
    if (x == 11) {  
        l.insert(l.end(), 29);  
    }  
    std::cout << x << ", ";  
}
```

Der gezeigte Code mit dem Container `std::vector<int>` liefert den Output<sup>1</sup>

```
2, 3, 5, 7, 0, 0, 0,
```

Die Einfügeoperation hat den Iterator ungültig gemacht, die Iteration läuft aber dennoch weiter und wird so oft ausgeführt, wie der Container Elemente enthält, ganz analog den Eigenschaften von LOOP. Die Dereferenzierung des Iterators liefert aber immer den falschen Wert 0. Auch erhält der Programmierer keine Warnung vom Compiler.

Verwendet man `std::list<int>` anstelle von `std::vector<int>`, arbeitet der Code wie erwartet und gibt

```
2, 3, 5, 7, 11, 13, 17, 23, 29,
```

aus. Die Zahl 29 wird in die Liste eingefügt und die Schleife führt eine zusätzliche Iteration aus, weil der Container jetzt ein Element mehr enthält.

### Die FreeRADIUS `unlang`

Der FreeRADIUS-Server ist gemäß Informationen auf seiner Website [14] der am weitesten verbreitete RADIUS-Server der Welt und wird bei Internet Service Providern und Telekommunikationsfirmen für die Authentisierung von Benutzern verwendet. Er ist eine der Schlüsseltechnologien hinter dem internationalen Netzwerk Eduroam zur Wi-Fi-Authentisierung an Hochschulen.

Der FreeRADIUS-Server verfügt über vielfältige Erweiterungsmöglichkeiten durch dynamisch ladbare Module, die die Nutzung der verschiedensten Authentisierungsmethoden, Datenbanken mit Benutzerinformationen und auch Programmiersprachen für die Implementation serverseitiger Funktionalität ermöglichen. Der Server braucht daher keine besonders leistungsfähige Konfigurationssprache. Im Gegenteil möchte man sicherstellen, dass der Server möglichst einfach bleibt und sein Verhalten möglichst vorhersagbar.

Aus diesen Gründen hat man sich bewusst dazu entschieden, eine Konfigurationssprache zu definieren, die *nicht* Turing-vollständig ist. Die `unlang`-Sprache [15] erreicht dies durch eine sorgfältige Beschränkung der verfügbaren Kontrollstrukturen. Es gibt eine `if-else`-Struktur und zur Vereinfachung ein `switch`-Statement, welches man natürlich auch mit `if-else` realisieren kann. Mit `if-else` allein kann man aber keine Iteration konstruieren. Die einzige Kontrollstruktur, die Iteration erlaubt, ist `foreach`, die in der Form

```
foreach &Attribute-Reference {
    ...
}
```

verwendet wird. Sie iteriert über die Werte eines Attributes. Die Menge der Attributwerte kann während der Ausführung von `foreach` nicht verändert werden, so dass die maximale Anzahl der Iterationen, die `foreach` ausführt, wie bei LOOP zu Beginn der Schleife bereits feststeht.

---

<sup>1</sup>Verwendet wurde der Apple clang Compiler Version 15.0.0.

## 14.3 Die Sprache WHILE

Die LOOP-Struktur war nicht flexibel genug, um Turing-Vollständigkeit zu erreichen. Der Grund dafür war die Tatsache, dass es in der LOOP-Sprache keine unbeschränkten Schleifen und daher auch keine nicht anhaltenden Programme geben kann. In diesem Abschnitt konstruieren wir daher eine Sprache mit einer unbeschränkten Schleifenstruktur.

### 14.3.1 Die WHILE-Schleife

Statt der LOOP-Sprache konstruieren wir jetzt eine Sprache, die die gleichen Grundelemente von Abschnitt 14.2.1 verwendet: Variablen mit natürlichen Zahlenwerten, Konstanten in  $\mathbb{N}$  und Addition und Subtraktion von Konstanten. Zusätzlich wird die folgende WHILE-Struktur

```
WHILE  $x_i > 0$  DO
    P
END
```

(14.5)

definiert. Sie führt den Block  $P$  so lange aus, als  $x_i > 0$  ist. Der Vergleich  $x_i > 0$  ist die einzige Bedingung, die zur Verfügung steht. Im Gegensatz zu LOOP wird erwartet, dass der Programmteil  $P$  beim Erreichen einer ausreichend großen Anzahl Iterationen die Variable  $x_i$  auf 0 setzt und damit die Schleife terminiert.

### Arithmetische Operationen

Mit der WHILE-Struktur können Addition und Multiplikation genauso implementiert werden, wie dies für LOOP möglich war.

Addition  $x_i := x_k + x_l$ :

```
xi := xk
y := xl
WHILE y > 0 DO
    xi := xi + 1
    y := y - 1
END
```

Multiplikation:  $x_i := x_k * x_l$

```
xi := 0
y := xl
WHILE y > 0 END
    xi := xi + xk
    y := y - 1
END
```

```
xi := 0
y := xl
WHILE y > 0 DO
```

```
    z := xk
    WHILE z > 0 DO
        xi := xi + 1
        z := z - 1
    END
    y := y - 1
END
```

In der letzten Implementation der Multiplikation ganz rechts ist die Addition  $x_i := x_k + x_l$ , die in der mittleren Implementation angenommen wurde (rot hinterlegt), ebenfalls als Schleife realisiert (rot).

## Bedingte Ausführung

Die WHILE-Konstruktion ermöglicht mit einer Hilfsvariable auch die bedingte Anweisung:

$$\left. \begin{array}{c} \text{IF } x_i \text{ DO} \\ \quad P \\ \text{END} \end{array} \right\} \quad \text{implementiert als} \quad \left\{ \begin{array}{c} y := x_i \\ \text{WHILE } y > 0 \text{ DO} \\ \quad y := 0 \\ \quad P \\ \text{END} \end{array} \right.$$

### 14.3.2 WHILE als Erweiterung von LOOP

Die LOOP-Struktur lässt sich mit dem WHILE-Konstrukt implementieren. Da WHILE bei jedem Schleifendurchlauf die Bedingung erneut prüft, muss eine Hilfsvariable verwendet werden, die den Schleifenzähler von Änderungen innerhalb des Schleifenkörpers isoliert:

$$\left. \begin{array}{c} \text{LOOP } x_i \text{ DO} \\ \quad P \\ \text{END} \end{array} \right\} \quad \text{implementieren als} \quad \left\{ \begin{array}{c} y := x_i \\ \text{WHILE } y > 0 \text{ DO} \\ \quad P \\ \quad y := y - 1 \\ \text{END} \end{array} \right. \quad (14.6)$$

Dies zeigt, dass die WHILE-Sprache eine echte Erweiterung der LOOP-Sprache ist. Alle unentgeltlichen Operationen der LOOP-Sprache stehen automatisch auch in WHILE zur Verfügung.

### 14.3.3 Turing-Vollständigkeit von WHILE

In der Sprache WHILE können Programme realisiert werden, die nicht terminieren, zum Beispiel

$$\begin{array}{c} y := 1 \\ \text{WHILE } y > 0 \text{ DO} \\ \quad \text{END} \end{array} \quad (14.7)$$

Das Programm tut nichts, bis  $y$  nicht mehr positiv ist, ein Fall, der nicht eintreten kann. Zwar ist das Halteproblem nicht mehr automatisch gelöst wie bei der LOOP-Sprache, doch garantiert eine Endlosschleife noch nicht, dass WHILE Turing-vollständig ist.

Um den Nachweis zu führen, dass WHILE tatsächlich Turing-vollständig ist, müsste ein Turing-Maschinensimulator in WHILE geschrieben werden. Dies dürfte nicht allzu schwierig sein, denn die WHILE-Struktur ist ja genau das, was sich der moderne Programmierer von den meisten Programmiersprachen gewohnt ist. Wir wählen aber einen anderen Weg. Wir werden im nächsten Abschnitt erst zeigen, dass sich jedes WHILE-Programm in ein äquivalentes GOTO-Programm transformieren lässt. Dies bildet die Vorgehensweise des Compilers einer Hochsprache ab, da der vom Compiler erzeugte Maschinencode eher einem GOTO-Programm gleicht. Wir werden auch die umgekehrte Richtung zeigen, dass sich also jedes GOTO-Programm in ein WHILE-Programm umwandeln lässt. Dies zeigt, dass WHILE und GOTO entweder beide Turing-vollständig sind oder beide nicht. In Abschnitt 14.4.4 werden wir dann einen Turing-Maschinensimulator in der GOTO-Sprache

	Beschreibung	C-äquivalent
>	inkrementiere den Zeiger	ptr++
<	dekrementiere den Zeiger	ptr--
+	inkrementiere den aktuellen Zellenwert	(*ptr)++
-	dekrementiere den aktuellen Zellenwert	(*ptr)--
.	gib den aktuellen Zellenwert aus	putchar(*ptr)
,	lese ein Zeichen in die aktuelle Zelle	*ptr = getchar();
[	springe nach vorne, hinter den passenden ]-Befehl, wenn der aktuelle Zellenwert 0 ist.	while (*ptr) {
]	springe zurück, hinter den passenden [-Befehl, wenn der aktuelle Zellenwert nicht 0 ist.	}

Tabelle 14.1: Die acht Brainfuck-Befehle mit ihrem C-Äquivalent.

vorstellen, was gleichzeitig die WHILE- und die GOTO-Sprache als Turing-vollständig beweisen wird.

#### 14.3.4 Brainfuck

Eine Turing-vollständige Sprache muss Zugang zu einem beliebig großen Speicher haben, und es muss eine unbeschränkte Schleifenkonstruktion wie WHILE geben. Brainfuck ist eine besonders kleine Sprache, die diese Kriterien erfüllt. Die Sprache wurde 1993 von Urban Müller entworfen. Sein Ziel war, dass die Sprache mit einem möglichst kleinen Compiler übersetzt werden kann. Tatsächlich ist der kleinste bekannte Compiler für MS-DOS unter 100 Bytes groß.

##### Das Brainfuck-Speichermodell

Die Sprache Brainfuck verwendet einen unbegrenzten Speicher aus einzelnen Speicherzellen, von denen typischerweise angenommen wird, dass sie jeweils ein Byte enthalten können. Es gibt allerdings auch Varianten von Brainfuck, die von unbeschränkt großen Zellen ausgehen, ähnlich den in Abschnitt 14.2.1 eingeführten Variablen.

Der Zugriff zu den einzelnen Speicherzellen erfolgt über einen impliziten Zeiger, den das Brainfuck-Programm inkrementieren oder dekrementieren kann. Dies ahmt ganz offensichtlich das Band einer Turing-Maschine nach, auf dessen Zellen ebenfalls nicht absolut, sondern nur relativ durch Verschiebung des Schreib-/Lesekopfes nach links oder rechts zugegriffen werden kann.

Um das Modell etwas zu konkretisieren, kann man sich den Zeiger als eine C-Variable vom Typ `unsigned char *ptr` vorstellen. Die Variable `ptr` zeigt auf Speicherzellen vom Typ `unsigned char` und kann inkrementiert oder dekrementiert werden.

##### Die Brainfuck-Befehle

Brainfuck verfügt über die in Tabelle 14.1 zusammenstellten Befehle. Die Befehle < und > dekrementieren und inkrementieren den Zeiger und wurden bereits beschrieben. Die

Brainfuck	Ook!	Abkürzung
>	Ook. Ook?	.?
<	Ook? Ook.	?.
+	Ook. Ook.	..
-	Ook! Ook!	!!
.	Ook! Ook.	!..
,	Ook. Ook!	.!
[	Ook! Ook?	!?
]	Ook? Ook!	?!

Tabelle 14.2: Übersetzung von Brainfuck in Ook!

Befehle + und - inkrementieren bzw. dekrementieren die aktuelle Zelle. Die Befehle . und , werden nur für Input und Output gebraucht und haben damit keinen Einfluss auf die Turing-Vollständigkeit der Sprache.

Die für Turing-Vollständigkeit nötige unbegrenzte Schleife wird durch die beiden Befehle [ und ] zur Verfügung gestellt. Der Schleifenkörper wird so lange ausgeführt, bis die aktuelle Zelle den Wert 0 hat. Interpretiert man den Pointer als eine C-Variable, entsprechen die Befehle [ und ] einer while-Schleife (dritte Spalte in Tabelle 14.1).

Das Programm

$$+[] \tag{14.8}$$

realisiert die Endlosschleife (14.7) in Brainfuck. Das + inkrementiert die aktuelle Zelle des mit 0 initialisierten Speichers. Die nachfolgende Schleife [] hat einen leeren Schleifenkörper. Sie testet immer wieder, ob die aktuelle Zelle 0 geworden ist, was natürlich nie geschehen wird. Das Programm steckt in einer Endlosschleife.

### Turing-Vollständigkeit von Brainfuck

Die Möglichkeit, eine Endlosschleife zu programmieren, garantiert noch nicht, dass die Sprache Turing-vollständig ist. Dazu muss gezeigt werden, dass sich in Brainfuck eine universelle Turing-Maschine realisieren lässt. Dies ist Daniel B. Cristofani gelungen.

Alternativ könnte man auch versuchen, ein Turing-Maschinenprogramm in ein Brainfuck-Programm zu übersetzen. Frans Faase hat eine solche Übersetzung konstruiert. Damit folgt ebenfalls, dass eine Brainfuck-Maschine jede beliebige Turing-Maschine ausführen kann.

### Ook!

Brainfuck wird mit acht verschiedenen Zeichen geschrieben. Ist es möglich, eine Turing-vollständige Sprache zu konzipieren, die nur drei Syntaxelemente verwendet? Gemäß David Morgan-Mar in [32] ist die Antwort ja. Seine Sprache Ook! hat nur die drei Syntaxelemente

$$\text{Ook.}, \quad \text{Ook!} \quad \text{und} \quad \text{Ook?} \tag{14.9}$$

Auf der Website [32] schreibt er, dass die Design-Prinzipien für die Sprache die folgenden seien:

- A programming language should be writable and readable by orang-utans.
- To this end, the syntax should be simple, easy to remember, and not mention the word “monkey”.
- Bananas are good.

Die Sprache Ook! ist nur eine neue Codierung von Brainfuck. Um die acht Zeichen von Brainfuck mit den drei Silben von (14.9) zu codieren, müssen jeweils zwei Silben ein Brainfuck-Zeichen codieren, wie dies in Tabelle 14.2 dargestellt ist.

Da es neun mögliche Zweiergruppen von Silben (14.9) gibt, fehlt Ook? Ook? in der Tabelle 14.2. Um das dritte Design-Prinzip abzudecken, definiert [32] die Bedeutung von Ook? Ook? als *Give the Memory Pointer a banana*.

### 14.3.5 Office

Ein Tabellenkalkulationsprogramm ist ein zellulärer Automat (Seite 10.4.3), in dessen Zellen der Benutzer beliebige Formeln eintragen kann. Durch geeignete Formeln, die die Übergangsfunktion implementieren, kann man jede beliebige Turing-Maschine mithilfe eines Spreadsheets berechnen. In Übungsaufgabe 14.2 wird dies für eine Turing-Maschine durchgeführt, die eine Binärzahl auf dem Band inkrementiert. Man darf daher schließen, dass ein Tabellenkalkulationsprogramm eine Turing-vollständige Programmierumgebung ist.

Die nicht ganz ernst gemeinte PowerPoint-Turing-Machine von Tom Wildenhain [43] illustriert, dass Turing-Vollständigkeit auch in anderen Office-Programm vorhanden ist.



## 14.4 Die Sprache GOTO

Frühe Programmiersprachen wie die erste Version von FORTRAN oder BASIC verfügen nicht über eine schachtelbare Kontrollstruktur wie WHILE. Auch der Maschinencode kann solche strukturierten Anweisungen nicht verwenden. Die WHILE-Sprache ist daher kein gutes Modell für real existierende Prozessoren oder für gewisse Programmiersprachen. Die GOTO-Instruktion, die in Abschnitt 14.4.1 eingeführt wird, ist besser dafür geeignet. In den Abschnitten 14.4.2 und 14.4.3 wird gezeigt, dass die beiden Sprachen WHILE und GOTO gleich mächtig sind. Schließlich wird in Abschnitt 14.4.4 ein Turing-Maschinensimulator für GOTO vorgestellt, was die Turing-Vollständigkeit beider Sprachen beweist.

### 14.4.1 Die GOTO-Sprache

Ein GOTO-Programm ist eine mit Marken versehene Liste von Anweisungen, wie sie in Abschnitt 14.2.1 definiert worden sind. Zusätzlich gibt es eine bedingte GOTO-Anweisung, mit der das Programm zu jeder beliebigen Marke verzweigen kann. Die Anweisung

$$M_k : \text{ IF } x_i = c \text{ THEN GOTO } M_l$$

setzt die Programmausführung bei  $M_l$  fort, wenn  $x_i = c$  ist. Man beachte den Unterschied zum IF in der LOOP-Sprache, wo die einzige Bedingung  $x_i > 0$  war.

Das Beispielprogramm

```

 $M_1 : x_1 := 3500$ 
 $M_2 : x_2 := 47$ 
 $M_3 : x_3 := 0$ 
 $M_4 : x_1 := x_1 - 47$ 
 $M_5 : x_2 := x_2 - 1$ 
 $M_6 : \text{IF } x_2 = 0 \text{ THEN GOTO } M_8$ 
 $M_7 : \text{IF } x_3 = 0 \text{ THEN GOTO } M_4$ 
 $M_8 :$ 
```

subtrahiert die Konstante 47 genau 47 mal von 3500, bis der bedingte Sprung in Zeile  $M_6$  die Iteration beendet. Am Ende des Programms hat  $x_1$  den Wert  $3500 - 47^2 = 3500 - 2209 = 1291$ .

Die einzige Vergleichsbedingung  $x_i = c$  erlaubt, verschiedene weitere Arten von Sprunganweisungen zu konstruieren:

- Unbedingte Sprunganweisung: Mit einer Hilfsvariable kann auch die unbedingte Sprunganweisung

$$M_n : \text{GOTO } M_k \quad \} \quad \text{als} \quad \left\{ \begin{array}{l} M_n : y := 0 \\ M_{n+1} : \text{IF } y = 0 \text{ THEN GOTO } M_k \end{array} \right.$$

implementiert werden.

- Negation: Die Bedingung kann mit einer Hilfsvariable auch negiert werden. Die Anweisung

$$M_k : \text{IF } \neg \text{Bedingung} \text{ THEN GOTO } M_l$$

kann als

$$\begin{aligned} M_k &: y := 0 \\ M_{k+1} &: \text{IF } \text{Bedingung} \text{ THEN GOTO } M_{k+3} \\ M_{k+2} &: y := 1 \\ M_{k+3} &: \text{IF } y = 1 \text{ THEN GOTO } M_l \end{aligned}$$

realisiert werden. Wenn die Bedingung erfüllt ist, wird  $M_{k+2}$  übersprungen. Damit hat  $y$  in der Zeile  $M_{k+3}$  nicht den Wert 1 und der Sprung zu  $M_l$  wird nicht ausgeführt.

- Bedingung  $x_i > 0$ : Da  $x_i$  nur natürliche Werte haben kann, ist die Bedingung  $x_i > 0$  die Negation der Bedingung  $x_i = 0$ . Die in der LOOP-Sprache eingeführte Bedingung ist also auch in GOTO vorhanden.

Im Folgenden wird diese erweiterten GOTO-Anweisungen als Teil der Sprache GOTO vorausgesetzt.

Während der Leser sicher vertraut ist mit Sprachen, die eine WHILE-Anweisung kennen, sind Sprachen nach dem Muster der GOTO-Sprache heute eher selten. Die Maschineninstruktionen moderner Prozessoren bilden aber im Wesentlichen eine Sprache dieser Art, wie der Code in Abbildung 14.3 rechts illustriert.

<pre>#include &lt;stdlib.h&gt; #include &lt;stdio.h&gt;  int main(int argc, char *argv[]) {      int i = 0;      while (i &lt; 10) {         printf("hello, world!\n");         i++;     }      return EXIT_SUCCESS; }</pre>	<pre>.globl _main                                ; -- Begin function main _main: .p2align 2 .cfi_startproc ; %bb.0:     stp x20, x19, [sp, #-32]!           ; 16-byte Folded Spill     stp x29, x30, [sp, #16]              ; 16-byte Folded Spill     add x29, sp, #16     mov w20, #10 Lloh0:  adrp x19, l_str@PAGE Lloh1:  add x19, x19, l_str@PAGEOFF LBB0_1:  mov x0, x19                      ; =&gt;This Inner Loop Header: Depth=1          b1 _puts          subs w20, w20, #1          b.ne LBB0_1 ; %bb.2:          mov w0, #0          ldp x29, x30, [sp, #16]          ; 16-byte Folded Reload          ldp x20, x19, [sp], #32          ; 16-byte Folded Reload          ret          .loh AdrpAdd Lloh0, Lloh1 .cfi_endproc ; -- End function .l_str: .section __TEXT,__cstring,cstring_literals         ; @str         .asciz "hello, world!" .subsections_via_symbols</pre>
--	--

Abbildung 14.3: Kompilation einer `while`-Schleife von C (links) in entsprechenden ARM-Assembler-Code (rechts). Der Assembler-Code hat eher den Charakter der GOTO-Sprache hat. Der Compiler hat die aufsteigende Iteration in eine absteigende Iteration optimiert. Grün hinterlegt ist die Instruktion, die den Zähler dekrementiert. Blau hinterlegt ist die bedingte Sprung-Operation zur Marke `LBB0_1`, die ausgeführt wird, solange das Resultat der Dekrementierung von Null verschieden ist, angezeigt durch das Suffix `ne` (not equal).

#### 14.4.2 WHILE-Programm in GOTO übersetzen

Ein WHILE-Programm kann immer in ein gleichwertiges GOTO-Programm übersetzt werden, indem man jede WHILE-Struktur nach dem Muster

$$\left. \begin{array}{c} \text{WHILE } x_i > 0 \text{ DO} \\ \quad P \\ \text{END} \end{array} \right\} \text{ übersetzt in } \left\{ \begin{array}{l} M_k : P \\ M_{k+1} : \text{IF } x_i = 0 \text{ THEN GOTO } M_{k+3} \\ M_{k+2} : \text{GOTO } M_k \\ M_{k+3} : \end{array} \right.$$

Eine ähnliche Transformation führt ein C-Compiler durch, der die `while`-Anweisung in einem C-Programm in Maschinencode übersetzt. In Abbildung 14.3 ist links ein einfaches C-Programm mit einer `while`-Schleife dargestellt. Rechts wird der erzeugte Assembler-Code gezeigt. Er sieht ungefähr wie GOTO aus, wobei nicht jede Zeile mit einer Marke versehen ist, sondern nur jene, die auch tatsächlich als Sprungziel vorkommen. Blau hinterlegt ist die bedingte Sprunganweisung, mit der der `printf`-Befehl wiederholt wird, bis die grün hinterlegte Dekrementinstruktion ein von Null verschiedenes Resultat hat.

#### 14.4.3 GOTO-Programm in WHILE übersetzen

Die umgekehrte Transformation von einem GOTO-Programm in ein WHILE-Programm ist ebenfalls möglich. Um die Umwandlung von GOTO in WHILE etwas kompakter zu schreiben, nehmen wir an, dass die IF-Konstruktion nicht nur für eine Bedingung der Art  $x_i > 0$

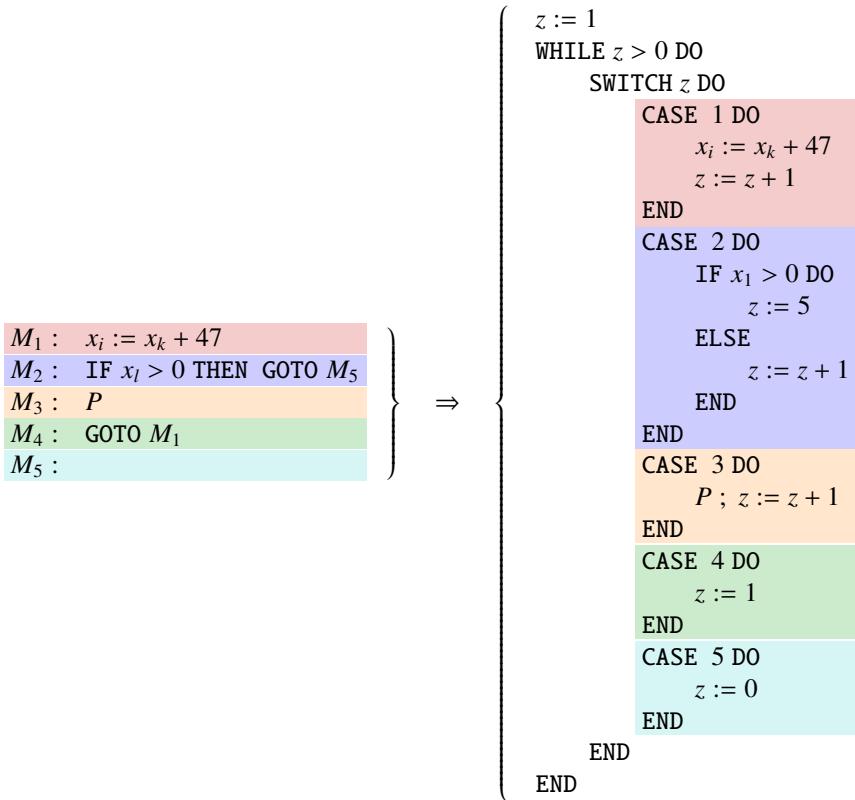


Abbildung 14.4: Umwandlung eines GOTO-Programms in ein WHILE-Programm mit Hilfe einer Variablen  $z$ , die die Funktion des Programmzählers übernimmt.

funktioniert, sondern auch für Bedingungen der Form  $x_i = c$ . Außerdem erlauben wir, dass mehrere Anweisungen, durch Semikolon getrennt, auf einer Zeile stehen können.

Als Ersatz für die Marken  $M_k$  wird im WHILE-Programm die Variable  $z$  reserviert. Sie übernimmt die Rolle eines Programmzählers. Die Übersetzung in Abbildung 14.4 verwendet die SWITCH-CASE-Konstruktion, die in jeder Sprache, die LOOP erweitert, automatisch existiert (Seite 368). Sie führt die Instruktionen bei der Marke  $M_k$  genau dann aus, wenn  $z = k$  ist. Bei jeder Instruktion, außer bei den Sprungoperationen, ist auch noch die Inkrementoperation  $z := z + 1$  nötig, die das Programm implizit zur nächsten Anweisung schreiten lässt.

#### 14.4.4 Turing-Maschinensimulator in GOTO

Die Sprache GOTO ist Turing-vollständig, wenn es dafür einen Turing-Maschinensimulator gibt. Ein solcher soll in diesem Abschnitt konstruiert werden.

## Das Band

Die Sprache GOTO verfügt über unendlich viele Variablen  $x_i$ , doch es gibt keine Möglichkeit, diese Variablen indirekt, also mit einem berechneten Index  $i$ , zu adressieren. Daher wird hier eine alternative Darstellungsweise für das Band verwendet. Da es keine Einschränkung für die Größe eines Variablenwertes gibt, kann eine einzelne Variable beliebig viel Information aufnehmen. Die Variable  $b$  soll den Bandinhalt darstellen.



Wir stellen uns die natürliche Zahl  $b$  als im Vierersystem geschrieben vor<sup>2</sup>. Die einzelnen Stellen bezeichnen wir mit  $b_i$ , sie sind Zahlen zwischen 0 und 3. Die Zahl  $b$  ist dann die Summe

$$b = b_0 + b_1 4 + b_2 4^2 + \cdots + b_{k-1} 4^{k-1} + b_k 4^k + b_{k+1} 4^{k+1} + \dots$$

Wir müssen nur die Zeichen „„, 0 und 1 codieren können, wir verwenden die Zuordnung

$$0 \mapsto \text{„}, \quad 1 \mapsto 0 \quad \text{und} \quad 2 \mapsto \text{1}. \quad (14.10)$$

## Die Position des Schreib-/Lesekopfes

Die Position des Schreib-/Lesekopfes ist die Stelle  $k$  der Zahl  $b$ . Die Wertigkeit dieser Stelle kann durch die Potenz  $h = 4^k$  wiedergegeben werden. Beide Zahlen  $k$  und  $h$  sind für die Beschreibung der Band-Operationen gleichermaßen nützlich.

## Die Band-Operationen

Die Turing-Maschine muss in jedem Schritt das aktuelle Zeichen  $b_k$  vom Band lesen. Mit den Variablen  $k$  und  $h = 4^k$  ist dies besonders einfach. Die ganzzahlige Division von  $b$  durch  $h$  ist

$$b/h = b_k + b_{k+1} 4 + b_{k+2} 4^2 + \dots,$$

der Wert von  $b_k$  kann also einfach dadurch bestimmt werden, dass der Rest von  $b/h$  bei Teilung durch 4 bestimmt wird.

Das Feld  $k$  löschen heißt, die Stelle  $b_k$  zu Null zu machen, was durch die Subtraktion

$$\bar{b} = b - b_k h = b - b_k 4^k = b_0 + b_1 4 + b_2 4^2 + \cdots + b_{k-1} 4^{k-1} + b_{k+1} 4^{k+1} + \dots$$

geschehen kann.

Das Feld  $k$  mit einem neuen Wert  $\tilde{b}_k$  beschreiben kann ebenfalls durch eine arithmetische Operation realisiert werden. Ist die Stelle  $k$  in  $b$  Null, also  $b_k = 0$ , dann wird diese Stelle durch die Operation

$$b' = \bar{b} + \tilde{b}_k h = (b - b_k h) + \tilde{b}_k h$$

mit dem Wert  $\tilde{b}_k$  besetzt. Damit sind alle Operationen auf dem Band durch einfache arithmetische Operationen, die in der GOTO-Sprache zur Verfügung stehen, realisiert.

<sup>2</sup>Da nur drei Zeichen codiert werden müssen, wäre die Dreiersystemdarstellung ausreichend. Eine Vierersystemdarstellung ist aber auf einem binären Computer viel einfacher zu realisieren, weil eine Stelle einer Zahl im Vierersystem genau zwei Stellen einer Binärzahl entspricht.

## Die Kopfbewegung

Der letzte Teil jedes Turing-Maschinenschrittes ist die Kopfbewegung. Sie entspricht einer Erhöhung oder Erniedrigung von  $k$  um 1 und einer Multiplikation von  $h$  mit 4 oder einer Division von  $h$  durch 4. Auch die Kopfbewegungen sind daher einfache arithmetische Operationen.

## Der Compiler

Die eben dargelegten arithmetischen Methoden ermöglichen jetzt, einen Compiler zu konstruieren, welcher Turing-Maschinenbeschreibungen in ein GOTO-Programm verwandelt. Eine Turing-Maschinenbeschreibung besteht aus Zuordnungen, die aus dem aktuellen Zustand und dem aktuellen Bandzeichen den neuen Zustand, das neue Bandzeichen und die Kopfbewegung ermitteln. Der Zustand kann als natürliche Zahl codiert werden, wir nehmen an, dass dafür die Variable  $z$  reserviert worden ist. Für die Bandzeichen wurde in (14.10) bereits eine Codierung durch Ganzzahlen festgelegt.

Die Kopfposition ist durch die Zahlen  $k$  und  $h = 4^k$  gegeben. Die Kopfbewegung wird durch Inkrement bzw. Dekrement von  $k$  und durch Multiplikation bzw. Division von  $h$  durch 4 erreicht. Diese Operationen sind für alle Turing-Maschinenübergänge gleich, es reicht daher, die auszuführende Kopfbewegung als Werte  $l = 1$ ,  $r = 0$  für links und  $l = 0$ ,  $r = 1$  für rechts zu speichern.

Die Turing-Maschinenregeln



werden dann durch die folgenden GOTO-Fragmente implementiert:

$M_i : \text{ IF } z = p \text{ DO}$	$M_i : \text{ IF } z = p \text{ DO}$
$M_{i+1} : \quad \text{IF } b_k = u \text{ DO}$	$M_{i+1} : \quad \text{IF } b_k = u \text{ DO}$
$M_{i+2} : \quad \quad \quad z := q$	$M_{i+2} : \quad \quad \quad z := q$
$M_{i+3} : \quad \quad \quad b_k := v$	$M_{i+3} : \quad \quad \quad b_k := v$
$M_{i+4} : \quad \quad \quad l := 1$	$M_{i+4} : \quad \quad \quad l := 0$
$M_{i+5} : \quad \quad \quad r := 0$	$M_{i+5} : \quad \quad \quad r := 1$
$M_{i+6} : \quad \quad \quad \text{END}$	$M_{i+6} : \quad \quad \quad \text{END}$
$M_{i+7} : \quad \quad \quad \text{END}$	$M_{i+7} : \quad \quad \quad \text{END}$

(14.12)

Der Compiler muss also für jeden Übergang der Turing-Maschine ein solches Fragment erzeugen.

Die Fragmente gemäß (14.12) werden in ein Programm eingefügt, welches zunächst die Variablen  $z$ ,  $b$ ,  $h$  und  $k$  initialisiert. Dann beginnt eine Schleife, in der zuerst die Zahlen  $b_k$  und  $b' = b - b_k h$  berechnet werden. Dann folgen alle Blöcke (14.12), wovon aber nur einer die Bedingungen erfüllt und somit ausgeführt wird. Am Ende muss dann der

Bandinhalt wieder zusammengesetzt werden und die Kopfbewegung ausgeführt werden:

$$\begin{aligned}
 M_j &: b = b_k * h + b' \\
 M_{j+1} &: \text{IF } l = 1 \text{ DO} \\
 M_{j+2} &: \quad k := k - 1 \\
 M_{j+3} &: \quad h := h/4 \\
 M_{j+4} &: \text{END} \\
 M_{j+5} &: \text{IF } r = 1 \text{ DO} \\
 M_{j+6} &: \quad k := k + 1 \\
 M_{j+7} &: \quad h := h * 4 \\
 M_{j+8} &: \text{END}
 \end{aligned}$$

### Eine virtuelle Maschine für GOTO-Programme

Um die Funktion des Compilers zu überprüfen, enthält das früher verlinkte Codepaket auch ein Programm, welches GOTO-Programme lesen und ausführen kann. Zur besseren Nachvollziehbarkeit wurden der virtuellen Maschine weitere Funktionen hinzugefügt, mit denen die aktuellen Werte von  $b$ ,  $z$ ,  $k$  und  $h$  in der gleichen Art dargestellt werden, wie der Turing-Maschinensimulator in C dies tut. Im verlinkten Video wird gezeigt, wie die virtuelle Maschine die nach GOTO kompilierte Turing-Maschine der binären Addition ausführt.

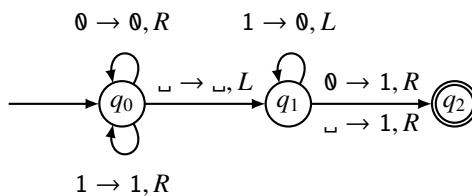


[autospr.ch/video/6](http://autospr.ch/video/6)

## Übungsaufgaben

**14.1.** Eine Tabellenkalkulation kann typischerweise nur mit Gleitkommazahlen rechnen und ist damit in der Stellenzahl auf weniger als 16 Stellen begrenzt. Konstruieren Sie ein Spreadsheet, welches zwei 25-stellige Dezimalzahlen addieren kann.

**14.2.** Das Zustandsdiagramm



beschreibt eine Turing-Maschine, die eine Binärzahl auf dem Band inkrementiert.

- Konstruieren Sie ein Spreadsheet, welches die Berechnung schrittweise durchführen kann (siehe Abbildung 14.5). In dieser Implementation wird ein Blank auf dem Band durch den Buchstaben B dargestellt.
- Überlegen Sie sich, ob Ihre Lösung sich auf einen Beweis für die Turing-Vollständigkeit des Spreadsheets verallgemeinern lässt.

	A	B	C	D	E	F	G	H	I	J
1	Zustand	Position	Input:							
2	0	2		B	1	1	0	0	1	1
3	0	3		B	1	1	0	0	1	1
4	0	4		B	1	1	0	0	1	1
5	0	5		B	1	1	0	0	1	1
6	0	6		B	1	1	0	0	1	1
7	0	7		B	1	1	0	0	1	1
8	0	8		B	1	1	0	0	1	1
9	1	7		B	1	1	0	0	1	1
10	1	6		B	1	1	0	0	1	0
11	1	5		B	1	1	0	0	0	0
12	2	6		B	1	1	0	1	0	0
13	2	6		B	1	1	0	1	0	0
14	2	6		B	1	1	0	1	0	0
15	2	6		R	1	1	0	1	0	0

Abbildung 14.5: Spreadsheet zur Simulation einer Turing-Maschine, die eine Binärzahl inkrementiert.

**14.3.** In der Sprache LOOP kann jede Variable direkt adressiert werden, und in der gleichen Instruktion können auch noch arithmetische Operationen ausgeführt werden. ‘Reinrassige’ RISC-CPUs arbeiten anders. Sie verwenden eine Load-Store-Architektur, die arithmetische Operationen nur zwischen Registern zulässt. Das Laden eines Registers mit einem Wert aus dem Hauptspeicher ist immer getrennt von Rechenoperationen.

Betrachten Sie folgende RISC-LOOP genannte Modifikation von LOOP. Statt direkt adressierbarer Variablen  $x_0, x_1, x_2, \dots$  gibt es einen Index  $i$ , der auf jede der Variablen zeigen kann. Der Zeiger ist zu Beginn des Programms auf 0 initialisiert und kann inkrementiert oder dekrementiert werden, dazu gibt es zwei neue Befehle INCR und DECR, die immer auf  $i$  wirken. Der Index  $i$  kann jedoch nicht direkt auf einen bestimmten Wert gesetzt werden.

Mit  $x_i$  kann jeweils auf diejenige Variable zugegriffen werden, auf die  $i$  zeigt.  $x_i$  kann aber nicht direkt in einem arithmetischen Ausdruck verwendet werden, vielmehr muss ihr Wert zuerst in das Register  $r$  geladen werden, welches als einziges für arithmetische Operationen zur Verfügung steht. Das folgende Programm berechnet, was in LOOP die Anweisung  $x_3 := x_2 + 4$  geschafft hat:

```
1:  INCR  
2:  INCR  
3:  r := xi  
4:  r := r + 4  
5:  INCR  
6:  xi := r
```

Auch die LOOP Anweisung kann nicht mehr jede beliebige Variable als Argument verwenden, sondern nur den aktuellen Wert von  $r$ . Zeigen Sie, dass RISC-LOOP trotzdem nicht weniger leistungsfähig als LOOP ist.

Lösungen: <https://autospr.ch/uebungen/AutoSpr-114.pdf>



# Kapitel 15

## Quantencomputer

Die Komplexitätsklassen P und NP und vor allem die NP-vollständigen Probleme, die in Kapitel 13 im Detail untersucht wurden, scheinen eine unüberwindliche technische Barriere zu sein. Der Unterschied zwischen P und NP war ja explizit so konstruiert worden, dass er unabhängig ist von der Art der Hardware. Heißt das, dass es keine Hoffnung gibt, dass Probleme in  $NP \setminus P$  je in polynomieller Zeit gelöst werden können?

Die Antwort wäre tatsächlich ja, wenn die Voraussetzungen, die wir in Kapitel 10 mit der Konstruktion der Turing-Maschinen formuliert haben, weiterhin gelten. Nur wenn wir eine gänzlich andere Art von Maschine zulassen, ist denkbar, dass gewisse Probleme schneller lösbar werden könnten. Ins Rampenlicht treten damit automatisch Quantencomputer, die die speziellen Eigenschaften der Dynamik kleinster Teilchen ausnützen können, um Berechnungen anzustellen.

Dieses Kapitel ist keine vollständige Einführung in die Theorie der Quantenberechnung oder der Quantencomputer. Ziel ist nur, zu zeigen, was an Quantencomputern so anders ist, das diesen “Umsturz von Naturgesetzen der Berechnung” möglich macht.

### 15.1 Analogcomputer

Die Turing-Maschine ist ein Beispiel für einen Universalcomputer, mit dem jede beliebige mathematische Berechnung möglich ist. Die Idee eines Universalcomputers ist jedoch älter, sie geht mindestens auf Charles Babbage und Ada Lovelace im 19. Jahrhundert zurück. Mit mechanischen Mitteln versuchte Babbage, die Berechnung der abstrakten Algorithmen der Arithmetik physikalisch zu realisieren und damit jede Berechnung, die die Mathematik beschreiben kann, durch eine Maschine ausführen zu lassen.

### 15.1.1 Berechnung und natürliche Prozesse

Die Turing-Maschine ist aber nicht die einzige Möglichkeit, zu Resultaten über ein mathematisches Modell zu kommen. Jede beliebige physikalische Realisierung eines mit mathematischen Methoden beschreibbaren Prozesses kann als eine Art Berechnungsmaschine betrachtet werden. Möglicherweise ist ihr Anwendungsbereich beschränkt. Es ist aber gut möglich, dass sie eine effizientere Lösung eines Modellierungsproblems und damit eine effizientere Art der Berechnung der Lösung eines mathematischen Problems darstellt, wie die folgenden Beispiele zeigen.

**Gezeitenberechnung:** Die Bewegung des Mondes bestimmt die Gezeiten vollständig.

Newton's Gravitationstheorie ermöglichte, die Mondbewegung mit großer Genauigkeit vorherzusagen. Edmond Halley ist berühmt dafür, im Frühjahr 1715 die Sonnenfinsternis vom 3. Mai 1715 mit einer Genauigkeit von 4 Minuten vorhergesagt zu haben. Der Totalitätspfad wichen nur etwa 32km ab. Trotz dieser Erfolge blieb die Berechnung der Gezeiten schwierig. Wegen der Bedeutung für die Schifffahrt und damit für den internationalen Handel gab es bedeutende wirtschaftliche Anreize, eine rechnerische Lösung zu finden.

William Thompson, der spätere Lord Kelvin, löste dieses Problem, indem er die Bewegungen der Gezeiten in einzelne Schwingungskomponenten zerlegte und einen mechanischen Rechner konstruierte, der diese Schwingungen simulieren und mechanisch addieren konnte. Solche Gezeitenrechner, die zum Teil über 50 verschiedene Frequenzkomponenten berücksichtigten, wurden bis in die Mitte des 20. Jahrhunderts verwendet. Sie konnten innerhalb weniger Stunden die Gezeiten in einem Hafen für ein ganzes Jahr berechnen.

**Strömungsdynamik und Windkanäle:** Die Luftströmung um ein Flugzeug oder ein Formel-1 Rennauto ist auch heute noch sehr schwierig zu berechnen, aber entscheidend für den wirtschaftlichen Erfolg eines Airliners oder den Sieg im Autorennen. Sowohl Flugzeugkonstrukteure wie auch Autobauer verlassen sich nicht allein auf numerische Simulation mit dem Computer, sie verwenden auch Windkanäle, um ein verkleinertes Modell der Strömung direkt zu ermitteln. Damit konstruieren sie eine Approximation der tatsächlichen Strömung, die in gewissen Aspekten genauer und schneller ist als alle mathematischen Methoden.

**Hydraulik und Modellexperimente:** Der Klimawandel führt zu einer Häufung von Extremwetterereignissen. Nicht immer ist die Berechnung der zu erwartenden Überschwemmungssituationen machbar. Daher werden auch heute noch Modelle von Flussläufen gebaut, um die Strömung der Wassermassen im Kleinen nachzubilden. Die Strömung im Modell ist den gleichen Naturgesetzen unterworfen. Man darf das Modell als spezialisierten Computer betrachten, der das Strömungsproblem löst.

Alle drei Beispiele zeigen, dass die naheliegende digitale Berechnung mit einem Universalcomputer leicht von einem analogen physikalischen Modell in Genauigkeit oder Effizienz übertroffen wird. Die Lösungsansätze sind völlig verschieden, der für Gezeitenberechnung konstruierte Computer hilft nicht, hydraulische Fragen zu beantworten. Gibt es einen gemeinsamen Kern, eine neue Art von Maschine, die viele solche Probleme mit der gleichen Hardware lösen kann?

### 15.1.2 Differentialgleichung und Analogcomputer

Sehr viele physikalische Phänomene können durch Differentialgleichungen beschrieben werden. Der Wurf eines Massepunktes der Masse  $m$  im homogenen Schwerefeld wird durch die Vektordifferentialgleichung

$$m \frac{d^2}{dt^2} \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} 0 \\ -g \end{pmatrix} \quad (15.1)$$

beschrieben. Für diesen einfachen Fall, der den Luftwiderstand vernachlässigt, gibt es eine Lösung in geschlossener Form. Mit Berücksichtigung des Luftwiderstandes entsteht ein Problem, für das die Mathematik keine Lösung in geschlossener Form anbieten kann.

Ein moderner Universalcomputer kann die Differentialgleichung mit geeigneten mathematischen Methoden sofort lösen. Bevor solche Computer verfügbar waren, bedeutete die Berechnung einer Lösung eine hochkomplexe Rechnung. In den zwei Weltkriegen des 20. Jahrhunderts hatten die sich bekriegenden Armeen ein unmittelbares praktisches Bedürfnis, diese Gleichungen als Modelle für die Flugbahnen ihrer Artilleriegeschosse zu lösen. Daher wurden mechanische oder elektronische Analoge gebaut, die das gleiche physikalische Verhalten an den Tag legen, wie es durch die Differentialgleichung (15.1) beschrieben wird.

Als besonders flexibel stellten sich elektronische Analogrechner heraus, die auf der Basis empfindlicher sogenannter Operationsverstärker und anderer Einheiten für die Multiplikation oder mathematische Funktionen durch neu Verdachten fast beliebig komplizierte Differentialgleichungen lösen konnten. Damit wurden sogar Flugsimulatoren gebaut und in der Industrie Ingenieursaufgaben rechnerisch bewältigt. Mitte des 20. Jahrhunderts, als gerade die ersten Digitalrechner gebaut wurden und IBMs Thomas Watson noch von einem Weltmarkt für fünf Computer sprach, waren diese Analogrechner vielen Ingenieuren ähnlich vertraut wie Rechenschieber.

Die elektronischen Analogcomputer kann man durchaus als eine Art universelle Computer betrachten. Ihre Bedienung und Programmierung weicht zwar dramatisch von dem ab, was wir uns von modernen digitalen Computern gewohnt sind, aber für den vorgesehenen Einsatzbereich bestechen sie durch besondere Effizienz. Natürlich können solche Computer keine Probleme lösen, die eine Turing-Maschine nicht auch lösen kann, denn man kann die vom Analogcomputer realisierten Differentialgleichungen ja auch numerisch mit einer Turing-Maschine simulieren. Aber für gewisse Arten von Problemen waren sie in ihrer Zeit den digitalen Rechnern weit überlegen.

### 15.1.3 Quantenprozesse

Die Analogcomputer des 20. Jahrhunderts nutzten Prozesse, die mit der klassischen Mechanik und Elektrodynamik beschrieben werden können. Auch die ersten digitalen Computer konnten mit der klassischen Physik verstanden werden. Inzwischen verwenden diese die Halbleitertechnologie, die auf der modernen Festkörperphysik basiert. Die Funktionsweise hat sich aber nicht wesentlich geändert.

In den Achtzigerjahren hat der Physiker Richard Feynman vorgeschlagen, die Eigenheiten der Quantenmechanik als Basis für eine neue Art von Computer zu nutzen. Die

Prinzipien der Quantenphysik waren zu dieser Zeit bereits gut etabliert, das Standardmodell war im Wesentlichen vollständig, auch wenn es erst mit der Entdeckung des Higgs-Bosons im Jahr 2012 vollständig experimentell bestätigt war. Sofort wurden auch die ersten Algorithmen entwickelt. Zum Beispiel erdachte Peter Shor 1984 einen Algorithmus, der ein Produkt von zwei Primzahlen effizienter faktorisieren kann, als dies einer Turing-Maschine möglich ist [55]. Die Schwierigkeit der Faktorisierung von Primzahlprodukten ist der Schlüssel zur Sicherheit verbreiteter Verschlüsselungsalgorithmen auf dem Internet. Ein Quantencomputer hat also das Potential, die gesamte Internet-Sicherheitstechnik zu unterwandern.

Die Prozesse, die in einem Quantencomputer ausgenutzt werden, sind in keiner Weise magisch. Sie können zum Beispiel durch die Schrödingergleichung, eine partielle Differentialgleichung für die Wellenfunktion des Quantensystems, beschrieben werden. Als solche können sie mit bekannten numerischen Methoden auf einem klassischen Computer berechnet werden, wenn auch nur mit großem rechnerischem Aufwand. Wie auch immer ein Quantencomputer aufgebaut sein wird, wenn es denn gelingen sollte, einen zu bauen, wird er nicht in der Lage sein, nicht entscheidbare Probleme (im Sinne des Kapitels 11) zu lösen. Er wird aber möglicherweise einzelne Aufgabenstellungen, die auf einer Turing-Maschine exponentielle Zeit benötigen in polynomieller Zeit lösen können.

Im Folgenden sollen ein paar Charakteristika der Quantenmechanik diskutiert werden, die als Basis für die Konstruktion eines universellen Quantencomputers wesentlich sind. Im nächsten Abschnitt 15.2 wird dann skizziert, wie ein Quantencomputer aufgebaut werden könnte.

## Zustandsmechanik

Die klassische Physik modelliert Bahnen von Teilchen oder die Zeitentwicklung von Feldern mithilfe von Differentialgleichungen. Diese Betrachtungsweise ist im Mikrokosmos nicht anwendbar. Die Idee einer klar bestimmten Position und Geschwindigkeit muss aufgegeben werden. Die Quantenmechanik ist daher eine Mechanik der Zustände. Bereits das Atommodell von Niels Bohr hat einen Teil der Prinzipien der Mechanik durch Regeln für Elektronenbahnen ersetzt, die nach der klassischen Mechanik nicht stabil sein könnten.

Physiker schreiben diese Zustände oft als sogenannte ket-Vektoren mit der Notation

$$|\psi\rangle, |0\rangle, |1\rangle, \dots, |n\rangle, \dots, |\uparrow\rangle, |\downarrow\rangle,$$

wobei zwischen den Klammerzeichen eine Charakterisierung des Zustands steht. Die Zustände im Atom werden zum Beispiel durch einige wenige Quantenzahlen beschrieben, die manchmal als Orbitale oder Schalen bezeichnet werden.

Die Quantenmechanik braucht als mathematisches Werkzeug die lineare Algebra. Im Folgenden wird angenommen, dass der Leser eine gewisse Vertrautheit mit Vektoren und Matrizen hat, wie sie ein Kurs der linearen Algebra wie das Buch [38] vermittelt.

## Vektorraum der Zustandsvektoren

Der Zustandsraum in der klassischen Mechanik besteht aus Punkten mit Koordinaten, die Orte und Geschwindigkeiten ausdrücken. Die Zustandsvektoren der Quantenmechanik bilden einen komplexen Vektorraum, d. h. sie können mit komplexen Zahlen multipliziert und

addiert werden. Die Linearkombination

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

der Vektoren  $|0\rangle$  und  $|1\rangle$  beschreibt einen gemischten Zustand, der sich aus den Zuständen  $|0\rangle$  und  $|1\rangle$  zusammensetzt. Die Koeffizienten  $\alpha$  und  $\beta$  können beliebige komplexe Zahlen sein. In populären Darstellungen wird dies manchmal ausgedrückt, als dass das Quantensystem in beiden Zuständen gleichzeitig sein könne. Dies ist irreführend und wird im Abschnitt zur Wahrscheinlichkeit weiter unten geklärt.

Je nach quantenmechanischem System gibt es eine Vielzahl verschiedener Vektorräume, die als Zustandsräume verwendet werden können. Die Wellenmechanik von Erwin Schrödinger verwendet eine Darstellung der Zustandsvektoren als Wellenfunktionen, die von den Ortskoordinaten abhängen. Die mathematisch gleichwertige Matrizenmechanik von Werner Heisenberg verwendet abstrakte Vektoren mit komplexen Komponenten.

## Qubits

Ein *Qubit* ist ein Quantensystem, welches zwei mögliche Basiszustände hat. Viele verschiedene Systeme werden zurzeit als mögliche Grundlage für den Bau eines Quantencomputers untersucht, zum Beispiel der Spin eines Elektrons oder die Polarisation von Photonen. Für die nachfolgenden Überlegungen ist nicht wichtig, wie ein Qubit realisiert wird. Qubits spielen für die Theorie der Quantencomputer eine ähnliche Rolle wie die Zellen des Bandes einer Turing-Maschine. Auch dort ist die Realisierung für die Frage, was sich mit einer Turing-Maschine berechnen lässt, nicht von Bedeutung. Wir schreiben die beiden Basiszustände auch als  $|+\rangle = |1\rangle$  und  $|-\rangle = |0\rangle$ .

## Zusammengesetzte Systeme

Qubits werden in einem Quantencomputer zu einem größeren Quantensystem kombiniert. Jedes Qubit hat seine zwei möglichen Basiszustände. Die Basiszustände eines Zwei-Qubit-Systems werden auch als

$$|+\rangle \otimes |+\rangle, \quad |+\rangle \otimes |-\rangle, \quad |-\rangle \otimes |+\rangle \quad \text{und} \quad |-\rangle \otimes |-\rangle$$

oder abgekürzt

$$|++\rangle, \quad |+-\rangle, \quad | -+\rangle \quad \text{bzw.} \quad |--\rangle$$

geschrieben. Für größere Systeme wird die Zahl der möglichen Zustände schnell sehr groß. Ein System mit  $n$  Qubits hat  $2^n$  Basiszustände. Das System muss aber nicht in einem Basiszustand sein, es ist möglich, dass das System in einer Überlagerung

$$|\psi\rangle = \sum_{i_1, \dots, i_k=0}^1 a_{i_1, \dots, i_k} |i_1, \dots, i_k\rangle, \quad a_{i_1, \dots, i_k} \in \mathbb{C},$$

aller Basiszustände ist.

Wir werden später zeigen, dass die Zeitentwicklung eines Quantensystems durch einen linearen Operator  $U$  beschrieben wird. Nach einer vorgegebenen Zeit wird das Quantensystem, welches ursprünglich im Zustand  $|\psi\rangle$  war, wegen der Linearität von  $U$  im Zustand

$$U|\psi\rangle = \sum_{i_1, \dots, i_k=1}^2 a_{i_1, \dots, i_k} U|i_1, \dots, i_k\rangle$$

sein.

Die rechte Seite dieser Gleichung besagt, dass der Operator  $U$  die Zeitentwicklung der exponentiell vielen Basiszustände  $|i_1, \dots, i_k\rangle$  als Teil der Rechnung mit  $|\psi\rangle$  parallel berechnet. Die Quantenmechanik ermöglicht somit, dass Berechnungen exponentiell skalieren.

### Skalarprodukt

Für die Zustandsvektoren gibt es noch eine weitere Operation, nämlich das Skalarprodukt. In der linearen Algebra schreibt man das Skalarprodukt meist mit einem Punkt oder mit runden Klammern als  $(\vec{u}, \vec{v})$ . In einer Basis kann man die Vektoren als komplexe Spaltenvektoren ausdrücken, das Skalarprodukt wird dann durch die Multiplikation des transponierten und konjuguierten ersten Faktors mit dem zweiten definiert:

$$(u, v) = \bar{u}^t v = \begin{pmatrix} \bar{u}_1 & \dots & \bar{u}_n \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \bar{u}_1 v_1 + \dots + \bar{u}_n v_n = \sum_{i=1}^n \bar{u}_i v_i.$$

Die Schreibweise mit ket-Vektoren der Quantenmechanik hält eine eigene Notation für die Operation  $u \mapsto \bar{u}^t$  bereit. Der dem ket-Vektor  $|\psi\rangle$  zugeordnete Vektor wird  $\langle\psi|$  geschrieben und bra-Vektor genannt. Die Umwandlung eines ket-Vektors in einen bra-Vektor beinhaltet eine komplexe Konjugation, es gilt daher für eine Linearkombination

$$\alpha|0\rangle + \beta|1\rangle \mapsto \bar{\alpha}\langle 0| + \bar{\beta}\langle 1|.$$

Das Skalarprodukt der Zustände  $|\varphi\rangle$  und  $|\psi\rangle$  wird  $\langle\varphi|\psi\rangle$  geschrieben und hat alle Eigenschaften eines komplexen Skalarprodukts:

1. Es ist hermitesch, d. h.

$$\langle\varphi|\psi\rangle = \overline{\langle\psi|\varphi\rangle}. \quad (15.2)$$

Vertauschung der Faktoren führt zu komplexer Konjugation. Dies hat auch zur Folge, dass das Skalarprodukt eines Zustandsvektors mit sich selbst eine reelle Zahl sein muss, denn  $\langle\psi|\psi\rangle = \overline{\langle\psi|\psi\rangle}$  ist gleichbedeutend damit, dass  $\langle\psi|\psi\rangle \in \mathbb{R}$ .

2. Es ist sesquilinear, d. h. es ist linear im zweiten Faktor

$$\langle\varphi|(a_1|\psi_1\rangle + a_2|\psi_2\rangle) = a_1\langle\varphi|\psi_1\rangle + a_2\langle\varphi|\psi_2\rangle.$$

Aus der Regel (15.2) ergibt sich die konjugierte Linearität im ersten Faktor.

3. Es ist positiv definit, d. h.  $\langle\psi|\psi\rangle \geq 0$  mit Gleichheit nur für  $|\psi\rangle = 0$ .

Das Skalarprodukt eines Zustandsvektors mit sich selbst wird auch die *Norm*  $\|\psi\|^2 = \langle\psi|\psi\rangle$  genannt.

Für zusammengesetzte Quantensysteme muss das Skalarprodukt nur für die Basisvektoren definiert werden. Dabei berechnet man das Skalarprodukt für jeden Faktor einzeln und multipliziert die Skalarprodukte:

$$\langle i_1, \dots, i_n | k_1, \dots, k_n \rangle = \langle i_1 | k_1 \rangle \cdot \langle i_2 | k_2 \rangle \cdot \dots \cdot \langle i_n | k_n \rangle \quad (15.3)$$

Für Überlagerungen kann das Skalarprodukt mithilfe der Linearität berechnet werden.

## Orthogonalität

Die Basiszustände der Qubits sind so gewählt, dass die Zustandsvektoren *orthogonal* sind, also

$$\langle i | k \rangle = \delta_{ik} = \begin{cases} 1 & \text{falls } i = k \\ 0 & \text{sonst.} \end{cases}$$

Der Grund dafür ist, dass sie Eigenvektoren des Energieoperators sind, doch diese Erklärung ist für die weitere Diskussion nicht notwendig. Wichtig ist nur, dass die Basiszustände die Norm 1 haben.

Aus der Definition (15.3) des Skalarproduktes für zusammengesetzte Quantensysteme folgt

$$\langle i_1, \dots, i_n | k_1, \dots, k_n \rangle = \prod_{r=1}^n \langle i_r | k_r \rangle = \prod_{r=1}^n \delta_{i_r k_r} = \begin{cases} 1 & \text{für } i_1 = k_1, i_2 = k_2, \dots, i_n = k_n \\ 0 & \text{sonst.} \end{cases}$$

Es sind also auch die Basisvektoren des zusammengesetzten Systems orthogonal.

## Wahrscheinlichkeit

Das Skalarprodukt in einem reellen Vektorraum misst, wie "ähnlich" zwei Vektoren sind. Wenn das Skalarprodukt verschwindet, sind die Vektoren so unähnlich wie möglich, sie zeigen in orthogonale Richtungen. Ist das Skalarprodukt  $\langle \vec{u}, \vec{v} \rangle = |\vec{u}| \cdot |\vec{v}|$ , dann haben die Vektoren die gleiche Richtung. Noch besser drückt diese Art von "Ähnlichkeit" der Quotient

$$\cos \alpha = \frac{\langle \vec{u}, \vec{v} \rangle}{|\vec{u}| \cdot |\vec{v}|} = \left( \frac{\vec{u}}{|\vec{u}|}, \frac{\vec{v}}{|\vec{v}|} \right)$$

aus, der für die Berechnung des Zwischenwinkels verwendet wird. Ganz rechts sind die Vektoren auf Länge 1 normiert worden. Wir erwarten daher, dass das Skalarprodukt nur für Vektoren der Länge 1 eine physikalisch interpretierbare Aussage über die Zustandsvektoren machen wird. Wir auferlegen daher den Zustandsvektoren der Quantenmechanik die zusätzliche *Normierungsbedingung*

$$\| |\psi\rangle \|^2 = \langle \psi | \psi \rangle = 1. \quad (15.4)$$

Ist die Bedingung noch nicht erfüllt, kann sie durch Division durch die Norm erzwungen werden.

Die Quantenmechanik interpretiert das Skalarprodukt  $\langle a|b\rangle$  als eine Aussage über das Resultat einer Messung. Ein System im normierten Zustand  $|b\rangle$  wird mit einem System im normierten Zustand  $|a\rangle$  verglichen. Man möchte wissen, inwiefern es sich wie ein System verhält, das im Zustand  $|a\rangle$  ist. Das Skalarprodukt ist eine beliebige komplexe Zahl mit Betrag  $\leq 1$ . Das Betragsquadrat  $|\langle a|b\rangle|^2 \in [0, 1]$  ist eine Zahl zwischen 0 und 1 und wird als die Wahrscheinlichkeit interpretiert, dass die Messung des Systems, welches ursprünglich im Zustand  $|b\rangle$  war, den Zustand  $|a\rangle$  findet.

Die Orthogonalitätseigenschaft der Basisvektoren der Qubits bedeutet also, dass ein Qubit im Zustand  $|0\rangle$  bei der Messung mit Wahrscheinlichkeit  $|\langle 1|0\rangle|^2 = \delta_{01}^2 = 0$  im Zustand 1 gefunden wird. Die Basiszustände des Qubits sind also so gewählt, dass sie bei einer Messung nicht verwechselt werden können.

Die Koeffizienten einer Überlagerung

$$|\psi\rangle = \sum_{i=1}^n \alpha_i |i\rangle$$

bekommen damit auch eine physikalische Interpretation. Das Skalarprodukt mit  $|k\rangle$  ist

$$\langle k| \sum_{i=1}^n \alpha_i |i\rangle = \sum_{i=1}^n \alpha_i \langle k|i\rangle = \sum_{i=1}^n \alpha_i \delta_{ik} = \alpha_k.$$

Daher ist das Betragsquadrat  $|\alpha_k|^2$  die Wahrscheinlichkeit, bei der Messung von  $|\psi\rangle$  den Zustand  $k$  vorzufinden.

### Vollständigkeit der Basis

Die Basisvektoren eines Qubits beschreiben alle möglichen Zustände, in denen das Quantensystem vorgefunden werden kann. Die Wahrscheinlichkeiten, bei der Messung eines Überlagerungszustandes  $|\psi\rangle$  die einzelnen Basiszustände zu finden, müssen sich daher zu 1 summieren:

$$1 = \sum_{k=1}^n |\langle k|\psi\rangle|^2 = \sum_{k=1}^n \left| \langle k| \sum_{i=1}^n \alpha_i |i\rangle \right|^2 = \sum_{k=1}^n \left| \sum_{i=1}^n \alpha_i \langle k|i\rangle \right|^2 = \sum_{k=1}^n \left| \sum_{i=1}^n \alpha_i \delta_{ik} \right|^2 = \sum_{k=1}^n |\alpha_k|^2.$$

### Unitarität

Operationen mit Zustandsvektoren werden in der Quantenmechanik durch lineare Operatoren beschrieben. Im Fall der Wellenmechanik können das partielle Differentialoperatoren sein, im Falle der Matrizenmechanik auch nur Matrizen.

Da Skalarprodukte als Wahrscheinlichkeiten dafür interpretiert werden, Systeme in einem bestimmten Zustand vorzufinden, darf sich das Skalarprodukt eines Zustandsvektors mit sich selbst, also die gesamte Wahrscheinlichkeit, nicht ändern, sie muss 1 bleiben. Für eine Matrix  $A$  würde dies bedeuten, dass die Skalarprodukte  $(A\vec{u}, A\vec{u}) = (\vec{u}, \vec{u})$  nicht verändert werden. Solche Matrizen werden *unitär* genannt.

Ein linearer Operator  $U$ , der auf ket-Vektoren wirkt, muss eine entsprechende Bedingung erfüllen. Schreiben wir  $U|\varphi\rangle = |\psi\rangle$ , dann muss

$$\langle \varphi|\varphi\rangle = \langle \psi|\psi\rangle$$

sein. Wir schreiben auch  $\langle \psi | = \langle \varphi | U^*$ . Für beliebige  $|u\rangle$  und  $|v\rangle$  kann man dann auch

$$\langle u | U^* U | v \rangle = \langle u | v \rangle \quad (15.5)$$

schreiben. Auch diese Bedingung besagt, dass  $U$  unitär ist. Da (15.5) die Operatorgleichung  $U^* U = I$  impliziert, ist ein solcher Operator immer auch invertierbar mit der Inversen  $U^*$ .

Die Schrödinger-Gleichung, die wir nicht im Detail studieren werden, beschreibt die Zeitentwicklung in einem Quantensystem. Da die Wahrscheinlichkeiten erhalten bleiben, muss auch die Schrödinger-Gleichung einen unitären Operator für die Zeitentwicklung zur Folge haben. Insbesondere ist die Zeitentwicklung in einem Quantensystem immer umkehrbar. Auch in der Quantenmechanik gilt also der aus der klassischen Mechanik bekannte Grundsatz, dass jeder Prozess auch in umgekehrter Zeitrichtung ablaufen könnte.

Im Gegensatz zur klassischen Mechanik, wo auch die ausgedehnte Beobachtung eines Systems zum Beispiel in einem Video nichts an der Umkehrbarkeit ändert, ändert jede Messung des Quantensystems den Zustand. Die Unitarität gilt nur zwischen zwei Messzeitpunkten.

## 15.2 Schaltungen und Quantenschaltungen

In Abschnitt 13.3.4 wurde das NP-vollständige Problem *CIRCUIT* eingeführt, welches wir damals auch als Hardwaredurchführung des *SAT*-Problems verstanden haben. Jede beliebige logische Funktion lässt sich aus Gattern für die Operationen UND, ODER und NICHT zusammenbauen. Fast noch wichtiger ist aber die Beobachtung, dass moderne Computer sich aus den genannten Gattern bauen lassen. Eine spektakuläre Demonstration dieser Tatsache gelang dem Apollo-Guidance Computer (AGC), den die Apollo-Raumschiffe für die Navigation in die Mondumlaufbahn und zurück verwendeten, und der dem Lunar Module (LM) die Landung auf dem Mond und den Wiederaufstieg ermöglichte. Der ab Apollo 7 eingesetzte Block-II AGC war mit ungefähr 5600 NOR-Gattern mit jeweils drei Inputs aufgebaut worden.

Der Erfolg des AGC und anderer Computer, die integrierte Schaltungen als Basis verwendeten, namentlich das IBM System/360, führte in den Sechzigerjahren zur Konzeption einer Familie von integrierten Logikbausteinen. Die 7400er-Reihe von Texas Instruments war speziell populär und war der de facto Standard. Eine große Zahl immer komplexerer Logikfunktionen wurden auf einem Chip integriert und machten mit der Zeit den Bau von Computern immer einfacher. Der 1970 eingeführte 74181, zum Beispiel, realisiert eine 4-Bit *arithmetic logic unit* (ALU). Mehrere solche Bausteine konnten zusammengeschaltet werden und so alle Rechen- und Logikfunktionen einer CPU übernehmen. Die Minicomputer-Entwicklungen der 70er Jahre wurden von der Verfügbarkeit solcher Bausteine getragen. In den 80er Jahren übernahmen die Mikroprozessoren, die eine ganze CPU auf einem einzigen Chip integrierten, die Führung.

Die logischen Gatterbausteine realisierten für die Digitalelektronik, was die Operationsverstärker für die analoge Elektronik geleistet hatten. Austauschbare Bausteine mit mathematisch klar definierter Funktion ermöglichen die Konstruktion von Maschinen mit vorhersagbarer Leistung.

### 15.2.1 Universelle Quantencomputerbausteine

Die Entwicklung vielfältig verwendbarer, standardisierter Logik-Bausteine gab der Entwicklung von Universalcomputern wesentliche Impulse. Für einen Quantencomputer könnte eine entsprechende Familie von zusammenschaltbaren Grundfunktionen einen Entwicklungsschub geben.

Die Familie von Operationen muss dazu ausreichend flexibel sein, damit jede beliebig komplexe Quantencomputerfunktion damit aufgebaut werden kann. Sie muss aber auch einfach genug sein, damit die industrielle Herstellung großer Stückzahlen von zuverlässigen Bausteinen zu tiefen Kosten möglich wird. Tatsächlich zeigt der Approximationssatz 15.1, dass jede unitäre Matrix beliebig genau durch die nachstehend beschriebenen Grundoperationen approximiert werden kann. Die Einschränkung auf unitäre Matrizen ist sinnvoll, weil die Zeitentwicklung eines Quantensystems nach Abschnitt 15.1.3 zwischen Messungen durch unitäre Matrizen beschrieben wird. Dies bedeutet aber auch, dass die Bausteine selbst unitär sind, sie müssen also umkehrbar sein.

Die klassischen logischen Operationen sind im Allgemeinen nicht umkehrbar. Ist der Ausgang einer UND-Verknüpfung mit zwei Eingängen falsch, lässt sich daraus nicht mehr rekonstruieren, welcher der beiden Eingänge falsch war, oder ob vielleicht sogar beide falsch waren. Selbst wenn einer der Eingänge als falsch bekannt ist, lässt sich immer noch nicht ermitteln, welchen logischen Wert der andere Input hatte. Die Logikgatter der Digitalelektronik sind also kein geeigneter Ausgangspunkt für die Konzeption von Quantengattern.

### 15.2.2 Die Hadamard-Operation

Damit das Parallelisierungspotential eines Quantencomputers ausgeschöpft werden kann, muss ein Überlagerungszustand von vielen Qubits bereitgestellt werden, der alle Basiszustände in bekannter Phasenlage enthält.

#### Ein Qubit

Die *Hadamard-Operation* oder das Hadamard-Gatter erzeugt aus einem Qubit im Basiszustand  $|0\rangle$  den Überlagerungszustand

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

Die unitäre Matrix, die dies bewerkstellt, ist

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

#### Hadamard-Operation auf einem $n$ -Qubit-Register

Aus der Hadamard-Operation für ein einzelnes Qubit kann jetzt auch eine Hadamard-Operation für ein  $n$ -Qubit-Register aufgebaut werden. Dazu lässt man die Matrix auf jedem Faktor operieren:

$$H^{\otimes n}|i_1, \dots, i_n\rangle = H|i_1\rangle \otimes \dots \otimes H|i_n\rangle.$$

Für ein 2-Qubit-Register ausgeschrieben bedeutet dies

$$\begin{aligned} H^{\otimes 2}|0,0\rangle &= H|0\rangle \otimes H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ &= \frac{1}{2}(|0\rangle \otimes |0\rangle + |0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle). \end{aligned}$$

Für allgemeines  $n$  erzeugt  $H^{\otimes n}$  aus dem Zustand  $|0 \dots 0\rangle$  eine Überlagerung aller  $2^n$  Basiszustände.

### Invertierbarkeit

Die Matrix  $H$  ist symmetrisch:  $H = H^t$ , aber auch orthogonal:

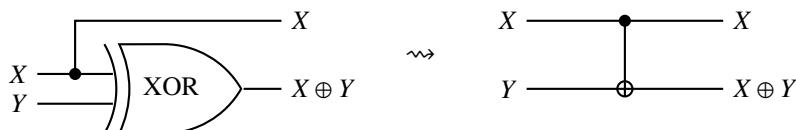
$$H^*H = H^tH = H^2 = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}^2 = \frac{1}{2} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = I.$$

Die Hadamard-Operation ist also sogar ihre eigene Inverse.

### 15.2.3 CNOT

Die XOR-Operation  $(X, Y) \mapsto X \oplus Y$  kann man auch als eine vom Eingang  $X$  gesteuerte Negation des Eingangs  $Y$  betrachtet werden. Wenn  $X$  falsch ist, passiert nichts mit  $Y$ , wenn  $X$  wahr ist, wird  $Y$  negiert. Ähnlich wie die UND-Operation ist auch XOR nicht umkehrbar, aus dem Resultat kann man nicht mehr ablesen, ob etwas invertiert worden ist oder nicht. Die Operation wird aber invertierbar, wenn man einen der Inputs kennt. Die Abbildung  $(X, Y) \mapsto (X, X \oplus Y)$  ist invertierbar.

Aus der XOR-Operation (Abbildung links) mit dem zusätzlichen Ausgang  $X$  kann jetzt die *CNOT-Quantenoperation* konstruiert werden, die als



(rechts) dargestellt wird. Die Wahrheitstabelle der XOR-Operation überträgt sich sofort auch in die unitäre Matrix  $C$ , die die CNOT-Operation beschreibt:

$X$	$Y$	$X$	$X \oplus Y$	
0	0	0	0	$ 0\rangle \otimes  0\rangle \mapsto  0\rangle \otimes  0 \oplus 0\rangle$
0	1	0	1	$ 0\rangle \otimes  1\rangle \mapsto  0\rangle \otimes  0 \oplus 1\rangle$
1	0	1	1	$ 1\rangle \otimes  0\rangle \mapsto  1\rangle \otimes  1 \oplus 1\rangle$
1	1	1	0	$ 1\rangle \otimes  1\rangle \mapsto  1\rangle \otimes  1 \oplus 0\rangle$

$\rightsquigarrow$

hat die Matrix  $C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ .

Auch diese Operation ist nicht nur invertierbar, sondern sogar ihre eigene Inverse, denn  $C^2 = I$ .

### 15.2.4 Das Toffoli-Gate

Die XOR-Operation war nicht invertierbar, konnte aber unitär gemacht werden, indem man einen der Inputs auch im Output verwendet hat. Die AND-Operation ist ebenfalls nicht invertierbar. Das *Toffoli-Gatter* macht sie durch zwei Maßnahmen unitär. Zunächst müssen die beiden Inputs der UND-Operation behalten werden. Damit hat man aber drei Output-Qubits statt nur zwei, und die Operation ist auch wieder nicht invertierbar.

Als zweite Maßnahme wird die UND-Verknüpfung dazu verwendet, um kontrolliert einen dritten Input zu invertieren. Die Operation ist also

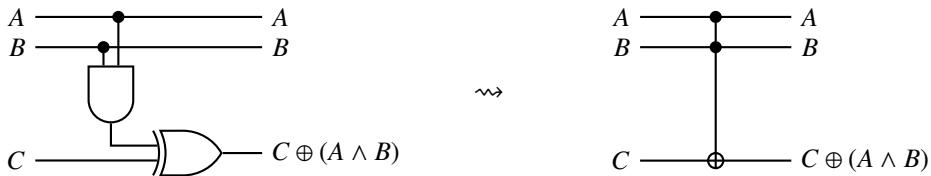
$$T : |a\rangle \otimes |b\rangle \otimes |c\rangle \mapsto |a\rangle \otimes |b\rangle \otimes |c \oplus (a \wedge b)\rangle.$$

So werden wieder drei Qubits auf drei Qubits abgebildet und es ist wieder möglich, dass die Operation invertierbar ist. Tatsächlich ist die Matrix dieser Operation

$A$	$B$	$C$	$(A \wedge B) \oplus C$	
0	0	0	0	
0	0	1	1	
0	1	0	0	
0	1	1	1	
1	0	0	0	
1	0	1	1	
1	1	0	1	
1	1	1	0	

$$\rightsquigarrow T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Rot hervorgehoben ist der Teil, in dem die Matrix von der Einheitsmatrix abweicht. Auch diese Operation ist symmetrisch und unitär. Die Operation wird oft mit dem Symbol



dargestellt (klassische logische Operation links, Quantenoperation rechts).

### 15.2.5 Phasenverschiebung

Alle bisher diskutierten Abbildungen waren nicht nur invertierbar und unitär, sondern sogar Involutionen, d. h. ihre eigene Inverse. Insbesondere hat sich die Phase der verschiedenen Zustände gegeneinander nicht verändert. Dies kann mit der  $90^\circ$ -Phasenverschiebung auf einem einzelnen Qubit mit der Matrix

$$P = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} : \left\{ \begin{array}{l} |0\rangle \mapsto |0\rangle \\ |1\rangle \mapsto i|1\rangle \end{array} \right.$$

geändert werden. Die Matrix ist unitär, denn

$$P^* P = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -i^2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I.$$

### 15.2.6 Der Approximationssatz

Die in den voranstehenden Abschnitten beschriebenen Quantengatter sind nicht nur theoretische Konstruktionen, sie konnten auch schon realisiert werden. Ob sie tatsächlich als Basis für die Konstruktion zukünftiger Quantencomputer verwendet werden, ist noch nicht klar, denn es gibt noch viele weitere Kandidaten für Quantengatter, aus denen die oben vorgestellten Gatter aufgebaut werden könnten oder die ihrerseits aus den vorgestellten Gattern aufgebaut werden könnten. Dies ist nicht unähnlich der Situation für die klassischen logischen Operationen. Die de-morganschen Regeln ermöglichen, klassische logische Operationen durch andere auszudrücken. Dies ermöglichte, den AGC allein mit NOR-Gattern zu realisieren. Ähnliches gilt für die verschiedenen Quantengatter.

Die vorgestellten Gatter zeigen aber, dass es tatsächlich einen Satz von universellen Grundbausteinen gibt, mit dem jede beliebige Quantenoperation approximiert werden kann. Es gilt nämlich der folgende Satz.

**Satz 15.1** (Deutsch-Kitaev). *Jede unitäre  $d \times d$ -Matrix kann durch ein Produkt von Hadamard-Operationen, Toffoli-Gattern und Phasenverschiebungen beliebig genau approximiert werden.*

Mehr Information zur Konstruktion von Quantencomputern und einen Beweis des Satzes findet man zum Beispiel in [3].

### 15.2.7 Aufbau eines universellen Quantencomputers

Die Quantengatter sind dazu geeignet, Quantenfunktionen zu berechnen. Sie erweitern damit die technischen Möglichkeiten, die zur Lösung rechnerischer Probleme zur Verfügung stehen. Wie konstruiert man sinnvollerweise eine Maschine, die diese neuen Fähigkeiten optimal nutzt?

Zu Beginn der PC-Ära waren die meisten Prozessoren nur zu Ganzzahlarithmetik in der Lage. Gleitkommaarithmetik musste in Software realisiert werden und war entsprechend langsam. PC-Bauer integrierten daher spezielle Koprozessoren, mit denen sich die Gleitkommaberechnungen beschleunigen ließen. Die Steuerung des Programmablaufs obliegt weiterhin dem Prozessor, der Koprozessor wird nur zugeschaltet, wenn Gleitkommarechnungen anstehen.

Moderne Graphikkarten (GPU) können ganz ähnlich für die hochparallelisierte Lösung anspruchsvoller numerischer Aufgaben verwendet werden. Sie sind zwar in beschränktem Umfang dazu in der Lage, Programmcode auszuführen, der gesamte Ablauf wird aber weiterhin durch den Prozessor gesteuert. Zum Beispiel kompiliert der Prozessor den von der GPU auszuführenden Code, überträgt den Code und die Ausgangsdaten in die GPU und holt am Ende der Berechnung die Resultate dort ab. Moderne Prozessoren integrieren sogar die GPU auf dem Prozessorchip.

Für Anwendungen der künstlichen Intelligenz sind spezielle “neural engines” entwickelt und den Prozessoren hinzugefügt worden, die auf die Auswertung eines neuronalen Netzwerks optimiert sind. Dieses “Koprozessor-Modell” wurde daher auch für einen universellen Quantencomputer vorgeschlagen.

Ein universeller Quantencomputer besteht also aus einem klassischen Computer, der die Verschaltung von geeigneten Quantengattern zur Implementation einer Quantenfunk-

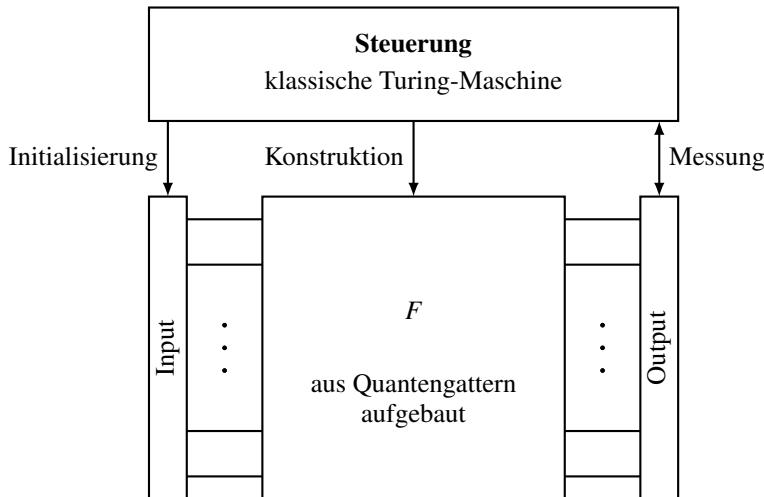


Abbildung 15.1: Aufbau eines universellen Quantencomputers. Eine klassische Turing-Maschine konstruiert die Quantenfunktion  $F$  aus Standardbausteinen, initialisiert das Input-Qubit-Register und kontrolliert die Messung des Output-Qubit-Registers.

tion steuert, die beteiligten Qubit-Register initialisiert und die dafür nötigen Messungen durchführt (Abbildung 15.1). Von der Realisierung eines solchen universellen Quantencomputers ist der Stand der Technik aber noch sehr weit entfernt.

## 15.3 Komplexitätsklassen

Die neue Art von Computer, die in Abschnitt 15.2 beschrieben worden ist, führt zu einer neuen Klasseneinteilung der Sprachen, die die in Kapitel 12 eingeführte Klassifizierung in P und NP verfeinert. Wie bereits in Abschnitt 15.1.3 ausgeführt, können nicht entscheidbare Probleme auch von einem Quantencomputer nicht gelöst werden, aber es besteht die Hoffnung, dass einige Probleme in  $\text{NP} \setminus \text{P}$  effizienter lösbar sind. Möglicherweise können sogar gewisse Probleme außerhalb von NP schneller gelöst werden. Es wäre aber eher überraschend, wenn ein Quantencomputer ein NP-vollständiges Problem in polynomieller Zeit lösen und damit alle NP-Probleme einer polynomiellen Lösung zugänglich machen könnte.

### 15.3.1 Probabilistische Algorithmen und BPP

In Abschnitt 3.4 wurde die Idee aufgeworfen, das Fehlen eines Orakels durch ‘‘Würfeln’’ zu kompensieren. Ein probabilistischer endlicher Automat ist zwar nicht deterministisch, aber er findet mit einer gewissen Wahrscheinlichkeit einen akzeptierenden Pfad für ein Wort durch den Automaten. Dieser Ansatz funktioniert auch für nichtdeterministische Turing-Maschinen und wird in der Praxis auch tatsächlich verwendet.

### Der Miller-Rabin-Test

Für viele Verschlüsselungsalgorithmen werden große Primzahlen benötigt. Ein nichtdeterministischer Algorithmus kann die Faktoren einer zusammengesetzten Zahl nichtdeterministisch finden und durch Ausmultiplizieren in polynomieller Zeit prüfen, dass sie zusammengesetzt ist. Der älteste bekannte deterministische Algorithmus für den Primzahltest ist die Probdivision, die eine Zahl durch alle potentiellen Faktoren zu teilen versucht. Wird kein Teiler gefunden, ist die Zahl prim. Dieser Algorithmus braucht exponentielle Zeit.

Der Miller-Rabin-Test verwendet einen probabilistischen Schritt und gelangt in polynomieller Zeit zu einer Schlussfolgerung darüber, ob eine Zahl prim ist, allerdings nur mit einer beschränkten Wahrscheinlichkeit. Er basiert auf dem kleinen Satz von Fermat.

**Satz 15.2** (kleiner Satz von Fermat). *Für jede Primzahl  $p$  und jede natürliche Zahl  $a$  mit  $0 < a < n$  gilt*

$$a^{p-1} \equiv 1 \pmod{p}. \quad (15.6)$$

Gilt die Folgerung (15.6) für eine ungerade Zahl  $n$  in der Form  $a^{n-1} \equiv 1 \pmod{n}$  nicht, dann kann  $n$  auch keine Primzahl sein.

**Satz 15.3** (Fermat-Test). *Eine natürliche Zahl  $n > 1$  ist keine Primzahl, wenn es eine natürliche Zahl  $a$ ,  $0 < a < n$ , gibt, für die*

$$a^{n-1} \not\equiv 1 \pmod{n} \quad (15.7)$$

gilt.  $a$  heißt ein Zertifikat für die Zusammengesettheit von  $n$ .

Möchte man überprüfen, ob  $n$  eine Primzahl ist, kann man zufällig eine Zahl  $a$  wählen und die Bedingung (15.7) nachrechnen. Abbildung 15.2 zeigt die Zertifikate  $a$  für eine zusammengesetzte Zahl  $n$  als schwarze Quadrate. Nicht überraschend enthalten die Spalten von Primzahlen gar keine solche Zertifikate. In den Spalten der zusammengesetzten Zahlen gibt es dagegen immer ziemlich viele schwarze Quadrate. Wählt man also zufällig eine Zahl  $a$  und erfüllt sie die Bedingung (15.6), dann entspricht dies einem weißen Quadrat in der Abbildung. Für eine zusammengesetzte Zahl ist die Wahrscheinlichkeit dafür eher klein, weil es meistens in der Spalte einer zusammengesetzten Zahl viel mehr schwarze als weiße Quadrate gibt. Man kann zeigen, dass für den Fermat-Test gilt: Die Wahrscheinlichkeit, dass für eine Zahl  $a$  die Bedingung (15.6) erfüllt ist, ist kleiner als  $\frac{1}{2}$ . Die Bedingung (15.6) garantiert also nicht automatisch, dass eine Zahl prim ist, aber es folgt mindestens eine Wahrscheinlichkeitsaussage.

Édouard Lucas ist es gelungen, den Fermat-Test umzukehren.

**Satz 15.4** (Lucas). *Eine natürliche Zahl  $n$  ist genau dann eine Primzahl, wenn es eine natürliche Zahl  $a$  gibt mit  $0 < a < n$ , so dass*

$$a^{n-1} \equiv 1 \pmod{n} \qquad \text{und} \qquad a^m \not\equiv 1 \pmod{n}$$

für alle Teiler  $m$  von  $n - 1$  gilt.

Die Schwierigkeit dieses Kriteriums ist, dass zu seiner Anwendung die Faktoren der großen Zahl  $n - 1$  bekannt sein müssen, was nicht einfacher ist als die Faktoren von  $n$ .

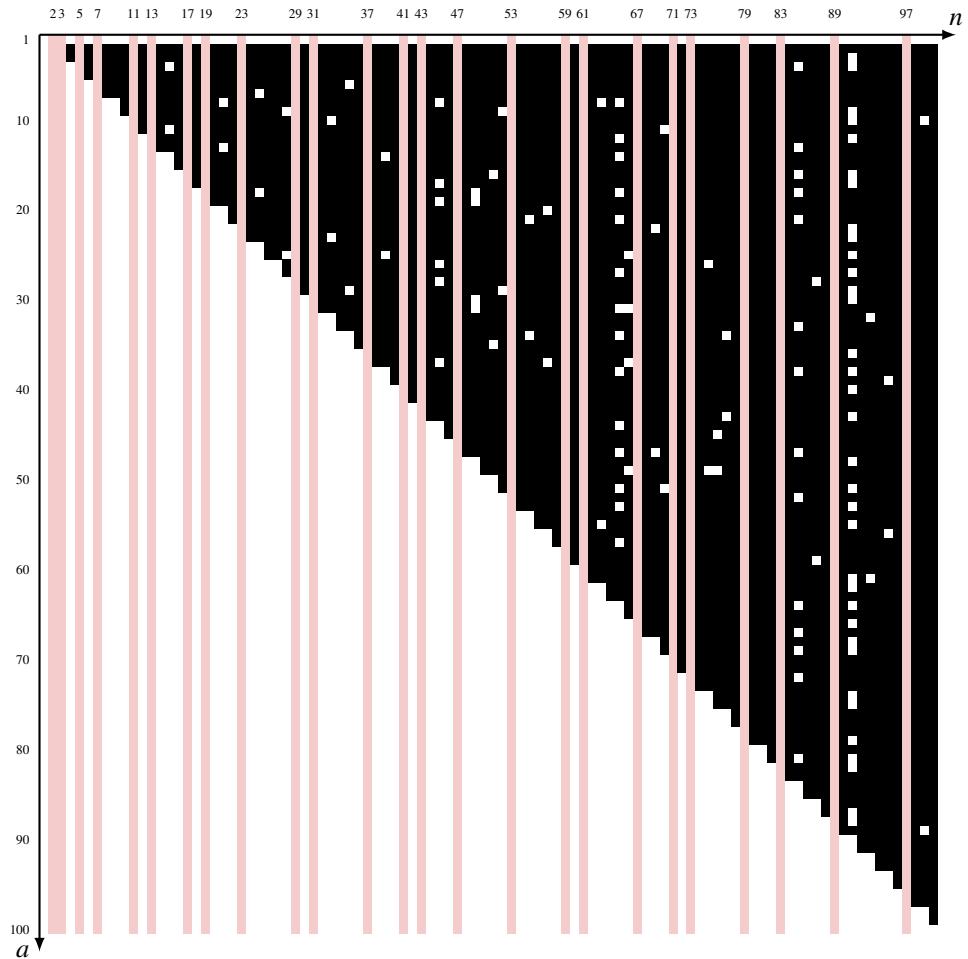


Abbildung 15.2: Zahlen  $a$ , die zertifizieren, dass  $n$  zusammengesetzt sind, sind als schwarze Quadrate dargestellt. Für Primzahlen gibt es keine Zahlen  $a$ , verdeutlicht durch die hellroten Streifen. Nichtprimzahlen haben eine große Zahl von Zertifikaten dafür, dass sie zusammengesetzt sind.

zu bestimmen. Gewisse Faktoren sind aber leicht zu finden, zum Beispiel kann man die Anzahl der Primfaktoren 2 von  $n - 1$  leicht zählen und so  $n - 1 = 2^s d$  schreiben, wobei  $d$  eine ungerade Zahl ist. Man kennt also mindestens die Teiler  $d, 2d, \dots, 2^s d$ . Wenn wir statt aller Teiler nur diese Teiler testen, werden wir nicht mehr mit Sicherheit schließen können, dass  $n$  eine Primzahl ist, aber es folgt wieder eine Wahrscheinlichkeitsaussage.

**Satz 15.5** (Miller-Rabin). *Sei  $n$  eine ungerade Zahl und  $a$  eine zufällig gewählte natürliche Zahl mit  $0 < a < n$ . Sei außerdem  $n - 1 = 2^s d$  mit  $d$  ungerade (die ungeraden Primfaktoren von  $n - 1$ ). Falls*

$$a^d \not\equiv \pm 1 \pmod{n} \quad \text{und} \quad a^{2^r d} \not\equiv -1 \pmod{n} \quad \forall r \in \mathbb{N} \quad (0 \leq r \leq s-1), \quad (15.8)$$

dann ist  $n$  zusammengesetzt. Die Wahrscheinlichkeit dafür, dass für eine zusammengesetzte Zahl  $n$  die Bedingungen

$$a^d \equiv \pm 1 \pmod{n} \quad \text{oder} \quad a^{2^r d} \equiv -1 \pmod{n} \quad \forall r \in \mathbb{N} \quad (0 \leq r \leq s-1), \quad (15.9)$$

erfüllt sind, ist  $> \frac{3}{4}$ .

Analog zur Darstellung für den Fermat-Test zeigt Abbildung 15.3 die Zertifikate dafür, dass eine ungerade Zahl  $n$  eine zusammengesetzte Zahl ist, als schwarze Rechtecke. Erfüllt  $n$  für ein zufällig gewähltes  $a$  die Bedingung (15.8), ist  $n$  mit Wahrscheinlichkeit  $< \frac{1}{4}$  eine Primzahl. Die verschiedenen möglichen Zahlen  $a$  sind zwar nicht im stochastischen Sinn unabhängig, trotzdem ist es zulässig zu argumentieren, dass eine große Zahl  $n$ , für die für  $k$  zufällig gewählte  $a$  die Bedingung (15.8) zutrifft, ungefähr mit Wahrscheinlichkeit  $1 - 4^{-k}$  eine Primzahl ist.

Vor der Entdeckung des polynomiellen AKS-Algorithmus [2] durch Agrawal, Kayal und Saxena wurden große Primzahlen für kryptographische Anwendungen dadurch gefunden, dass man bei einer großen ungeraden Zahl  $n$  anfing und sie dem Miller-Rabin-Test unterwarf. Falls die Zahl den Test nicht bestand, hat man sie um 2 vergrößert und erneut getestet. So konnte man in polynomieller Zeit eine Zahl finden, die mit hoher Wahrscheinlichkeit eine Primzahl war. Es bleibt aber immer eine positive Wahrscheinlichkeit, dass die Zahl keine Primzahl ist.

## Probabilistische Algorithmen

Der Primzahltest von Miller-Rabin ist ein Repräsentant einer neuen Klasse von Algorithmen und damit auch einer neuen Komplexitätsklasse. Der Algorithmus läuft in polynomieller Zeit und entscheidet eine Sprache, nämlich ob eine Zahl prim ist oder nicht. Er verwendet aber einen durch den Zufall bestimmten Schritt. Die Schlussfolgerung ist daher auch nicht in jedem Fall korrekt, die Wahrscheinlichkeit für einen Fehler ist nicht 0.

Der Primzahltest ist bei weitem nicht der einzige Algorithmus, der Zufallsschritte enthält. Monte-Carlo-Simulationen erzeugen Zufallsstichproben und berechnen daraus Approximationen um Resultate zu bekommen, die anders rechnerisch nur schwer zu erhalten sind. Sind zum Beispiel  $X$  und  $Y$  im Intervall  $[0, 1]$  gleichverteilte Zufallszahlen, dann ist die Wahrscheinlichkeit

$$P(X^2 + Y^2 < 1)$$

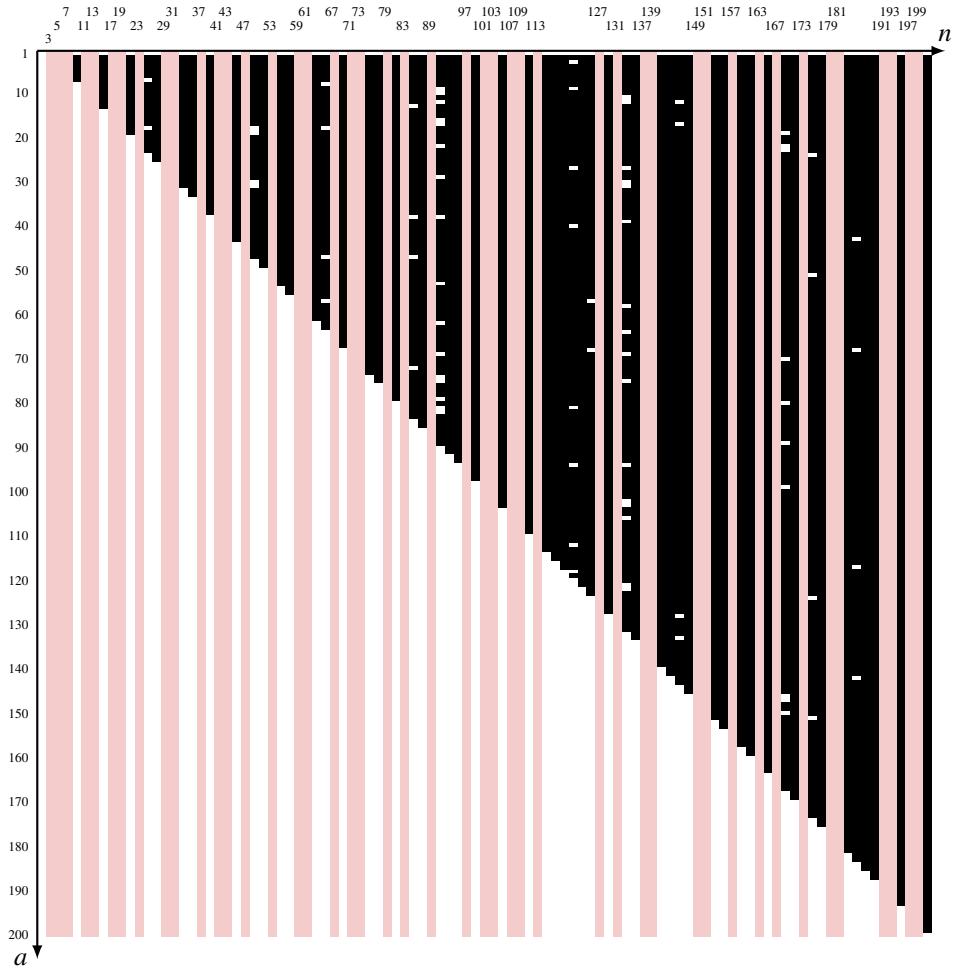


Abbildung 15.3: Zahlen  $a$ , die zertifizieren, dass die ungerade Zahl  $n$  nach dem Rabin-Miller-Kriterium zusammengesetzt ist, sind als schwarze Rechtecke dargestellt. Für Primzahlen gibt es keine Zahlen  $a$ , verdeutlicht durch die hellroten Streifen. Nichtprimzahlen haben eine große Zahl von Zertifikaten dafür, dass sie zusammengesetzt sind. Die Wahrscheinlichkeit, dass eine zusammengesetzte Zahl das Rabin-Miller-Kriterium (15.9) erfüllt, ist klein.

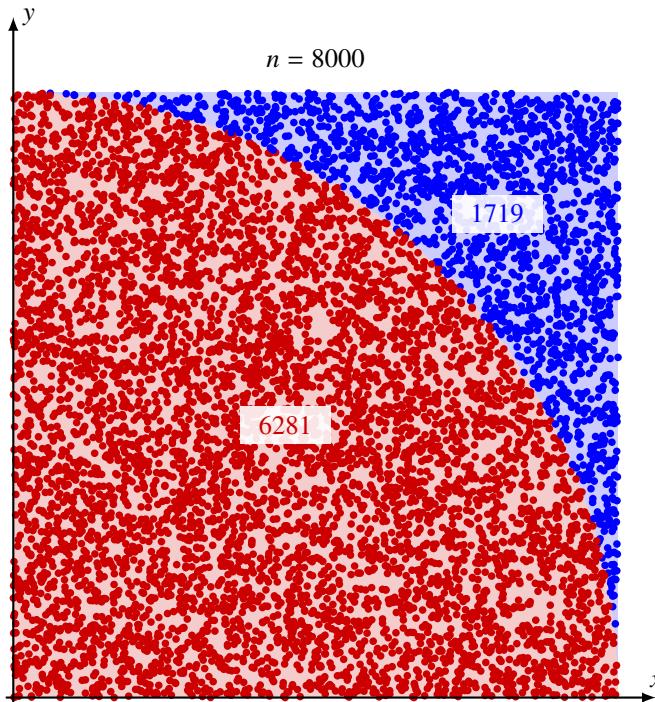


Abbildung 15.4: Monte-Carlo-Simulation zur Berechnung von  $\pi$ . Die Wahrscheinlichkeit, dass ein zufälliger Punkt  $P = (x, y)$  mit im Intervall  $[0, 1]$  gleichverteilten Koordinaten in den Einheitskreis fällt, ist  $\frac{\pi}{4}$ . Von 8000 zufällig gewählten Punkten fallen in diesem Beispiel 6281 in den Einheitskreis, was die Näherung  $\frac{\pi}{4} = 0.785398 \approx 0.78512$  ergibt.

die Wahrscheinlichkeit, dass ein zufällig im Einheitsquadrat gewählter Punkt in den Viertelkreis (Abbildung 15.4) fällt. Da der Flächeninhalt des Viertelkreises den Anteil  $\frac{\pi}{4} \approx 0.7853$  des Quadrates ausmacht, kann man durch Zählen der Punkte im Viertelkreis eine Approximation von  $\pi$  erhalten.

Monte-Carlo-Simulationen sind in den Naturwissenschaften, im Ingenieurswesen und im Versicherungswesen weit verbreitet.

### Die Klasse BPP

Wenn ein probabilistischer Algorithmus vom Zufall keinen Gebrauch macht, bleibt offenbar ein deterministischer Algorithmus mit polynomieller Laufzeit übrig. Probabilistische Algorithmen stellen also eine Erweiterung der polynomiellen Algorithmen dar. Sie sind damit in der Lage, jedes Problem in  $P$  zu lösen, aber noch ein paar weitere, denen nicht ohne Hilfe eines zufallsbestimmten Schrittes in polynomieller Zeit beizukommen ist. Dies führt uns auf eine neue Komplexitätsklasse.

**Definition 15.6** (Komplexitätsklasse BPP). *Eine Sprache  $L$  gehört zur Klasse BPP (englisch bounded error probability polynomial time), wenn es eine probabilistische Turing-*

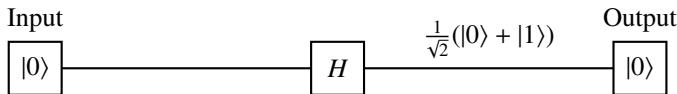


Abbildung 15.5: Quantencomputerschaltung, die zufällig 0 und 1 mit gleicher Wahrscheinlichkeit erzeugen kann. Sie zeigt, dass ein Quantencomputer mindestens so leistungsfähig ist wie probabilistische Algorithmen.

*Maschine M gibt, für die gilt:*

1. *M läuft auf allen Inputs in polynomieller Zeit.*
2. *Falls  $w \in L$  ist, ist die Wahrscheinlichkeit  $P(M \text{ akzeptiert } w) \geq \frac{2}{3}$ .*
3. *Falls  $w \notin L$  ist, ist die Wahrscheinlichkeit  $P(M \text{ akzeptiert } w \text{ nicht}) \geq \frac{1}{3}$ .*

Die Klasse BPP ist eine Erweiterung der Klasse P. Die Beziehung zwischen BPP und NP ist jedoch nicht bekannt. Es konnte bis jetzt weder  $BPP \subset NP$  noch  $NP \subset BPP$  gezeigt werden. Die letzte Ungleichung würde bedeuten, dass jedes NP-Problem mithilfe des Zufalls in polynomieller Zeit gelöst werden kann. Nach aktueller Erfahrung scheint dies höchst unwahrscheinlich zu sein.

### 15.3.2 Die Klasse BQP

Ein universeller Quantencomputer, wie er in Abschnitt 15.2.6 beschrieben worden ist, verfügt gegenüber einer Turing-Maschine offensichtlich über zusätzliche Fähigkeiten. Selbst wenn er die Quantenschaltungen nicht nutzt, ist er mindestens eine Turing-Maschine. Er kann also alle Probleme in der Klasse P in polynomieller Zeit lösen.

#### Zufallszahlen aus einer Quantenschaltung

Eine sehr einfache Form einer Quantenschaltung kann wie in Abbildung 15.5 aufgebaut werden. Das Inputregister ist ein 1-Qubit-Register, welches mit dem Zustand  $|0\rangle$  initialisiert wird. Mit der Hadamard-Operation  $H$  wird daraus der Überlagerungszustand

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

Das Outputregister versucht, den Zustand  $|0\rangle$  zu messen, dies ergibt

$$\langle 0|\psi\rangle = \frac{1}{\sqrt{2}}(\langle 0|0\rangle + \underbrace{\langle 0|1\rangle}_{=0}) = \frac{1}{\sqrt{2}},$$

da die Vektoren  $|0\rangle$  und  $|1\rangle$  orthogonal sind. Der quadrierte Betrag des Skalarproduktes  $\langle 0|\psi\rangle$  ist aber die Wahrscheinlichkeit, dass tatsächlich  $|0\rangle$  gemessen wird. Die beiden Ereignisse “ $|0\rangle$  wird gemessen” und “ $|0\rangle$  wird nicht gemessen” treten also mit gleicher Wahrscheinlichkeit

$$|\langle 0|\psi\rangle|^2 = \left| \frac{1}{\sqrt{2}} \right|^2 = \frac{1}{2}$$

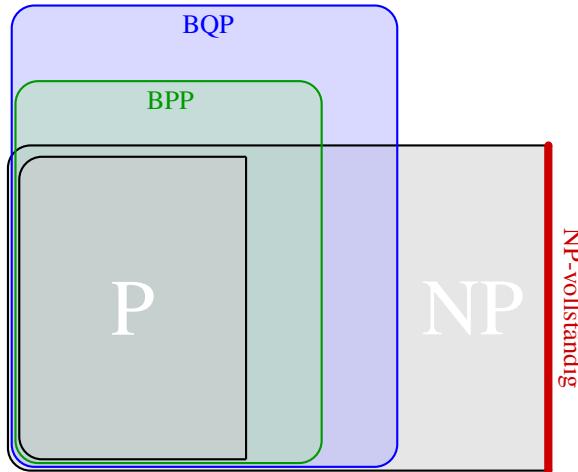


Abbildung 15.6: Die Komplexitätsklassen BPP und BQP im Vergleich zu P und NP. Die Klasse BPP enthält die polynomiellen Probleme und zusätzliche Probleme in NP. Es ist aber nicht bekannt, wie BPP und NP genau zueinander stehen, die Darstellung ist eine Vermutung. Da ein Quantencomputer mindestens Zufallszahlen erzeugen kann, kann er alle Probleme in BPP in polynomieller Zeit lösen, es ist also  $BPP \subset BQP$ . Es ist unwahrscheinlich, dass  $NP \subset BQP$  ist, aber ebenfalls nicht bekannt.

ein. Die Quantenschaltung von Abbildung 15.5 gibt einem Quantencomputer die Fähigkeit, Zufallszahlen zu erzeugen.

### Quantencomputer und BPP

Ein Quantencomputer ist also eine Turing-Maschine, die dank der Quantenschaltung von Abbildung 15.5 mindestens noch über die Fähigkeit verfügt, Zufallszahlen zu erzeugen. Wenn er keine weiteren Quantenfähigkeiten nutzt, kann er nur mit diesen Zufallszahlen immerhin jeden beliebigen probabilistischen Algorithmus in polynomieller Zeit ausführen. Die Klasse der Probleme, die ein Quantencomputer in polynomieller Zeit lösen kann, enthält also mindestens die Klasse BPP.

### Die Klasse BQP

Die Messung eines Quantenzustandes ist nicht ohne zufällige Messfehler möglich. Selbst wenn die Quantenberechnung fehlerfrei wäre, würde der Messprozess zufällige Messfehler einführen. Diese Fehler können aber durch wiederholte Messung erkannt und die Wahrscheinlichkeit ihres Auftretens quantifiziert werden. Wir erwarten daher, dass ein Quantencomputer ähnlich wie eine probabilistische Turing-Maschine falsche Resultate mit einer beschränkten Wahrscheinlichkeit liefern wird.

Ein Quantencomputer muss als erstes die Beschreibung der zu verwendenden Quantengatter produzieren, die für die Quantenberechnung verwendet werden sollen. Dass es eine solche Beschreibung gibt, ist in erster Linie dem Satz 15.1 zu verdanken. Sie zu finden muss aber effizient möglich sein. Die Klasse der von einem Quantencomputer effizient

lösbarer Probleme wird also durch diesen Schritt ein erstes Mal eingeschränkt.

Die Realisierung der Quantenschaltung, die Initialisierung des Inputregisters und die Messung des Outputregisters wird in der Abschätzung der Laufzeit nicht berücksichtigt. Diese Schritte werden daher als instantan im Vergleich zu den von der deterministischen Turing-Maschine ausgeführten Berechnungsstufen angenommen. Als physikalischen Grund für diese Wahl kann man anführen, dass Quantensysteme sehr leicht durch Umgebungseinflüsse gestört werden können und daher nur für sehr kurze Zeit als isolierte Quantensysteme behandelt werden können. Dies ist auch als das Problem der Dekohärenz bekannt. Aus diesen Elementen entsteht eine neue Komplexitätsklasse für Probleme, die sich von einem Quantencomputer lösen lassen.

**Definition 15.7** (Komplexitätsklasse BQP). *Die Klasse BQP umfasst die Funktionen, die von einem Quantencomputer mit beschränkter Fehlerwahrscheinlichkeit in polynomieller Zeit berechnet werden können.*

Abbildung 15.6 zeigt, wie die beiden Komplexitätsklassen BPP und BQP zu den bereits aus Kapitel 12 bekannten Klassen vermutlich stehen. Es ist nicht bekannt und eher unwahrscheinlich, dass BQP ein NP-vollständiges Problem enthält. Eine polynomielle Lösung eines NP-vollständigen Problems würde eine universelle, polynomielle Lösung aller NP-Probleme ermöglichen. Man kann also nicht erwarten, dass Quantencomputer, sollten sie erfolgreich gebaut werden können, die gesamte Klasse NP in polynomieller Zeit berechenbar machen werden. Aber sie können einige spezielle Probleme wie das Faktorisierungsproblem, die für klassische Computer zu aufwendig sind, in polynomieller Zeit lösen.



## Anhang A

# Grundlagen und Bezeichnungen

In diesem Anhang werden einige Grundlagen zusammengefasst, die meistens in anderen Fächern des Grundstudiums wie zum Beispiel einem Kurs über diskrete Mathematik behandelt werden. Darüber hinaus soll er als Referenz für die Notation dienen, die in diesem Buch verwendet wird.

### A.1 Logik

Die mathematische Logik ist ein weitschweifiges und tiefgründiges Gebiet, welches viel zur sorgfältigen Grundlegung der Mathematik beigetragen hat. Einen kleinen Einblick in die von der mathematischen Logik studierten Aspekte erhält der Leser in Kapitel 9, wo auch ein paar für das Thema dieses Buches wichtige Entwicklungen des späten 19. und des frühen 20. Jahrhunderts zur Sprache kommen. Außer an diesen speziellen Stellen liefert die Logik diesem Buch vor allem eine unmissverständliche Notation für logische Aussagen und damit die Basis für weitere Konstruktionen zum Beispiel der in Abschnitt A.2 angesprochenen Mengenlehre.

#### A.1.1 Prädikate

Ein *Prädikat* ist eine Funktion, die die Werte **wahr** oder **falsch** annehmen kann. Man spricht auch vom Wahrheitswert des Prädikats. Ein Prädikat kann von Variablen  $x_1, x_2, \dots, x_n$  abhängen, die irgendwelche mathematische Objekte sein können. Ein solches Prädikat wird  $P(x_1, x_2, \dots, x_n)$  geschrieben und heißt ein  $n$ -stelliges Prädikat.

Besonders einfache Prädikate sind die logischen Variablen, also Variablen, die nur die Werte **wahr** oder **falsch** annehmen können. Logische Variablen werden auch *Literale* genannt.

Über den Wahrheitswert eines einstelliges Prädikat  $P(x)$  kann man erst dann eine Aussage machen, wenn der Wert der Variable  $x$  bekannt ist. Wenn eine Aussage für alle möglichen Werte der Variablen wahr sein soll, dann kann man dies mit dem sogenannten Allquantor ausdrücken. Die Schreibweise

$$\forall x \in A(P(x)), \quad \text{gelesen als: Für alle } x \in A \text{ gilt } P(x),$$

bedeutet, dass das Prädikat  $P(x)$  für alle Werte  $x \in A$  wahr sein muss. Wenn das Prädikat für mindestens einen Wert  $x \in A$  gelten soll, schreibt man dies

$\exists x \in A(P(x))$ , gelesen als: Es gibt ein  $x \in A$ , für das gilt  $P(x)$ ,

genannt der Existenzquantor.

Ist  $P(x_1, \dots, x_n)$  ein  $n$ -stelliges Prädikat, dann sind die Prädikate

$$\begin{aligned} Q(x_2, \dots, x_n) &= \forall x_1 \in A(P(x_1, x_2, \dots, x_n)) \\ \text{und} \quad R(x_2, \dots, x_n) &= \exists x_1 \in A(P(x_1, x_2, \dots, x_n)) \end{aligned}$$

nur noch  $n - 1$ -stellig.

In etwas informellerer mathematischer Sprechweise werden die Quantoren manchmal auch nachgestellt und eher wie eine Abkürzung als "für alle" bzw. "es gibt" gelesen. Ein Beispiel ist die Formel (3.6).

## A.1.2 Logische Verknüpfungen

Mithilfe logischer Verknüpfungen können aus Prädikaten neue Prädikate zusammengebaut werden:

**UND-Verknüpfung:**  $P \wedge Q$  ist genau dann wahr, wenn  $P$  und  $Q$  wahr sind.

**ODER-Verknüpfung:**  $P \vee Q$  ist genau dann wahr, wenn  $P$  oder  $Q$  wahr ist.

**Negation:**  $\neg P$  ist genau dann wahr, wenn  $P$  falsch ist.

**XOR-Verknüpfung:**  $P \veebar Q$  ist genau dann wahr, wenn genau eines der Prädikate  $P$  und  $Q$  wahr ist.

**Implikation:**  $P \Rightarrow Q = \neg P \vee Q$ , wie man zum Beispiel mit einer Wahrheitstabelle nachprüfen kann.

Für  $n$  Prädikate  $P_1, P_2, \dots, P_n$  kann man die Verknüpfung auch nach der Art des Summenzeichens als

$$\begin{aligned} P_1 \wedge P_2 \wedge \cdots \wedge P_n &= \bigwedge_{i=1}^n P_i \\ P_1 \vee P_2 \vee \cdots \vee P_n &= \bigvee_{i=1}^n P_i \end{aligned}$$

schreiben.

Ist  $x_i$  eine logische Variable, dann wird die Negation von  $x_i$  auch mit  $\neg x_i = \bar{x}_i$  geschrieben und  $\bar{x}_i$  ebenfalls wie eine Variable als Literale behandelt. Die Ausdrücke, die durch logische Verknüpfung aus logischen Variablen entstehen werden auch *logische Formeln* genannt.

### A.1.3 Rechenregeln

Für die UND- und ODER-Verknüpfungen gelten die Distributivgesetze

$$\begin{aligned} P \wedge (Q \vee R) &= (P \wedge Q) \vee (P \wedge R) \\ P \vee (Q \wedge R) &= (P \vee Q) \wedge (P \vee R). \end{aligned} \quad (\text{A.1})$$

Die de-morganschen Gesetze regeln, wie sich die UND- und ODER-Verknüpfung mit der Negation verträgt. Sie besagen:

$$\begin{aligned} \neg(P \wedge Q) &= \neg P \vee \neg Q \\ \neg(P \vee Q) &= \neg P \wedge \neg Q \end{aligned} \quad (\text{A.2})$$

Mit den Distributivgesetzen (A.1) und den de-morganschen Regeln (A.2) lässt sich jede logische Formel ausschließlich mit der Negation und der UND- bzw. der ODER-Verknüpfung schreiben.

### A.1.4 Normalformen

Die Distributivgesetze (A.1) ermöglichen, die Verknüpfung in der Klammer nach außen zu bringen. Durch wiederholte Anwendung kann man eine sogenannte Normalform erreichen. Man sagt, eine logische Formel ist in *konjunktiver Normalform*, wenn sei eine UND-Verknüpfung von Ausdrücken ist, die ODER-Verknüpfungen sind:

$$\varphi = (a_1 \vee \cdots \vee a_{k_1}) \wedge (a_{k_1+1} \vee \cdots \vee a_{k_2}) \wedge \cdots \wedge (a_{k_{n-1}+1} \vee \cdots \vee a_{k_n}).$$

Die  $a_i$  sind Literale. Die einzelnen Klammern heißen auch die *Klauseln*. Die konjunktive Normalform findet besondere Anwendung im Problem 3SAT, welches in der Komplexitätstheorie in Kapitel 13 eine bedeutende Rolle spielt.

Die disjunktive Normalform ist eine ODER-Verknüpfung von Ausdrücken, die UND-Verknüpfungen sind:

$$\varphi = (a_1 \wedge \cdots \wedge a_{k_1}) \vee (a_{k_1+1} \wedge \cdots \wedge a_{k_2}) \vee \cdots \vee (a_{k_{n-1}+1} \wedge \cdots \wedge a_{k_n}).$$

## A.2 Mengen

Für die Zwecke dieses Buches reicht die sogenannte naive Mengenlehre aus. Diese kann zwar zu Widersprüchen führen, wie sie zum Beispiel durch das Pinocchio-Paradoxon in Abschnitt 11.3.1 illustriert werden. Diese Limitierungen werden aber nicht einmal im Kapitel 11 zu Schwierigkeiten führen. Diese Art der Mengenlehre wurde von Georg Cantor eingeführt, der Mengen wie folgt definiert hat:

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

Dies bedeutet, dass Mengen immer dadurch studiert werden, dass man sich die Elemente einer Menge genauer anschaut. Wenn ein Objekt  $x$  Element der Menge  $A$  ist, schreibt man dies  $x \in A$ .  $x \notin A$  bedeutet, dass  $x$  nicht in  $A$  ist.

Wohlbekannt sind die Mengen der natürlichen Zahlen  $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ , die Menge der ganzen Zahlen  $\mathbb{Z}$ , die Menge der Brüche oder rationalen Zahlen  $\mathbb{Q}$  und die Menge der reellen Zahlen  $\mathbb{R}$ .

### A.2.1 Konstruktion von Mengen

Die einfachste Form, eine Menge anzugeben, ist ihre Elemente als Liste

$$\begin{aligned}\Sigma &= \{\emptyset, 1\} \\ \Gamma &= \{\emptyset, 1, \sqcup\} \\ \mathbb{N} &= \{0, 1, 2, \dots\}\end{aligned}$$

aufzuzählen. Dies wird auch die *aufzählende Form* genannt. Gibt man kein Element an, erhält man die *leere Menge*, die auch  $\{\} = \emptyset$  geschrieben wird.

Ist  $G$  eine Menge und  $P(x)$  ein einstelliges Prädikat, welches auf Elemente von  $G$  angewendet werden kann, dann kann mithilfe des Prädikates eine neue Menge

$$A = \{x \in G \mid P(x)\}$$

konstruiert werden, die aus allen Elementen besteht, die das Prädikat erfüllen. Wenn klar ist, aus welcher Menge  $G$  die Elemente  $x$  stammen können, darf die Angabe von  $G$  auch weggelassen werden. Die Menge der natürlichen Zahlen kann zum Beispiel durch das Prädikat  $P(x) = (x \geq 0)$  als  $\mathbb{N} = \{x \in \mathbb{Z} \mid x \geq 0\}$  definiert werden. Zu jeder Teilmenge  $A \subset G$  gibt es das einstellige Prädikat  $P(x)$ , welches genau dann war ist, wenn  $x \in A$  ist. Mengen und Prädikate sind also verschiedene Darstellungen für das gleiche Konzept.

### A.2.2 Mengenoperationen

Da Mengen und einstellige Prädikate austauschbar sind, muss es den logischen Verknüpfungen entsprechende Mengenoperationen geben. Die Operationen werden durch sogenannte Venn-Diagramme wie in Abbildung A.1 visualisiert.

**Schnittmenge:** Die Schnittmenge  $A \cap B$  besteht aus den Elementen, die sowohl in  $A$  als auch in  $B$  liegen:

$$A \cap B = \{x \in A \mid x \in B\} = \{x \in B \mid x \in A\} = \{x \mid x \in A \wedge x \in B\}.$$

Dies ist die UND-Verknüpfung der Prädikate  $x \in A$  und  $x \in B$ .

**Vereinigungsmenge:** Die Vereinigungsmenge besteht aus den Elementen, die in  $A$  oder in  $B$  liegen, also

$$A \cup B = \{x \mid x \in A \vee x \in B\}.$$

Dies ist die ODER-Verknüpfung der Prädikate  $x \in A$  und  $x \in B$ .

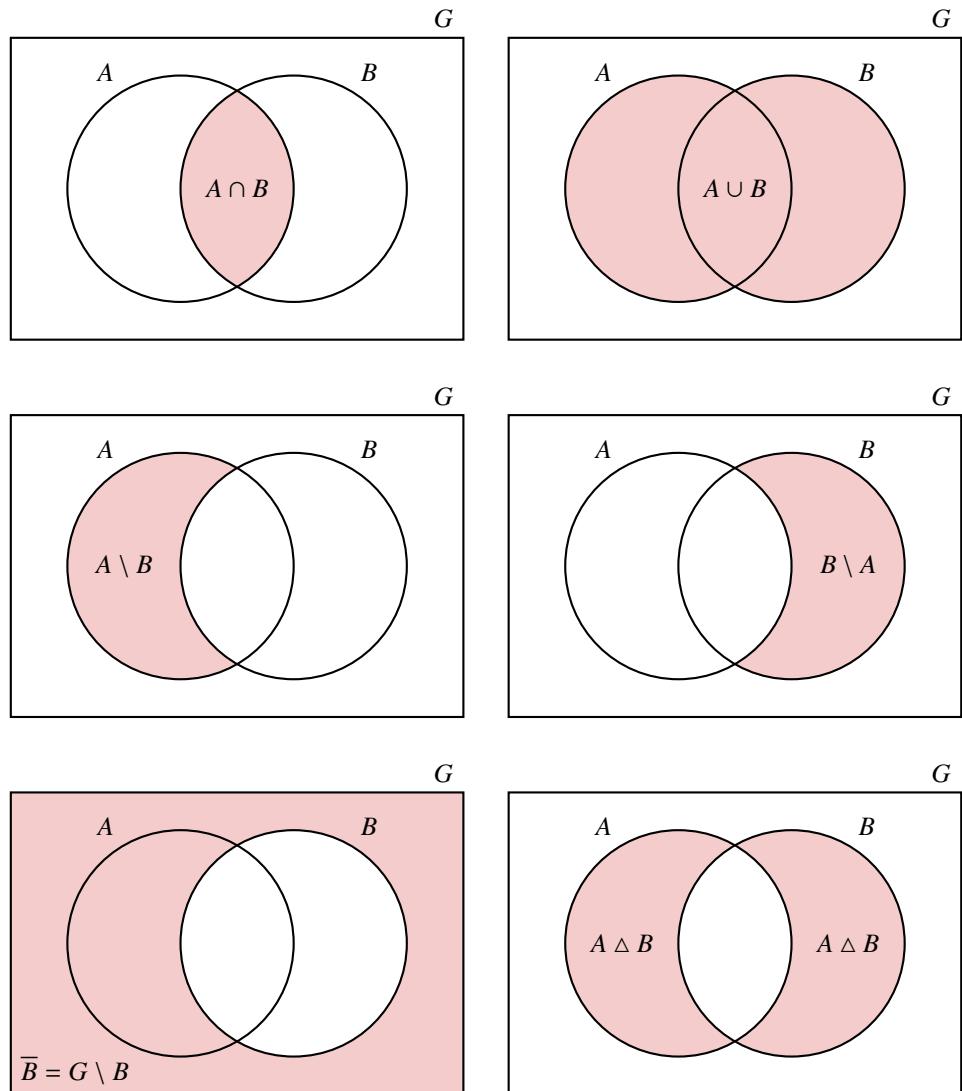


Abbildung A.1: Die grundlegenden Mengenoperationen visualisiert als Venn-Diagramme von links oben: Schnittmenge  $A \cap B$ , Vereinigungsmenge  $A \cup B$ , Differenzmenge  $A \setminus B$  und  $B \setminus A$ , Komplement  $\overline{B}$  der Menge  $B$  und symmetrische Differenz  $A \Delta B$ .

**Differenz:** Die Differenzmenge  $A \setminus B$  besteht aus den Elementen von  $A$ , die nicht in  $B$  liegen:

$$\begin{aligned} A \setminus B &= \{x \in A \mid x \notin B\} = \{x \in A \mid \neg(x \in B)\}, \\ B \setminus A &= \{x \in B \mid x \notin A\} = \{x \in B \mid \neg(x \in A)\}. \end{aligned}$$

**Komplement:** Das Komplement der Menge  $B$  innerhalb der Grundmenge  $G$  ist der Spezialfall

$$\overline{B} = G \setminus B = \{x \in G \mid x \notin B\} = \{x \in G \mid \neg(x \in B)\}$$

der Differenzmenge. Dies ist die Negation des Prädikats  $x \in B$ .

**Symmetrische Differenz:** Die symmetrische Differenz zweier Mengen  $A$  und  $B$  besteht aus den Elementen, die in nur genau einer der Mengen enthalten sind. Sie kann auch

$$A \Delta B = (A \setminus B) \cup (B \setminus A) = \{x \in A \cup B \mid x \in A \vee x \in B\}.$$

geschrieben werden. Dies ist die XOR-Verknüpfung der Prädikate  $x \in A$  und  $x \in B$ . Die symmetrische Differenz zweier Mengen ist genau dann leer, wenn die beiden Mengen gleich sind.

Ist  $A_1, A_2, \dots, A_n$  eine Folge von Mengen, dann kann deren Schnittmenge bzw. Vereinigungsmenge ähnlich dem Summenzeichen als

$$\begin{aligned} A_1 \cap A_2 \cap \cdots \cap A_n &= \bigcap_{i=1}^n A_i \\ A_1 \cup A_2 \cup \cdots \cup A_n &= \bigcup_{i=1}^n A_i \end{aligned}$$

geschrieben werden. Falls die Mengen  $A_i$  durch Prädikate  $P_i(x)$  definiert sind, gilt auch

$$\begin{aligned} \bigcap_{i=1}^n A_i &= \bigcap_{i=1}^n \{x \mid P_i(x)\} = \left\{x \mid \bigwedge_{i=1}^n P_i(x)\right\} \\ \bigcup_{i=1}^n A_i &= \bigcup_{i=1}^n \{x \mid P_i(x)\} = \left\{x \mid \bigvee_{i=1}^n P_i(x)\right\}. \end{aligned}$$

Für eine Familie  $(A_i)_{i \in I}$  von Mengen kann die Schnittmenge bzw. die Vereinigung der Familie auch als

$$\begin{aligned} \bigcap_{i \in I} A_i &= \{x \mid \forall i \in I (x \in A_i)\} \\ \bigcup_{i \in I} A_i &= \{x \mid \exists i \in I (x \in A_i)\} \end{aligned}$$

formuliert werden.

Die Distributivgesetze für die logischen Verknüpfungen werden zu den Distributivgesetzen

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C) \end{aligned}$$

für die Mengenoperationen.

### A.2.3 Kartesisches Produkt

Das kartesische Produkt zweier Mengen  $A$  und  $B$  ist die Menge aller Paare

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\},$$

deren erste Komponente in  $A$  und die zweite in  $B$  liegt. Das kartesische Produkt von  $n$  Faktoren  $A_1, \dots, A_n$  ergibt die Menge

$$A_1 \times A_2 \times \dots \times A_n = \bigtimes_{i=1}^n A_i = \{(a_1, a_2, \dots, a_n) \mid a_1 \in A_1 \wedge a_2 \in A_2 \wedge \dots \wedge a_n \in A_n\}$$

aller  $n$ -Tupel. Das kartesische Produkt von  $n$  identischen Mengen  $A$  wird auch als  $A^n$  geschrieben und besteht aus  $n$ -Tupeln

$$A^n = \{(a_1, \dots, a_n) \mid a_1, \dots, a_n \in A\},$$

deren Komponenten alle in  $A$  sind.

Zur Vervollständigung der Notation setzen wir  $A^0$  als die Menge der 0-Tupel

$$A^0 = \{\emptyset\},$$

das einzige 0-Tupel ist das *leere Tupel*  $\emptyset$ , es hat keine Komponenten.

In Abschnitt 1.1.2 wird die Menge

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$$

aller Wörter als Vereinigung der  $k$ -Tupel von Elementen von  $\Sigma$  für alle natürlichen  $k$  konstruiert.

### A.2.4 Abbildungen

Eine Abbildung

$$f: A \rightarrow B : a \mapsto f(a)$$

ist eine Vorschrift, die jedem Element  $a \in A$  genau ein Element  $f(a) \in B$  zuordnet.

**Definition A.1** (surjektiv, injektiv, bijektiv). *Die Abbildung  $f: A \rightarrow B$  heißt surjektiv, wenn es für jedes  $b \in B$  ein  $a \in A$  gibt derart, dass  $f(a) = b$ . Die Abbildung heißt injektiv, wenn aus  $a_1 \neq a_2$  auch  $f(a_1) \neq f(a_2)$  folgt. Die Abbildung heißt bijektiv, wenn sie sowohl surjektiv wie auch injektiv ist.*

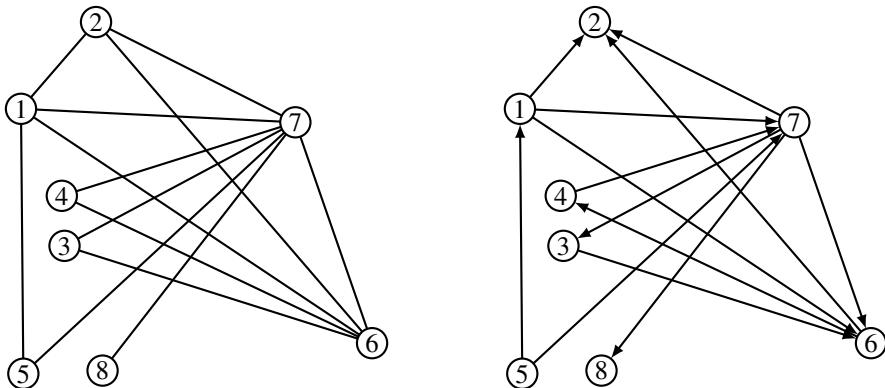


Abbildung A.2: Ungerichteter Graph (links) und gerichteter Graph (rechts).

## A.3 Graphen

Graphen sind kombinatorische Strukturen aus Knoten, die durch Kanten miteinander verbunden sind. Sehr viele kombinatorischen Probleme können in der Form einer Fragestellung über Graphen formuliert werden. Viele der in Kapitel 13 als NP-vollständig erkannten Probleme sind Probleme über Graphen. In den folgenden Abschnitten geht es vor allem um die Definition und einheitliche Notation.

Graphen können auch effizient mithilfe von Matrizen dargestellt werden. Damit lässt sich zum Beispiel die Zahl der Wege einer bestimmten Länge durch den Graphen berechnen. Diese algebraischen Aspekte werden zum Beispiel in [38, Kapitel 14] diskutiert.

### A.3.1 Ungerichtete Graphen

Ein ungerichteter Graph ist in Abbildung A.2 visualisiert. Er besteht aus Knoten, die mit den Zahlen 1 bis 8 nummeriert sind, und Kanten, die als Verbindung eingezeichnet sind. Eine abstrakte Definition dieser Idee ist die Folgende.

**Definition A.2** (Ungerichteter Graph). *Ein (ungerichteter) Graph ist ein Paar  $(V, E)$  bestehend aus einer Menge  $V$  von Knoten oder Vertices und einer Menge von Kanten oder Edges. Eine Kante ist eine zweielementige Menge  $\{a, b\} \subset V$  von Knoten, die auch die beiden Enden der Kante genannt werden.*

Da eine Kante  $e = \{a, b\} \in E$  eine Menge ist, müssen  $a$  und  $b$  verschieden sein:  $a \neq b$ . Es ist nicht zulässig, einen Knoten mit sich selbst zu verbinden. Eine Kante von  $a$  zu  $a$  müsste als  $\{a, a\} = \{a\}$  geschrieben werden, dies ist aber keine zweielementige Menge.

**Definition A.3** (Grad eines Knotens). *Sei  $G = (V, E)$  ein Graph und  $v \in V$  ein Knoten. Der Grad des Knotens  $v$  ist die Anzahl der Knoten, mit denen  $v$  verbunden ist:*

$$\deg(v) = |\{a \in V \mid \{a, v\} \in E\}|.$$

Da jede Kante genau zwei Endpunkte hat, zählt die Summe

$$\sum_{v \in V} \deg(v) = 2|E|$$

der Grade aller Knoten jede Kante zweimal.

Die Teilnehmer des sozialen Netzwerks Facebook können als Knoten eines Graphen betrachten werden. Zwei Knoten werden durch eine Kante verbunden, wenn sie auf Facebook befreundet sind. Da die Freundschaftsrelation symmetrisch ist, sind alle Kanten ungerichtet.

### A.3.2 Gerichtete Graphen

Das soziale Netzwerk Twitter oder X verwendet eine asymmetrische Relation zwischen Teilnehmern. Ein Teilnehmer  $a$  kann ein *Follower* eines Teilnehmers  $b$  sein, auch ohne dass  $b$  ein Follower von  $a$  ist. Die Follower-Relation ist also asymmetrisch und kann daher nicht mit einem ungerichteten Graphen modelliert werden. Dies führt auf die folgende Definition eines gerichteten Graphen (Abbildung A.2).

**Definition A.4** (Gerichteter Graph). *Ein gerichteter Graph ist ein Paar  $(V, E)$  bestehend aus einer Menge  $V$  von Knoten oder Vertices und einer Menge  $E \subset V \times V$  von Kanten. Eine Kante ist ein Paar  $e = (a, b) \in V \times V$ ,  $a$  heißt der Anfangspunkt und  $b$  heißt der Endpunkt der Kante.*

Im Gegensatz zu der Situation bei einem ungerichteten Graph ist es bei einem gerichteten Graph zulässig, eine Kante der Form  $(a, a)$  zu konstruieren, die als Pfeil von  $a$  zurück auf  $a$  dargestellt wird.

### A.3.3 Beschriftete Graphen

Die Zustandsdiagramme für endliche Automaten (Kapitel 1), Stackautomaten (Kapitel 7) und Turing-Maschinen (Kapitel 10) sind gerichtete Graphen, aber die Übergänge sind zusätzlich mit einer Beschriftung versehen. Dabei ist die Beschriftung nicht nur etwas, was man nachträglich einer Kante hinzufügen kann. Es kann zwei unterschiedlich beschriftete Kanten zwischen  $a$  und  $b$  geben. Das kann zum Beispiel wie folgt definiert werden.

**Definition A.5** (BESCHRIFTETER GRAPH). *Ein beschrifteter Graph ist ein Paar  $(V, E)$  bestehend aus einer Menge  $V$  von Knoten und einer Menge  $E$  von Kanten. Eine Kante ist ein Paar  $(\{a, b\}, l)$  aus einer zweielementigen Menge von Knoten und einer Beschriftung  $l \in L$  in einer Menge  $L$  von Beschriftungen, auch Labels genannt.*

Ein mit ganzen Zahlen beschrifteter Graph ist zum Beispiel Gegenstand des NP-vollständigen Problems *STEINER-TREE*, welches in Abschnitt 13.4.9 diskutiert wird.

**Definition A.6** (BESCHRIFTETER, GERICHTETER GRAPH). *Ein beschrifteter, gerichteter Graph ist ein Paar  $(V, E)$  bestehend aus einer Menge  $V$  von Knoten und einer Menge  $E$  von beschrifteten Kanten. Eine Kante ist ein Tripel  $(a, b, l) \in V \times V \times L$ , wobei  $L$  eine Menge von Beschriftungen ist.*

Das Zustandsdiagramm eines deterministischen endlichen Automaten wie in Abschnitt 1.2.2 definiert, ist ein gerichteter, beschrifteter Graph mit  $V = Q$  und Beschriftungen aus der Menge  $L = \Sigma$  der Zeichen. Zu jedem Zustand  $q \in Q$  und jedem Zeichen  $a \in \Sigma$  gibt es eine gerichtete Kante  $(q, \delta(q, a), a) \in Q \times Q \times \Sigma$ , die mit  $a$  beschriftet ist.

## A.4 Wahrscheinlichkeitsrechnung

In Abschnitt 3.4 wird gezeigt, wie Zufallsprozesse verwendet werden können, um Nicht-determinismus zu ersetzen. Das Resultat sind Algorithmen, die Resultate nur mit einer gewissen Wahrscheinlichkeit finden. Die für deren Berechnung nötigen Regeln werden im Folgenden zusammengestellt.

### A.4.1 Ereignisse

Die Wahrscheinlichkeitsrechnung behandelt den Ausgang von Experimenten, die vom Zufall beeinflusst sind. Die Menge  $\Omega$  ist die Menge aller möglichen *Versuchsausgänge*, die Elemente von  $\Omega$  heißen auch die *Elementareignisse*. Die Menge  $\Omega$  kann auch unendlich sein.

Meistens interessieren nicht einzelne Versuchsausgänge, sondern nur Zusammenfassungen von Versuchsausgängen, die eine Gemeinsamkeit haben. Solche Teilmengen  $A \subset \Omega$  heißen *Ereignisse*. Nicht jede Teilmenge muss ein Ereignis sein. Wir bezeichnen die Menge der Ereignisse mit  $\mathcal{E}$ .

Die Ereignisse müssen die folgenden Axiome erfüllen:

1. Die Vereinigung zweier Ereignisse ist ein Ereignis:

$$A, B \in \mathcal{E} \quad \Rightarrow \quad A \cup B \in \mathcal{E}.$$

2. Die Differenzmenge von Ereignissen ist ein Ereignis:

$$A, B \in \mathcal{E} \quad \Rightarrow \quad A \setminus B \in \mathcal{E}.$$

3. Die Menge  $\Omega$  ist ein Ereignis,  $\Omega \in \mathcal{E}$ .  $\Omega$  heißt auch das *sichere Ereignis*.

4. Die Vereinigung einer abzählbaren Familie  $A_i, i \in \mathbb{N}$ , von Ereignissen ist wieder ein Ereignis:

$$A_0 \cup A_1 \cup \dots = \bigcup_{i=0}^{\infty} A_i \in \mathcal{E}.$$

Aus den Axiomen folgen sofort weitere Ereignisse. Die leere Menge  $\emptyset$  ist die Differenz  $\emptyset = \Omega \setminus \Omega$ , also ein Ereignis. Es wird das *unmöglichere Ereignis* genannt. Mit jedem Ereignis  $A \in \mathcal{E}$  ist auch das Komplement  $\bar{A} = \Omega \setminus A$  ein Ereignis.

Ereignisse werden oft auch durch Bedingungen an die Versuchsresultate beschrieben. Eine Funktion  $\Omega \rightarrow \mathbb{R}$  heißt eine *Zufallsvariable*. Zum Beispiel ist die Menge  $A = \{\omega \mid X(\omega) > 0\}$  der Versuchsausgänge  $\omega$ , für die die Zufallsvariable  $X(\omega)$  einen positiven Wert hat, ein Ereignis.

## A.4.2 Wahrscheinlichkeit

Die *Wahrscheinlichkeit* ist eine Funktion, die einem Ereignis einen Zahlenwert zuordnet, der die Zuversicht misst, mit der das Eintreten des Ereignisses erwartet wird. Wie schreiben die Wahrscheinlichkeit als Funktion  $P: \mathcal{E} \rightarrow \mathbb{R}$ . Sie erfüllt die folgenden Axiome:

1.  $0 \leq P(A) \leq 1$  für alle  $A \in \mathcal{E}$ .
2. Für das sichere Ereignis gilt:  $P(\Omega) = 1$ .
3. Für eine abzählbare Familie paarweise disjunkter Ereignisse  $A_i, i \in \mathbb{N}$ , ist

$$P\left(\bigcup_{i=0}^n A_i\right) = \sum_{i=0}^n P(A_i).$$

Aus den Axiomen folgen sofort weitere Resultate. Für das unmögliche Ereignis ist

$$\Omega = \emptyset \cup \Omega \Rightarrow P(\Omega) = P(\emptyset \cup \Omega) = P(\emptyset) + P(\Omega) \Rightarrow P(\emptyset) = 0.$$

Für ein Ereignis  $A \in \mathcal{E}$  folgt

$$\Omega = A \cup \bar{A} \Rightarrow P(\Omega) = P(A \cup \bar{A}) = P(A) + P(\bar{A}) \Rightarrow 1 = P(A) + P(\bar{A})$$

oder  $P(\bar{A}) = 1 - P(A)$ . Für zwei Ereignisse  $A, B \in \mathcal{E}$  folgt wegen  $A = (A \setminus B) \cup (A \cap B)$  und  $B = (B \setminus A) \cup (A \cap B)$

$$\left. \begin{array}{l} P(A) = P(A \setminus B) + P(A \cap B) \\ P(B) = P(B \setminus A) + P(A \cap B) \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} P(A \cup B) = P(A \setminus B) + P(B \setminus A) + P(A \cap B) \\ \quad \quad \quad = P(A) + P(B) - P(A \cap B). \end{array} \right.$$

## A.4.3 Bedingte Wahrscheinlichkeit und Unabhängigkeit

Die bedingte Wahrscheinlichkeit entsteht, wenn man statt allen möglichen Versuchsausgänge  $\Omega$  nur eine Teilmenge  $B \in \mathcal{E}$  berücksichtigt. In einem Experiment ignoriert man so alle Durchführungen, die nicht mit Eintreten des Ereignisses  $B$  enden. Die Wahrscheinlichkeit eines Ereignisses unter dieser Einschränkung ist die *bedingte Wahrscheinlichkeit*

$$P(A|B) = \frac{P(A \cap B)}{P(B)}.$$

Aus der Definition folgt sofort der *Satz von Bayes*:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A \cap B)}{P(A)} \cdot \frac{P(A)}{P(B)} = P(B|A) \cdot \frac{P(A)}{P(B)}.$$

Die Rechenregeln von Abschnitt A.4.2 können  $P(A \cap B)$  nicht berechnen. Die bedingte Wahrscheinlichkeit

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \Rightarrow P(A \cap B) = P(A|B)P(B)$$

zeigt den Grund dafür. Wenn  $P(A)$  und  $P(A|B)$  verschieden sind, dann kommt es offenbar darauf an, ob man nur Experimente mit Versuchsausgang in  $B$  anschaut. Die Wahrscheinlichkeit des Ereignisses  $A$  ist abhängig davon, ob das Ereignis  $B$  eingetreten ist.

Wenn die bedingte Wahrscheinlichkeit  $P(A|B)$  nicht von  $B$  abhängt, dann ist  $P(A|B) = P(A)$  und damit folgt

$$P(A \cap B) = P(A|B)P(B) = P(A)P(B). \quad (\text{A.3})$$

Man sagt, die Ereignisse  $A$  und  $B$  sind *unabhängig*, wenn die Regel (A.3) gilt.

#### A.4.4 Bernoulli-Experimente und Binomialverteilung

Ein *Bernoulli-Experiment* ist ein Experiment mit nur zwei Ausgängen  $A$  und  $\bar{A}$ . Für ein solches Experiment ist  $\Omega$  eine zweielementige Menge. Die beiden möglichen Ausgänge haben die Wahrscheinlichkeit  $P(A) = p$  und  $P(\bar{A}) = 1 - p$ . Ein Beispiel ist der Wurf einer Münze, oder in Abschnitt 3.4 ein probabilistischer endlicher Automat, der ein Wort akzeptiert oder nicht.

Führt man  $n$  unabhängig Experiment durch, dann sei  $A_i$  das Ereignis, dass  $A$  im Versuch mit der Nummer  $i$  eingetreten ist. Die Wahrscheinlichkeit, dass das Ereignis  $A$  genau  $k$  mal eingetreten ist, ist

$$P(A \text{ tritt genau } k \text{ mal ein}) = \binom{n}{k} p^k (1-p)^{n-k}.$$

Sie heißt die Binomialverteilung.

#### A.5 $O$ -Notation

Beim Vergleich der Abhängigkeit der Laufzeit  $t(n)$  von der Größe  $n$  der Problemstellung interessieren die genauen Werte für jedes  $n$  nicht, es geht nur um das asymptotische Verhalten für  $n \rightarrow \infty$ . Seien also zwei Funktionen  $f(n)$  und  $g(n)$  gegeben. Man möchte sagen können, dass  $f(n)$  mindestens so schnell wächst wie  $g(n)$ .

Die Forderung

$$f(n) \geq g(n) \quad \forall n \in \mathbb{N} \quad (\text{A.4})$$

wäre dafür viel zu strikt, wie das Beispiel  $f(n) = n^2$  und  $g(n) = c + 2n$  mit  $c > 0$  zeigt. Die Funktion  $f$  wächst quadratisch,  $g$  wächst nur linear,  $f$  wächst also für große  $n$  tatsächlich schneller als  $g$ . Trotzdem ist die Ungleichung (A.4) für  $n < 2 + \sqrt{1+c}$  verletzt. Für die Beurteilung des Wachstums für  $n \rightarrow \infty$  können einzelne Werte  $n$ , für die (A.4) verletzt ist, ignoriert werden.

Die beiden Funktionen

$$f(n) = n + \frac{1}{n} \quad \text{und} \quad g(n) = n + 1 \quad (\text{A.5})$$

haben als Unterschied

$$f(n) - g(n) = \frac{1}{n} - 1,$$

dessen Betrag für  $n \rightarrow \infty$  nie größer als 1 wird, man sollte also sagen dürfen, dass die beiden Funktionen  $f$  und  $g$  gleich schnell wachsen. Sie verletzen aber die Ungleichung (A.4) für all  $n$ .

Das Problem kann gelöst werden, indem man den Quotienten betrachtet. Tatsächlich gilt

$$\frac{f(n)}{g(n)} = \frac{n + \frac{1}{n}}{n + 1} = \frac{1 + \frac{1}{n^2}}{1 + \frac{1}{n}} < \frac{1 + \frac{2}{n} + \frac{1}{n^2}}{1 + \frac{1}{n}} = \frac{(1 + \frac{1}{n})^2}{1 + \frac{1}{n}} = 1 + \frac{1}{n}.$$

Für große  $n$  ist der Quotient nie größer als 2. Man könnte also (A.4) durch die Forderung ersetzen, dass

$$\lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| \leq M \quad (\text{A.6})$$

ist.

Die Forderung (A.6) funktioniert jedoch für die beiden Funktionen

$$f(n) = n + n(-1)^n \quad \text{und} \quad g(n) = 2n + n(-1)^n \quad (\text{A.7})$$

nicht, denn deren Quotient ist

$$\left| \frac{f(n)}{g(n)} \right| = \frac{n + n(-1)^n}{2n + n(-1)^n} = \frac{1 + (-1)^n}{2 + (-1)^n} = \begin{cases} \frac{2}{3} & \text{für } n \text{ gerade} \\ \frac{0}{1} & \text{für } n \text{ ungerade.} \end{cases}$$

Daraus kann man ablesen, dass der Quotient nicht konvergiert. Für die geraden  $n$  wachsen beide Funktionen linear, wir möchten sie also als gleich schnell wachsend betrachten können. Damit werden wir auf die folgende Definition geführt.

**Definition A.7** (Wachstum). *Gegeben sind die Funktionen  $f(n)$  und  $g(n)$ . Man sagt  $f$  wächst höchstens so schnell wie  $g$ , wenn es eine Konstante  $M \in \mathbb{R}$  derart gibt, dass*

$$\limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| \leq M. \quad (\text{A.8})$$

**Definition A.8** (*O*-Notation). *Sei  $f(n)$  eine Funktion, die höchstens so schnell wächst wie  $g(n)$ , dann schreibt man dafür  $f(n) = O(g(n))$ .*

Die folgende Eigenschaft folgt direkt aus der Definition. Wenn für zwei divergente ganzzahlige Funktionen  $f(n)$  und  $g(n)$  eine Ungleichung der Form

$$f(n) \leq Mg(n) + c$$

gilt, dann ist  $f(n) \leq O(g(n))$ .

# Literatur

- [1] Aviv Adler u. a. “Tatamibari is NP-complete”. In: *10th International Conference on Fun with Algorithms* (2021).
- [2] Manindra Agrawal, Neeraj Kayal und Nitin Saxena. *PRIMES is in P*. 2002. URL: [https://www.cse.iitk.ac.in/users/manindra/algebra/primality\\_v6.pdf](https://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf).
- [3] Sanjeev Arora und Boaz Barak. *Computational complexity. A modern approach*. Cambridge University Press, 2009. ISBN: 978-0-521-42426-4. URL: <https://www.cambridge.org/9780521424264>.
- [4] *avremu: An AVR emulator written in pure LaTeX*. URL: <https://gitlab.brokenpipe.de/stettberger/avremu> (besucht am 29.07.2024).
- [5] Fabrice Bellard. *JSLinux - Javascript PC Emulator running Linux*. URL: <https:////jslinux.org/> (besucht am 29.07.2024).
- [6] *brk, sbrk - change data segment size. Linux manual page*. Aug. 2024. URL: <https://man7.org/linux/man-pages/man2/brk.2.html>.
- [7] Martin Buchholz. *JEP draft: Predictable regex performance*. 2021. URL: <https://openjdk.org/jeps/8260688>.
- [8] James Clark und Steve DeRose. *XML Path Language (XPath). Version 1.0*. 1999. URL: <https://www.w3.org/TR/1999/REC-xpath-19991116/>.
- [9] *Comparison of parser generators*. Juli 2024. URL: [https://en.wikipedia.org/wiki/Comparison\\_of\\_parser\\_generators](https://en.wikipedia.org/wiki/Comparison_of_parser_generators).
- [10] Mike Davey. *A Turing Maschine. In The Classic Style*. Juli 2021. URL: <https://aturingmachine.com/>.
- [11] *dd - convert and copy a file. Linux manual page*. Juni 2024. URL: <https://linux.die.net/man/1/mt>.
- [12] Apostolos Doxiadis und Christos H. Papadimitriou. *Logicomix. An epic search for truth*. New York: Bloomsbury, 2009. ISBN: 978-1-59691-452-0.
- [13] Philippe Flajolet und Robert Sedgewick. *Analytic combinatorics*. Cambridge University Press, 2009. doi: <https://doi.org/10.1017/CBO9780511801655>.
- [14] *FreeRADIUS. We authenticate the Internet*. URL: <https://www.freeradius.org/> (besucht am 29.07.2024).

- [15] *FreeRADIUS man pages. unlang.* Dez. 2017. URL: <https://www.freeradius.org/radiusd/man/unlang.html>.
- [16] Erich Friedman. *Spiral Galaxies is NP-complete.* 2021. URL: <https://erich-friedman.github.io/papers/spiral.pdf>.
- [17] Michael R. Garey und David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* New York: W. H. Freeman und Company, 1979.
- [18] *GNU Bash.* URL: <https://www.gnu.org/software/bash/> (besucht am 30.08.2024).
- [19] *GNU Octave.* URL: <https://octave.org/> (besucht am 30.08.2024).
- [20] *Group shot with Fleming standing.* URL: <https://www.cfa.harvard.edu/news/wolbach-library-covid-19-and-harvards-acclaimed-women-computers> (besucht am 24.07.2024).
- [21] *Homebrew. The Missing Package Manager for macOS.* Juni 2024. URL: <https://brew.sh>.
- [22] University of Southern California Information Sciences Institute. “Internet Protocol. DARPA Internet Program Protocol Specification”. In: (1981).
- [23] University of Southern California Information Sciences Institute. “Transmission Control Protocol. DARPA Internet Program Protocol Specification”. In: (1981).
- [24] Clay Mathematics Institute. *P vs NP.* Juni 2024. URL: <https://www.claymath.org/millennium/p-vs-np>.
- [25] Richard M. Karp. “Reducibility among combinatorial problems”. In: *Complexity of Computer Computations.* Hrsg. von R. E. Miller und J. W. Thatcher. New York: Plenum Press, 1972. URL: <https://cgi.di.uoa.gr/~sgk/teaching/grad/handouts/karp.pdf>.
- [26] Berhard Korte und Jens Vygen. *Kombinatorische Optimierung. Theorie und Algorithmen.* Springer Berlin, Heidelberg, 2008. doi: <https://doi.org/10.1007/978-3-540-76919-4>.
- [27] William Lewis. *Turing Machine Multiplication.* Sep. 2019. URL: [https://www.youtube.com/watch?v=S\\_TaUt6o4dg](https://www.youtube.com/watch?v=S_TaUt6o4dg).
- [28] *List of NP-complete problems.* Juni 2024. URL: [https://en.wikipedia.org/wiki/List\\_of\\_NP-complete\\_problems](https://en.wikipedia.org/wiki/List_of_NP-complete_problems).
- [29] Yuri V. Matiyasevich. *Hilbert's tenth problem.* Foundations of Computing. MIT Press, 1993. ISBN: 978-0-262-13295-4.
- [30] *Maxima. A Computer Algebra System.* Juni 2024. URL: <https://maxima.sourceforge.io>.
- [31] Ralph Merkle und Martin Hellman. “Hiding information and signatures in trapdoor knapsacks”. In: *IEEE Transactions in Information Theory* 24.5 (1978), S. 525–530.
- [32] David Morgan-Mar. *Ook!* 2022. URL: <https://www.dangermouse.net/esoteric/ook.html> (besucht am 14.08.2024).

- [33] *mt - control magnetic tape drive operation. Linux manual page.* Juni 2024. URL: <https://linux.die.net/man/1/mt>.
- [34] Randall Munroe. *Formal languages.* 2012. URL: <https://xkcd.com/1090/> (besucht am 16. 10. 2024).
- [35] Randall Munroe. *NP-Complete.* 2007. URL: <https://xkcd.com/287/> (besucht am 16. 10. 2024).
- [36] Randall Munroe. *Regular expressions.* 2007. URL: <https://xkcd.com/208/> (besucht am 16. 10. 2024).
- [37] Randall Munroe. *xkcd. A webcomic of romance, sarcasm, math, and language.* 2005. (Besucht am 16. 10. 2024).
- [38] Andreas Müller. *Lineare Algebra: Eine praxisorientierte Einführung. Mathematische Grundlagen, praxisrelevante Methoden und technische Anwendungen.* Springer-Vieweg, 2023. ISBN: 978-3-662-67865-7. doi: <https://doi.org/10.1007/978-3-662-67866-4>.
- [39] Andreas Müller. “Polynomial fill-in puzzles or how to make sense of the Cook-Levin theorem”. In: *Elemente der Mathematik* 77 (2022), S. 53–62.
- [40] Andreas Müller. *Teilbarkeit mit endlichen Automaten.* 2024. URL: <https://autospqr.ch/teilbar>.
- [41] Numb3rs. *Uncertainty Principle.* 2005. URL: <https://www.imdb.com/title/tt0663233/>.
- [42] Adesuwa Okyomon. 2021. URL: <https://www.scienceworld.ca/stories/science-star-wars-can-planet-have-two-suns/> (besucht am 01. 10. 2024).
- [43] *On the Turing Completeness of PowerPoint (SIGBOVIK).* URL: <https://youtu.be/uNjxe8ShM-8> (besucht am 02. 09. 2024).
- [44] Ruhsan Onder und Zeki Bayram. “XSLT Version 2.0 is Turing-complete: A purely transformation based proof”. In: Lecture Notes in Computer Science 4094 (2006), S. 275–276. URL: [https://staff.emu.edu.tr/zekibayram/Documents/papers/XSLT\\_Turing\\_complete\\_Onder\\_Bayram.pdf](https://staff.emu.edu.tr/zekibayram/Documents/papers/XSLT_Turing_complete_Onder_Bayram.pdf).
- [45] Matt Parker. *Stand-up Maths.* URL: <https://www.youtube.com/@standupmaths>.
- [46] *Perl.* URL: <https://www.perl.org/> (besucht am 30. 08. 2024).
- [47] *php.* URL: <https://www.php.net/> (besucht am 30. 08. 2024).
- [48] Tibor Radó. “On non-computable functions”. In: *Bell System Technical Journal* 41.3 (1962), S. 877–884. doi: [10.1002/j.1538-7305.1962.tb00480.x](https://doi.org/10.1002/j.1538-7305.1962.tb00480.x).
- [49] *re2c.* Juni 2024. URL: <https://re2c.org>.
- [50] *readelf - display information about ELF files. Linux manual page.* Aug. 2024. URL: <https://man7.org/linux/man-pages/man1/readelf.1.html>.
- [51] Paul Rendell. *Turing Machine Universality of the Game of Life.* Bd. 18. Emergence, Complexity and Computation. 2016. ISBN: 978-3-319-19841-5. doi: [10.1007/978-3-319-19842-2](https://doi.org/10.1007/978-3-319-19842-2).

- [52] *Schweizer Weltatlas. 33.2 Politische Gliederung*. Juni 2024. URL: <https://www.schweizerweltatlas.ch/listings/0332/>.
- [53] Adi Shamir. “A polynomial-time algorithm for breaking the basic Merkle-Hellman cryptosystem”. In: *Information Theory* 30.5 (1984), S. 699–704. doi: [doi : 10.1109/SFCS.1982.5](https://doi.org/10.1109/SFCS.1982.5).
- [54] Stuart M. Shieber. “Evidence against the context-freeness of natural language”. In: *Linguistics and Philosophy* 8 (1985), S. 333–343.
- [55] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorizations and Discrete Logarithms on a Quantum Computer”. In: *SIAM Journal on Computing* 26.5 (1997).
- [56] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.
- [57] *The Power of 10: Rules for Developing Safety-Critical Code*. URL: [https://en.wikipedia.org/wiki/The\\_Power\\_of\\_10:\\_Rules\\_for\\_Developing\\_Safety-Critical\\_Code](https://en.wikipedia.org/wiki/The_Power_of_10:_Rules_for_Developing_Safety-Critical_Code) (besucht am 29.07.2024).
- [58] Adrian D. Thurston. *Ragel State Machine Compiler*. Juni 2024. URL: <https://www.colm.net/open-source/ragel/>.
- [59] Alan Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 43.1 (1936), S. 230–265.
- [60] *Turing machine simulator (LaTeX)*. URL: [https://literateprograms.org/turing\\_machine\\_simulator\\_latex\\_.html](https://literateprograms.org/turing_machine_simulator_latex_.html) (besucht am 29.07.2024).
- [61] Larry Wall. *Usenet article <1994Jul21.173737.16853@netlabs.com>*. 1994. URL: <https://groups.google.com/g/comp.lang.clos/c/edcyFOBLMeo/m/MgB1ogL7AHcJ> (besucht am 15.09.2024).
- [62] *With Fifth Busy Beaver, Researchers Approach Computation’s Limits*. 2024. URL: <https://www.quantamagazine.org/amateur-mathematicians-find-fifth-busy-beaver-turing-machine-20240702/> (besucht am 15.07.2024).

# Index

- \*-Operation, 74, 124
- + Operation, 76, 82
- 0-1 integer programming, 344
- 3D-MATCHING*, 343, 356
- 3SAT*, 322, 331, 343
- $\emptyset$ , 412
- A Programming Language, 121
- ableitbar, 112
- ableiten, 112
- Ableitung, 112
- Ableitungsdreieck, 148
- abzählbar unendlich, 206
- $A_{CFG}$ , 266
- Ada Lovelace, 219
- Addition, 366
- Addition, binär, 226
- $A_{DEA}$ , 265
- Adi Shamir, 337
- $A_{\epsilon CFG}$ , 288
- $A_{\epsilon DEA}$ , 265
- AGC, 395
- AI, 318
- AI-Schnittstelle, 50
- AKS-Algorithmus, 403
- Aktivieren, 260
- Akzeptanzproblem
  - CFG, 266
  - DEA, 265
  - speziell, 265
  - TM, 268
- akzeptieren
  - NEA, 53
  - nichtdeterministische TM, 299
  - nichtdeterministischer Automat, 63
  - regulärer Ausdruck, 81
- Stackautomat, 158
- Turing-Maschine, 223
- Wort, 9
- akzeptierte Sprache, 9, 63
- NEA, 58
- Akzeptierzustand, 6, 53, 222
- Algebra, 4
- ALGOL 58, 138
- Algorithmus
  - genetisch, 317
- $ALL_{CFG}$ , 285
- $ALL_{DEA}$ , 287
- $ALL_{PDA}$ , 285
- Alphabet, 3, 6
- Alternative, 72, 124
- ALU, 338, 395
- American Standard Code for Information Interchange, 2
- Analogrechner, 389
- analytische Kombinatorik, 13
- APL, 121
- Apollo, 395
- Apollo-Guidance-Computer, 395
- Appel, Kenneth, 294
- Approximation, 316
- Approximationssatz, 399
- äquivalent, polynomiell, 295, 315
- arithmetic logic unit, 338, 395
- Array, 371
- Artificial Intelligence, 318
- ASCII, 2, 232
- Assembler, 138
- Assembler und GOTO, 379
- assoziativ, 72, 74
- $A_{TM}$ , 268
- Atmel, 364

- Atommodell, 390  
 aufzählbar, 245  
 aufzählende Form, 412  
 Aufzähler, 245  
 Ausfüllrätsel, polynomiell, 304, 305  
 Auswertung, 118  
 Automat  
     endlich  
     deterministisch, 6  
     nichtdeterministisch, 53  
     probabilistisch, 68  
     komplementär, 7  
     zellulär, 251  
 Automat, deterministischer endlicher, 6  
 AVR, 364  
 awk, 88  
 B, Programmiersprache, 171  
 Babbage, Charles, 219  
 Backtracking, 55, 300  
 Backus, John, 138  
 Backus-Naur-Form, 139  
 Backus-Normal-Form, 139  
 banana, 377  
 Bandalphabet, 217  
 Bandstation, 233  
 Bash, 171  
 BASIC, 377  
 Basiszustand, 391  
 Bayes, Satz von, 419  
 BCD, 231  
 BCPL, 108, 362  
 Bellard, Fabrice, 363  
 berechenbar, 263  
 berechenbare Funktion, 263  
 Berechnungsgeschichte, 228  
 Bernoulli-Experiment, 68, 420  
 beschränkte Iteration, 77, 83  
 Biber, fleißig, 262  
 Big Bang, 262  
 bijektiv, 206, 415  
 binary coded decimal, 231  
 binary integer programming, 344  
 Binomialverteilung, 420  
 binäre Addition, 226  
 Binärzahl, 15  
 BIP, 343, 344  
 Bison, 171  
 Bohr, Niels, 390  
 bohren, 357  
 BPP, Komplexitätsklasse, 405  
 BQP, Komplexitätsklasse, 408  
 bra-Vektor, 392  
 Brainfuck, 375  
 Breakpoint, 273  
 Buch, 216  
 busy beaver, 262  
 Bytecode, 362  
 C, 108, 363  
 C++, 88, 108, 363  
 C++ Standard Library, 371  
 C++11, 371  
 Calculator, 172  
 Cantor, Georg, 411  
 CFG, 111  
 CFL, 112  
 $CFL_{TM}$ , 278  
 Charles Babbage, 219  
 Chomsky-Normalform, 128, 143  
     Umwandlung in, 129  
 chromatische Zahl, 294  
*CIRCUIT*, 339  
 clang, 372  
 Clay Research Institute, 308  
*CLIQUE*, 324, 343  
 $k$ -Clique, 296  
*CLIQUE-COVER*, 343, 345  
 $k$ -Cliquenproblem, 296  
 CNF, 128  
 CNOT, 397  
 Cocke-Younger-Kasami-Algorithmus, 137, 144  
 Codierstandards, 370  
 Collatz-Folge, 243  
 Collatz-Funktion, 243  
 Compiler, 362  
 compound statement, 138  
 Computeralgebrasystem, 259  
 constraint solver, 318  
 Container, 371  
 context free grammar, 111

- context free language, 112  
Cook, Stephen, 318  
Corbett, Robert, 171  
Cristofani, Daniel B., 376  
CYK-Algorithmus, 137, 144
- David Hilbert, 204, 257  
de-morgansche Gesetze, 411  
Deaktivieren, 260  
Debugger, 273  
Dekohärenz, 408  
Dennis Ritchie, 141  
deterministisch, 6, 7  
deterministischer endlicher Automat, 6  
Differenz, symmetrisch, 264, 414  
Differenzmenge, 18, 414  
digitale Signatur, 308  
diophantisch, 258  
disjunktive Normalform, 411  
diskrete Logarithmus, 315  
Distributivgesetz  
    Logik, 411  
    Mengenlehre, 415  
Division, 203, 367  
Domino, 281  
Donald Knuth, 139  
Drehkreuz, 5  
Dreieckszahlen, 208  
durchnummrieren, 206  
dynamic programming, 137
- $\varepsilon$ , 4  
 $\varepsilon$ -erweitertes Alphabet, 62  
E-Mail, 1  
 $\varepsilon$ -Regel, 127, 129  
 $\varepsilon$ -Übergang, 63  
EBCDIC, 232  
 $E_{DEA}$ , 263  
Eduroam, 372  
Eigenvektor, 393  
eindeutige Grammatik, 116  
Elektronik, 357  
elementare Funktion, 259  
Elementarereignisse, 418  
ELF, 238  
EMI, 1
- Emil Leon Post, 281  
Emoji, 3  
endlich, 206  
Energieoperator, 393  
entscheidbar, 262  
entscheidbare Sprache, 262  
Entscheider, 246, 258, 261, 262  
Entscheider, nichtdeterministisch, 299  
 $EQ_{CFG}$ , 266, 286, 287  
 $EQ_{DEA}$ , 264  
Ereignis, 418  
    sicher, 418  
    unmöglich, 418  
Erfüllbarkeit, 325  
Erfüllbarkeitsproblem, 318  
erfüllungäquivalent, 323  
erkannte Sprache, 223, 241  
erkennen, 223, 241  
erzeugte Sprache, 112  
erzeugtes Wort, 148  
 $E_{TM}$ , 275  
ETSI, 1  
ETX, 1  
European Telecommunication Standards Institute, 1  
*EXACT-CLIQUE-COVER*, 345, 346  
*EXACT-COVER*, 343, 350  
Excel, 304  
expr, 88  
extensible stylesheet language for transformations, 363
- Faase, Frans, 376  
Facebook, 296, 417  
Faktorisierung, 315, 390  
*FEEDBACK-ARC-SET*, 343, 352  
*FEEDBACK-NODE-SET*, 343, 352  
Fehlerzustand, 11, 50  
Fermat, Pierre de, 401  
Fermat-Test, 401  
Fernschreiber, 2, 233  
Feynman, Richard, 389  
FFT, 306  
Filenamen, 77  
Filter, LDAP, 152  
fleißige Biber, 262

- Formel, logische, 410  
FORTRAN, 138, 377  
Frank Wood, 231  
FreeRADIUS, 372  
Fünfzehnerrest, 52  
Färbeproblem, 293  
führende Null, 10  
  
Game of Life, 251, 341  
ganze gaußsche Zahl, 209  
ganze Zahl, 207  
Ganzzahl, 10  
Ganzzahlen, 50  
Gatter, 338, 395  
Gauß-Algorithmus, 306  
gcc, 5  
genetischer Algorithmus, 317  
Gerard J. Holzmann, 370  
gerichteter Graph, 352  
gerichteter Zyklus, 352  
Giuseppe Peano, 205  
Gleichheitsproblem  
    CFG, 266  
    DEA, 264  
Gleitkommaarithmetik, 399  
GNU-C, 5  
GNU-Projekt, 171  
GOTO, 377  
GPU, 399  
Grammatik, 111  
Graph, gerichtet, 352  
Graphikkarte, 399  
Grenzfall, 49  
Gruppierung, 82, 108  
  
Hadamard-Gatter, 396  
Hadamard-Operation, 396  
Haken, Wolfgang, 294  
Halteproblem für LOOP, 369  
 $\text{HALT}_{\text{TM}}$ , 272  
 $\text{HALT}_{\text{TM}}$ , 270  
 $\text{HAMCIRCUIT}$ , 343, 352  
hamiltonscher Pfad, 352  
hamiltonscher Zyklus, 352  
 $\text{HAMPATH}$ , 352  
Handakuten, 20  
  
Handelsreisender, 357  
Harvard-Architektur, 237  
Heiratsproblem, 356  
Heisenberg, Werner, 391  
Hello world, 363  
hermitesch, 392  
Heuristik, 317  
Higgs-Boson, 390  
Hilbert, David, 204, 257  
hilbertsche Probleme, 258  
Hiragana, 20  
 $\text{HITTING-SET}$ , 343, 350  
Holzmann, Gerard J., 370  
Hotelmanager, 203  
HTTP-Request, 2  
  
IBM 2401, 233  
IBM 704, 138  
ICM 1900, 257  
IEEE 754, 231  
 $\text{IF-THEN-ELSE}$ , 368  
Imperator, 203  
Implementationseigenschaft, 273  
 $\text{INFINITE}_{\text{DEA}}$ , 288  
Informationslücke, 50  
Initialisierung, 222  
injektiv, 206, 415  
Internationaler Mathematikerkongress 1900, 257  
Internet Protocol, 11  
Internet-Sicherheit, 390  
Interpreter, 362  
IP, 11  
Iteration, beschränkt, 77, 83  
Iterator, 371  
  
Japanisch, 20  
Java, 5, 90, 108, 362  
Java-Bytecode, 362  
JavaScript, 108, 362, 363  
JIT-Compiler, 362  
John Backus, 138  
Johnson, Stephen C., 171  
JPL, 370  
JSON, 2, 260  
Juri Matijasewitsch, 258

- Just-in-Time-Compiler, 362  
*k-CLIQUE*, 324, 343  
*k-CLIQUE-COVER*, 343, 345  
*k-EXACT-CLIQUE-COVER*, 345, 346  
*k-EXACT-COVER*, 343  
*k-FEEDBACK-ARC-SET*, 343, 352  
*k-FEEDBACK-NODE-SET*, 343, 352  
*k-SET-COVERING*, 343, 348  
*k-SET-PACKING*, 343, 347  
*k-VERTEX-COLORING*, 343, 346  
*k-VERTEX-COLORING*, 315, 326  
*k-VERTEX-COVER*, 343, 346  
Kanji, 3  
Kanton, 294  
Karp, Richard, 343  
kartesisches Produkt, 415  
    Mächtigkeit, 208  
ket-Vektor, 390  
Klammerausdruck, 108, 160  
    mit Bison, 177  
Klasse BPP, 405  
Klasse BQP, 408  
Klasse NP, 307  
Klasse P, 306  
Klausel, 322, 324, 411  
Knuth, Donald, 139  
Kombinationszeichen, 20  
kombinatorisches Optimierungsproblem, 357  
Kommunikationsnetz, 357  
Kommunikationsprotokoll, 11  
Komplement, 414  
komplementär, 7  
komplexe Ebene, 209  
Konflikt  
    reduce/reduce, 170  
    shift/reduce, 170  
Konjugation, komplex, 392  
konjunktive Normalform, 322, 411  
Konrad Zuse, 138  
Konstantheitsproblem, 259  
kontextfrei, 112  
    Grammatik, 111  
    Sprache, 112  
kontextsensitiv, 112  
Kontrollstruktur, 121  
Kopfbewegung, 236  
Koprozessor, 399  
korrespondierende Felder, 149  
Kreuzung, 339  
kyrillisch, 3  
 $L(A)$ , 9, 58, 63  
 $L(G)$ , 112  
 $L(M)$ , 223, 241, 262  
 $L(q)$ , 21  
 $L(r)$ , 79  
 $L(w)$ , 28  
Larry Wall, 90, 152  
Laufzeit, nichtdeterministische TM, 300  
LDAP, 152  
leere Menge, 412  
leeres Tupel, 415  
leeres Wort, 4  
Leerheitsproblem, 275  
Leiterplatte, 357  
Leonid Levin, 318  
Levin, Leonid, 318  
lex, 88, 171  
LIKE, 77  
linksrekursiv, 117  
Linux, 240  
Lisp, 152  
Liste, 371  
Literal, 322, 324, 409  
LM, 395  
Lochkarte, 232  
Lochstreifen, 232  
LogFile, 49  
logische Formel, 410  
logische Variable, 409  
look-ahead, 171  
LOOP  
    Addition, 366  
    Kontrollstruktur, 366  
    Multiplikation, 366  
Lovelace, Ada, 219  
LTO, 233  
Lucas, Édouard, 401  
Lunar Module, 395  
Lösungszertifikat, 301

- Magnetband, 217, 233  
 Maschineninstruktion, 137  
 Maskierungszeichen, 81  
 Mathematik, numerische, 1  
 Matijassewitsch, Juri, 258  
 Matrizenmechanik, 391  
 Matt Parker, 304  
 $MAX\text{-}CUT$ , 343, 354  
 Maxima, 259  
 Mehrbandmaschine, 236  
 Mehrspurmaschine, 232  
 Memorybus, 234  
 Menge aller DEAs  
     Mächtigkeit, 210  
 Mengenoperation, 18  
 Merkle-Hellman  
     Satz von, 336  
 Messung, 394  
 Metazeichen, 82  
 Mikrokontroller, 364  
 Mikroprozessor, 234, 395  
 Millennium-Probleme, 308  
 Miller-Rabin, Satz von, 403  
 Miller-Rabin-Test, 403  
 Minesweeperkonsistenz, 337  
 Minicomputer, 395  
 Minimalautomat, 24  
 $MIRROR_{DEA}$ , 288  
 Mobiltelefon, 2  
 monkey, 377  
 Monte-Carlo-Simulation, 403  
 Morgan-mar, David, 376  
 Morse-Alphabet, 3  
 MS-DOS, 375  
 Müller, Urban, 375  
 Multiplikation, 227, 366  
 Mustersuche, 79  
 Myhill-Nerode-Automat, 30  
     durch drei teilbare Binärzahlen, 32  
     gerade Länge, 31  
 $n$ -Tupel, 415  
 NASA, 370  
 Naur, Peter, 139  
 NEA, 53  
     verallgemeinert, 91  
 NEA mit  $\varepsilon$ -Übergängen, 63  
 Negation  
     bedingte Anweisung, 368  
 netstat, 11  
 network file system, 260  
 Netzwerktechnik, 357  
 NFS, 260  
 nicht kontextfrei, 5  
 nicht regulär, 5  
 Nichtdeterminismus, 50  
 nichtdeterministisch, 50  
     Turing-Maschine, 297  
 nichtdeterministischer endlicher Automat, 53  
 nichttriviale Eigenschaft, 278  
 Nikoli, 341  
 Nixie-Röhre, 231  
 Norm, 393  
 Normalform  
     disjunktiv, 411  
     konjunktiv, 322, 341, 411  
 Normierungsbedingung, 393  
 NP, Komplexitätsklasse, 307  
 NP-vollständig, 314  
 Numb3rs, 338  
 Numbers (Tabellenkalkulation), 304  
 Numerik, 1  
 numerische Mathematik, 1  
 O-code, 362  
 O-Notation, 421  
 Objective-C, 260  
 Octave, 171  
 Office, 377  
 ONC, 260  
 Ook, 376  
 Open Network Computing, 260  
 OpenOffice Calc, 304  
 Operation  
     regulär, 76  
 Operationen, regulär, 124  
 Operationsverstärker, 389, 395  
 Operator, linear, 392  
 Optimierungsproblem, 357  
     kombinatorisch, 357  
 Option, 76, 82  
 Orakel, 56, 74, 297, 308, 317

- orang-utan, 377  
Orbital, 390  
orthogonal, 393
- P, Komplexitätsklasse, 306  
p-Code, 362  
Paar, 415  
Palindrom, 159, 164, 226  
Palindrome  
    mit Bison, 178  
Paris, 257  
Parker, Matt, 304  
Parse-Tree, 114  
*PARTITION*, 343, 353  
PC, 2  
PDA, 157  
Peano, Giuseppe, 205  
Peano-Axiome, 205  
Perl, 85, 90, 171  
Peter Naur, 139  
Pfad, 193  
Pfad, hamiltonscher, 352  
Phasenverschiebung, 398  
PHP, 141, 171  
PIC, 221  
Pierre de Fermat, 401  
planar, 294  
Plankalkül, 138  
Planung, 358  
Platzhalter, 111  
Plausibilisierung, 10  
politische Karte, 294  
polynomiell äquivalent, 295, 315  
polynomielle Laufzeit, 291  
polynomielle Reduktion, 293  
polynomieller Verifizierer, 302  
polynomielles Ausfüllrätsel, 304, 305  
position independent code, 221  
positiv definit, 392  
Post, Emil Leon, 281  
Post-Korrespondenz-Problem, 282  
Potenzmenge, 58  
    Mächtigkeit, 212  
Power of 10, 370  
PowerPoint, 377  
Primzahl, 245  
Primzahltest, 401  
probabilistisch, 49  
probabilistischer endlicher Automat, 68  
Probedivision, 401  
Problem des Handelsreisenden, 357  
Produkt, kartesisch, 415  
Produktautomat, 18  
Produktübergangsfunktion, 17  
produzierte Sprache, 112  
Programmiersprache, 137, 362  
Prozess, 218  
Prädikat, 409  
Prädikat, *n*-stellig, 409  
Pseudocode, 55  
Pumpeigenschaft  
    kontextfreie Sprache, 195  
    reguläre Sprache, 41  
Pumping-Lemma, 108  
    kontextfreie Sprache, 195  
    reguläre Sprache, 42  
Pumping-Length  
    kontextfreie Sprache, 195  
    reguläre Sprache, 42  
Punktoperation, 117  
Pushdown-Automat, 157  
pythagoräisches Tripel, 258  
Python, 91, 362
- $\mathbb{Q}$ , 209  
Q Kontinuum, 204  
QIC-11, 233  
Quantencomputer, 390, 399  
Quantenmechanik, 389  
Quantenzahlen, 390  
Quartiermeister, 203  
Qubit, 391  
Queue, 371
- Radó, Tibor, 262  
rationale Zahl, 209  
Raumschiff, 203  
RE2, 89  
Rebellenallianz, 203  
Rechenschieber, 389  
rechtsrekursiv, 117  
Record, 217

- reduce/reduce-Konflikt, 170  
 Reduktion, 272  
     polynomiell, 293  
 Regel, 111  
 regulärer Ausdruck, 79  
 $REGULAR_{TM}$ , 277  
 reguläre Operation, 76  
 reguläre Operationen, 124  
 reguläre Sprache, 13  
 rekursiv, 144  
 rekursiv aufzählbar, 246  
 REST, 260  
 RFC 793, 11  
 Richard Karp, 343  
 RISC-LOOP, 384  
 Risch-Algorithmus, 259  
 Ritchie, Dennis, 141  
 Robert Corbett, 171  
 RPC, 260  
 Rucksack, 330  
     stark wachsend, 333  
 Rucksackproblem, 329, 330  
 $\Sigma^*$ , 4  
     Mächtigkeit, 210  
 SAT, 318, 343  
 Satisfiability, 318  
 Satz  
     von Bayes, 419  
     von Cook-Levin, 318  
     von Deutsch-Kitaev, 399  
     von Fermat, kleiner, 401  
     von Lucas, 401  
     von Merkle-Hellman, 336  
     von Miller-Rabin, 403  
     von Myhill-Nerode, 30  
     von Rice, 278  
 Schale, 390  
 Scheduling, 358  
 Schnittmenge, 18, 412  
     einer Familie, 414  
 Schreib-/Lesekopf, 217  
 Schrödinger, Erwin, 391  
 Schrödinger-Gleichung, 395  
 Schweiz, 294  
 sed, 88  
 Segmentanzeige, 231  
 $SEQUENCING$ , 343, 358  
 Serialisierung, 260  
 sesquilinear, 392  
 $SET-COVERING$ , 343, 348  
 $SET-PACKING$ , 343, 347  
 Shamir, Adi, 337  
 shift/reduce-Konflikt, 170  
 Shor, Peter, 390  
 Signatur, digital, 308  
 Skalarprodukt, 392  
 Skilift, 5  
 SMS, 1  
 SMSC, 1  
 SN7445, 231  
 SN7446, 231  
 Socket, 11  
 Sortieren, 306  
 soziale Medien, 3  
 Speicherzyklus, 234  
 Spiral Galaxies, 341  
 Splitter, 339  
 Sprache, 4  
     aufzählbar, 245  
     endlich, 14  
     entscheidbar, 262  
     erkannte, 241  
     kontextfrei, 112  
     regulär, 13  
 Spracheigenschaft, 275  
 Spreadsheet, 250, 304, 377  
 SQL, 77  
 Stackautomat, 157  
 Stackoperation, 156  
 Stammfunktion, 259  
 Stand-up Maths, 304  
 Standardmodell, 390  
 stark wachsender Rucksack, 333  
 Startvariable, 111, 127, 128  
 Startzustand, 6, 53  
`std::for_each`, 371  
`std::list<int>`, 372  
`std::vector<int>`, 372  
 $STEINER-TREE$ , 343, 357  
 Stephen C. Johnson, 171

- Stephen Cook, 318  
Sternoperation, 74  
Steuerung, 220  
Stormtrooper, 203  
Strichoperation, 117  
Strömungssimulation, 1  
*STUNDENPLAN*, 315, 329  
Stundenplan, 293  
STX, 1  
*SUBSET-SUM*, 330, 343  
Suchfunktion, 79  
Sudoku, 300  
Sun Microsystems, 362  
surjektiv, 206, 415  
SWITCH-CASE, 368, 380  
symmetrische Differenz, 18, 264, 414  
Syntaxbaum, 114, 174  
Syntaxdreieck, 190  
system call, 248
- Tabellendarstellung, 8  
Tablet, 2  
Tatamibari, 341  
TCP, 11  
Template, 363  
Terminalsymbol, 111  
Terminalsymbolregel, 128  
Thompson-NEA, 56, 61  
Tibor Radó, 262  
Toffoli-Gatter, 398  
Tom Wildenhain, 377  
Topographie, 294  
Transmission Control Protocol, 11  
Traveling salesman problem, 357  
Tupel, 415  
    leer, 415  
Turing-erkennbar, 241  
Turing-Maschine, 223  
    nichtdeterministisch, 297  
Turing-vollständig, 362  
Twitter, 417
- überabzählbar unendlich, 210  
Übergangsfunktion, 6, 223  
    nichtdeterministisch, 53  
    Produkt-, 17
- Übergangsfunktion für Wörter, 9, 57  
Übergangsfunktion für Zustandsmengen, 57  
Überlagerung, 391  
UCP, 1  
UCSD-Pascal, 362  
*UHAMCIRCUIT*, 343, 352  
*UHAMPATH*, 352  
unentgeltliche Anweisungen, 367  
Unicode, 3  
Unit-Rule, 127  
Unit-Testing, 280  
unitär, 394  
universelle Turing-Maschine, 249  
unlang, 372  
Unterdreieck, 191  
ünäre Darstellung, 210  
ünäre Multiplikation, 227  
*USES-STATE*<sub>TM</sub>, 273
- Variable, 111, 365  
Variable, logische, 409  
Vektor  
    bra-, 392  
    ket-, 390  
Venn-Diagramm, 412  
Verankerungszeichen, 84  
Vereinigung, 18  
Vereinigungsmenge, 18, 412  
    einer Familie, 414  
Vergleich, 26  
Verifizierer, 301  
    polynomiell, 302  
Verkettung, 73, 124  
Verschlüsselung, 390  
Versuchsausgänge, 418  
*VERTEX-COLORING*, 315, 326, 343, 346  
*k*-Vertex-Coloring, 293  
*VERTEX-COVER*, 343, 346  
verwerfen  
    Turing-Maschine, 223  
Vier-Farben-Satz, 294  
VNEA, 91  
von Neumann-Architektur, 238
- $|w|$ , 4  
 $|w|_a$ , 4

Wahrscheinlichkeit, 49, 394, 419

bedingt, 419

Wall, Larry, 90, 152

Wellenfunktion, 391

Wellenmechanik, 391

Wertetabelle, 8

Wetterprognose, 1

Wildenhain, Tom, 377

Wire, 339

Wood, Frank, 231

Wort, 3

Wort, leer, 4

Wortlänge, 4

Wortproblem, 265

X, 417

x86-Emulator, 363

XKCD, vii, 79, 107, 329

XOR, 414

XSLT, 363

Yacc, 171

$\mathbb{Z}$ , 207

zehntes hilbertsches Problem, 258

Zeichenklasse, 83, 85

zellulärer Automat, 251

Zertifikat, 401

Zertifikat, Lösungs-, 301

Zufall, 317

Zufallsvariable, 418

Zuse, Konrad, 138

Zustand, 6

Quantenmechanik, 390

Zustandsdiagramm, 8

Zwischensprache, 362

Zwischenwinkel, 393

Zyklus, gerichtet, 352

Zyklus, hamiltonscher, 352