Thomas Worsch

Grundbegriffe der Informatik

Vorlesung im Wintersemester 2010/2011

Fakultät für Informatik, KIT

INHALTSVERZEICHNIS (KURZ)

Inhaltsverzeichnis (kurz) i

Inhaltsverzeichnis (lang) iii

1 Prolog 1

2 Signale, Nachrichten, Informationen, Daten 5

3 Alphabete, Abbildungen, Aussagenlogik 9

4 Wörter 17

5 Formale Sprachen 29

6 Der Begriff des Algorithmus 35

7 Dokumente 49

8 Kontextfreie Grammatiken 55

9 Speicher 67

10 übersetzungen und Codierungen 73

11 Graphen 89

12 Erste Algorithmen in Graphen 101

13 Quantitative Aspekte von Algorithmen 119

14 Endliche Automaten 135

15 Reguläre Ausdrücke und rechtslineare Grammatiken 147

16 Turingmaschinen 157

17 Relationen 177

Index 193

Literatur 199

Colophon 201

INHALTSVERZEICHNIS (LANG)

Inhaltsverzeichnis (kurz) i

Inhaltsverzeichnis (lang) iii

1 Prolog 1
- 1.1 Aufbau der Vorlesung und Ziele 1
- 1.2 Quellen 2
- Literatur 3

2 Signale, Nachrichten, Informationen, Daten 5
- 2.1 Signal 5
- 2.2 Übertragung und Speicherung 6
- 2.3 Nachricht 6
- 2.4 Information 7
- 2.5 Datum 8
- 2.6 Zusammenfassung 8

3 Alphabete, Abbildungen, Aussagenlogik 9
- 3.1 Alphabete 10
  - 3.1.1 Beispiel ASCII 10
  - 3.1.2 Beispiel Unicode 10
- 3.2 Relationen und Abbildungen 12
- 3.3 Logisches 13
- 3.4 Zusammenfassung und Ausblick 16

4 Wörter 17
- 4.1 Wörter 17
- 4.2 Das leere Wort 18
- 4.3 Mehr zu Wörtern 19
- 4.4 Konkatenation von Wörtern 20
  - 4.4.1 Konkatenation mit dem leeren Wort 21
  - 4.4.2 Eigenschaften der Konkatenation 22
  - 4.4.3 Beispiel: Aufbau von E-Mails 23
  - 4.4.4 Iterierte Konkatenation 24
- 4.5 Vollständige Induktion 27
- 4.6 Binäre Operationen 27
- 4.7 Zusammenfassung 28

5 **Formale Sprachen 29**
5.1 Formale Sprachen 29
5.2 Operationen auf formalen Sprachen 30
5.2.1 Produkt oder Konkatenation formaler Sprachen 30
5.2.2 Konkatenationsabschluss einer formalen Sprache 32
5.3 Zusammenfassung 33

6 **Der Begriff des Algorithmus 35**
6.1 Lösen einer Sorte quadratischer Gleichungen 36
6.2 Zum informellen Algorithmusbegriff 37
6.3 Zur Korrektheit des Algorithmus zur Lösung einer Sorte quadratischer Gleichungen 38
6.4 Wie geht es weiter? 39
6.5 Ein Algorithmus zur Multiplikation nichtnegativer ganzer Zahlen 40
6.6 Der Algorithmus zur Multiplikation nichtnegativer ganzer Zahlen mit einer Schleife 44

7 **Dokumente 49**
7.1 Dokumente 49
7.2 Struktur von Dokumenten 50
7.2.1 LaTeX 50
7.2.2 HTML und XHTML 52
7.2.3 Eine Grenze unserer bisherigen Vorgehensweise 53
7.3 Zusammenfassung 54

8 **Kontextfreie Grammatiken 55**
8.1 Rekursive Definition syntaktischer Strukturen 55
8.2 Kontextfreie Grammatiken 60
8.3 Relationen (Teil 2) 64
8.4 Ein Nachtrag zu Wörtern 65
8.5 Ausblick 66

9 **Speicher 67**
9.1 Bit und Byte 67
9.2 Speicher als Tabellen und Abbildungen 68
9.2.1 Hauptspeicher 68
9.3 Binäre und dezimale Größenpräfixe 71
9.4 Ausblick 72

10 **übersetzungen und Codierungen 73**

10.1 Von Wörtern zu Zahlen und zurück 73
10.1.1 Dezimaldarstellung von Zahlen 73
10.1.2 Andere Zahldarstellungen 74
10.1.3 Von Zahlen zu ihren Darstellungen 76
10.2 Von einem Alphabet zum anderen 77
10.2.1 Ein Beispiel: Übersetzung von Zahldarstellungen 77
10.2.2 Homomorphismen 79
10.2.3 Beispiel Unicode: UTF-8 Codierung 81
10.3 Huffman-Codierung 82
10.3.1 Algorithmus zur Berechnung von Huffman-Codes 83
10.3.2 Weiteres zu Huffman-Codes 87
10.4 Ausblick 88

11 Graphen 89
11.1 Gerichtete Graphen 89
11.1.1 Graphen und Teilgraphen 89
11.1.2 Pfade und Erreichbarkeit 92
11.1.3 Isomorphie von Graphen 93
11.1.4 Ein Blick zurück auf Relationen 94
11.2 Ungerichtete Graphen 95
11.2.1 Anmerkung zu Relationen 97
11.3 Graphen mit Knoten- oder Kantenmarkierungen 98
11.3.1 Gewichtete Graphen 99

12 Erste Algorithmen in Graphen 101
12.1 Repräsentation von Graphen im Rechner 101
12.2 Berechnung der 2-Erreichbarkeitsrelation und Rechnen mit Matrizen 105
12.2.1 Matrixmultiplikation 107
12.2.2 Matrixaddition 108
12.3 Berechnung der Erreichbarkeitsrelation 109
12.3.1 Potenzen der Adjazenzmatrix 110
12.3.2 Erste Möglichkeit für die Berechnung der Wegematrix 111
12.3.3 Zählen durchzuführender arithmetischer Operationen 113
12.3.4 Weitere Möglichkeiten für die Berechnung der Wegematrix 113
12.4 Algorithmus von Warshall 115
12.5 Ausblick 118
Literatur 118

13 Quantitative Aspekte von Algorithmen 119
13.1 Ressourcenverbrauch bei Berechnungen 119
13.2 Groß-O-Notation 120
13.2.1 Ignorieren konstanter Faktoren 121
13.2.2 Notation für obere und untere Schranken des Wachstums 125
13.2.3 Die furchtbare Schreibweise 126
13.2.4 Rechnen im O-Kalkül 127
13.3 Matrixmultiplikation 129
13.3.1 Rückblick auf die Schulmethode 129
13.3.2 Algorithmus von Strassen 130
13.4 Asymptotisches Verhalten „implizit" definierter Funktionen 131
13.5 Unterschiedliches Wachstum einiger Funktionen 133
13.6 Ausblick 134
Literatur 134

14 Endliche Automaten 135
14.1 Erstes Beispiel: ein Getränkeautomat 135
14.2 Mealy-Automaten 138
14.3 Moore-Automaten 140
14.4 Endliche Akzeptoren 142
14.4.1 Beispiele formaler Sprachen, die von endlichen Akzeptoren akzeptiert werden können 143
14.4.2 Ein formale Sprache, die von keinem endlichen Akzeptoren akzeptiert werden kann 145
14.5 Ausblick 146

15 Reguläre Ausdrücke und rechtslineare Grammatiken 147
15.1 Reguläre Ausdrücke 147
15.2 Rechtslineare Grammatiken 151
15.3 Kantorowitsch-Bäume und strukturelle Induktion 152
15.4 Ausblick 156
Literatur 156

16 Turingmaschinen 157
16.1 Alan Mathison Turing 157
16.2 Turingmaschinen 157
16.2.1 Berechnungen 160
16.2.2 Eingaben für Turingmaschinen 162
16.2.3 Ergebnisse von Turingmaschinen 163

16.3 Berechnungskomplexität 166
16.3.1 Komplexitätsmaße 166
16.3.2 Komplexitätsklassen 167
16.4 Unentscheidbare Probleme 169
16.4.1 Codierungen von Turingmaschinen 169
16.4.2 Das Halteproblem 170
16.4.3 Die Busy-Beaver-Funktion 172
16.5 Ausblick 174
Literatur 175

17 Relationen 177
17.1 Äquivalenzrelationen 177
17.1.1 Definition 177
17.1.2 Äquivalenzrelationen von Nerode 178
17.1.3 Äquivalenzklassen und Faktormengen 179
17.2 Kongruenzrelationen 180
17.2.1 Verträglichkeit von Relationen mit Operationen 181
17.2.2 Wohldefiniertheit von Operationen mit Äquivalenzklassen 182
17.3 Halbordnungen 183
17.3.1 Grundlegende Definitionen 183
17.3.2 „Extreme“ Elemente 186
17.3.3 Vollständige Halbordnungen 187
17.3.4 Stetige Abbildungen auf vollständigen Halbordnungen 188
17.4 Ordnungen 190
17.5 Ausblick 192

Index 193

Literatur 199

Colophon 201

1 PROLOG

Mark Twain wird der Ausspruch zugeschrieben:

> „Vorhersagen sind schwierig, besonders wenn sie die Zukunft betreffen.“

Wie recht er hatte kann man auch an den folgenden Zitaten sehen:

1943: „I think there is a world market for maybe five computers.“ (Thomas Watson, IBM)

1949: „Computers in the future may weigh no more than 1.5 tons.“ (Popular Mechanics)

1977: „There is no reason for any individual to have a computer in their home.“ (Ken Olson, DEC)

1981: „640K ought to be enough for anybody.“ (Bill Gates, Microsoft, bestreitet den Ausspruch)

2000: Es wurden mehr PCs als Fernseher verkauft.

Das lässt sofort die Frage aufkommen: Was wird am Ende Ihres Studiums der Fall sein? Sie können ja mal versuchen, auf einem Zettel aufzuschreiben, was in fünf Jahren am Ende Ihres Masterstudiums, das Sie hoffentlich an Ihr Bachelorstudium anschließen, wohl anders sein wird als heute, den Zettel gut aufheben und in fünf Jahren lesen.

Am Anfang Ihres Studiums steht jedenfalls die Veranstaltung „Grundbegriffe der Informatik“, die unter anderem verpflichtend für das erste Semester der Bachelorstudiengänge Informatik und Informationswirtschaft an der Universität Karlsruhe vorgesehen ist, zu denen man sich seit Wintersemester 2008/2009 einschreiben konnte.

Der vorliegende Text ist ein Vorlesungsskript zu dieser Veranstaltung, wie ich sie im Wintersemester 2009/2010 lese(n werde).

1.1 AUFBAU DER VORLESUNG UND ZIELE

Für diese Vorlesung stehen 15 Termine zu je 90 Minuten zur Verfügung. Der Vorlesungsinhalt auf eine Reihe überschaubarer inhaltlicher Einheiten aufgeteilt. Am Ende werden es vermutlich 18 Einheiten sein.

Die Vorlesung hat vordergründig mehrere Ziele. Zum einen sollen, wie der Name der Vorlesung sagt, eine ganze Reihe wichtiger Begriffe und Konzepte gelernt werden, die im Laufe des Studiums immer und immer wieder auftreten; typische Beispiele sind Graphen und endliche Automaten. Zum zweiten sollen parallel dazu einige Begriffe und Konzepte vermittelt werden, die man vielleicht eher der

Mathematik zuordnen würde, aber ebenfalls unverzichtbar sind. Drittens sollen die Studenten mit wichtigen Vorgehensweisen bei der Definition neuer Begriffe und beim Beweis von Aussagen vertraut gemacht werden. Induktives Vorgehen ist in diesem Zusammenhang wohl zu allererst zu nennen.

Andere Dinge sind nicht explizit Thema der Vorlesung, werden aber (hoffentlich) doch vermittelt. So bemühe ich mich mit diesem Skript zum Beispiel auch, klar zu machen,

- dass man präzise formulieren und argumentieren kann und muss,
- dass Formalismen ein Hilfsmittel sind, um *gleichzeitig verständlich (!) und präzise* formulieren zu können, und
- wie man ordentlich und ethisch einwandfrei andere Quellen benutzt und zitiert.

Ich habe versucht, der Versuchung zu widerstehen, prinzipiell wie in einem Nachschlagewerk im Haupttext überall einfach nur lange Listen von Definitionen, Behauptungen und Beweisen aneinander zu reihen. Gelegentlich ist das sinnvoll, und dann habe ich es auch gemacht, sonst aber hoffentlich nur selten.

Der Versuch, das ein oder andere anders und hoffentlich besser zu machen ist auch dem Buch „Lernen“ von Manfred Spitzer (2002) geschuldet. Es sei allen als interessante Lektüre empfohlen.

1.2 QUELLEN

Bei der Vorbereitung der Vorlesung habe ich mich auf diverse Quellen gestützt: Druckerzeugnisse und andere Quellen im Internet, die gelesen werden wollen, sind in den Literaturverweisen aufgeführt.

Explizit an dieser Stelle erwähnt seien die Bücher von Goos (2006) und Abeck (2005), die Grundlage waren für die Vorlesung „Informatik I“, den Vorgänger der diesjährigen Vorlesungen „Grundbegriffe der Informatik“ und „Programmieren“.

Gespräche und Diskussionen mit Kollegen sind nirgends zitiert. Daher sei zumindest an dieser Stellen pauschal allen gedankt, die – zum Teil womöglich ohne es zu wissen – ihren Teil beigetragen haben.

Für Hinweise auf Fehler und Verbesserungsmöglichkeiten bin ich allen Lesern dankbar.

Thomas Worsch, im Oktober 2009.

LITERATUR

Abeck, Sebastian (2005). *Kursbuch Informatik, Band 1*. Universitätsverlag Karlsruhe.

Goos, Gerhard (2006). *Vorlesungen über Informatik: Band 1: Grundlagen und funktionales Programmieren*. Springer-Verlag.

Spitzer, Manfred (2002). *Lernen: Gehirnforschung und Schule des Lebens*. Spektrum Akademischer Verlag.

2 SIGNALE, NACHRICHTEN, INFORMATIONEN, DATEN

Das Wort *Informatik* ist ein Kunstwort, das aus einem Präfix des Wortes *Information* und einem Suffix des Wortes *Mathematik* zusammengesetzt ist.

So wie es keine scharfe Grenzen zwischen z. B. Physik und Elektrotechnik gibt, sondern einen fließenden Übergang, so ist es z. B. auch zwischen Informatik und Mathematik und zwischen Informatik und Elektrotechnik. Wir werden hier nicht versuchen, das genauer zu beschreiben. Aber am Ende Ihres Studiums werden Sie vermutlich ein klareres Gefühl dafür entwickelt haben.

Was wir in dieser ersten Einheit klären wollen, sind die offensichtlich wichtigen Begriffe *Signal*, *Nachricht*, *Information* und *Datum*.

2.1 SIGNAL

Wenn Sie diese Zeilen vorgelesen bekommen, dann klappt das, weil Schallwellen vom Vorleser zu Ihren Ohren gelangen. Wenn Sie diese Zeilen lesen, dann klappt das, weil Lichtwellen vom Papier oder dem Bildschirm in Ihr Auge gelangen. Wenn Sie diesen Text auf einer Braillezeile (siehe Abbildung 2.1) ertasten, dann klappt das, weil durch Krafteinwirkung die Haut Ihrer Finger leicht deformiert wird. In allen Fällen sind es also physikalische Vorgänge, die ablaufen und

Abbildung 2.1: Eine Braillezeile, Quelle: http://commons.wikimedia.org/wiki/Image:Refreshable_Braille_display.jpg (10.10.2008)

im übertragenen oder gar wörtlichen Sinne einen „Eindruck“ davon vermitteln, was mitgeteilt werden soll.

Den Begriff *Mitteilung* wollen wir hier informell benutzen und darauf vertrauen, dass er von allen passend verstanden wird (was auch immer hier „passend“ be-

Signal

deuten soll). Die Veränderung einer (oder mehrerer) physikalischer Größen (zum Beispiel Schalldruck) um etwas mitzuteilen nennt man ein *Signal*.

Unter Umständen werden bei der Übermittlung einer Mitteilung verschiedene Signale benutzt: Lichtwellen dringen in die Augen des Vorlesers, was elektrische Signale erzeugt, die dazu führen, dass Schallwellen erzeugt werden, die ins Ohr des Hörers dringen. Dort werden dann ... und so weiter.

2.2 Übertragung und Speicherung

Schallwellen, Lichtwellen, usw. bieten die Möglichkeit, eine Mitteilung von einem Ort zu einem anderen zu übertragen. Damit verbunden ist (jedenfalls im alltäglichen Leben) immer auch das Vergehen von Zeit.

Inschrift

Es gibt eine weitere Möglichkeit, Mitteilungen von einem Zeitpunkt zu einem späteren zu „transportieren“: Die *Speicherung* als *Inschrift*. Die Herstellung von Inschriften mit Papier und Stift ist uns allen geläufig. Als es das noch nicht gab, benutzte man z. B. Felswände und Pinsel. Und seit einigen Jahrzehnten kann man auch magnetisierbare Schichten „beschriften“.

Aber was wird denn eigentlich gespeichert? Auf dem Papier stehen keine Schall- oder Lichtwellen oder andere Signale. Außerdem kann man verschiedene Inschriften herstellen, von denen Sie ganz selbstverständlich sagen würden, dass „da die gleichen Zeichen stehen“.

Um sozusagen zum Kern dessen vorzustoßen „was da steht“, bedarf es eines Aktes in unseren Köpfen. Den nennt man *Abstraktion*. Jeder hat vermutlich eine gewisse Vorstellung davon, was das ist. Ich wette aber, dass Sie gerade als Informatik-Studenten zum Abschluss Ihres Studiums ein sehr viel besseres Verständnis davon haben werden, was es damit genau auf sich hat. So weit sich der Autor dieses Textes erinnern kann (ach ja ...), war die zunehmende Rolle, die Abstraktion in einigen Vorlesungen spielte, sein Hauptproblem in den ersten beiden Semestern. Aber auch das ist zu meistern.

Im Fall der Signale und Inschriften führt Abstraktion zu dem Begriff, auf den wir als nächstes etwas genauer eingehen wollen:

2.3 Nachricht

Offensichtlich kann man etwas (immer Gleiches) auf verschiedene Arten, d. h. mit Hilfe verschiedener Signale, übertragen, und auch auf verschiedene Weisen speichern.

Das Wesentliche, das übrig bleibt, wenn man z. B. von verschiedenen Medien für die Signalübertragung oder Speicherung absieht, nennt man eine *Nachricht*.

Nachricht

Das, was man speichern und übertragen kann, sind also Nachrichten. Und: Man kann Nachrichten verarbeiten. Das ist einer der zentralen Punkte in der Informatik.

Mit Inschriften werden wir uns ein erstes Mal genauer in Einheit 3 genauer beschäftigen und mit Speicherung in Einheit 9. Beschreibungen, wie Nachrichten in gewissen Situationen zu verarbeiten sind, sind *Programme* (jedenfalls die einer bestimmten Art). Dazu erfahren Sie unter anderem in der parallel stattfindenden Vorlesung *Programmieren* mehr.

2.4 INFORMATION

Meist übertragt man Nachrichten nicht um ihrer selbst willen. Vielmehr ist man üblicherweise in der Lage, Nachrichten zu interpretieren und ihnen so eine *Bedeutung* zuzuordnen. Dies ist die einer Nachricht zugeordnete sogenannte *Information*.

Interpretation
Information

Wie eine Nachricht interpretiert wird, ist aber nicht eindeutig festgelegt. Das hängt vielmehr davon ab, welches „Bezugssystem“ der Interpretierende benutzt. Der Buchhändler um die Ecke wird wohl den

Text `1001` interpretieren als Zahl Tausendundeins,

aber ein Informatiker wird vielleicht eher den

Text `1001` interpretieren als Zahl Neun.

Ein Rechner hat „keine Ahnung“, was in den Köpfen der Menschen vor sich geht und welche Interpretationsvorschriften sie anwenden. Er verarbeitet also im obigen Sinne Nachrichten und keine Informationen.

Das heißt aber nicht, dass Rechner einfach immer nur sinnlose Aktionen ausführen. Vielmehr baut und programmiert man sie gerade so, dass zum Beispiel die Transformation von Eingabe- zu Ausgabenachrichten bei einer bestimmten festgelegten Interpretation zu einer beabsichtigten Informationsverarbeitung passt:

Rechner:	`42`	`17`	$\xrightarrow[\text{ausführung}]{\text{Programm-}}$	`59`
	Interpretation↓	Interpre↓tation		Interpre↓tation
Mensch:	zweiundvierzig	siebzehn	$\xrightarrow[\text{Addition}]{\text{Rechnen}}$	neunundfünfzig

2.5 DATUM

Die umgangssprachlich meist anzutreffende Bedeutung des Wortes „Datum“ ist die eines ganz bestimmten Tages, z. B. „2. Dezember 1958“. Ursprünglich kommt das Wort aus dem Lateinischen, wo „datum“ „das Gegebene“ heißt.

In der Informatik ist mit einem Datum aber oft etwas anderes gemeint: Syntaktisch handelt es sich um die Singularform des vertrauten Wortes „Daten“.

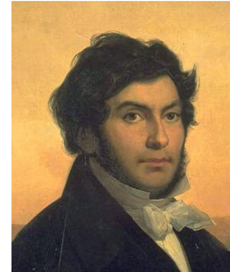
Unter einem Datum wollen wir ein Paar verstehen, das aus einer Nachricht und einer zugehörigen Information besteht.

Wenn man sich auf bestimmte feste Interpretationsmöglichkeiten von Nachrichten und eine feste Repräsentation dieser Möglichkeiten als Nachrichten geeinigt hat, kann man auch ein Datum als Nachricht repräsentieren (i. e. speichern oder übertragen).

2.6 ZUSAMMENFASSUNG

Signale übertragen und *Inschriften* speichern *Nachrichten*. Die Bedeutung einer Nachricht ist die ihr zugeordnete *Information* — im Rahmen es gewissen Bezugssystems für die Interpretation der Nachricht. Auf der Grundlage eines Bezugssystems ist ein *Datum* ein Paar, das aus einer Nachricht und der zugehörigen Information besteht.

3 ALPHABETE, ABBILDUNGEN, AUSSAGENLOGIK



In der Einheit über Signale, Nachrichten, ... haben wir auch über *Inschriften* gesprochen. Ein typisches Beispiel ist der Rosetta-Stein (Abb. 3.1), der für Jean-François Champollion *die* Hilfe war, um die Bedeutung ägyptischer Hieroglyphen zu entschlüsseln. Auf dem Stein findet man Texte in drei Schriften: in Hieroglyphen, in demotischer Schrift und auf Altgriechisch in griechischen Großbuchstaben.

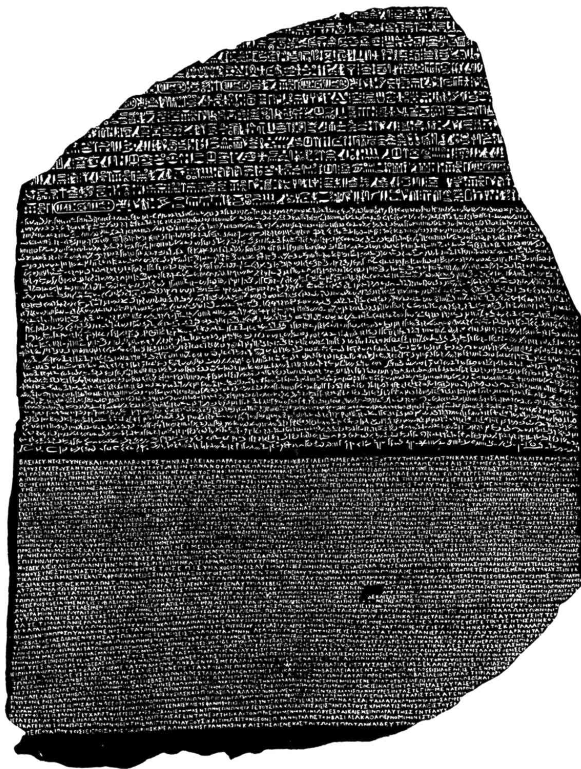


Abbildung 3.1: Der Rosetta-Stein, heute im Britischen Museum, London. Bildquelle: `http://www.gutenberg.org/files/16352/16352-h/images/p1.jpg` (19.10.08)

Wir sind gewohnt, lange Inschriften aus (Kopien der) immer wieder gleichen Zeichen zusammenzusetzen. Zum Beispiel in europäischen Schriften sind das die Buchstaben, aus denen Wörter aufgebaut sind. Im asiatischen Raum gibt es Schriften mit mehreren Tausend Zeichen, von denen viele jeweils für etwas stehen, was

wir als Wort bezeichnen würden.

3.1 ALPHABETE

Alphabet

Unter einem *Alphabet* wollen wir eine endliche Menge sogenannter *Zeichen* oder *Symbole* verstehen, die nicht leer ist. Was dabei genau „Zeichen“ sind, wollen wir nicht weiter hinterfragen. Es seien einfach die elementaren Bausteine, aus denen Inschriften zusammengesetzt sind. Hier sind einfache Beispiele:

- $A = \{\texttt{|}\}$
- $A = \{\texttt{a}, \texttt{b}, \texttt{c}\}$
- $A = \{\texttt{0}, \texttt{1}\}$
- Manchmal erfindet man auch Zeichen: $A = \{\texttt{1}, \texttt{0}, \rotatebox{180}{\texttt{1}}\}$
- $A = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}\}$

Gelegentlich nimmt man aber auch einen etwas abstrakteren Standpunkt ein und sieht zum Beispiel jeden der folgenden „Kästen“ als jeweils *ein* Zeichen eines gewissen Alphabetes an:

[int] [adams] [=] [42] [;]

3.1.1 Beispiel ASCII

Ein wichtiges Alphabet ist der sogenannte *ASCII*-Zeichensatz. Die Abkürzung steht für *American Standard Code for Information Interchange*. Diese Spezifikation umfasst insbesondere eine Liste von 94 „druckbaren“ und einem „unsichtbaren“ Zeichen, die man z. B. in Emails verwenden darf. Außerdem hat jedes Zeichen eine Nummer aus dem Bereich der natürlichen Zahlen zwischen 32 und 126. Die vollständige Liste findet man in Tabelle 3.1. Wie man dort sieht, fehlen in diesem Alphabet etliche Buchstaben aus nichtenglischen Alphabeten, wie zum Beispiel ä, ç, è, ğ, ñ, œ, ß, ů usw., von Kyrillisch, Japanisch und vielen anderen außereuropäischen Schriften ganz zu schweigen.

Auf ein Zeichen in Tabelle 3.1 sei ausdrücklich hingewiesen, nämlich das mit Nummer 32. Das ist das „Leerzeichen“. Man gibt es normalerweise auf einer Rechnertastatur ein, indem man die extrabreite Taste ohne Beschriftung drückt. Auf dem Bildschirm wird dafür in der Regel *nichts* dargestellt. Damit man es trotzdem sieht und um darauf aufmerksam zu machen, dass das ein Zeichen ist, ist es in der Tabelle als ␣ dargestellt.

3.1.2 Beispiel Unicode

Der Unicode Standard (siehe auch http://www.unicode.org) definiert mehrere Dinge. Das wichtigste und Ausgangspunkt für alles weitere ist eine umfassende

		40	(	50	2	60	<	70	F
		41	)	51	3	61	=	71	G
32	␣	42	*	52	4	62	>	72	H
33	!	43	+	53	5	63	?	73	I
34	"	44	,	54	6	64	@	74	J
35	#	45	-	55	7	65	A	75	K
36	$	46	.	56	8	66	B	76	L
37	%	47	/	57	9	67	C	77	M
38	&	48	0	58	:	68	D	78	N
39	'	49	1	59	;	69	E	79	O
80	P	90	Z	100	d	110	n	120	x
81	Q	91	[	101	e	111	o	121	y
82	R	92	\	102	f	112	p	122	z
83	S	93	]	103	g	113	q	123	{
84	T	94	^	104	h	114	r	124	|
85	U	95	_	105	i	115	s	125	}
86	V	96	`	106	j	116	t	126	~
87	W	97	a	107	k	117	u		
88	X	98	b	108	l	118	v		
89	Y	99	c	109	m	119	w		

Tabelle 3.1: Die „druckbaren“ Zeichen des ASCII-Zeichensatzes (einschließlich Leerzeichen)

Liste von Zeichen, die in der ein oder anderen der vielen heute gesprochenen Sprachen (z. B. in Europa, im mittleren Osten, oder in Asien) benutzt wird. Die Seite http://www.unicode.org/charts/ vermittelt einen ersten Eindruck von der existierenden Vielfalt.

Das ist mit anderen Worten ein Alphabet, und zwar ein großes: es umfasst rund 100 000 Zeichen.

Der Unicode-Standard spezifiziert weitaus mehr als nur einen Zeichensatz. Für uns sind hier zunächst nur die beiden folgenden Aspekte wichtig[1]:

1. Es wird eine große (aber endliche) Menge A_U von Zeichen festgelegt, und
2. eine Nummerierung dieser Zeichen, jedenfalls in einem gewissen Sinne.

Punkt 1 ist klar. Hinter der Formulierung von Punkt 2 verbirgt sich genauer folgendes: Jedem Zeichen aus A_U ist eine nichtnegative ganze Zahl zugeordnet, der

[1]Hinzu kommen in Unicode noch viele andere Aspekte, wie etwa die Sortierreihenfolge von Buchstaben (im Schwedischen kommt zum Beispiel ö *nach* z, im Deutschen kommt ö *vor* z), Zuordnung von Groß- zu Kleinbuchstaben und umgekehrt (soweit existent), und vieles mehr.

auch sogenannte *Code Point* des Zeichens. Die Liste der benutzten Code Points ist allerdings nicht „zusammenhängend".

Jedenfalls liegt eine Beziehung zwischen Unicode-Zeichen und nichtnegativen ganzen Zahlen vor. Man spricht von einer Relation. (Wenn Ihnen die folgenden Zeilen schon etwas sagen: schön. Wenn nicht, gedulden Sie sich bis Abschnitt 3.2 wenige Zeilen weiter.)

Genauer liegt sogar eine Abbildung $f : A_U \to \mathbb{N}_0$ vor. Sie ist

- eine Abbildung, weil jedem Zeichen nur *eine* Nummer zugewiesen wird,
- injektiv, weil verschiedenen Zeichen verschiedene Nummern zugewiesen werden,
- aber natürlich nicht surjektiv (weil A_U nur endlich viele Zeichen enthält).

Entsprechendes gilt natürlich auch für den ASCII-Zeichensatz.

3.2 RELATIONEN UND ABBILDUNGEN

Die Beziehung zwischen den Unicode-Zeichen in A_U und nichtnegativen ganzen Zahlen kann man durch die Angabe aller Paare (a, n), für die $a \in A_U$ ist und n der zu a gehörenden Code Point, vollständig beschreiben. Für die Menge U aller dieser Paare gilt also $U \subseteq A_U \times \mathbb{N}_0$.

kartesisches Produkt

Allgemein heißt $A \times B$ *kartesisches Produkt* der Mengen A und B. Es ist die Menge *aller* Paare (a, b) mit $a \in A$ und $b \in B$:

$$A \times B = \{(a, b) \mid a \in A \land b \in B\}$$

Relation
binäre Relation

Eine Teilmenge $R \subseteq A \times B$ heißt auch eine *Relation*. Manchmal sagt man noch genauer *binäre* Relation; und manchmal noch genauer „von A in B".

Die durch Unicode definierte Menge $U \subseteq A_U \times \mathbb{N}_0$ hat „besondere" Eigenschaften, die nicht jede Relation hat. Diese (und ein paar andere) Eigenschaften wollen wir im folgenden kurz aufzählen und allgemein definieren:

1. Zum Beispiel gibt es für jedes Zeichen $a \in A_U$ (mindestens) ein $n \in \mathbb{N}_0$ mit $(a, n) \in U$.

   linkstotal

   Allgemein nennt man eine Relation $R \subseteq A \times B$ *linkstotal*, wenn für jedes $a \in A$ (mindestens) ein $b \in B$ existiert mit $(a, b) \in R$.
2. Für kein Zeichen $a \in A_U$ gibt es mehrere $n \in \mathbb{N}_0$ mit der Eigenschaft $(a, n) \in U$.

   rechtseindeutig

   Allgemein nennt man eine Relation $R \subseteq A \times B$ *rechtseindeutig*, wenn es für kein $a \in A$ zwei $b_1 \in B$ und $b_2 \in B$ mit $b_1 \neq b_2$ gibt, so dass sowohl $(a, b_1) \in R$ als auch $(a, b_2) \in R$ ist.

3. Relationen, die linkstotal und rechtseindeutig sind, kennen Sie auch unter anderen Namen: Man nennt sie *Abbildungen* oder auch *Funktionen* und man schreibt dann üblicherweise $R : A \to B$. Es heißt dann A der *Definitionsbereich* und B der *Zielbereich* der Abbildung.

   Definitionsbereich
   Zielbereich

   Gelegentlich ist es vorteilhaft, sich mit Relationen zu beschäftigen, von denen man nur weiß, dass sie rechtseindeutig sind. Sie nennt man manchmal *partielle Funktionen*. (Bei ihnen verzichtet man also auf die Linkstotalität.)

   partielle Funktion
4. Außerdem gibt es bei Unicode keine zwei verschiedene Zeichen a_1 und a_2, denen der gleiche Code Point zugeordnet ist.

   Eine Relation $R \subseteq A \times B$ heißt *linkseindeutig*, wenn für alle $(a_1, b_1) \in R$ und alle $(a_2, b_2) \in R$ gilt:

   linkseindeutig

$$\text{wenn } a_1 \neq a_2, \text{ dann } b_1 \neq b_2 .$$

5. Eine Abbildung, die linkseindeutig ist, heißt *injektiv*.

   injektiv
6. Der Vollständigkeit halber definieren wir auch gleich noch, wann eine Relation $R \subseteq A \times B$ *rechtstotal* heißt: wenn für jedes $b \in B$ ein $a \in A$ existiert, für das $(a, b) \in R$ ist.

   rechtstotal
7. Eine Abbildung, die rechtstotal ist, heißt *surjektiv*.

   surjektiv
8. Eine Abbildung, die sowohl injektiv als auch surjektiv ist, heißt *bijektiv*.

   bijektiv

3.3 LOGISCHES

Im vorangegangenen Abschnitt stehen solche Dinge wie:

„Die Abbildung $U : A_U \to \mathbb{N}_0$ ist injektiv."

Das ist eine Aussage. Sie ist *wahr*.

„Die Abbildung $U : A_U \to \mathbb{N}_0$ ist surjektiv."

ist auch eine Aussage. Sie ist aber *falsch* und deswegen haben wir sie auch nicht getroffen.

Aussagen sind Sätze, die „objektiv" wahr oder falsch sind. Allerdings bedarf es dazu offensichtlich einer Interpretation der Zeichen, aus denen die zu Grunde liegende Nachricht zusammengesetzt ist.

Um einzusehen, dass es auch umgangssprachliche Sätze gibt, die nicht wahr oder falsch (sondern sinnlos) sind, mache man sich Gedanken zu Folgendem: „Ein Barbier ist ein Mann, der genau alle diejenigen Männer rasiert, die sich nicht selbst rasieren." Man frage sich insbesondere, ob sich ein Barbier selbst rasiert ...

Und wir bauen zum Beispiel in dieser Vorlesung ganz massiv darauf, dass es keine Missverständnisse durch unterschiedliche Interpretationsmöglichkeiten gibt.

Das ist durchaus nicht selbstverständlich: Betrachten Sie das Zeichen $\mathbb{N}$. Das schreibt man üblicherweise für eine Menge von Zahlen. Aber ist bei dieser Menge die 0 dabei oder nicht? In der Literatur findet man beide Varianten (und zumindest für den Autor dieser Zeilen ist nicht erkennbar, dass eine deutlich häufiger vorkäme als die andere).

Häufig setzt man aus einfachen Aussagen, im folgenden kurz mit $\mathcal{A}$ und $\mathcal{B}$ bezeichnet, kompliziertere auf eine der folgenden Arten zusammen:

Negation: „Nicht $\mathcal{A}$“
: Dafür schreiben wir auch kurz $\neg\mathcal{A}$.

logisches Und: „$\mathcal{A}$ und $\mathcal{B}$“
: Dafür schreiben wir auch kurz $\mathcal{A} \wedge \mathcal{B}$.

logisches Oder: „$\mathcal{A}$ oder $\mathcal{B}$“
: Dafür schreiben wir auch kurz $\mathcal{A} \vee \mathcal{B}$.

logische Implikation: „Wenn $\mathcal{A}$, dann $\mathcal{B}$“
: Dafür schreiben wir auch kurz $\mathcal{A} \Rightarrow \mathcal{B}$.

Ob so eine zusammengesetzte Aussage wahr oder falsch ist, hängt dabei *nicht* vom konkreten Inhalt der Aussagen ab! Wesentlich ist nur, welche Wahrheitswerte die Aussagen $\mathcal{A}$ und $\mathcal{B}$ haben, wie in der folgenden Tabelle dargestellt. Deswegen beschränkt und beschäftigt man sich dann in der Aussagenlogik mit sogenannten *aussagenlogischen Formeln*, die nach obigen Regeln zusammengesetzt sind und bei denen statt elementarer Aussagen einfach *Aussagevariablen* stehen, die als Werte „wahr“ und „falsch“ sein können.

aussagenlogische Formeln

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$
falsch	falsch	wahr	falsch	falsch	wahr
falsch	wahr	wahr	falsch	wahr	wahr
wahr	falsch	falsch	falsch	wahr	falsch
wahr	wahr	falsch	wahr	wahr	wahr

Das meiste, was in obiger Tabelle zum Ausdruck gebracht wird, ist aus dem alltäglichen Leben vertraut. Nur auf wenige Punkte wollen wir explizit eingehen:

- Das „Oder“ ist „inklusiv“ (und nicht „exklusiv“): Auch wenn A und B *beide* wahr sind, ist $A \vee B$ wahr.
- Man kann für komplizierte Aussagen anhand der obigen Tabellen „ausrechnen“, wenn sie wahr sind und wann falsch.

  Zum Beispiel ergibt einfaches Rechnen und scharfes Hinsehen, dass die Aussagen $\neg(A \wedge B)$ und $(\neg A) \vee (\neg B)$ immer gleichzeitig wahr bzw. falsch sind.

  Solche Aussagen nennt man *äquivalent*.

äquivalente Aussagen

- Gleiches gilt für $\neg\neg A$ und A.
- Die Implikation $A \Rightarrow B$ ist auf jeden Fall wahr, wenn A falsch ist, unabhängig vom Wahrheitsgehalt von B, insbesondere auch dann, wenn B falsch ist. Zum Beispiel ist die Aussage

  Wenn $0 = 1$ ist, dann ist $42 = -42$.

  wahr.

  Das wird unter Umständen noch etwas klarer, wenn man überlegt, was man sich denn unter dem „Gegenteil" von $A \Rightarrow B$ vorstellen sollte. Das ist doch wohl $A \wedge \neg B$. Also ist $A \Rightarrow B$ äquivalent zu $\neg(A \wedge \neg B)$, und das ist nach obigem äquivalent zu $(\neg A) \vee (\neg\neg B)$ und das zu $(\neg A) \vee B$.
- Dabei haben wir jetzt so getan, als dürfe man selbstverständlich in einer Aussage einen Teil durch einen äquivalenten Teil ersetzen. Das darf man auch, denn von Bedeutung ist ja immer nur der Wahrheitswert einer Formel und nicht ihr syntkatischer Aufbau.

Alles was wir bisher in diesem Abschnitt betrachtet haben, gehört zu dem Bereich der *Aussagenlogik*. Wir werden sie im beschriebenen Sinne naiv verwenden und in Zukunft zum Beispiel Definitionen wie die für Injektivität und Surjektivität von Funktionen entsprechend kompakter schreiben können.

Außerdem gibt es die sogenannte *Prädikatenlogik*. (Genauer gesagt interessiert uns Prädikatenlogik erster Stufe, ohne dass wir die Bedeutung dieser Bezeichnung jetzt genauer erklären wollen oder können.)

Aus der Prädikatenlogik werden wir — wieder ganz naiv — die sogenannten *Quantoren* verwenden:

Quantoren

Allquantor $\forall$ *Existenzquantor* $\exists$

In der puren Form hat eine quantifizierte Aussage eine der Formen

$$\forall x\ A(x) \qquad \text{oder} \qquad \exists x\ A(x)$$

Dabei soll $A(x)$ eine Aussage sein, die von einer Variablen x abhängt (oder jedenfalls abhängen kann). A kann weitere Quantoren enthalten.

Die Formel $\forall x\ A(x)$ hat man zu lesen als: „Für alle x gilt: $A(x)$". Und die Formel $\exists x\ A(x)$ hat man zu lesen als: „Es gibt ein x mit: $A(x)$".

Zum Beispiel ist die Formel

$$\forall x\ (x \in \mathbb{N}_0 \Rightarrow \exists y\ (y \in \mathbb{N}_0 \wedge y = x + 1))$$

wahr.

Sehr häufig hat man wie in diesem Beispiel den Fall, dass eine Aussage nicht für alle x gilt, sondern nur für x aus einer gewissen Teilmenge M. Statt

$$\forall x\ (x \in M \Rightarrow B(x))$$

schreibt man oft kürzer

$$\forall x \in M : \; B(x)$$

wobei der Doppelpunkt nur die Lesbarkeit verbessern soll. Obiges Beispiel wird damit zu

$$\forall x \in \mathbb{N}_0 : \exists y \in \mathbb{N}_0 : y = x + 1$$

3.4 ZUSAMMENFASSUNG UND AUSBLICK

In dieser Einheit wurde der Begriff *Alphabet*, eingeführt, die im weiteren Verlauf der Vorlesung noch eine große Rolle spielen werden. Die Begriffe *Wort* und *formale Sprache* werden in den nächsten Einheiten folgen.

Mehr über Schriften findet man zum Beispiel über die WWW-Seite `http://www.omniglot.com/writing/` (1.10.08).

Als wichtige technische Hilfsmittel wurden die Begriffe *binäre Relation*, sowie *injektive*, *surjektive* und *bijektive Abbildungen* definiert, und es wurden informell einige Schreibweisen zur kompakten aber lesbaren Notation von Aussagen eingeführt.

Ein bisschen *Aussagenlogik* haben wir auch gemacht.

4 WÖRTER

4.1 WÖRTER

Jeder weiß, was ein Wort ist: Ein *Wort über einem Alphabet A* ist eine Folge von Zeichen aus A. Aber gerade weil jeder weiß, was das ist, werden wir uns im folgenden eine Möglichkeit ansehen, eine formale Definition des Begriffes „Wort“ zu geben. Sinn der Übung ist aber nicht, eine einfache Sache möglichst kompliziert darzustellen, sondern an einem Beispiel, das niemandem ein Problem bereitet, Dinge zu üben, die in späteren Einheiten noch wichtig werden.

Wort über einem Alphabet A

Vorher aber noch kurz eine Bemerkung zu einem Punkt, an dem wir sich der Sprachgebrauch in dieser Vorlesung (und allgemeiner in der Theorie der formalen Sprachen) vom umgangssprachlichen unterscheidet: Das *Leerzeichen*. Übrigens benutzt man es heutzutage (jedenfalls z. B. in europäischen Schriften) zwar ständig — früher aber nicht! Aber wie der Name sagt, fassen wir auch das Leerzeichen als ein Zeichen auf. Damit man es sieht, schreiben wir manchmal explizit `␣` statt einfach nur Platz zu lassen.

Konsequenz der Tatsache, dass wir das Leerzeichen, wenn wir es denn überhaupt benutzen, als ein ganz normales Symbol auffassen, ist jedenfalls, dass z. B. `Hallo␣Welt` *eine* Folge von Zeichen, also nur *ein* Wort ist (und nicht zwei).

Was man für eine mögliche technische Definition von „Wort“ braucht, ist im wesentlichen eine Formalisierung von „Liste“ (von Symbolen). Für eine bequeme Notation definieren wir zunächst: Für jede natürliche Zahl $n \geq 0$ sei $\mathbb{G}_n = \{i \in \mathbb{N}_0 \mid 0 \leq i \wedge i < n\}$ die Menge der n kleinsten nichtnegativen ganzen Zahlen. Zum Beispiel ist $\mathbb{G}_4 = \{0, 1, 2, 3\}$, $\mathbb{G}_1 = \{0\}$ und $\mathbb{G}_0 = \{\}$.

Dann wollen wir jede *surjektive* Abbildung $w : \mathbb{G}_n \to A$ als ein *Wort* auffassen. Die Zahl n heiße auch die *Länge* des Wortes, für die man manchmal kurz $|w|$ schreibt. (Es ist in Ordnung, wenn Sie im Moment nur an Längen $n \geq 1$ denken. Auf den Fall des sogenannten leeren Wortes ε mit Länge $n = 0$ kommen wir im nachfolgenden Abschnitt gleich noch zu sprechen.)

Wort, formalistisch definiert

Länge eines Wortes

Das Wort (im umgangssprachlichen Sinne) $w =$ `hallo` ist dann also formal die Abbildung $w : \mathbb{G}_5 \to \{\mathtt{a}, \mathtt{h}, \mathtt{l}, \mathtt{o}\}$ mit $w(0) = \mathtt{h}$, $w(1) = \mathtt{a}$, $w(2) = \mathtt{l}$, $w(3) = \mathtt{l}$ und $w(4) = \mathtt{o}$.

Im folgenden werden wir uns erlauben, manchmal diese formalistische Sicht auf Wörter zu haben, und manchmal die vertraute von Zeichenfolgen. Dann ist insbesondere jedes einzelne Zeichen auch schon ein Wort. Formalismus, vertraute Sichtweise und das Hin- und Herwechseln zwischen beidem ermöglich dreierlei:

- präzise Argumentationen, wenn andernfalls nur vages Händewedeln möglich wäre,

- leichteres Vertrautwerden mit Begriffen und Vorgehensweisen bei Wörtern und formalen Sprachen, und
- das langsame Vertrautwerden mit Formalismen.

Menge aller Wörter

A^*

Ganz häufig ist man in der Situation, dass man ein Alphabet A gegeben hat und über die *Menge aller Wörter* reden möchte, in denen höchstens die Zeichen aus A vorkommen. Dafür schreibt man A^*. Ganz formalistisch gesehen ist das also Menge aller surjektiven Abbildungen $w : \mathbb{G}_n \to B$ mit $n \in \mathbb{N}_0$ und $B \subseteq A$. Es ist aber völlig in Ordnung, wenn Sie sich einfach Zeichenketten vorstellen.

4.2 DAS LEERE WORT

Beim Zählen ist es erst einmal natürlich, dass man mit eins beginnt: 1, 2, 3, 4, ... Bei Kindern ist das so, und geschichtlich gesehen war es bei Erwachsenen lange Zeit auch so. Irgendwann stellte sich jedoch die Erkenntnis ein, dass die Null auch ganz praktisch ist. Daran hat sich jeder gewöhnt, wobei vermutlich eine gewisse Abstraktion hilfreich war; oder stellen Sie sich gerade vor, dass vor Ihnen auf dem Tisch 0 Elefanten stehen?

Ebenso umfasst unsere eben getroffene Definition von „Wort" den Spezialfall der Wortlänge $n = 0$. Auch ein Wort der Länge 0 verlangt zugegebenermaßen ein bisschen Abstraktionsvermögen. Es besteht aus 0 Symbolen. Deshalb sieht man es so schlecht.

Wenn es Ihnen hilft, können Sie sich die formalistische Definition ansehen: Es ist $\mathbb{G}_0 = \{\}$ die leere Menge; und ein Wort der Länge 0 enthält keine Zeichen. Formalisiert als surjektive Abbildung ergibt das dann ein $w : \{\} \to \{\}$.

Wichtig:

- Wundern Sie sich nicht, wenn Sie sich über $w : \{\} \to \{\}$ erst einmal wundern. Sie werden sich an solche Dinge schnell gewöhnen.
- Vielleicht haben Sie ein Problem damit, dass der Definitionsbereich oder/und der Zielbereich von $w : \{\} \to \{\}$ die leere Menge ist. Das ist aber nicht wirklich eines: Man muss nur daran denken, dass Abbildungen besondere Relationen sind.
- Es gibt nur *eine* Relation $R \subseteq \{\} \times \{\} = \{\}$, nämlich $R = \{\}$. Als Menge von Paaren aufgefasst ist dieses R aber linkstotal und rechtseindeutig, also tatsächlich eine Abbildung; und die ist sogar rechtstotal. Also ist es richtig von *dem leeren Wort* zu sprechen.

leeres Wort

Nun gibt es ähnlich wie schon beim Leerzeichen ein ganz praktisches Problem: Da das leere Wort aus 0 Symbolen besteht, „sieht man es nicht". Das führt leicht zu Verwirrungen. Man will aber gelegentlich ganz explizit darüber sprechen und schreiben. Deswegen vereinbaren wir, dass wir für das leere Wort ε schreiben.

Beachten Sie, dass wir in unseren Beispielen Symbole unseres Alphabetes immer blau darstellen, ε aber nicht. Es ist nie Symbol das gerade untersuchten Alphabetes. Wir benutzen dieses Zeichen aus dem Griechischen als etwas, das Sie immer interpretieren (siehe Einheit 2) müssen, nämlich als das leere Wort.

4.3 MEHR ZU WÖRTERN

Für die Menge aller Wörter einer festen Länge n über einem Alphabet A schreiben wir auch A^n. Wenn zum Beispiel das zu Grunde liegende Alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ ist, dann ist:

A^n

$$\begin{aligned} A^0 &= \{\varepsilon\} \\ A^1 &= \{\mathtt{a}, \mathtt{b}\} \\ A^2 &= \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\} \\ A^3 &= \{\mathtt{aaa}, \mathtt{aab}, \mathtt{aba}, \mathtt{abb}, \mathtt{baa}, \mathtt{bab}, \mathtt{bba}, \mathtt{bbb}\} \end{aligned}$$

Vielleicht haben nun manche die Idee, dass man auch erst die A^n hätte definieren können, und dann festlegen:

$$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots$$

Das unschöne daran sind die „$\cdots$“: Im vorliegenden Fall mag ja noch klar sein, was gemeint ist. Da wir aber darauf achten wollen, dass Sie sich nichts angewöhnen, was im allgemeinen zu Problemen führen könnte (und dafür sind Pünktchen, bei denen man darauf baut, dass der Leser schon die passende Interpretation haben möge, prädestiniert), wollen wir lieber folgende Schreibweise benutzen:

$$A^* = \bigcup_{i=0}^{\infty} A^i$$

Allerdings sind hier nun zwei Anmerkungen ganz wichtig:

- Sie können mit Recht fragen, was denn präzise so etwas wie

  $$\bigcup_{i=0}^{\infty} M_i$$

  bedeuten soll, wenn M_0, M_1, M_2, $\ldots$ (schon wieder die Pünktchen ...) unendlich viele Menge sind. Das hier:

  $$\bigcup_{i=0}^{\infty} M_i = \{x \mid \exists i : x \in M_i\}$$

also alle Elemente, die in mindestens einem M_i enthalten sind.

- Das ∞-Zeichen in obiger Schreibweise ist leider gefährlich. Es kann so missverstanden werden, als könne i auch „den Wert Unendlich" annehmen. *Das ist **nicht** so!* Gemeint ist nur, dass i alle Werte aus dem Bereich der ganzen Zahlen ab $i = 0$ durchläuft. Und jede dieser Zahlen ist *endlich*; es sind aber unendlich viele.

4.4 KONKATENATION VON WÖRTERN

Zahlen kann man zum Beispiel addieren oder multiplizieren. Man spricht auch davon, dass die Addition und Multiplikation zweistellige oder binäre Operationen sind.

Konkatenation

Für Wörter definieren wir nun auch eine ganz einfache aber wichtige binäre Operation: die sogenannte *Konkatenation* von Wörtern. Das ist einfach die Hintereinanderschreibung zweier Wörter. Als Operationssymbol verwendet man üblicherweise wie bei der Multiplikation von Zahlen den Punkt „·". Also zum Beispiel:

$$\texttt{SCHRANK} \cdot \texttt{SCHLÜSSEL} = \texttt{SCHRANKSCHLÜSSEL}$$

oder

$$\texttt{SCHLÜSSEL} \cdot \texttt{SCHRANK} = \texttt{SCHLÜSSELSCHRANK}$$

Oft lässt man wie bei der Multiplikation auch den Konkatenationspunkt weg.

Wie man sieht, kommt es (im Gegensatz zur Multiplikation von Zahlen) auf die Reihenfolge an: Ein `SCHRANKSCHLÜSSEL` ist etwas anderes als ein `SCHLÜSSELSCHRANK`.

Konkatenation zweier Wörter formal

Nun wollen wir die *Konkatenation zweier Wörter formal* definieren.

4.1 Definition. Es seien zwei beliebige Wörter $w_1 : \mathbb{G}_m \to A_1$ und $w_2 : \mathbb{G}_n \to A_2$ gegeben. Dann ist

$$\begin{aligned} w_1 \cdot w_2 : \mathbb{G}_{m+n} &\to A_1 \cup A_2 \\ i &\mapsto \begin{cases} w_1(i) & \text{falls } 0 \leq i < m \\ w_2(i-m) & \text{falls } m \leq i < m+n \end{cases} \end{aligned}$$

Ist das eine sinnvolle Definition? Oder vielmehr: Ist das überhaupt eine Definition? Und wird hier ein Wort definiert?

- Als erstes hat man sich zu überlegen, dass die Ausdrücke $w_1(i)$ für $0 \leq i < m$ und $w_2(i-m)$ für $m \leq i < m+n$ stets definiert sind. Das ist so.
- Zweitens stammen die in der Fallunterscheidung vorgeschriebenen Funktionswerte tatsächlich aus dem Bereich $A_1 \cup A_2$: denn $w_1(i)$ ist stets aus A_1 und $w_2(i-m)$ ist stets aus A_2.

- Drittens muss man sich klar machen, dass die Fallunterscheidung von der Art ist, dass für jedes $i \in \mathbb{G}_{m+n}$ nur genau *ein* Funktionswert festgelegt wird und nicht mehrere verschiedene.
- Und schließlich muss man sich noch klar machen, dass wieder ein *Wort* definiert wird: Dafür muss die Abbildung $w_1 \cdot w_2 : \mathbb{G}_{m+n} \to A_1 \cup A_2$ surjektiv sein. Das ist sie auch. Denn für jedes $a \in A_1 \cup A_2$ gilt (mindestens) eine der folgenden Möglichkeiten:
  - $a \in A_1$: Dann gibt es aber, da w_1 ein Wort ist, also eine surjektive Abbildung, ein $i_1 \in \mathbb{G}_m$ mit $w_1(i_1) = a$. Also ist $(w_1 w_2)(i_1) = w_1(i_1) = a$.
  - $a \in A_2$: Dann gibt es aber, da w_2 ein Wort ist, also eine surjektive Abbildung, ein $i_2 \in \mathbb{G}_n$ mit $w_2(i_2) = a$. Also ist $(w_1 w_2)(m + i_2) = w_2(i_2) = a$.

Als letztes sei noch angemerkt, dass man an der Definition sofort sieht:

4.2 Lemma. Für jedes Alphabet A gilt:

$$\forall w_1 \in A^* \; \forall w_2 \in A^* : |w_1 w_2| = |w_1| + |w_2|$$

4.4.1 Konkatenation mit dem leeren Wort

Gefragt, was des Besondere an der Zahl Null ist, antworten zumindest manche Leute, dass es die Eigenschaft hat:

$$\forall x \in \mathbb{N}_0 : x + 0 = x \land 0 + x = x$$

Man sagt auch, die Null sei das *neutrale Element* bezüglich der Addition. *neutrales Element*

Etwas Ähnliches wie die Null für natürliche Zahlen gibt es bei Wörtern: Das leere Wort ist das neutrale Element bezüglich Konkatenation.

4.3 Lemma. Für jedes Alphabet A gilt:

$$\forall w \in A^* : w \cdot \varepsilon = w \land \varepsilon \cdot w = w \; .$$

Anschaulich ist das wohl klar: Wenn man ein Wort w nimmt und hinten dran der Reihe nach noch alle Symbole des leeren Wortes „klebt", dann „ändert sich an w nichts".

Aber da wir auch eine formale Definition von Wörtern haben, können wir das auch präzise beweisen ohne auf Anführungszeichen und „ist doch wohl klar" zurückgreifen zu müssen. Wie weiter vorne schon einmal erwähnt: Wir machen das nicht, um Einfaches besonders schwierig darzustellen (so etwas hat angeblich Herr Gauß manchmal gemacht . . .), sondern um an einem einfachen Beispiel etwas zu üben, was Sie durch Ihr ganzes Studium begleiten wird: Beweisen.

4.4 Beweis. Die erste Frage, die sich stellt, ist: Wie beweist man das für alle denkbaren(?) Alphabete A? Eine Möglichkeit ist: Man geht von einem wie man sagt „beliebigen aber festen" Alphabet A aus, über das man *keinerlei weitere* Annahmen macht und zeigt, dass die Aussage für dieses A gilt.

Tun wir das: Sei im folgenden A ein beliebiges aber festes Alphabet.

Damit stellt sich die zweite Frage: Wie beweist man, dass die Behauptung für alle $w \in A^*$ gilt? Im vorliegenden Fall funktioniert das gleiche Vorgehen wieder: Man geht von einem beliebigen Wort w aus, über das man keinerlei Annahmen macht.

Sei also im folgenden w ein beliebiges aber festes Wort aus A^*, d. h. eine surjektive Abbildung $w : \mathbb{G}_m \to B$ mit $B \subseteq A$.

Außerdem wissen wir, was das leere Wort ist: $\varepsilon : \mathbb{G}_0 \to \{\}$.

Um herauszufinden, was $w' = w \cdot \varepsilon$ ist, können wir nun einfach losrechnen: Wir nehmen die formale Definition der Konkatenation und setzen unsere „konkreten" Werte ein. Dann ist also w' eine Abbildung $w' : \mathbb{G}_{m+0} \to B \cup \{\}$, also schlicht $w' : \mathbb{G}_m \to B$. Und für alle $i \in \mathbb{G}_m$ gilt für die Funktionswerte laut der formalen Definition von Konkatenation für alle $i \in \mathbb{G}_m$:

$$\begin{aligned} w'(i) &= \begin{cases} w_1(i) & \text{falls } 0 \leq i < m \\ w_2(i-m) & \text{falls } m \leq i < m+n \end{cases} \\ &= \begin{cases} w(i) & \text{falls } 0 \leq i < m \\ \varepsilon(i-m) & \text{falls } m \leq i < m+0 \end{cases} \\ &= w(i) \end{aligned}$$

Also haben w und w' die gleichen Definitions- und Zielbereiche und für alle Argumente die gleichen Funktionswerte, d. h. an allen Stellen die gleichen Symbole. Also ist $w' = w$. ∎

4.4.2 Eigenschaften der Konkatenation

Wenn man eine neue binäre Operation definiert, stellt sich immer die Frage nach möglichen Rechenregeln. Weiter oben haben wir schon darauf hingewiesen, dass man bei der Konkatenation von Wörtern nicht einfach die Reihenfolge vertauschen darf. (Man sagt auch, die Konkatenation sei *nicht kommutativ*.)

Was ist, wenn man mehrere Wörter konkateniert? Ist für jedes Alphabet A und alle Wörter w_1, w_2 und w_3 aus A^* stets

$$(w_1 \cdot w_2) \cdot w_3 = w_1 \cdot (w_2 \cdot w_3)\ ?$$

Die Antwort ist: Ja. Auch das kann man stur nachrechnen.

Das bedeutet, dass man bei der Konkatenation mehrerer Wörter keine Klammern setzen muss. Man sagt auch, die Konkatenation sei eine *assoziative Operation* (siehe Unterabschnitt 4.6).

4.4.3 Beispiel: Aufbau von E-Mails

Die Struktur von E-Mails ist in einem sogenannten RFC festgelegt. *RFC* ist die Abkürzung für *Request For Comment*. Man findet alle RFCs zum Beispiel unter `http://tools.ietf.org/html/`.

RFC

Request For Comment

Die aktuelle Fassung der Spezifikation von E-Mails findet man in *RFC 2822* (`http://tools.ietf.org/html/rfc2822`, 28.10.09). Wir zitieren und komentieren[1] einige Ausschnitte aus der Einleitung von Abschnitt 2.1 dieses RFC, wobei die Nummerierung/Strukturierung von uns stammt:

RFC 2822

1. • *„This standard specifies that messages are made up of characters in the US-ASCII range of 1 through 127."*
   • Das Alphabet, aus dem die Zeichen stammen müssen, die in einer E-Mail vorkommen, ist der US-ASCII-Zeichensatz mit Ausnahme des Zeichens mit der Nummer 0.
2. • *„Messages are divided into lines of characters. A line is a series of characters that is delimited with the two characters carriage-return and line-feed; that is, the carriage return (CR) character (ASCII value 13) followed immediately by the line feed (LF) character (ASCII value 10). (The carriage-return/line-feed pair is usually written in this document as "CRLF".)"*
   • Eine Zeile (*line*) ist eine Folge von Zeichen, also ein Wort, das mit den beiden „nicht druckbaren" Symbolen `CR` `LF` endet.

     An anderer Stelle wird im RFC übrigens spezifiziert, dass als Zeile im Sinne des Standards nicht beliebige Wörter zulässig sind, sondern nur solche, deren Länge kleiner oder gleich 998 ist.
3. • *A message consists of*
     – *[...] the header of the message [...] followed,*
     – *optionally, by a body."*
   • Eine E-Mail (*message*) ist die Konkatenation von Kopf (*header*) der E-Mail und Rumpf (*body*) der E-Mail. Dass der Rumpf optional ist, also sozusagen fehlen darf, bedeutet nichts anderes, als dass der Rumpf auch das leere Wort sein darf. (Aus dem Rest des RFC ergibt sich, dass der Kopf nie das leere Wort sein kann.)

[1] Wir erlauben uns kleine Ungenauigkeiten, weil wir nicht den ganzen RFC zitieren wollen.

Aber das ist noch nicht ganz vollständig. Gleich anschließend wird der RFC genauer:

4.
- – *„The header is a sequence of lines of characters with special syntax as defined in this standard.*
  – *The body is simply a sequence of characters that follows the header and*
  – *is separated from the header by an empty line (i.e., a line with nothing preceding the CRLF). [...]"*
- Es gilt also:
  – Der Kopf einer E-Mail ist die Konkatenation (de facto mehrerer) Zeilen.
  – Der Rumpf einer E-Mail ist (wie man an anderer Stelle im RFC nachlesen kann) ebenfalls die Konkatenation von Zeilen. Es können aber auch 0 Zeilen oder 1 Zeile sein.
  – Eine Leerzeile (*empty line*) ist das Wort `CR` `LF`.
  – Eine Nachricht ist die Konkatenation von Kopf der E-Mail, einer Leerzeile und Rumpf der E-Mail.

4.4.4 Iterierte Konkatenation

Von den Zahlen kennen Sie die Potenzschreibweise x^3 für $x \cdot x \cdot x$ usw. Das wollen wir nun auch für die Konkatenation von Wörtern einführen. Die Idee ist so etwas wie

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

Aber da stehen wieder diese Pünktchen ... Wie kann man die vermeiden? Was ist mit $k = 1$ (immerhin stehen da ja drei w auf der rechten Seite)? Und was soll man sich für $k = 0$ vorstellen?

induktive Definition

Potenzen von Wörtern

Ein möglicher Ausweg ist eine sogenannte *induktive Definition*. Für *Potenzen von Wörtern* geht das so:

$$w^0 = \varepsilon$$
$$\forall k \in \mathbb{N}_0 : \; w^{k+1} = w^k \cdot w$$

Man definiert also

- explizit, was ein Exponent 0 bedeuten soll, nämlich $w^0 = \varepsilon$, (es hat sich einfach als nützlich erwiesen, das gerade so festzulegen), und
- wie man w^{n+1} ausrechnen kann, wenn man schon w^n kennt.

Damit kann man z. B. ausrechnen, was w^1 ist:

$$w^1 = w^{0+1} = w^0 \cdot w = \varepsilon \cdot w = w$$

Und dann:

$$w^2 = w^{1+1} = w^1 \cdot w = w \cdot w$$

Und so weiter. Man gewinnt den Eindruck, dass für jede nichtnegative ganze Zahl n als Exponent eindeutig festgelegt ist, was w^n sein soll. Das ist in der Tat so. Machen Sie sich das in diesem Beispiel klar!

4.5 Lemma. Für jedes Alphabet A und jedes Wort $w \in A^*$ gilt:

$$\forall n \in \mathbb{N}_0 : |w^n| = n|w| \ .$$

Wie kann man das beweisen? Immer wenn in einer Aussage „etwas" eine Rolle spielt, das induktiv definiert wurde, sollte man in Erwägung ziehen, für den Beweis *vollständige Induktion* zu benutzen.

vollständige Induktion

Sehen wir uns mal ein paar einfache Spezialfälle der Behauptung an:

- $n = 0$: Das ist einfach: $|w^0| = |\varepsilon| = 0 = 0 \cdot |w|$.
- $n = 1$: Natürlich könnten wir einfach oben nachsehen, dass $w^1 = w$ ist, und folgern $|w^1| = |w|$. Oder wir wiederholen im wesentlichen die obige Rechnung:

$$\begin{aligned} |w^1| = |w^{0+1}| &= |w^0 \cdot w| \\ &= |w^0| + |w| && \text{siehe Lemma 4.2} \\ &= 0|w| + |w| && \text{siehe Fall } n = 0 \\ &= 1|w| \end{aligned}$$

  Man kann auch sagen: Weil die Behauptung für $n = 0$ richtig war, konnten wir sie auch für $n = 1$ beweisen.
- $n = 2$: Wir gehen analog zu eben vor:

$$\begin{aligned} |w^2| = |w^{1+1}| &= |w^1 \cdot w| \\ &= |w^1| + |w| && \text{siehe Lemma 4.2} \\ &= 1|w| + |w| && \text{siehe Fall } n = 1 \\ &= 2|w| \end{aligned}$$

  Man kann auch sagen: Weil die Behauptung für $n = 1$ richtig war, konnten wir sie auch für $n = 2$ beweisen.

- Sie erkennen nun wohl das Muster: Weil w^{n+1} mit Hilfe von w^n definiert wurde, kann man die Richtigkeit der Behauptung für $|w^{n+1}|$ aus der für $|w^n|$ folgern.
- Also gilt das folgende: Wenn wir mit M die Menge aller natürlichen Zahlen n bezeichnen, für die die Behauptung $|w^n| = n|w|$ gilt, dann wissen wir also:
  1. $0 \in M$
  2. $\forall n \in \mathbb{N}_0 : \ (n \in M \Rightarrow n+1 \in M)$
- Es ist ein Faktum der Mathematik, dass eine Menge M, die nur natürliche Zahlen enthält und die Eigenschaften 1 und 2 von eben hat, gerade die Menge $\mathbb{N}_0$ ist. (Wenn man die natürlichen Zahlen axiomatisch definiert, ist das sogar gerade eine der Forderungen.) Auf der genannten Tatsache fußt das Beweisprinzip der *vollständigen Induktion*, das wir nun erst einmal in der einfachsten Form vorführen.

vollständige Induktion

4.6 Beweis. Wir schreiben nun im wesentlichen noch einmal das gleiche wie oben in der für Induktionsbeweise üblichen Form auf.

Sei im folgenden A ein beliebiges aber festes Alphabet und $w \in A^*$ ein beliebiges aber festes Wort.

Induktionsanfang $n = 0$: Zu zeigen ist: $|w^0| = 0 \cdot |w|$.

Das geht ganz ausführlich aufgeschrieben so:

$$\begin{aligned} |w^0| &= |\varepsilon| && \text{nach Defintion von } w^0 \\ &= 0 = 0 \cdot |w| \ . \end{aligned}$$

Induktionsschritt $n \to n+1$: Zu zeigen ist:

Für alle n gilt: Wenn $|w^n| = n|w|$ ist, dann ist auch $|w^{n+1}| = (n+1)|w|$.

Wie kann man zeigen, dass eine Aussage für *alle* natürlichen Zahlen n gilt? Eine Möglichkeit ist jedenfalls wieder, von einem „beliebigen, aber festen“ n auszugehen und für „dieses“ n zu zeigen:

$$|w^n| = n|w| \Rightarrow |w^{n+1}| = (n+1)|w| \ .$$

Mit anderen Worten macht man nun die

Induktionsannahme oder Induktionsvoraussetzung: für ein beliebiges aber festes n gilt: $|w^n| = n|w|$.

Zu leisten ist nun mit Hilfe dieser Annahme der Nachweis, dass auch $|w^{n+1}| = (n+1)|w|$. Das nennt man den

Induktionsschluss: In unserem Fall:

$$\begin{aligned} |w^{n+1}| &= |w^n \cdot w| && \text{nach Definition 4.1} \\ &= |w^n| + |w| && \text{nach Lemma 4.2} \\ &= n|w| + |w| && \text{nach Induktionsvoraussetzung} \\ &= (n+1)|w| \end{aligned}$$

4.5 vollständige induktion

Zusammenfassend wollen wir noch einmal festhalten, dass das Prinzip der vollständigen Induktion auf der folgenden Tatsache beruht:

4.7 (Prinzip der vollständigen Induktion) Wenn man für eine Aussage $\mathcal{A}(n)$, die von einer Zahl $n \in \mathbb{N}_0$ abhängt, weiß

vollständige Induktion

es gilt	$\mathcal{A}(0)$
und es gilt	$\forall n \in \mathbb{N}_0 : (\mathcal{A}(n) \Rightarrow \mathcal{A}(n+1))$

dann gilt auch:

$$\forall n \in \mathbb{N}_0 : \ \mathcal{A}(n) \ .$$

Den Beweis schreibt man typischerweise in folgender Struktur auf
Induktionsanfang: Man zeigt, dass $\mathcal{A}(0)$ gilt.
Induktionsvoraussetzung: für beliebiges aber festes $n \in \mathbb{N}_0$ gilt: $\mathcal{A}(n)$.
Induktionsschluss: Man zeigt, dass auch $\mathcal{A}(n+1)$ gilt
(und war typischerweise unter Verwendung von $\mathcal{A}(n)$).

4.6 binäre operationen

Unter einer binären Operation auf einer Menge M versteht man eine Abbildung $f : M \times M \to M$. Üblicherweise benutzt man aber ein „Operationssysmbol" wie das Pluszeichen oder den Multiplikationspunkt und setzt ihn zwischen die Argumente: Statt $+(3,8) = 11$ schreibt man normalerweise $3 + 8 = 11$.

Allgemein heißt eine binäre Operation $\diamond : M \times M \to M$ genau dann *kommutativ*, wenn gilt:

kommutative Operation

$$\forall x \in M \ \forall y \in M : x \diamond y = y \diamond x \ .$$

Eine binäre Operation $\diamond : M \times M \to M$ nennt man genau dann *assoziativ*, wenn gilt:

assoziative Operation

$$\forall x \in M \ \forall y \in M \ \forall z \in M : (x \diamond y) \diamond z = x \diamond (y \diamond z) \ .$$

Wir haben gesehen, dass die Konkatenation von Wörtern eine assoziative Operation ist, die aber *nicht* kommutativ ist.

4.7 ZUSAMMENFASSUNG

In dieser Einheit wurde eingeführt, was wir unter einem *Wort* verstehen wollen, und wie *Konkatenation* und *Potenzen* von Wörtern definiert sind.

Als wichtiges technisches Hilfsmittel haben wir Beispiele *induktiver Defintionen* gesehen und das Beweisprinzip der *vollständigen Induktion*.

 © worsch 2008–2010

5 FORMALE SPRACHEN

5.1 FORMALE SPRACHEN

Eine natürliche Sprache umfasst mehrere Aspekte, z. B. Aussprache und Stil, also z. B. Wortwahl und Satzbau. Dafür ist es auch notwendig zu wissen, welche Formulierungen syntaktisch korrekt sind. Neben den anderen genannten und ungenannten Punkten spielt *syntaktsiche Korrektheit* auch in der Informatik an vielen Stellen eine Rolle.

Bei der Formulierung von Programmen ist das jedem klar. Aber auch der Text, der beim Senden einer Email über das Netz transportiert wird oder der Quelltext einer HTML-Seite müssen bestimmten Anforderungen genügen. Praktisch immer, wenn ein Programm Eingaben liest, sei es aus einer Datei oder direkt vom Benutzer, müssen diese Eingaben gewissen Regeln genügen, sofern sie weiterverarbeitet werden können sollen. Wird z. B. vom Programm die Darstellung einer Zahl benötigt, dann ist vermutlich „`101`“ in Ordnung, aber „`a*&W`“ nicht. Aber natürlich (?) sind es bei jeder Anwendung andere Richtlinien, die eingehalten werden müssen.

Es ist daher nicht verwunderlich, wenn

- syntaktische Korrektheit,
- Möglichkeiten zu spezifizieren, was korrekt ist und was nicht, und
- Möglichkeiten, syntaktische Korrektheit von Texten zu überprüfen,

von großer Bedeutung in der Informatik sind.

Man definiert: Eine *formale Sprache* (über einem Alphabet A) ist eine Teilmenge $L \subseteq A^*$.

formale Sprache

Immer, wenn es um syntaktische Korrektheit geht, bilden die syntaktisch korrekten Gebilde eine formale Sprache L, während die syntaktisch falschen Gebilde eben *nicht* zu L gehören.

Beispiele:

- Es sei $A = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \texttt{-}\}$. Die formale Sprache der Dezimaldarstellungen ganzer Zahlen enthält zum Beispiel die Wörter „`1`“, „`-22`“ und „`192837465`“, aber nicht „`2-3--41`“.
- Die formale Sprache der syntaktisch korrekten Java-Programme über dem Unicode-Alphabet enthält zum Beispiel nicht das Wort „`[2] class int)(`“ (aber eben alle Java-Programme).

5.2 OPERATIONEN AUF FORMALEN SPRACHEN

5.2.1 Produkt oder Konkatenation formaler Sprachen

Wir haben schon definiert, was die Konkatenation zweier Wörter ist. Das erweitern wir nun auf eine übliche Art und Weise auf Mengen von Wörtern: Für zwei formale Sprachen L_1 und L_2 heißt

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

Produkt von Sprachen

das *Produkt der Sprachen L_1 und L_2*.

5.1 Lemma. Für jede formale Sprache L ist

$$L \cdot \{\varepsilon\} = L = \{\varepsilon\} \cdot L \; .$$

5.2 Beweis. Einfaches Nachrechnen:

$$\begin{aligned} L \cdot \{\varepsilon\} &= \{w_1 w_2 \mid w_1 \in L \land w_2 \in \{\varepsilon\}\} \\ &= \{w_1 w_2 \mid w_1 \in L \land w_2 = \varepsilon\} \\ &= \{w_1 \varepsilon \mid w_1 \in L\} \\ &= \{w_1 \mid w_1 \in L\} \\ &= L \end{aligned}$$

Analog zeigt man $L = \{\varepsilon\} \cdot L$. ∎

In Abschnitt 4.4.3 haben wir ein erstes Mal über den Aufbau von E-Mails gesprochen. Produkte formaler Sprachen könnte man nun benutzen, um folgende Festlegung zu treffen:

- Die formale Sprache L_{email} der syntaktisch korrekten E-Mails ist

$$L_{email} = L_{header} \cdot L_{leer} \cdot L_{body}$$

- Dabei sei
  - L_{header} die formale Sprache der syntaktisch korrekten E-Mail-Köpfe,
  - L_{leer} die formale Sprache, die nur die Leerzeile enthält, also $L_{leer} =$ { CR LF } und
  - L_{body} die formale Sprache der syntaktisch korrekten E-Mail-Rümpfe.

L_{header} und L_{body} muss man dann natürlich auch noch definieren. Eine Möglichkeit, das bequem zu machen, werden wir in einem späteren Kapitel kennenlernen.

Potenzen L^k

Wie bei Wörtern will man *Potenzen L^k* definieren. Der „Trick" besteht darin,

für den Fall $k = 0$ etwas Sinnvolles zu finden — Lemma 5.1 gibt einen Hinweis. Die Definition geht (wer hätte es gedacht?) wieder induktiv:

$$L^0 = \{\varepsilon\}$$
$$\forall k \in \mathbb{N}_0 : \ L^{k+1} = L \cdot L^k$$

Wie auch schon bei der Konkatenation einzelner Wörter kann man auch hier wieder nachrechnen, dass z. B. gilt:

$$\begin{aligned} L^1 &= L \\ L^2 &= L \cdot L \\ L^3 &= L \cdot L \cdot L \end{aligned}$$

Genau genommen hätten wir in der dritten Zeile $L \cdot (L \cdot L)$ schreiben müssen. Aber Sie dürfen glauben (oder nachrechnen), dass sich die Assoziativität vom Produkt von Wörtern auf das Produkt von Sprachen überträgt.

Als einfaches Beispiel betrachte man $L = \{\mathtt{aa}, \mathtt{b}\}$. Dann ist

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^1 &= \{\mathtt{aa}, \mathtt{b}\} \\ L^2 &= \{\mathtt{aa}, \mathtt{b}\} \cdot \{\mathtt{aa}, \mathtt{b}\} = \{\mathtt{aa} \cdot \mathtt{aa}, \mathtt{aa} \cdot \mathtt{b}, \mathtt{b} \cdot \mathtt{aa}, \mathtt{b} \cdot \mathtt{b}\} \\ &= \{\mathtt{aaaa}, \mathtt{aab}, \mathtt{baa}, \mathtt{bb}\} \\ L^3 &= \{\mathtt{aa} \cdot \mathtt{aa} \cdot \mathtt{aa},\ \mathtt{aa} \cdot \mathtt{aa} \cdot \mathtt{b},\ \mathtt{aa} \cdot \mathtt{b} \cdot \mathtt{aa},\ \mathtt{aa} \cdot \mathtt{b} \cdot \mathtt{b}, \\ &\qquad \mathtt{b} \cdot \mathtt{aa} \cdot \mathtt{aa},\ \mathtt{b} \cdot \mathtt{aa} \cdot \mathtt{b},\ \mathtt{b} \cdot \mathtt{b} \cdot \mathtt{aa},\ \mathtt{b} \cdot \mathtt{b} \cdot \mathtt{b}\} \\ &= \{\mathtt{aaaaaa}, \mathtt{aaaab}, \mathtt{aabaa}, \mathtt{aabb}, \mathtt{baaaa}, \mathtt{baab}, \mathtt{bbaa}, \mathtt{bbb}\} \end{aligned}$$

In diesem Beispiel ist L endlich. Man beachte aber, dass die Potenzen auch definiert sind, wenn L unendlich ist. Betrachten wir etwa den Fall

$$L = \{\mathtt{a}^n \mathtt{b}^n \mid n \in \mathbb{N}_+\} ,$$

es ist also (angedeutet)

$$L = \{\mathtt{ab}, \mathtt{aabb}, \mathtt{aaabbb}, \mathtt{aaaabbbb}, \dots\} .$$

Welche Wörter sind in L^2? Die Definion besagt, dass man alle Produkte $w_1 w_2$ von Wörtern $w_1 \in L$ und $w_2 \in L$ bilden muss. Man erhält also (erst mal ungenau

hingeschrieben)

$$\begin{aligned} L^2 = \quad & \{\mathtt{ab} \cdot \mathtt{ab}, \mathtt{ab} \cdot \mathtt{aabb}, \mathtt{ab} \cdot \mathtt{aaabbb}, \dots\} \\ \cup\, & \{\mathtt{aabb} \cdot \mathtt{ab}, \mathtt{aabb} \cdot \mathtt{aabb}, \mathtt{aabb} \cdot \mathtt{aaabbb}, \dots\} \\ \cup\, & \{\mathtt{aaabbb} \cdot \mathtt{ab}, \mathtt{aaabbb} \cdot \mathtt{aabb}, \mathtt{aaabbb} \cdot \mathtt{aaabbb}, \dots\} \\ & \vdots \end{aligned}$$

Mit anderen Worten ist

$$L^2 = \{\mathtt{a}^{n_1}\mathtt{b}^{n_1}\mathtt{a}^{n_2}\mathtt{b}^{n_2} \mid n_1 \in \mathbb{N}_+ \land n_2 \in \mathbb{N}_+\} .$$

Man beachte, dass bei die Exponenten n_1 „vorne“ und n_2 „hinten“ verschieden sein dürfen.

Für ein Alphabet A und für $i \in \mathbb{N}_0$ hatten wir auch die Potenzen A^i definiert. Und man kann jedes Alphabet ja auch als eine formale Sprache auffassen, die genau alle Wörter der Länge 1 über A enthält. Machen Sie sich klar, dass die beiden Definitionen für Potenzen konsistent sind, d. h. A^i ergibt immer die gleiche formale Sprache, egal, welche Definition man zu Grunde legt.

5.2.2 Konkatenationsabschluss einer formalen Sprache

Bei Alphabeten hatten wir neben den A^i auch noch A^* definiert und darauf hingewiesen, dass für ein Alphabet A gilt:

$$A^* = \bigcup_{i=0}^{\infty} A^i .$$

Konkatenationsabschluss L^* von L
ε-freier Konkatenationsabschluss L^+ von L

Das nehmen wir zum Anlass nun den *Konkatenationsabschluss L^* von L* und den *ε-freien Konkatenationsabschluss L^+ von L* definieren:

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

Wie man sieht, ist $L^* = L^0 \cup L^+$. In L^* sind also alle Wörter, die sich als Produkt einer beliebigen Zahl (einschließlich 0) von Wörtern schreiben lassen, die alle Element von L sind.

Als Beispiel betrachten wieder $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}_+\}$. Weiter vorne hatten wir schon gesehen:

$$L^2 = \{\mathtt{a}^{n_1}\mathtt{b}^{n_1}\mathtt{a}^{n_2}\mathtt{b}^{n_2} \mid n_1 \in \mathbb{N}_+ \land n_2 \in \mathbb{N}_+\} .$$

Analog ist

$$L^3 = \{\mathtt{a}^{n_1}\mathtt{b}^{n_1}\mathtt{a}^{n_2}\mathtt{b}^{n_2}\mathtt{a}^{n_3}\mathtt{b}^{n_3} \mid n_1 \in \mathbb{N}_+ \land n_2 \in \mathbb{N}_+ \land n_3 \in \mathbb{N}_+\} .$$

Wenn wir uns erlauben, Pünktchen zu schreiben, dann ist allgemein

$$L^i = \{\mathtt{a}^{n_1}\mathtt{b}^{n_1} \cdots \mathtt{a}^{n_i}\mathtt{b}^{n_i} \mid n_1 \dots, n_i \in \mathbb{N}_+\} .$$

Und für L^+ könnte man vielleicht notieren:

$$L^+ = \{\mathtt{a}^{n_1}\mathtt{b}^{n_1} \cdots \mathtt{a}^{n_i}\mathtt{b}^{n_i} \mid i \in \mathbb{N}_+ \land n_1 \dots, n_i \in \mathbb{N}_+\} .$$

Aber man merkt (hoffentlich!) an dieser Stelle doch, dass uns $^+$ und * die Möglichkeit geben, etwas erstens präzise und zweitens auch noch kürzer zu notieren, als wir es sonst könnten.

Zum Abschluss wollen wir noch darauf hinweisen, dass die Bezeichnung ε-freier Konkatenationsabschluss für L^+ leider (etwas?) irreführend ist. Wie steht es um das leere Wort bei L^+ und L^*? Klar sollte inzwischen sein, dass für *jede* formale Sprache L gilt: $\varepsilon \in L^*$. Das ist so, weil ja $\varepsilon \in L^0 \subseteq L^*$ ist. Nun läuft zwar in der Definition von L^+ die Vereinigung der L^i nur ab $i = 1$. Es kann aber natürlich sein, dass $\varepsilon \in L$ ist. In diesem Fall ist dann aber offensichtlich $\varepsilon \in L = L^1 \subseteq L^+$. Also kann L^+ sehr wohl das leere Wort enthalten.

Außerdem sei erwähnt, dass die Definition von L^* zur Folge hat, dass gilt:

$$\{\}^* = \{\varepsilon\}$$

5.3 ZUSAMMENFASSUNG

In dieser Einheit wurden *formale Sprachen* eingeführt, ihr *Produkt* und der *Konkatenationsabschluss*.

Wir haben gesehen, dass man damit jedenfalls manche formalen Sprachen kurz und verständlich beschreiben kann. Dass diese Notationsmöglichkeiten auch in der Praxis Verwendung finden, werden wir in der nächsten Einheit sehen.

Manchmal reicht das, was wir bisher an Notationsmöglichkeiten haben, aber noch nicht. Deshalb werden wir in späteren Einheiten mächtigere Hilfsmittel kennenlernen.

6 DER BEGRIFF DES ALGORITHMUS

Muhammad ibn Mūsā al-Khwārizmī lebte ungefähr von 780 bis 850. Abbildung 6.1 zeigt eine (relativ beliebte weil copyright-freie) Darstellung auf einer russischen Briefmarke anlässlich seines (jedenfalls ungefähr) 1200. Geburtstages. Im Jahr 830 (oder wenig früher) schrieb al-Khwārizī ein wichtiges Buch mit dem

Abbildung 6.1: Muhammad ibn Mūsā al-Khwārizmī; Bildquelle: http://commons.wikimedia.org/wiki/Image:Abu_Abdullah_Muhammad_bin_Musa_al-Khwarizmi.jpg (4.11.2010)

Titel „Al-Kitāb al-mukhtaṣar fī hīsāb al-ğabr wa'l-muqābala". (An anderer Stelle findet man als Titel „Al-Kitāb al-mukhtaṣar fī ḥisāb al-jabr wa-l-muqābala".) Die deutsche Übersetzung lautet in etwa „Das kurzgefasste Buch zum Rechnen durch Ergänzung und Ausgleich". Aus dem Wort „al-ğabr" bzw. „al-jabr" entstand später das Wort *Algebra*. Inhalt des Buches ist unter anderem die Lösung quadratischer Gleichungen mit einer Unbekannten.

Einige Jahre früher (825?) entstand das Buch, das im Original vielleicht den Titel „Kitāb al-Jam' wa-l-tafrīq bi-ḥisāb al-Hind" trug, auf Deutsch "Über das Rechnen mit indischen Ziffern". Darin führt al-Khwārizī die aus dem Indischen stammende Zahl Null in das arabische Zahlensystem ein und führt die Arbeit mit Dezimalzahlen vor. Von diesem Buch existieren nur noch Übersetzungen, zum Beispiel auf Lateinisch aus dem vermutlich 12. Jahrhundert. Abbildung 6.2 zeigt einen Ausschnitt aus einer solchen Übersetzung.

Von diesen Fassungen ist kein Titel bekannt, man vermutet aber, dass er „Algoritmi de numero Indorum" oder „Algorismi de numero Indorum" gelautet haben könnte, also ein Buch „des Al-gorism über die indischen Zahlen". Das „i" am Ende von „Algorismi" wurde später fälschlicherweise als Pluralendung des Wortes

Herkunft das Wortes Algorithmus

Algorithmus angesehen.

Womit wir etwas zur Ethymologie eines der wichtigsten Begriffe der Informatik gesagt hätten.

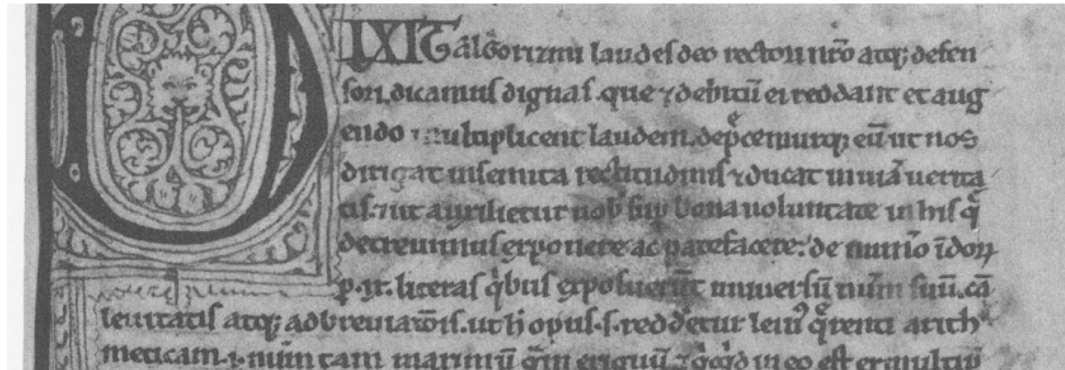


Abbildung 6.2: Ausschnitt einer Seite einer lateinischen Übersetzung der Arbeit von al-Khwārizmī über das „Rechnen mit indischen Ziffern"; Bildquelle: http://en.wikipedia.org/wiki/Image:Dixit_algorizmi.png (4.11.2010)

6.1 Lösen einer Sorte quadratischer Gleichungen

Angenommen, man hat eine quadratische Gleichung der Form $x^2 + bx = c$, wobei b und c positive Zahlen sind. Dann, so al-Khwarizmi, kann man die positive Lösung dieser Gleichung bestimmen, indem man nacheinander wie folgt rechnet:

$$h \leftarrow b/2 \tag{6.1}$$

$$q \leftarrow h^2 \tag{6.2}$$

$$s \leftarrow c + q \tag{6.3}$$

$$w \leftarrow \sqrt{s} \tag{6.4}$$

$$x \leftarrow w - h \tag{6.5}$$

(Natürlich ist die Beschreibung der durchzuführenden Rechnungen bei al-Khwarizmi anders.)

Wir haben hier eine Notation gewählt, bei der in jeder Zeile ein Pfeil $\leftarrow$ steht. Links davon steht ein symbolischer Name. Rechts vom Pfeil steht ein arithmetischer Ausdruck, in dem zum einen die beiden Namen b und c für die Eingangs-

größen vorkommen, zum anderen symbolische Namen, die schon einmal in einer *darüberliegenden* Zeile auf der linken Seite vorkamen.

Durch eine solche *Zuweisung* wird die linke Seite zu einem Namen für den Wert, den man aus der rechten Seite ausrechnen kann.

Man kann Schritt für Schritt alle Zuweisungen „ausführen". Es passieren keine Unglücke (s ist nie negativ.) Man kann sich überlegen, dass am Ende x immer einen Wert bezeichnet, der die quadratische Gleichung $x^2 + bx = c$ erfüllt.

6.2 ZUM INFORMELLEN ALGORITHMUSBEGRIFF

Das gerade besprochene Beispiel besitzt einige Eigenschaften, die man allgemein beim klassischen Algorithmusbegriff fordert:

- Der Algorithmus besitzt eine *endliche Beschreibung* (ist also ein Wort über einem Alphabet).
- Die Beschreibung besteht aus *elementaren Anweisungen*, von denen jede offensichtlich effektiv in einem Schritt ausführbar ist.
- *Determinismus:* Zu jedem Zeitpunkt ist eindeutig festgelegt, welches die nächste elementare Anweisung ist, und diese Festlegung hängt nur ab
  - von schon berechneten Ergebnissen und
  - davon, welches die zuletzt ausgeführte elementare Anweisung war.
- Aus einer *endlichen Eingabe* wird eine *endliche Ausgabe* berechnet.
- Dabei werden *endliche viele Schritte* gemacht, d. h. nur endlich oft eine elementare Anweisung ausgeführt.
- Der Algorithmus funktioniert für *beliebig große Eingaben.*
- Die *Nachvollziehbarkeit/Verständlichkeit* des Algorithmus steht für jeden (mit der Materie vertrauten) außer Frage.

Zwei Bemerkungen sind hier ganz wichtig:

- So plausibel obige Forderungen sind, so informell sind sie aber andererseits: Was soll z. B. „offensichtlich effektiv ausführbar" heißen? Für harte Beweise benötigt man einen präziseren Algorithmusbegriff.
- Im Laufe der Jahre hat sich herausgestellt, dass es durchaus auch interessant ist, Verallgemeinerungen des oben skizzierten Algorithmenbegriffes zu betrachten. Dazu gehören zum Beispiel
  - randomisierte Algorithmen, bei denen manchmal die Fortsetzung nicht mehr eindeutig ist, sondern abhängig von Zufallsereignissen,
  - Verfahren, bei denen nicht von Anfang an alle Eingaben zur Verfügung stehen, sondern erst nach und nach (z. B. Cacheverwaltung in Prozessoren), und
  - Verfahren, die nicht terminieren (z. B. Ampelsteuerung).

6.3 Zur Korrektheit des Algorithmus zur Lösung einer Sorte quadratischer Gleichungen

Al-Khwarizmi gibt einen sehr schönen Beweis dafür an, dass die Rechnungen in (6.1)-(6.5) das Ergebnis liefern. Er beruht auf einer geometrischen Überlegung (siehe z. B. `http://www-history.mcs.st-and.ac.uk/~history/Biographies/Al-Khwarizmi.html`, 4.11.2010).

Man kann in diesem konkreten Fall auch einfach den am Ende berechneten Wert z. B. in die linke Seite der Ausgangsgleichung einsetzen und nachrechnen, dass sich als Ergebnis c ergibt. Wir schreiben die fünf Zuweisungen noch einmal auf und fügen nach jeder eine logische Formel ein, die eine gültige Aussage über berechnete Werte macht. Den Formeln ist ein doppelter Aufstrich // vorangestellt, mit dem in Java Kommentare gekennzeichnet werden. Man nennt so etwas auch eine *Zusicherung*:

Zusicherung

// $b > 0 \wedge c > 0$

$h \leftarrow b/2$

// $h = b/2$

$q \leftarrow h^2$

// $q = b^2/4$

$s \leftarrow c + q$

// $s = c + b^2/4$

$w \leftarrow \sqrt{s}$

// $w = \sqrt{c + b^2/4}$

$x \leftarrow w - h$

// $x = \sqrt{c + b^2/4} - b/2$

// $x^2 + bx = (\sqrt{c + b^2/4} - b/2)^2 + b(\sqrt{c + b^2/4} - b/2)$

// $x^2 + bx = c + b^2/4 - b\sqrt{c + b^2/4} + b^2/4 + b\sqrt{c + b^2/4} - b^2/2$

// $x^2 + bx = c$

Genauer und besser ist es, wenn man zum Verständnis einer Zusicherung nicht alle vorherigen noch einmal studieren muss, sondern nur die unmittelbar vorangehende. Damit dann in unserem Beispiel trotzdem klar ist, was nach der Zuweisung

$x \leftarrow w - h$ der Fall ist, muss man die Information, dass $h = b/2$ ist, Schritt für Schritt mitführen. Das ergibt folgendes Bild:

// $b > 0 \land c > 0$

$h \leftarrow b/2$

// $h = b/2$

$q \leftarrow h^2$

// $q = b^2/4 \land h = b/2$

$s \leftarrow c + q$

// $s = c + b^2/4 \land h = b/2$

$w \leftarrow \sqrt{s}$

// $w = \sqrt{c + b^2/4} \land h = b/2$

$x \leftarrow w - h$

// $x = \sqrt{c + b^2/4} - b/2$

// $x^2 + bx = (\sqrt{c + b^2/4} - b/2)^2 + b(\sqrt{c + b^2/4} - b/2)$

// $x^2 + bx = c + b^2/4 - b\sqrt{c + b^2/4} + b^2/4 + b\sqrt{c + b^2/4} - b^2/2$

// $x^2 + bx = c$

6.4 WIE GEHT ES WEITER?

Wir wollen im Laufe der Vorlesung zum Beispiel sehen, wie man zumindest in nicht allzu schwierigen Fällen *allgemein* vorgehen kann, um sich davon zu überzeugen, dass solche Folgen von Rechnungen das richtige Ergebnis liefern. Das gehört dann zum Thema *Verifikation* von Algorithmen.

Dafür benötigt man zumindest die folgenden „Zutaten“:

- einen präzisen Algorithmenbegriff,
- eine präzise Spezifikation des „richtigen Verhaltens“ (bei uns meist der Zusammenhang zwischen gegebenen Eingaben und dazu gehörenden Ausgaben) und
- präzise Methoden, um z. B. zu beweisen, dass das Verhalten eines Algorithmus der Spezifikation genügt.
- Dazu kommt in allen Fällen auch eine präzise Notation, die zumindest bei der Verarbeitung durch Rechner nötig ist.

In einigen der nachfolgenden Einheiten werden wir diese Punkte aufgreifen und in verschiedenen Richtungen weiter vertiefen.

- Präzisierungen des Algorithmenbegriffes gibt es viele, und Sie werden schon in diesem Semester mehrere kennenlernen:
  - Sie kennen inzwischen z. B. Grundzüge einer Programmiersprache. Die ist praktisch, wenn man tatsächlich Algorithmen so aufschreiben will, dass sie ein Rechner ausführen können soll.
  - Die ist aber andererseits unpraktisch, wenn man z. B. beweisen will, dass ein bestimmtes Problem durch keinen Algorithmus gelöst werden kann. Dann sind einfachere Modelle wie Registermaschinen oder Turingmaschinen besser geeignet.
- „Ordentliche“ Notationen dafür, was „das richtige Verhalten“ eines Algorithmus ist, gibt es viele. Wie im vorangegangenen Abschnitt werden wir gleich und in einer späteren Einheit logische Formeln benutzen.
- Dort werden wir auch eine Methode kennenlernen, um zu beweisen, dass ein Algorithmus „das Richtige tut“.
- Präzise Notationen sind nicht nur wichtig, um sich etwas sicherer sein zu können, keine Fehler gemacht zu haben. Sie sind auch unabdingbar, wenn man Aufgaben dem Rechner übertragen will. Und dazu gehören nicht nur „Rechnungen“, sondern z. B. auch
  - die Analyse von Beschreibungen aller Art (z. B. von Programmen oder Ein-/Ausgabe-Spezifikationen) auf syntaktische Korrektheit
  - Beweise (etwa der Korrektheit von Algorithmen bezüglich spezifizierter Anforderungen).

6.5 Ein Algorithmus zur Multiplikation nichtnegativer ganzer Zahlen

Betrachten wir als erstes den in Abbildung 6.3 dargestellten einfachen Algorithmus, der als Eingaben eine Zahl $a \in \mathbb{G}_8$ und eine Zahl $b \in \mathbb{N}_0$ erhält. Variablen X_0 und Y_0 werden mit a und b initialisiert und Variable P_0 mit 0.

Es werden zwei binäre Operationen **div** und **mod** benutzt, die wie folgt definiert sind: Die Operation **mod** liefere für zwei Argumente x und y als Funktionswert x **mod** y den Rest der ganzzahligen Division von x durch y. Und die Operation **div** liefere für zwei Argumente x und y als Funktionswert x **div** y den Wert der ganzzahligen Division von x durch y. Mit anderen Worten gilt für nichtnegative ganze Zahlen x und y stets:

$$x = y \cdot (x \textbf{ div } y) + (x \textbf{ mod } y) \qquad \text{und} \qquad 0 \leq (x \textbf{ mod } y) < y \tag{6.6}$$

```
// Eingaben: a ∈ G_8, b ∈ N_0
P_0 ← 0
X_0 ← a
Y_0 ← b
x_0 ← X_0 mod 2
// — Algorithmusstelle — i = 0
P_1 ← P_0 + x_0 · Y_0
X_1 ← X_0 div 2
Y_1 ← 2 · Y_0
x_1 ← X_1 mod 2
// — Algorithmusstelle — i = 1
P_2 ← P_1 + x_1 · Y_1
X_2 ← X_1 div 2
Y_2 ← 2 · Y_1
x_2 ← X_2 mod 2
// — Algorithmusstelle — i = 2
P_3 ← P_2 + x_2 · Y_2
X_3 ← X_2 div 2
Y_3 ← 2 · Y_2
x_3 ← X_3 mod 2
// — Algorithmusstelle — i = 3
```

Abbildung 6.3: Ein einfacher Algorithmus für die Multiplikation einer kleinen Zahl a mit einer anderen Zahl b.

Machen wir eine Beispielrechnung für den Fall $a = 6$ und $b = 9$. Für die in Abbildung 6.3 mit dem Wort „Algorithmusstelle“ markierten Zeilen sind in der nachfolgenden Tabelle in jeweils einer Zeile die Werte angegeben, die dann die Variablen P_i, X_i, Y_i und x_i haben. Am Ende erhält man in P_3 den Wert 54. Das ist das Produkt der Eingabewerte 6 und 9.

	P_i	X_i	Y_i	x_i
$i = 0$	0	6	9	0
$i = 1$	0	3	18	1
$i = 2$	18	1	36	1
$i = 3$	54	0	72	0

Im folgenden wollen wir auf einen Beweis hinarbeiten, der zeigt, dass das immer so ist: Unter den Voraussetzungen, die zu Beginn zugesichert werden enthält nach Beendigung des Algorithmus das zuletzt ausgerechnete P_3 den Wert $P_3 = a \cdot b$. Wie geht das?

Da P_3 unter anderem mit Hilfe von P_2 ausgerechnet wird, ist es vermutlich hilfreich auch etwas darüber zu wissen; und über P_1 auch, usw. Analog sollte man am besten etwas über alle x_i wissen sowie über alle X_i und Y_i.

Angenommen, uns gelingt es, „etwas Passendes" hinzuschreiben, d. h. eine logische Formel, die eine Aussage $\mathcal{A}_i$ über die interessierenden Werte P_i, x_i, X_i und Y_i macht. Was dann? Sie ahnen es vermutlich: vollständige Induktion.

Induktionsbeweise sind am Anfang nicht immer ganz leicht. Aber hier stehen wir vor einem weiteren Problem: Wir müssen erst einmal Aussagen $\mathcal{A}_i$ finden, die wir erstens beweisen können, und die uns zweitens zum gewünschten Ziel führen. Passende Aussagen zu finden ist nicht immer ganz einfach und Übung ist sehr(!) hilfreich. Hinweise kann man aber oft durch Betrachten von Wertetabellen wie weiter vorne angegeben finden. Im vorliegenden Fall liefert ein bisschen Herumspielen irgendwann die Beobachtung, dass jedenfalls im Beispiel für jedes $i \in \mathbb{G}_4$ die folgende Aussage wahr ist:

$$\forall i \in \mathbb{G}_4 : X_i \cdot Y_i + P_i = a \cdot b$$

Das formen wir noch in eine Aussage für alle nichtnegativen ganzen Zahlen um:

$$\forall i \in \mathbb{N}_0 : i < 4 \Longrightarrow X_i \cdot Y_i + P_i = a \cdot b$$

Wir beweisen nun also durch vollständige Induktion die Formel $\forall i \in \mathbb{N}_0 : \mathcal{A}_i$, wobei $\mathcal{A}_i$ die Aussage ist:

$$i < 4 \Longrightarrow X_i \cdot Y_i + P_i = a \cdot b \; .$$

Induktionsanfang $i = 0$: Aufgrund der Initialisierungen der Variablen ist klar: $X_0 Y_0 + P_0 = ab + 0 = ab$.

Induktionsvoraussetzung: für ein beliebiges aber festes i gelte: $i < 4 \Longrightarrow X_i \cdot Y_i + P_i = a \cdot b$.

Induktionsschluss: zu zeigen ist nun:

$i + 1 < 4 \Longrightarrow X_{i+1} \cdot Y_{i+1} + P_{i+1} = a \cdot b$.

Sei also $i + 1 < 4$ (andernfalls ist die Implikation ohnehin wahr). Dann ist auch $i < 4$ und nach Induktionsvoraussetzung gilt: $X_i \cdot Y_i + P_i = a \cdot b$.

Wir rechnen nun:

$$\begin{aligned} X_{i+1} \cdot Y_{i+1} + P_{i+1} &= (X_i \ \mathbf{div}\ 2) \cdot 2Y_i + P_i + x_i Y_i \\ &= (X_i \ \mathbf{div}\ 2) \cdot 2Y_i + P_i + (X_i \ \mathbf{mod}\ 2) Y_i \\ &= (2(X_i \ \mathbf{div}\ 2) + (X_i \ \mathbf{mod}\ 2)) Y_i + P_i \\ &= X_i Y_i + P_i \\ &= ab\ . \end{aligned}$$

Dabei gelten die beiden ersten Gleichheiten wegen der Zuweisungen im Algorithmus, die vierte wegen Gleichung 6.6 und die letzte nach Induktionsvoraussetzung.

Wissen wir nun, dass am Ende des Algorithmus $P = ab$ ist? Offensichtlich noch nicht: Wir haben bisher nur bewiesen, dass nach der letzten Anweisung gilt: $P_3 + X_3 Y_3 = ab$. Der weiter vorne angegebenen Wertetabelle entnimmt man, dass jedenfalls in der Beispielrechnung am Ende $X_3 = 0$ gilt. Wenn es gelingt zu beweisen, dass auch das für *alle* Eingaben $a \in \mathbb{G}_8$ und $b \in \mathbb{N}_0$ gilt, dann sind wir fertig. Wie beweist man das? Wie Sie sich vielleicht schon bewusst gemacht haben, werden die X_i der Reihe nach immer kleiner. Und zwar immer um mindestens die Hälfte, denn $X_i \ \mathbf{div}\ 2 \leq X_i/2$. Mit anderen Worten:

$$\begin{aligned} X_0 &\leq a \\ X_1 &\leq X_0/2 \leq a/2 \\ X_2 &\leq X_1/2 \leq a/4 \\ &\vdots \end{aligned}$$

Ein Induktionsbeweis, der so einfach ist, dass wir ihn hier schon nicht mehr im Detail durchführen müssen, zeigt: $\forall i \in \mathbb{N}_0 : i < 4 \Longrightarrow X_i \leq a/2^i$. Insbesondere ist also $X_2 \leq a/4$. Nun ist aber nach Voraussetzung $a < 8$, und folglich $X_2 < 8/4 = 2$. Da X_2 eine ganze Zahl ist, ist $X_2 \leq 1$. Und daher ist das zuletzt berechnete $X_3 = X_2 \ \mathbf{div}\ 2 = 0$.

6.6 DER ALGORITHMUS ZUR MULTIPLIKATION NICHTNEGATIVER GANZER ZAHLEN MIT EINER SCHLEIFE

Nun schreiben wir als den Algorithmus etwas anders auf. Man kann nämlich folgendes beobachten: Wenn man erst einmal der Reihe nach x_{i+1}, P_{i+1}, X_{i+1} und Y_{i+1} berechnet hat, braucht man x_i, P_i, X_i und Y_i nicht mehr. Deswegen kann man die Indizes weglassen, und erhält die vereinfachte Darstellung aus Abbildung 6.4.

Schleife

Nun hat man dreimal genau den gleichen Algorithmustext hintereinander. Dafür führen wir als Abkürzung eine *Schleife* ein, genauer gesagt eine sogenannte **for**-Schleife. Generell schreiben wir das in der folgenden Struktur auf:

for ⟨*Schleifenvariable*⟩ ← ⟨*Startwert*⟩ **to** ⟨*Endwert*⟩ **do**
    ⟨*sogenannter Schleifenrumpf, der*⟩
    ⟨*aus mehreren Anweisungen bestehen darf*⟩
od

Die Bedeutung soll sein, dass der Schleifenrumpf nacheinander für jeden Wert der ⟨*Schleifenvariable*⟩ durchlaufen wird: als erstes für den ⟨*Startwert*⟩. Bei jedem weiteren Durchlauf werde die ⟨*Schleifenvariable*⟩ um 1 erhöht. Der letzte Durchlauf finde für den ⟨*Endwert*⟩ statt. Außerdem wollen wir vereinbaren, dass der Schleifenrumpf überhaupt nicht durchlaufen werde, falls der angegebene ⟨*Endwert*⟩ echt kleiner ist als der ⟨*Anfangswert*⟩.

Abbildung 6.5 zeigt, was sich in unserem Beispiel ergibt. Es ist ein besonders einfacher Fall, in dem der Schleifenrumpf gar nicht die Schleifenvariable (hier i) benutzt. Im allgemeinen soll das aber erlaubt sein.

Wir haben am Ende des Algorithmus noch die inzwischen bewiesene Zusicherung $P = ab$ notiert.

Interessant ist es nun, sich noch einmal klar zu machen, was nach dem Entfernen der Indizes aus den Aussagen $\mathcal{A}_i \equiv P_i + X_iY_i = ab$ wird: Alle sehen gleich aus: $P + XY = ab$. Inhaltlich war $\mathcal{A}_i$ aber eine Aussage darüber, was „an Algorithmusstelle i" gilt, also wie wir nun sagen können, nach i Schleifendurchläufen bzw. vor dem $i+1$-ten Schleifendurchlauf. Wir hatten bewiesen, dass aus der Gültigkeit von $\mathcal{A}_i$ die von $\mathcal{A}_{i+1}$ folgt. Nach dem Entfernen der Indizes sind aber $\mathcal{A}_i$ und $\mathcal{A}_{i+1}$ identisch. Wir haben also gezeigt:

> Wenn die Aussage $P + XY = ab$ vor dem einmaligen Durchlaufen des Schleifenrumpfes gilt, dann gilt sie auch hinterher wieder.

```
//   Eingaben: a ∈ G_8, b ∈ N_0
X ← a
Y ← b
P ← 0
x ← X mod 2
//   — Algorithmusstelle — i = 0
P ← P + x · Y
X ← X div 2
Y ← 2 · Y
x ← X mod 2
//   — Algorithmusstelle — i = 1
P ← P + x · Y
X ← X div 2
Y ← 2 · Y
x ← X mod 2
//   — Algorithmusstelle — i = 2
P ← P + x · Y
X ← X div 2
Y ← 2 · Y
x ← X mod 2
//   — Algorithmusstelle — i = 3
```

Abbildung 6.4: Vereinfachter Darstellung des Algorithmus aus Abbildung 6.3.

variante

Man sagt auch, dass diese Aussage eine *Schleifeninvariante* ist.

Der Induktionsanfang war nichts anderes als der Nachweis, dass die Schleifeninvariante vor dem ersten Betreten der Schleife stets gilt. Der Induktionsschritt war der Nachweis, dass die Wahrheit der Schleifeninvariante bei jedem Durchlauf erhalten bleibt.

Wenn also die Schleife jemals zu einem Ende kommt (und etwas anderes ist bei einer **for**-Schleife wie eben beschrieben gar nicht möglich), dann gilt die Schleifeninvariante auch zum Schluss.

```
// Eingaben: a ∈ G_8, b ∈ N_0
X ← a
Y ← b
P ← 0
x ← X mod 2
for i ← 0 to 2 do
    // — Algorithmusstelle — i
    P ← P + x · Y
    X ← X div 2
    Y ← 2 · Y
    x ← X mod 2
    // — Algorithmusstelle — i + 1
od
// Ergebnis: P = a · b
```

Abbildung 6.5: Multiplikationsalgorithmus mit einer Schleife für größenbeschränkten Faktor a.

Zum Abschluss wollen wir den Algorithmus nun noch so verallgemeinern, dass er für alle $a \in \mathbb{N}_0$ funktioniert und nicht nur für Zahlen kleiner als 8. Wenn man sich die obigen Beweise alle noch einmal ansieht, stellt man fest, dass die Bedingung $a < 8$ nur an einer Stelle verwendet wurde, nämlich beim Nachweis, dass $X_3 = 0$ ist.

Für einen Anfangswert von z. B. 4711 ist man natürlich nach drei Schleifendurchläufen noch nicht bei 0. Damit man für größere Anfangswerte „irgendwann“ bei 0 angelangt, muss man öfter den Wert X_i halbieren. Es ist also eine größere Anzahl n von Schleifendurchläufen notwendig. Wieviele es sind, sieht man an der schon besprochenen Ungleichung $X_i \leq a/2^i$. Man ist fertig, wenn vor dem letzten Durchlauf gilt: $X_{n-1} \leq 1$. Das ist sicher richtig, wenn $a/2^{n-1} \leq 1$ ist, also $a \leq 2^{n-1}$. Im Fall $a > 0$ gilt das, wenn $n - 1 \geq \log_2 a$. Im Fall $a = 0$ ist $a \leq 2^{n-1}$ auch immer wahr. Um diese Fallunterscheidung nicht immer aufschreiben zu müssen, vereinbaren wir: $\log_2 0 = 0$.

Insgesamt ergibt sich der in Abbildung 6.6 dargestellte Algorithmus.

```
// Eingaben: a ∈ ℕ0, b ∈ ℕ0
X ← a
Y ← b
P ← 0
x ← X mod 2
n ← 1 + ⌈log2 a⌉
for i ← 0 to n − 1 do
    P ← P + x · Y
    X ← X div 2
    Y ← 2 · Y
    x ← X mod 2
od
// Ergebnis: P = a · b
```

Abbildung 6.6: Algorithmus zu Multiplikation beliebiger nichtnegativer ganzer Zahlen.

7 DOKUMENTE

7.1 DOKUMENTE

Die Idee zu dieser Einheit geht auf eine Vorlesung „Informatik A“ von Till Tantau zurück (siehe http://www.tcs.mu-luebeck.de/lehre/2008-ws/info-a/wiki/Vorlesung, 6.11.2009).

Im alltäglichen Leben hat man mit vielerlei Inschriften zu tun: Briefe, Kochrezepte, Zeitungsartikel, Vorlesungsskripte, Seiten im WWW, Emails, und so weiter. Bei vielem, was wir gerade aufgezählt haben, und bei vielem anderen kann man drei verschiedene Aspekte unterscheiden, die für den Leser eine Rolle spielen, nämlich

- den *Inhalt* des Textes,
- seine *Struktur* und
- sein *Erscheinungsbild*, die (äußere) *Form*.

Inhalt

Struktur

Erscheinungsbild, Form

Stünde die obige Aufzählung so auf dem Papier:

- den INHALT des Textes,
- seine STRUKTUR und
- sein ERSCHEINUNGSBILD, die (äußere) FORM.

dann hätte man an der Form etwas geändert (Wörter in Großbuchstaben statt kursiv), aber nicht am Inhalt oder an der Struktur. Hätten wir dagegen geschrieben:

> „[...] den *Inhalt* des Textes, seine *Struktur* und sein *Erscheinungsbild*, die (äußere) *Form*.“

dann wäre die Struktur eine andere (keine Liste mehr, sondern Fließtext), aber der Inhalt immer noch der gleiche. Und stünde hier etwas über

- *Balaenoptera musculus* (Blauwal),
- *Mesoplodon carlhubbsi* (Hubbs-Schnabelwal) und
- *Physeter macrocephalus* (Pottwal),

dann wäre offensichtlich der Inhalt ein anderer.

Von Ausnahmen (welchen?) abgesehen, ist Autoren und Lesern üblicherweise vor allem am Inhalt gelegen. Die Wahl einer bestimmten Struktur und Form hat primär zum Ziel, den Leser beim Verstehen des Inhalts zu unterstützen. Mit dem Begriff *Dokumente* in der Überschrift sollen Texte gemeint sein, bei denen man diese drei Aspekte unterscheiden kann.

Dokumente

Wenn Sie später beginnen, erste nicht mehr ganz kleine Dokumente (z. B. Seminarausarbeitungen oder die Bachelor-Arbeit) selbst zu schreiben, werden Sie merken, dass die Struktur, oder genauer gesagt das Auffinden einer geeigneten Struktur, auch zu Ihrem, also des Autors, Verständnis beitragen kann. Deswegen ist es bei solchen Arbeiten unseres Erachtens sehr ratsam, *früh damit zu beginnen,*

ein Rat für Ihr weiteres Studium

etwas aufzuschreiben, weil man so gezwungen wird, über die Struktur nachzudenken.

Programme fallen übrigens auch in die Kategorie dessen, was wir hier Dokumenten nennen.

7.2 Struktur von Dokumenten

Auszeichnungssprache
markup language

Oft ist es so, dass es Einschränkungen bei der erlaubten Struktur von Dokumenten gibt. Als Beispiel wollen wir uns zwei sogenannte *Auszeichnungssprachen* (im Englischen *markup language*) ansehen. Genauer gesagt werden wir uns dafür interessieren, wie Listen in LaTeX aufgeschrieben werden müssen, und wie Tabellen in XHTML.

7.2.1 LaTeX

LaTeX, ausgesprochen „Latech", ist (etwas ungenau, aber für unsere Belange ausreichend formuliert) eine Erweiterung eines Textsatz-Programmes (iniTeX), das von Donald Knuth entwickelt wurde (http://www-cs-faculty.stanford.edu/~knuth/, 6.11.2009).

Es wird zum Beispiel in der Informatik sehr häufig für die Verfassung von wissenschaftlichen Arbeiten verwendet, weil unter anderem der Textsatz mathematischer Formeln durch TeX von hervorragender Qualität ist, ohne dass man dafür viel tun muss. Sie ist deutlich besser als alles, was der Autor dieser Zeilen jemals in Dokumenten gesehen hat, die mit Programmen wie z. B. OpenOffice.org o. ä. verfasst wurden. Zum Beispiel liefert der Text

```
\[ 2 - \sum_{i=0}^{k} i 2^{-i} = (k+2) 2^{-k} \]
```

die Ausgabe

$$2 - \sum_{i=0}^{k} i 2^{-i} = (k+2) 2^{-k}$$

Auch der vorliegende Text wurde mit LaTeX gemacht. Für den Anfang dieses Abschnittes wurde z. B. geschrieben:

```
\section{Struktur von Dokumenten}
```

woraus LaTeX die Zeile

7.2 Struktur von Dokumenten

am Anfang dieser Seite gemacht hat. Vor dem Text der Überschrift wurde also automatisch die passende Nummer eingefügt und der Text wurde in einer Kapitälchenschrift gesetzt. Man beachte, dass z. B. die Auswahl der Schrift *nicht* in der

Eingabe mit vermerkt ist. Diese Angabe findet sich an anderer Stelle, und zwar an *einer* Stelle, an der das typografische Aussehen *aller* Abschnittsüberschriften (einheitlich) festgelegt ist.

Ganz grob kann ein LaTeX-Dokument z. B. die folgende Struktur haben:

```
\documentclass[11pt]{report}
% so schreibt man Kommentare
% dieser Teil heißt Präambel des Dokumentes
% Unterstützung für Deutsch,
%    z.B. richtige automatische Trennung
\usepackage[german]{babel}
% Angabe des Zeichensatzes, in dem der Text ist
\usepackage[latin1]{inputenc}
% für das Einbinden von Grafiken
\usepackage{graphicx}
\begin{document}
  % und hier kommt der eigentliche Text .....
\end{document}
```

Eine Liste einfacher Punkte sieht in LaTeX so aus:

Eingabe	Ausgabe
`\begin{itemize}`	
`\item Inhalt`	• Inhalt
`\item Struktur`	• Struktur
`\item Form`	• Form
`\end{itemize}`	

Vor den aufgezählten Wörtern steht jeweils ein dicker Punkt. Auch dieser Aspekt der äußeren Form ist *nicht* dort festgelegt, wo die Liste steht, sondern an *einer* Stelle, an der das typografische Aussehen *aller* solcher Listen (einheitlich) festgelegt ist. Wenn man in seinen Listen lieber alle Aufzählungspunkte mit einem sogenannten Spiegelstrich „–“ beginnen lassen möchte, dann muss man nur an einer Stelle (in der Präambel) die Definition von `\item` ändern.

Wollte man die formale Sprache L_{itemize} aller legalen Texte für Listen in LaTeX aufschreiben, dann könnte man z. B. zunächst geeignet die formale Sprache L_{item} aller Texte spezifizieren, die hinter einem Aufzählungspunkt vorkommen dürfen. Dann wäre

$L_{\text{itemize}} = \{\texttt{\textbackslash begin\{itemize\}}\} \left(\{\texttt{\textbackslash item}\} L_{\text{item}}\right)^+ \{\texttt{\textbackslash end\{itemize\}}\}$

Dabei haben wir jetzt vereinfachend so getan, als wäre es kein Problem, L_{item} zu definieren. Tatsächlich ist das aber zumindest auf den ersten Blick eines, denn

ein Aufzählungspunkt in LaTeX darf seinerseits wieder eine Liste enthalten. Bei naivem Vorgehen würde man also genau umgekehrt für die Definition von L_{item} auch auf L_{itemize} zurückgreifen (wollen). Wir diskutieren das ein kleines bisschen ausführlicher in Unterabschnitt 7.2.3.

7.2.2 HTML und XHTML

HTML ist die Auszeichnungssprache, die man benutzt, wenn man eine WWW-Seite (be)schreibt. Für HTML ist formaler als für LaTeX festgelegt, wie syntaktisch korrekte solche Seiten aussehen. Das geschieht in einer sogenannten *document type definition*, kurz *DTD*.

Hier ist ein Auszug aus der DTD für eine Version von XHTML. Sie dürfen sich vereinfachend vorstellen, dass das ist im wesentlichen eine noch striktere Variante von HTML ist. Das nachfolgenden Fragment beschreibt teilweise, wie man syntaktisch korrekt eine Tabelle notiert.

```
<!ELEMENT table (caption?, thead?, tfoot?, (tbody+|tr+))>
<!ELEMENT caption  %Inline;>
<!ELEMENT thead    (tr)+>
<!ELEMENT tfoot    (tr)+>
<!ELEMENT tbody    (tr)+>
<!ELEMENT tr       (th|td)+>
<!ELEMENT th       %Flow;>
<!ELEMENT td       %Flow;>
```

Wir können hier natürlich nicht auf Details eingehen. Einige einfache aber wichtige Aspekte sind aber mit unserem Wissen schon verstehbar. Die Wörter wie `table`, `thead`, `tr`, usw. dürfen wir als bequem notierte Namen für formale Sprachen auffassen. Welche, das wollen wir nun klären.

Die Bedeutung von `*` und `+` ist genau das, was wir als Konkatenationsabschluss und ε-freien Konkatenationsabschluss kennen gelernt haben. Die Bedeutung des Kommas `,` in der ersten Zeile ist die des Produktes formaler Sprachen. Die Bedeutung des senkrechten Striches `|` in der sechsten Zeile ist Vereinigung von Mengen.

Das Fragezeichen ist uns neu, hat aber eine ganz banale Bedeutung: In der uns geläufigen Notation würden wir definieren: $L^? = L^0 \cup L^1 = \{\varepsilon\} \cup L$. Mit anderen Worten: Wenn irgendwo $L^?$ notiert ist, dann kann an dieser Stelle ein Wort aus L stehen, oder es kann fehlen. Das Auftreten eines Wortes aus L ist also sozusagen optional.

Nun können Sie zur Kenntnis nehmen, dass z. B. die Schreibweise

```
<!ELEMENT tbody (tr)+ >
```

die folgende formale Sprache festlegt:

$$L_{\texttt{tbody}} = \{\texttt{<tbody>}\} \cdot L_{\texttt{tr}}^{+} \cdot \{\texttt{</tbody>}\}$$

Das heißt, ein Tabellenrumpf (*table body*) beginnt mit der Zeichenfolge <tbody>, endet mit der Zeichenfolge </tbody>, und enthält dazwischen eine beliebige positive Anzahl von Tabellenzeilen (*table rows*). Und die erste Zeile aus der DTD besagt

$$L_{\texttt{table}} = \{\texttt{<table>}\} \cdot L_{\texttt{caption}}^{?} \cdot L_{\texttt{thead}}^{?} \cdot L_{\texttt{tfoot}}^{?} \cdot L_{\texttt{tbody}}^{+} \cdot \{\texttt{</table>}\}$$

das heißt, eine Tabelle (*table*) ist von den Zeichenketten <table> und </table> umschlossen und enthält innerhalb in dieser Reihenfolge

- optional eine Überschrift (*caption*),
- optional einen Tabellenkopf (*table head*),
- optional einen Tabellenfuß (*table foot*) und
- eine beliebige positive Anzahl von Tabellenrümpfen (siehe eben).

Insgesamt ergibt sich, dass zum Beispiel

```
<table>
  <tbody>
    <tr>  <td>1</td> <td>a</td>  </tr>
    <tr>  <td>2</td> <td>b</td>  </tr>
  </tbody>
</table>
```

eine syntaktisch korrekte Tabelle.

In Wirklichkeit gibt es noch zusätzliche Aspekte, die das Ganze formal verkomplizieren und bei der Benutzung flexibler machen, aber das ganz Wesentliche haben wir damit jedenfalls an einem Beispiel beleuchtet.

7.2.3 Eine Grenze unserer bisherigen Vorgehensweise

Dass wir eben mit Hilfe von Produkt und Konkatenationsabschluss formaler Sprachen in einigen präzise Aussagen machen konnten, hing auch der Einfachheit dessen, was es zu spezifizieren galt. Es wurde, jedenfalls in einem intuitiven Sinne, immer etwas von einer komplizierteren Art aus Bestandteilen einfacherer Art zusammengesetzt.

Es gibt aber auch den Fall, dass man sozusagen größere Dinge einer Art aus kleineren Bestandteilen zusammensetzen will, die aber von der gleichen Art sind. Auf Listen, deren Aufzählungspunkte ihrerseits wieder Listen enthalten dürfen, hatten wir im Zusammenhang mit LaTeX schon hingewiesen.

Ein anderes typisches Beispiel sind korrekt geklammerte arithmetische Ausdrücke. Sehen wir einmal von den Operanden und Operatoren ab und konzentrieren uns auf die reinen Klammerungen. Bei einer syntaktisch korrekten Klammerung gibt es zu jeder Klammer auf „weiter hinten" die „zugehörige" Klammer zu. Insbesondere gilt:

- Man kann beliebig viele korrekte Klammerungen konkatenieren und erhält wieder eine korrekte Klammerung.
- Man kann um eine korrekte Klammerung außen herum noch ein Klammerpaar schreiben (Klammer auf ganz vorne, Klammer zu ganz hinten) und erhält wieder eine korrekte Klammerung.

Man würde also gerne in irgendeinem Sinne L_{Klammer} mit L_{Klammer}^* und mit $\{\mathtt{(}\} \cdot L_{\text{Klammer}} \cdot \{\mathtt{)}\}$ in Beziehung setzen. Es ist aber nicht klar wie. Insbesondere würden bei dem Versuch, eine Art Gleichung hinzuschreiben, sofort die Fragen im Raum stehen, ob die Gleichung überhaupt lösbar ist, und wenn ja, ob die Lösung eindeutig ist.

7.3 ZUSAMMENFASSUNG

In dieser Einheit haben wir über *Dokumente* gesprochen. Sie haben einen *Inhalt*, eine *Struktur* und ein *Erscheinungsbild*.

Formale Sprachen kann man z. B. benutzen, um zu spezifizieren, welche Struktur(en) ein legales, d. h. syntaktisch korrektes Dokument haben darf, sofern die Strukturen hinreichen einfach sind. Was man in komplizierten Fällen machen kann, werden wir in einer späteren Einheit kennenlernen.

8 KONTEXTFREIE GRAMMATIKEN

8.1 REKURSIVE DEFINITION SYNTAKTISCHER STRUKTUREN

Wir hatten in Einheit 7 über Dokumente schon darauf hingewiesen, dass die Beschreibung formaler Sprachen nur mit Hilfe einzelner Symbole und der Operation Vereinigung, Konkatenation und Konkatenationsabschluss manchmal möglich ist, aber manchmal anscheinend auch nicht.

Tatsächlich ist es manchmal unmöglich. Wie man zu einen harten Beweis für diese Aussage kommen kann, werden wir in einer späteren Einheit sehen.

Als typisches Beispiel eines Problemfalles sehen wir uns einen Ausschnitt der Definition der Syntax von Java an (siehe die Seite über Anweisungen unter `http://java.sun.com/docs/books/jls/third_edition/html/statements.html`, 13.11.08). Dort steht im Zusammenhang mit der Festlegung, was in Java eine Anweisung sei, unter anderem folgende fünf Teile:

1	Block:
	`{` BlockStatements$_{\text{opt}}$ `}`
2	BlockStatements:
	BlockStatement
	BlockStatements BlockStatement
3	BlockStatement:
	Statement
	......
4	Statement:
	StatementWithoutTrailingSubstatement
	......
5	StatementWithoutTrailingSubstatement:
	Block
	......

Tabelle 8.1: Auszug aus der Spezifikation der Syntax von Java

Wir werden die Wörter aus obiger Spezifikation von normalem Text unterscheiden, indem wir sie in spitzen Klammern und kursiv setzen, wie z. B. bei ⟨*Block*⟩, um das Lesen zu vereinfachen (und um schon mal deutlich zu machen, dass es sich z. B. bei ⟨*Block*⟩ um etwas handelt, was wir als *ein* Ding ansehen wollen).

Man macht keinen Fehler, wenn man das zum Beispiel erst mal so liest:

1. Ein ⟨*Block*⟩ hat die Struktur: Ein Zeichen `{`, eventuell gefolgt von ⟨*BlockStatements*⟩ (das tiefgestellte Suffix „opt“ besagt, dass das optional ist), gefolgt von einem Zeichen `}`.
2. ⟨*BlockStatements*⟩ sind von einer der Strukturen
   - ein einzelnes ⟨*BlockStatement*⟩ oder
   - ⟨*BlockStatements*⟩ gefolgt von einem ⟨*BlockStatement*⟩
3. Ein ⟨*BlockStatement*⟩ kann ein ⟨*Statement*⟩ sein (oder anderes ...).
4. Ein ⟨*Statement*⟩ kann ein ⟨*StatementWithoutTrailingSubstatement*⟩ sein (oder anderes, zum Beispiel etwas „ganz einfaches“ ...).
5. Ein ⟨*StatementWithoutTrailingSubstatement*⟩ kann ein ⟨*Block*⟩ sein (oder anderes ...).

Diese Formulierungen beinhalten Rekursion: Bei der Beschreibung der Struktur von ⟨*BlockStatements*⟩ wird direkt auf ⟨*BlockStatements*⟩ Bezug genommen.

Außerdem wird bei der Definition von ⟨*Block*⟩ (indirekt) auf ⟨*Statement*⟩ verwiesen und bei der Definition von ⟨*Statement*⟩ (indirekt) wieder auf ⟨*Block*⟩.

Wir werden uns der Antwort auf die Frage, wie man diese Rekursionen eine vernünftige Bedeutung zuordnen kann, in mehreren Schritten nähern.

Als erstes schälen wir den für uns gerade wesentlichen Aspekt noch besser heraus. Schreiben wir einfach X statt ⟨*Block*⟩, ⟨*Statement*⟩ o.ä., und schreiben wir lieber runde Klammern `(` und `)` statt der geschweiften, weil wir die die ganze Zeit als Mengenklammern benutzen wollen. (Sie fragen sich, warum nicht einen Ausschnitt aus der Grammatik für arithmetische Ausdrücke genommen haben, in denen sowieso schon runde Klammern benutzt werden? Die Antwort ist: Der Ausschnitt aus der Syntaxbeschreibung wäre viel länger und unübersichtlicher geworden.)

Dann besagt die Definition unter anderem:

K1 Ein X kann etwas “ganz einfaches“ sein. Im aktuellen Zusammenhang abstrahieren wir mal ganz stark und schreiben für dieses Einfache einfach das leere Wort ε.

K2 Ein X kann ein Y sein oder auch ein X gefolgt von einem Y; also kann ein X von der Form YY sein. Jedes Y seinerseits kann wieder ein X sein. Also kann ein X auch von der Form XX sein.

K3 Wenn man ein X hat, dann ist auch (X) wieder ein X.

K4 Schließlich tun wir so, als dürfe man in die Definition auch hineininterpretieren: Es ist nichts ein X, was man nicht auf Grund der obigen Festlegungen als solches identifizieren kann.

Und nun? Wir könnten jetzt zum Beispiel versuchen, mit X eine formale Sprache L zu assoziieren, und folgende Gleichung hinschreiben:

$$L = \{\varepsilon\} \cup LL \cup \{(\}L\{)\} \tag{8.1}$$

Dabei wäre die Hoffnung, dass die Inklusion $L \supseteq \dots$ die ersten drei aufgezählten Punkte widerspiegelt, und die Inklusion $L \subseteq \dots$ den letzten Punkt. Wir werden sehen, dass diese Hoffnung zum Teil leider trügt.

Die entscheidenden Fragen sind nun natürlich:

1. Gibt es überhaupt eine formale Sprache, die Gleichung 8.1 erfüllt?

   Das hätten wir gerne, und wir werden sehen, dass es tatsächlich so ist, indem wir eine Lösung der Gleichung konstruieren werden.
2. Und falls ja: Ist die formale Sprache, die Gleichung 8.1 erfüllt, nur durch die Gleichung eindeutig festgelegt?

   Das hätten wir auch gerne, wir werden aber sehen, dass das *nicht* so ist. Das bedeutet für uns die zusätzliche Arbeit, dass wir „irgendwie" eine der Lösungen als die uns interessierende herausfinden und charakterisieren müssen. Das werden wir im nächsten Unterabschnitt machen.

Zum Abschluss dieses Unterabschnittes wollen wir eine Lösung von Gleichung 8.1 konstruieren. Wie könnte man das tun? Wir „tasten uns an eine Lösung heran".

Das machen wir, indem wir

- erstens eine ganze Folge $L_0, L_1, \dots$ formaler Sprachen L_i für $i \in \mathbb{N}_0$ definieren, der Art, dass jedes L_i offensichtlich jedenfalls einige der gewünschten Wörter über dem Alphabet $\{(,)\}$ enthält, und
- zweitens dann zeigen, dass die Vereinigung aller dieser L_i eine Lösung von Gleichung 8.1 ist.

Also:

- Der Anfang ist ganz leicht: $L_0 = \{\varepsilon\}$.
- und weiter machen kann man so: für $i \in \mathbb{N}_0$ sei

  $L_{i+1} = L_i L_i \cup \{(\}L_i\{)\}$.

Wir behaupten, dass $L = \bigcup_{i=0}^{\infty} L_i$ Gleichung 8.1 erfüllt.

Zum Beweis der Gleichheit überzeugen wir uns davon, dass beide Inklusionen erfüllt sind.

Zunächst halten wir fest: $\varepsilon \in L_0$. Außerdem ist für alle $i \in \mathbb{N}_0$ auch $L_i L_i \subseteq L_{i+1}$, wenn also $\varepsilon \in L_i$, dann auch $\varepsilon = \varepsilon\varepsilon \in L_i L_i$, also $\varepsilon \in L_{i+1}$. Also gilt für alle $i \in \mathbb{N}_0$: $\varepsilon \in L_i$. Und folglich gilt für alle $i \in \mathbb{N}_0$: $L_i = L_i\{\varepsilon\} \subseteq L_i L_i$, also ist für alle $i \in \mathbb{N}_0$: $L_i \subseteq L_{i+1}$.

$L \subseteq \{\varepsilon\} \cup LL \cup \{(\}L\{)\}$: Da $\varepsilon \in L_0 \subseteq L$ ist, ist $L = L\{\varepsilon\} \subseteq LL$.

$L \supseteq \{\varepsilon\} \cup LL \cup \{(\}L\{)\}$: sei $w \in \{\varepsilon\} \cup LL \cup \{(\}L\{)\}$.

1. Fall: $w = \varepsilon$: $w = \varepsilon \in L_0 \subseteq L$.

2. Fall: $w \in LL$: Dann ist $w = w_1 w_2$ mit $w_1 \in L$ und $w_2 \in L$. Es gibt also Indizes i_1 und i_2 mit $w_1 \in L_{i_1}$ und $w_2 \in L_{i_2}$. Für $i = \max(i_1, i_2)$ ist also $w_1 \in L_i$ und $w_2 \in L_i$, also $w = w_1 w_2 \in L_i L_i \subseteq L_{i+1} \subseteq L$.

3. Fall: $w \in \{(\}L\{)\}$: Für ein $i \in \mathbb{N}_0$ ist dann $w \in \{(\}L_i\{)\} \subseteq L_{i+1} \subseteq L$.

Damit haben wir gezeigt, dass Gleichung 8.1 (mindestens) eine Lösung hat.

Um sehen, dass die konstruierte Lösung keineswegs die einzige ist, muss man sich nur klar machen, dass $\{(,)\}^*$ auch eine Lösung der Gleichung ist, aber eine andere.

- Dass es sich um eine Lösung handelt, sieht man so: „$\subseteq$" zeigt man wie oben; „$\supseteq$" ist trivial, da $\{(,)\}^*$ eben *alle* Wörter sind.
- Dass es eine andere Lösung ist, sieht man daran, dass z. B. das Wort (((zwar in $\{(,)\}^*$ liegt, aber *nicht* in der oben konstruierten Lösung. Die enthält nämlich jedenfalls nur Wörter, in denen gleich viele (und) vorkommen. (Diese Behauptung gilt nämlich für alle L_i, wie man durch vollständige Induktion zeigen kann.)

Kommen wir zurück zu unserer zuerst konstruierten Lösung $L = \bigcup_{i=0}^{\infty} L_i$. Zählen wir für die ersten vier L_i explizit auf, welche Wörter jeweils neu hinzukommen:

$$\begin{aligned}
L_0 &= \{\varepsilon\} \\
L_1 \setminus L_0 &= \{\ ()\ \} \\
L_2 \setminus L_1 &= \{\ ()()\ ,\ (())\ \} \\
L_3 \setminus L_2 &= \{\ ()()()\ ,\ (())()\ ,\ ()(())\ , \\
&\qquad ()()()()\ ,\ ()()(())\ ,\ (())()()\ ,\ (())(())\ , \\
&\qquad (()())\ ,\ ((()))\ \}
\end{aligned}$$

Dabei gilt zum Beispiel:

- Die Erklärung für $(())()() \in L_3$ ist, dass $(()) \in L_2$ und $()() \in L_2$ und Regel K2.
  - Die Erklärung für $(()) \in L_2$ ist, dass $() \in L_1$ und Regel K3.
    - Die Erklärung für $() \in L_1$ ist, dass $\varepsilon \in L_0$ und Regel K3.
  - Die Erklärung für $()() \in L_2$ ist, dass $() \in L_1$ und $() \in L_1$ ist und Regel K2
    - Die Erklärung für $() \in L_1$ ist, dass $\varepsilon \in L_0$ und Regel K3.
    - Die Erklärung für $() \in L_1$ ist, dass $\varepsilon \in L_0$ und Regel K3.

Das kann man auch in graphischer Form darstellen: siehe Abbildung 8.1. Ersetzt man dann noch überall die $w \in L_i$ durch „X", ergibt sich Abbildung 8.2.

Eine solche Darstellung nennt man in der Informatik einen Baum (kurioserweise wird die „Wurzel" üblicherweise oben dargestellt und die „Blätter" unten). Auf Bäume und andere Graphen als Untersuchungsgegenstand werden wir in einer späteren Einheit zu sprechen kommen. Vorläufig genügt es uns, dass wir solche Bilder malen können.

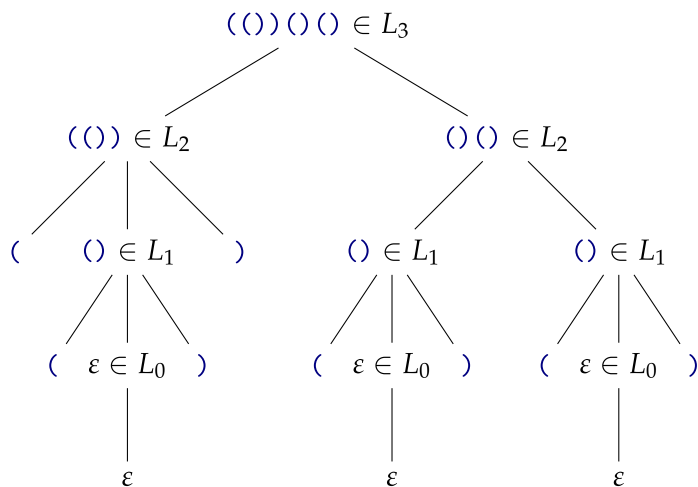


Abbildung 8.1: Eine „Begründung“, warum $(())()() \in L_3$ ist.

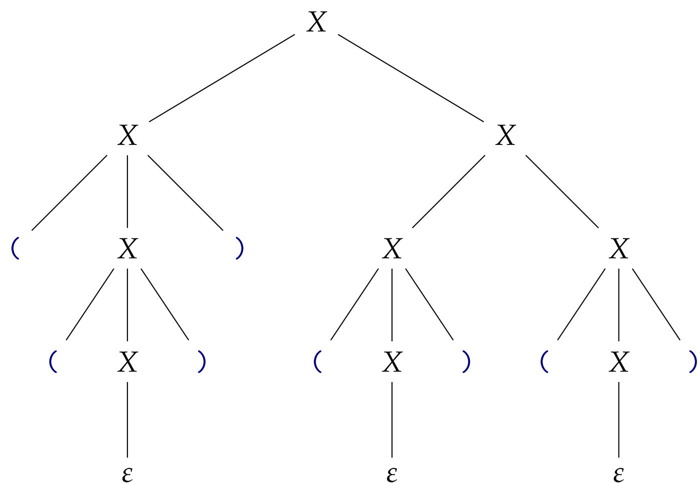


Abbildung 8.2: Eine abstraktere Darstellung der „Begründung“, warum $(())()() \in L_3$ ist.

Zu kontextfreien Grammatiken ist es nun nur noch ein kleiner Schritt: Statt wie eben im Baum Verzweigungen von unten nach oben zu als Rechtfertigungen zu interpretieren, geht man umgekehrt vor und interpretiert Verzweigungen als Stellen, an denen man „Verfeinerungen“ vornimmt, indem man rein syntaktisch zum Beispiel ein X *ersetzt* durch (X) oder XX.

8.2 kontextfreie grammatiken

kontextfreie Grammatik

Eine *kontextfreie Grammatik* $G = (N, T, S, P)$ ist durch vier Bestandteile gekennzeichnet, die folgende Bedeutung haben:

Nichtterminalsymbole

- N ist ein Alphabet, dessen Elemente *Nichtterminalsymbole* heißen.

Terminalsymbole

- T ist ein Alphabet, dessen Elemente *Terminalsymbole* heißen. Kein Zeichen darf in beiden Alphabeten vorkommen; es muss stets $N \cap T = \emptyset$ sein.

Startsymbol

- $S \in N$ ist das sogenannte *Startsymbol* (also ein Nichtterminalsymbol).

Produktionen

- $P \subseteq N \times V^*$ ist eine *endliche* Menge sogenannter *Produktionen*. Dabei ist $V = N \cup T$ die Menge aller Symbole überhaupt.

$X \to w$

Gilt für ein Nichtterminalsymbol X und ein Wort w, dass $(X, w) \in P$ ist, dann schreibt man diese Produktion üblicherweise in der Form $X \to w$ auf. Eine solche Produktion besagt, dass man ein Vorkommen des Zeichens X durch das Wort w ersetzen darf, und zwar „egal wo es steht", d. h. ohne auf den Kontext zu achten.

Ableitungsschritt

$u \Rightarrow v$

Das ist dann das, was man einen *Ableitungsschritt* gemäß einer Grammatik nennt. Formal definiert man das so: Aus einem Wort $u \in V^*$ ist in einem Schritt ein Wort $v \in V^*$ ableitbar, in Zeichen $u \Rightarrow v$, wenn es Wörter $w_1, w_2 \in V^*$ und eine Produktion $X \to w$ in P gibt, so dass $u = w_1 X w_2$ und $v = w_1 w w_2$.

Betrachten wir als Beispiel die Grammatik $G = (\{X\}, \{\mathtt{a}, \mathtt{b}\}, X, P)$ mit der Produktionenmenge $P = \{X \to \varepsilon, X \to \mathtt{a}X\mathtt{b}\}$. Dann gilt zum Beispiel $\mathtt{aba}X\mathtt{ba}XXXX \Rightarrow \mathtt{aba}X\mathtt{baa}X\mathtt{b}XXX$, wie man an der folgenden Aufteilung sieht:

$$\underbrace{\mathtt{aba}X\mathtt{ba}}_{w_1} X \underbrace{XXX}_{w_2} \Rightarrow \underbrace{\mathtt{aba}X\mathtt{ba}}_{w_1} \mathtt{a}X\mathtt{b} \underbrace{XXX}_{w_2}$$

Ebenso gilt $\mathtt{aba}X\mathtt{ba}XXXX \Rightarrow \mathtt{abaa}X\mathtt{bba}XXXX$:

$$\underbrace{\mathtt{aba}}_{w_1} X \underbrace{\mathtt{ba}XXXX}_{w_2} \Rightarrow \underbrace{\mathtt{aba}}_{w_1} \mathtt{a}X\mathtt{b} \underbrace{\mathtt{ba}XXXX}_{w_2}$$

Man kann also unter Umständen aus dem gleichen Wort verschiedene Wörter in einem Schritt ableiten.

Infixschreibweise

Die Definition von $\Rightarrow$ legt eine Relation zwischen Wörtern über dem Alphabet $V = N \cup T$ fest. Man könnte also auch schreiben: $R_\Rightarrow \subseteq V^* \times V^*$ oder gar $\Rightarrow \subseteq V^* \times V^*$. In diesem Fall, wie z. B. auch bei der $\leq$-Relation, ist es aber so, dass man die sogenannte *Infixschreibweise* bevorzugt: Man schreibt $5 \leq 7$ und nicht $(5, 7) \in R_\leq$ o. ä. Im allgemeinen ist $\Rightarrow$ weder links- noch rechtstotal und weder links- noch rechtseindeutig.

Ableitung

Da die linke Seite einer Produktion immer ein Nichtterminalsymbol ist, wird bei einem Ableitungsschritt nie ein Terminalsymbol ersetzt. Wo sie stehen, ist „die Ableitung zu Ende" (daher der Name *Terminal*symbol). Eine *Ableitung* (oder auch

Ableitungsfolge) ist eine Folge von Ableitungsschritten, deren Anzahl irrelevant ist. Die folgenden Definitionen kommen Ihnen vermutlich schon vertrauter vor als vor einigen Wochen. Für alle $u, v \in V^*$ gelte

$$u \Rightarrow^0 v \text{ genau dann, wenn } u = v$$
$$\forall i \in \mathbb{N}_0 : (u \Rightarrow^{i+1} v \text{ genau dann, wenn } \exists w \in V^* : u \Rightarrow w \Rightarrow^i v)$$
$$u \Rightarrow^* v \text{ genau dann, wenn } \exists i \in \mathbb{N}_0 : u \Rightarrow^i v$$

Bei unserer Beispielgrammatik gilt zum Beispiel

$$X \Rightarrow \mathtt{a}X\mathtt{b} \Rightarrow \mathtt{aa}X\mathtt{bb} \Rightarrow \mathtt{aaa}X\mathtt{bbb} \Rightarrow \mathtt{aaabbb}$$

Also hat man z. B. die Beziehungen $X \Rightarrow^* \mathtt{aa}X\mathtt{bb}$, $\mathtt{a}X\mathtt{b} \Rightarrow^* \mathtt{aaa}X\mathtt{bbb}$, $X \Rightarrow^* \mathtt{aaabbb}$ und viele andere. Weiter geht es in obiger Ableitung nicht, weil überhaupt keine Nichtterminalsymbole mehr vorhanden sind.

Sozusagen das Hauptinteresse bei einer Grammatik besteht in der Frage, welche Wörter aus Terminalsymbolen aus dem Startsymbol abgeleitet werden können. Ist $G = (N, T, S, P)$, so ist die *von einer Grammatik erzeugte formale Sprache*

von einer Grammatik erzeugte formale Sprache

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\} .$$

Eine formale Sprache, die man mit einer kontextfreien Grammatik erzeugen kann, heißt auch *kontextfreie Sprache*.

kontextfreie Sprache

Was ist bei unserer Beispielgrammatik $G = (\{X\}, \{\mathtt{a}, \mathtt{b}\}, X, \{X \to \varepsilon, X \to \mathtt{a}X\mathtt{b}\}$? Eben haben wir gesehen: $\mathtt{aaabbb} \in L(G)$. Die Ableitung ist so strukturiert, dass man als Verallgemeinerung leicht sieht, dass für alle $i \in \mathbb{N}_0$ gilt: $\mathtt{a}^i\mathtt{b}^i \in L(G)$. Der Beweis (Sie wissen schon wie ...) wird leichter, wenn man mit der allgemeineren Behauptung beginnt:

$$\forall i \in \mathbb{N}_0 : (X \Rightarrow^* \mathtt{a}^i\mathtt{b}^i \land X \Rightarrow^* \mathtt{a}^i X \mathtt{b}^i)$$

Daraus folgt, dass $\{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}_0\} \subseteq L(G)$ ist. Umgekehrt kann man zeigen, dass gilt:

$$\forall i \in \mathbb{N}_0 : \text{ wenn } X \Rightarrow^{i+1} w, \text{ dann } w = \mathtt{a}^i\mathtt{b}^i \lor w = \mathtt{a}^{i+1} X \mathtt{b}^{i+1}$$

Daraus folgt $L(G) \subseteq \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}_0\}$. Insgesamt ergibt sich also:

$$L(G) = \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}_0\} .$$

Um eine etwas kompaktere Notation zu haben, fasst man manchmal mehrere Produktionen mit gleicher linker Seite zusammen, indem man das Nichtterminalsymbol und den Pfeil nur einmal hinschreibt, und die rechten Seiten durch senkrechte

© worsch 2008–2010

Striche getrennt aufführt. Für unsere Beispielgrammatik sieht die Produktionenmenge dann so aus:

$$P = \{ X \to \mathtt{a}X\mathtt{b} \mid \varepsilon \}$$

Und damit können wir nun auch die richtige Interpretation für die Fragmente aus Tabelle 8.1 angeben: Es handelt sich um Produktionen einer kontextfreien Grammatik.

- Jedes der Wörter „Block“ ist *ein* Nichtterminalsymbol. Deswegen haben wir Schreibweisen wie $\langle Block \rangle$ vorgezogen, um deutlicher zu machen, dass wir so etwas als *ein* nicht weiter zerlegbares Objekt betrachten wollen.
- Der Doppelpunkt entspricht unserem Pfeil $\to$ und trennt linke Seite einer Produktion von rechten Seiten.
- Jede eingerückte Zeile ist eine rechte Seite einer Produktion.
- Aufeinander folgende Zeilen muss man sich durch senkrechte Striche $\mid$ getrennt denken.

Der zweite Teil

2	BlockStatements:	
		BlockStatement
		BlockStatements BlockStatement

der Tabelle aus dem ersten Unterabschnitt repräsentiert also z. B. die Produktionen

$$\begin{aligned} \langle BlockStatements \rangle &\to \langle BlockStatement \rangle \\ &\quad \mid \langle BlockStatements \rangle \langle BlockStatement \rangle \end{aligned}$$

Und der erste Teil

1	Block:	
		{ BlockStatements$_{opt}$ }

der Tabelle repräsentiert die Produktionen

$$\langle Block \rangle \to \mathtt{\{} \ \langle BlockStatements \rangle \ \mathtt{\}} \mid \mathtt{\{} \ \mathtt{\}}$$

Wie man sieht, stehen (jedenfalls manche) Nichtterminalsymbole für strukturelle Konzepte der Programmiersprache. Im Idealfall wäre es dann so, dass man mit der kontextfreie Grammatik für Java alle syntaktisch korrekten Javaprogramme ableiten kann, aber auch nur diese und nichts anderes. Die Realität ist etwas

komplizierter: Was man mit der kontextfreie Grammatik nicht ableiten kann, ist bestimmt kein Javaprogramm. Aber man kann Dinge ableiten, die keine korrekten Programme sind (weil man manche Forderungen, wie z. B. „alle vorkommenden Variablen müssen vorher deklariert worden sein", überhaupt nicht mit Hilfe kontextfreier Grammatiken ausdrücken kann).

Wenn man sich davon überzeugen will, dass ein Wort w aus dem Startsymbol ableitbar ist, ist das Aufschreiben einer langen Ableitungsfolge ist manchmal sehr mühsam aber nicht sonderlich erhellend. Die Grammatik für unser Klammerproblem ist ja sehr übersichtlich: $(\{X\}, \{(,)\}, X, \{X \to XX|(X)|\varepsilon\})$. Aber

$$X \Rightarrow XX \Rightarrow (X)X \Rightarrow (X)XX \Rightarrow (X)X(X) \Rightarrow ((X))X(X)$$
$$\Rightarrow ((X))X() \Rightarrow ((X))(X)() \Rightarrow (())(X)() \Rightarrow (())()()$$

findet jedenfalls der Autor dieser Zeilen wenig hilfreich. Das kommt natürlich zum Teil daher, dass hier mal vorne und mal weiter hinten ein Ableitungsschritt gemacht wurde. Aber da bei kontextfreien Grammatiken eben Ersetzungen vom Kontext unabhängig sind, kann man umsortieren und zum Beispiel immer möglichst weit links abzuleiten. Dann erhält man eine sogenannte *Linksableitung*; in unserem Beispiel:

$$X \Rightarrow XX \Rightarrow (X)X \Rightarrow ((X))X \Rightarrow (())X \Rightarrow (())XX$$
$$\Rightarrow (())(X)X \Rightarrow (())()X \Rightarrow (())()(X) \Rightarrow (())()()$$

Noch nützlicher ist es manchmal, stattdessen den zu dieser Ableitung gehörenden sogenannten *Ableitungsbaum* aufzumalen. Im vorliegenden Fall erhält man gerade die Darstellung aus Abbildung 8.2. Machen Sie sich den Zusammenhang klar! (An dieser Stelle verzichten wir noch bewusst darauf, Ableitungsbäumen und ihren Zusammenhang mit Ableitungen zu formalisieren, weil es unseres Erachtens mehr verwirrt als erleuchtet.)

Ableitungsbaum

Die Wörter, die die Grammatik $(\{X\}, \{(,)\}, X, \{X \to XX|(X)|\varepsilon\})$ erzeugt, nennt man übrigens auch *wohlgeformte oder korrekte Klammerausdrücke*.. Das sind die Folgen von öffnenden und schließenden Klammern, die sich ergeben, wenn man z. B. in „normalen" arithmetische Ausdrücken alles außer den Klammern weglässt. Umgangssprachlich aber trotzdem präzise lassen sie sich etwa so definieren:

wohlgeformte oder korrekte Klammerausdrücke

- Das leere Wort ε ist ein korrekter Klammerausdruck.
- Wenn w_1 und w_2 korrekte Klammerausdrücke sind, dann auch w_1w_2.
- Wenn w ein korrekter Klammerausdruck ist, dann auch (w).
- Nichts anderes ist korrekter Klammerausdruck.

Machen Sie sich klar, wie eng der Zusammenhang zwischen dieser Festlegung und der Grammatik ist.

Typ-2-Grammatiken

Zum Abschluss dieses Unterabschnittes sei noch erwähnt, dass kontextfreie Grammatiken auch *Typ-2-Grammatiken* heißen. Daneben gibt es auch noch:

- Typ-3-Grammatiken: Auf sie werden wir später in der Vorlesung noch eingehen.
- Typ-1-Grammatiken und Typ-0-Grammatiken: Was es damit auf sich hat, werden Sie in anderen Vorlesungen kennenlernen.

8.3 relationen (teil 2)

Einige formale Aspekte dessen, was wir im vorangegangenen Unterabschnitt im Zusammenhang mit der Ableitungsrelation $\Rightarrow$ und mit $\Rightarrow^*$ getan haben, sind im Folgenden noch einmal allgemein aufgeschrieben, ergänzt um weitere Erläuterungen. Weil wir schon ausführliche Beispiele gesehen haben, und weil Sie nicht mehr ganz an Anfang des Semesters stehen und sich auch schon an einiges gewöhnt haben, beginnen wir, an manchen Stellen etwas kompakter zu schreiben. Das bedeutet insbesondere, dass Sie an einigen Stellen etwas mehr selbstständig mitdenken müssen. Tun Sie das! Und seien Sie ruhig *sehr* sorgfältig; mehr Übung im Umgang mit Formalismen kann wohl nicht schaden.

Sind $R \subseteq M_1 \times M_2$ und $S \subseteq M_2 \times M_3$ zwei Relationen, dann heißt

$$S \circ R = \{(x,z) \in M_1 \times M_3 \mid \exists y \in M_2 : (x,y) \in R \land (y,z) \in S\}$$

Produkt von Relationen

das *Produkt der Relationen R* und *S*. Machen Sie sich bitte klar, dass diese Schreibweise mit der Komposition von Funktionen kompatibel ist, die Sie kennen. Machen Sie sich bitte auch klar, dass das Relationenprodukt eine assoziative Operation ist.

Mit I_M bezeichnen wir die Relation

$$\mathrm{I}_M = \{(x,x) \mid x \in M\}$$

Wie man schnell sieht, ist das die identische Abbildung auf der Menge M. Deswegen macht man sich auch leicht klar, dass für jede binäre Relation R auf einer Menge M, also für $R \subseteq M \times M$ gilt:

$$R \circ \mathrm{I}_M = R = \mathrm{I}_M \circ R$$

Potenzen einer Relation

Ist $R \subseteq M \times M$ binäre Relation auf einer Menge M, dann definiert man *Potenzen*

R^i wie folgt:

$$R^0 = \mathrm{I}_M$$
$$\forall i \in \mathbb{N}_0 : R^{i+1} = R^i \circ R$$

Die sogenannte *reflexiv-transitive Hülle* einer Relation R ist

reflexiv-transitive Hülle

$$R^* = \bigcup_{i=0}^{\infty} R^i$$

Die reflexiv-transitive Hülle R^* einer Relation R hat folgende Eigenschaften:

- R^* ist reflexiv. Was das bedeutet, werden wir gleich sehen.
- R^* ist transitiv. Was das bedeutet, werden wir gleich sehen.
- R^* ist die kleinste Relation, die R enthält und reflexiv und transitiv ist.

Eine Relation heißt *reflexiv*, wenn $\mathrm{I}_M \subseteq R$ ist.

reflexive Relation

Eine Relation heißt *transitiv*, wenn gilt:

transitive Relation

$$\forall x \in M : \forall y \in M : \forall z \in M : xRy \land yRz \Longrightarrow xRz$$

Bitte bringen Sie den logischen Implikationspfeil $\Longrightarrow$ in dieser Formel nicht mit dem Ableitungspfeil $\Rightarrow$ durcheinander. Wir werden versuchen, die Pfeile immer unterschiedlich lang zu machen, aber das ist natürlich ein nur bedingt tauglicher Versuch besserer Lesbarkeit. Was mit einem Doppelpfeil gemeint ist, muss man immer aus dem aktuellen Kontext herleiten.

Dass R^* stets eine reflexive Relation ist, ergibt sich daraus, dass ja $\mathrm{I}_M = R^0 \subseteq R^*$ ist.

Man kann sich überlegen, dass für alle $i, j \in \mathbb{N}_0$ gilt: $R^i \circ R^j = R^{i+j}$. Wenn man das weiß, dann kann man auch leicht beweisen, dass R^* stets eine transitive Relation ist. Denn sind $(x, y) \in R^*$ und $(y, z) \in R^*$, dann gibt es i und $j \in \mathbb{N}_0$ mit $(x, y) \in R^i$ und $(y, z) \in R^j$. Also ist dann $(x, z) \in R^i \circ R^j = R^{i+j} \subseteq R^*$.

Hinter der Formulierung, R^* sei die *kleinste* Relation, die R umfasst und reflexiv und transitiv ist, steckt folgende Beobachtung: Wenn S eine beliebige Relation ist, die reflexiv und transitiv ist und R umfasst, also $R \subseteq S$, dann ist sogar $R^* \subseteq S$.

8.4 Ein Nachtrag zu Wörtern

Gelegentlich ist es hilfreich, eine kompakte Notation für die Zahl der Vorkommen eines Zeichens $x \in A$ in einem Wort $w \in A^*$ zu haben. Wir definieren daher für

alle Alphabete A und alle $x \in A$ Funktionen $N_x : A^* \to \mathbb{N}_0$, die wie folgt festgelegt sind:

$$N_x(\varepsilon) = 0$$

$$\forall y \in A : \forall w \in A^* : N_x(yw) = \begin{cases} 1 + N_x(w) & \text{falls } y = x \\ N_x(w) & \text{falls } y \neq x \end{cases}$$

Dann ist zum Beispiel

$$\begin{aligned} N_{\mathtt{a}}(\mathtt{abbab}) &= 1 + N_{\mathtt{a}}(\mathtt{bbab}) = 1 + N_{\mathtt{a}}(\mathtt{bab}) = 1 + N_{\mathtt{a}}(\mathtt{ab}) \\ &= 1 + 1 + N_{\mathtt{a}}(\mathtt{b}) = 1 + 1 + N_{\mathtt{a}}(\varepsilon) = 1 + 1 + 0 = 2 \end{aligned}$$

8.5 AUSBLICK

Es sollte klar sein, dass kontextfreie Grammatiken im Zusammenhang mit der Syntaxanalyse von Programmiersprachen eine wichtige Rolle spielen. Allgemein bieten Grammatiken eine Möglichkeit zu spezifizieren, wann etwas syntaktisch korrekt ist.

Die Relation $\Rightarrow^*$ ist ein klassisches Beispiel der reflexiv-transitiven Hülle einer Relation. Eine (etwas?) andere Interpretation von Relationen, Transitivität, usw. werden wir in der Einheit über sogenannte Graphen kennenlernen.

9 SPEICHER

Etwas aufzuschreiben (vor ganz langer Zeit in Bildern, später dann eben mit Zeichen) bedeutet, etwas zu speichern. Weil es naheliegend und einfach ist, reden wir im Folgenden nur über Zeichen.

Wenn man über Speicher reden will, muss man über Zeit reden. Speichern bedeutet, etwas zu einem Zeitpunkt zu schreiben und zu einem späteren Zeitpunkt zu lesen.

Die ersten Sätze dieser Vorlesungseinheit über Speicher haben Sie vermutlich von oben nach unten Ziele für Zeile und in jeder Zeile von links nach rechts gelesen. Für diese Reihenfolge ist der Text auch gedacht. Man könnte aber auch relativ einfach nur jede dritte Zeile lesen. Mit ein bisschen mehr Mühe könnte man auch nachlesen, welches das zweiundvierzigste Zeichen in der dreizehnten Zeile ist. Und wenn Sie wissen wollen, was auf Seite 33 in Zeile 22 Zeichen Nummer 11 ist, dann bekommen Sie auch das heraus.

Man spricht von *wahlfreiem Zugriff*.

wahlfreier Zugriff

Auch Rechner haben bekanntlich Speicher, üblicherweise welche aus Halbleitermateralien und welche aus magnetischen Materialien. Auch für diese Speicher können Sie sich vorstellen, man habe wahlfreien Zugriff. Die gespeicherten Werte sind Bits bzw. Bytes. Diejenigen „Gebilde“, die man benutzt, um eines von mehreren gespeicherten „Objekten“ auszuwählen, nennt man *Adressen*.

9.1 BIT UND BYTE

Das Wort „Bit“ hat verschiedene Bedeutungen. Auf die zweite werden wir vielleicht in einer späteren Einheit eingehen. Die erste ist: Ein *Bit* ist ein Zeichen des Alphabetes {`0`,`1`}.

Bit

Unter einem *Byte* wird heute üblicherweise ein Wort aus acht Bits verstanden (früher war das anders und unterschiedlich). Deswegen wird in manchen Bereichen das deutlichere Wort *Octet* statt Byte bevorzugt. Das gilt nicht nur für den frankophonen Sprachraum, sondern zum Beispiel auch in den sogenannten *Requests for Comments* der *Internet Engineering Task Force* (siehe `http://www.ietf.org/rfc.html`, 19.11.08).

Byte

Octet

RFC
IETF

So wie man die Einheiten Meter und Sekunde mit m bzw. s abkürzt, möchte man manchmal auch Byte und Bit abkürzen. Leider gibt es da Unklarheiten:

- Für Bit wird manchmal die Abkürzung „b“ benutzt. Allerdings ist die „b“ schon die Abkürzung für eine Flächeneinheit, das „barn“ (wovon Sie vermutlich noch nie gehört haben). Deswegen schreibt man manchmal auch „bit“.

- Für Bytes wird oft die Abkürzung „B“ benutzt, obwohl auch „B“ schon die Abkürzung für eine andere Einheit ist (das Bel; Sie haben vielleicht schon von deziBel gehört).
- Für Octets wird die Abkürzung „o“ benutzt.

9.2 SPEICHER ALS TABELLEN UND ABBILDUNGEN

Um den aktuellen Gesamtzustand eines Speichers vollständig zu beschreiben muss man für jede Adresse, zu der etwas gespeichert ist, angeben, welcher Wert unter dieser Adresse abgelegt ist. Das kann man sich zum Beispiel vorstellen als eine große Tabelle mit zwei Spalten: in der linken sind alle Adressen aufgeführt und in der rechten die zugehörigen Werte (siehe Abbildung 9.1).

Adresse 1	Wert 1
Adresse 2	Wert 2
Adresse 3	Wert 3
Adresse 4	Wert 4
⋮	⋮
Adresse n	Wert n

(a) allgemein

000	10110101
001	10101101
010	10011101
011	01110110
100	00111110
101	10101101
110	00101011
111	10101001

(b) Halbleiterspeicher

Abbildung 9.1: Speicher als Tabelle

Mathematisch kann man eine solche Tabelle zum Beispiel auffassen als eine Abbildung, deren Definitionsbereich die Menge aller Adressen und deren Wertebereich die Menge aller möglichen speicherbaren Werte ist:

$$m : \text{Adr} \to \text{Val}$$

9.2.1 Hauptspeicher

Betrachten wir als erstes einen handelsüblichen Rechner mit einem Prozessor, der 4 GiB adressieren kann und mit einem Hauptspeicher dieser Größe ausgerüstet ist. Bei Hauptspeicher ist die Menge der Adressen fest, nämlich alle Kombinationen

von 32 Bits, und was man unter einer Adresse zugreifen kann, ist ein Byte. Jeder Speicherinhalt kann also beschrieben werden als Abbildung

$$m : \{\texttt{0}, \texttt{1}\}^{32} \to \{\texttt{0}, \texttt{1}\}^{8}$$

Der in einem Speicher im Zustand m an einer Adresse $a \in \text{Adr}$ gespeicherte Wert ist dann gerade $m(a)$.

Die Menge der Adressen ist hier fest, und bezeichnet im wesentlichen einen physikalischen Ort auf dem Chip, an dem ein bestimmtes Byte abgelegt ist. Das entspricht einer Angabe wie

Am Fasanengarten 5
76131 Karlsruhe

auf einem Brief.

Nicht nur den Zustand eines Speichers, sondern auch das Auslesen eines Wertes kann man als Abbildung formalisieren. Argumente für eine solche Funktion Auslesefunktion memread sind der gesamte Speicherinhalt m des Speichers und die Adresse a aus der ausgelesen wird, und Resultat ist der in m an Adresse a gespeicherte Wert. Also:

$$\begin{aligned} \text{memread} : \text{Mem} \times \text{Adr} &\to \text{Val} \\ (m, a) &\mapsto m(a) \end{aligned}$$

Dabei sei Mem die Menge aller möglichen Speicherzustände, also die Menge aller Abbildungen von Adr nach Val.

Hier sind nun zwei allgemeine Bemerkungen angezeigt:

1. Man lasse sich nicht dadurch verwirren, dass die Funktion memread eine (andere) Funktion m als Argument bekommt. Das Beispiel soll gerade klar machen, dass daran nichts mystisch ist.
2. Wir werden noch öfter die *Menge aller Abbildungen* der Form $f : A \to B$ von einer Menge A in eine Menge B notieren wollen. Hierfür benutzen wir die Schreibweise B^A. (Anlass für diese Schreibweise war vielleicht die Tatsache, dass für endliche Mengen A und B gilt: $|B^A| = |B|^{|A|}$.)

Wir hätten oben also auch schreiben können:

$$\begin{aligned} \text{memread} : \text{Val}^{\text{Adr}} \times \text{Adr} &\to \text{Val} \\ (m, a) &\mapsto m(a) \end{aligned}$$

Das Speichern eines neuen Wertes v an eine Adresse a in einem Speicher m kann man natürlich auch als Funktion notieren. Es sieht dann ein wenig komplizierter aus als beim Lesen:

$$\begin{aligned} \text{memwrite} : \text{Val}^{\text{Adr}} \times \text{Adr} \times \text{Val} &\to \text{Val}^{\text{Adr}} \\ (m, a, v) &\mapsto m' \end{aligned}$$

Dabei ist m' dadurch festgelegt, dass für alle $a' \in \text{Adr}$ gilt:

$$m'(a') = \begin{cases} m(a') & \text{falls } a' \neq a \\ v & \text{falls } a' = a \end{cases}$$

Was ist „das wesentliche“ an Speicher? Der allerwichtigste Aspekt überhaupt ist sicherlich dieser:

- Wenn man einen Wert v an Adresse a in einen Speicher m hineinschreibt und danach den Wert an Adresse a im durch das Schreiben entstandenen Speicher liest, dann ist das wieder der Wert v. Für alle $m \in \text{Mem}$, $a \in \text{Adr}$ und $v \in \text{Val}$ gilt:

$$\text{memread}(\text{memwrite}(m, a, v), a) = v \tag{9.1}$$

Aber man kann sich „Implementierungen“ von Speicher vorstellen, die zwar diese Eigenschaft haben, aber trotzdem nicht unseren Vorstellungen entsprechen. Eine zweite Eigenschaft, die bei erstem Hinsehen plausibel erscheint, ist diese:

- Was man beim Lesen einer Speicherstelle als Ergebnis erhält, ist nicht davon abhängig, was vorher an einer anderen Adresse gespeichert wurde. Für alle $m \in \text{Mem}$, $a \in \text{Adr}$, $a' \in \text{Adr}$ mit $a' \neq a$ und $v' \in \text{Val}$ gilt:

$$\text{memread}(\text{memwrite}(m, a', v'), a) = \text{memread}(m, a) \tag{9.2}$$

So wie wir oben memwrite definiert haben, gilt diese Gleichung tatsächlich.

Allerdings gibt es Speicher, bei denen diese zunächst plausibel aussehende Bedingung *nicht* erfüllt ist, z. B. sogenannte Caches (bzw. allgemeiner Assoziativspeicher). Für eine adäquate Formalisierung des Schreibens in einen solchen Speicher müsste man also die Defintion von memwrite ändern!

Wir können hier nur kurz andeuten, was es z. B. mit Caches auf sich hat. Spinnen wir die Analogie zu Adressen auf Briefen noch etwas weiter: Typischerweise ist auch der Name des Empfängers angegeben. So etwas wie „Thomas Worsch“ ist aber nicht eine weitere Spezifizierung des physikalischen Ortes innerhalb von „Fasanengarten 5, 76131 Karlsruhe“, an den der Brief transportiert werden soll, sondern eine “inhaltliche“ Beschreibung. Ähnliches gibt es auch bei Speichern. Auch

sie haben eine beschränkte Größe, aber im Extremfall wird für die Adressierung der gespeicherten Wörter sogar überhaupt keine Angabe über den physikalischen Ort gemacht, sondern nur eine Angabe über den Inhalt. Und dann kann Speichern eines Wertes dazu führen, dass ein anderer Wert aus dem Speicher entfernt wird.

Zum Schluss noch kurz eine Anmerkung zur Frage, wozu Gleichungen wie 9.1 und 9.2 gut sein können. Diese Gleichungen sagen ja etwas darüber aus, „wie sich Speicher verhalten soll“. Und daher kann man sie als Teile der *Spezifikation* dessen auffassen, was Speicher überhaupt sein soll. Das wäre ein einfaches Beispiel für etwas, was Sie unter Umständen später in Ihrem Studium als algebraische Spezifikation von Datentypen kennen lernen werden.

Wenn Sie nicht der Spezifizierer sondern der Implementierer von Speicher sind, dann liefern Ihnen Gleichungen wie oben Beispiele für Testfälle. Testen kann nicht beweisen, dass eine Implementierung korrekt ist, sondern „nur“, dass sie falsch ist. Aber das ist auch schon mehr als hilfreich.

9.3 BINÄRE UND DEZIMALE GRÖSSENPRÄFIXE

Früher waren Speicher klein (z. B. ein paar Hundert Bits), heute sind sie groß: Wenn zum Beispiel die Adressen für einen Speicherriegel aus 32 Bits bestehen, dann kann man $2^{32} = 4\,294\,967\,296$ Bytes speichern. Festplatten sind noch größer; man bekommt heute im Laden problemlos welche, von denen der Hersteller sagt, sie fassten 1 Terabyte. Was der Hersteller damit (vermutlich) meint sind 1 000 000 000 000 Bytes.

Solche Zahlen sind nur noch schlecht zu lesen. Wie bei sehr großen (und sehr kleinen) Längen-, Zeit- und anderen Angaben auch, benutzt man daher Präfixe, um zu kompakteren übersichtlicheren Darstellungen zu kommen (Kilometer (km), Mikrosekunde (µs), usw.)

10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-15}	10^{-18}
1000^{-1}	1000^{-2}	1000^{-3}	1000^{-4}	1000^{-5}	1000^{-6}
milli	mikro	nano	pico	femto	atto
m	μ	n	p	f	a
10^{3}	10^{6}	10^{9}	10^{12}	10^{15}	10^{18}
1000^{1}	1000^{2}	1000^{3}	1000^{4}	1000^{5}	1000^{6}
kilo	mega	giga	tera	peta	exa
k	M	G	T	P	E

Im Jahre 1999 hat die *International Electrotechnical Commission* „binäre Präfixe“ ana-

log zu kilo, mega, usw. eingeführt, die aber nicht für Potenzen von 1000, sondern für Potenzen von 1024 stehen. Motiviert durch die Kunstworte „kilobinary“, „megabinary”, usw. heißen die Präfixe *kibi*, *mebi*, usw., abgekürzt Ki, Mi, usw.

2^{10}	2^{20}	2^{30}	2^{40}	2^{50}	2^{60}
1024^1	1024^2	1024^3	1024^4	1024^5	1024^6
kibi	mebi	gibi	tebi	pebi	exbi
Ki	Mi	Gi	Ti	Pi	Ei

9.4 AUSBLICK

- In Vorlesungen über Technische Informatik werden Sie lernen, wie z. B. Caches aufgebaut sind.
- In Vorlesungen über Systemarchitektur und Prozessorarchitektur werden Sie lernen wozu und wie Caches und Assoziativspeicher in modernen Prozessoren eingesetzt werden. Dazu gehört dann auch die Frage, was man tut, wenn mehrere Prozessoren auf einem Chip integriert sind und gemeinsam einen Cache nutzen sollen.
- Der Aufsatz *What Every Programmer Should Know About Memory* von Ulrich Drepper enthält auf 114 Seiten viele Informationen, unter anderem einen 24-seitigen Abschnitt über Caches. Die Arbeit steht online unter der URL `http://people.redhat.com/drepper/cpumemory.pdf` (12.11.09) zur Verfügung.

10 ÜBERSETZUNGEN UND CODIERUNGEN

Von natürlichen Sprachen weiß man, dass man *übersetzen* kann. Beschränken wir uns im weiteren der Einfachheit halber als erstes auf Inschriften. Was ist dann eine Übersetzung? Das hat zwei Aspekte:

1. eine Zuordnung von Wörtern einer Sprache zu Wörtern einer anderen Sprache, die
2. die schöne Eigenschaft hat, dass jedes Ausgangswort und seine Übersetzung die gleiche Bedeutung haben.

Als erstes schauen wir uns ein einfaches Beispiel für Wörter und ihre Bedeutung an: verschiedene Methoden der Darstellung natürlicher Zahlen.

10.1 VON WÖRTERN ZU ZAHLEN UND ZURÜCK

10.1.1 Dezimaldarstellung von Zahlen

Wir sind gewohnt, natürliche Zahlen im sogenannten Dezimalsystem notieren, das aus Indien kommt (siehe auch Einheit 6 über den Algorithmusbegriff):

- Verwendet werden die Ziffern des Alphabetes $Z_{10} = \{\mathtt{0}, \dots, \mathtt{9}\}$.
- Die Bedeutung $\text{num}_{10}(x)$ einer einzelnen Ziffer x als Zahl ist durch die Tabelle

x	0	1	2	3	4	5	6	7	8	9
$\text{num}_{10}(x)$	0	1	2	3	4	5	6	7	8	9

  gegeben.
- Für die Bedeutung eines ganzen Wortes $x_{k-1} \cdots x_0 \in Z_{10}^*$ von Ziffern wollen wir $\text{Num}_{10}(x_{k-1} \cdots x_0)$ schreiben. In der Schule hat man gelernt, dass das gleich

$$10^{k-1} \cdot \text{num}_{10}(x_{k-1}) + \cdots 10^1 \cdot \text{num}_{10}(x_1) + 10^0 \cdot \text{num}_{10}(x_0)$$

  ist. Wir wissen inzwischen, wie man die Pünktchen vermeidet. Und da

$$\begin{aligned} &10^{k-1} \cdot \text{num}_{10}(x_{k-1}) + \cdots 10^1 \cdot \text{num}_{10}(x_1) + 10^0 \cdot \text{num}_{10}(x_0) \\ &\quad = 10 \left(10^{k-2} \cdot \text{num}_{10}(x_{k-1}) + \cdots 10^0 \cdot \text{num}_{10}(x_1)\right) + 10^0 \cdot \text{num}_{10}(x_0) \end{aligned}$$

  ist, definiert man:

$$\begin{aligned} \text{Num}_{10}(\varepsilon) &= 0 \\ \forall w \in Z_{10}^* \; \forall x \in Z_{10} : \text{Num}_{10}(wx) &= 10 \cdot \text{Num}_{10}(w) + \text{num}_{10}(x) \end{aligned}$$

10.1.2 Andere Zahldarstellungen

Gottfried Wilhelm Leibniz

Gottfried Wilhelm Leibniz wurde am 1. Juli 1646 in Leipzig geboren und starb am 14. November 1716 in Hannover. Er war Philosoph, Mathematiker, Physiker, Bibliothekar und vieles mehr. Leibniz baute zum Beispiel die erste Maschine, die zwei Zahlen multiplizieren konnte.
Leibniz hatte erkannt, dass man die natürlichen Zahlen nicht nur mit den Ziffern 0, . . . , 9 notieren kann, sondern dass dafür 0 und 1 genügen. Er hat dies in einem Brief vom 2. Januar 1697 an den Herzog von Braunschweig-Wolfenbüttel beschrieben (siehe auch http://www.hs-augsburg.de/~harsch/germanica/Chronologie/17Jh/Leibniz/lei_bina.html, 22.11.2010) und im Jahre 1703 in einer Zeitschrift veröffentlicht. Abbildung 10.1 zeigt Beispielrechnungen mit Zahlen in Binärdarstellung aus dieser Veröffentlichung.

Pour l'Addition par exemple.	110 | 6 111 | 7 1101 | 13	101 | 5 1011 | 11 10000 | 16	1110 | 14 10001 | 17 11111 | 31
Pour la Souſtraction.	1101 | 13 111 | 7 110 | 6	10000 | 16 1011 | 11 101 | 5	11111 | 31 10001 | 17 1110 | 14
Pour la Multiplication.	11 | 3 11 | 3 11 11 1001 | 9	101 | 5 11 | 3 101 101 1111 | 15	101 | 5 101 | 5 101 1010 11001 | 25
Pour la Diviſion.	15 | 1111 3 | 1111 11	101 | 5	

Abbildung 10.1: Ausschnitt aus dem Aufsatz „Explication de l'Arithmétique Binaire" von Leibniz, Quelle: http://commons.wikimedia.org/wiki/Image:Leibniz_binary_system_1703.png (22.11.2010)

Binärdarstellung

Bei der *Binärdarstellung* von nichtnegativen ganzen Zahlen geht man analog zur

Dezimaldarstellung vor. Als Ziffernmenge benutzt man $Z_2 = \{\texttt{0}, \texttt{1}\}$ und definiert

$$\begin{aligned} \text{num}_2(\texttt{0}) &= 0 \\ \text{num}_2(\texttt{1}) &= 1 \\ \text{Num}_2(\varepsilon) &= 0 \end{aligned}$$

sowie $$\forall w \in Z_2^* \; \forall x \in Z_2 : \text{Num}_2(wx) = 2 \cdot \text{Num}_2(w) + \text{num}_2(x)$$

Damit ist dann z. B.

$$\begin{aligned} \text{Num}_2(\texttt{1101}) &= 2 \cdot \text{Num}_2(\texttt{110}) + 1 \\ &= 2 \cdot (2 \cdot \text{Num}_2(\texttt{11}) + 0) + 1 \\ &= 2 \cdot (2 \cdot (2 \cdot \text{Num}_2(\texttt{1}) + 1) + 0) + 1 \\ &= 2 \cdot (2 \cdot (2 \cdot 1 + 1) + 0) + 1 \\ &= 2^3 \cdot 1 + 2^2 \cdot 1 + 2^1 \cdot 0 + 2^0 \cdot 1 \\ &= 13 \end{aligned}$$

Bei der *Hexadezimaldarstellung* oder Sedezimaldarstellung benutzt man 16 Ziffern des Alphabets $Z_{16} = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}\}$. (Manchmal werden statt der Großbuchstaben auch Kleinbuchstaben verwendet.)

Hexadezimaldarstellung

x	0	1	2	3	4	5	6	7
$\text{num}_{16}(x)$	0	1	2	3	4	5	6	7
x	8	9	A	B	C	D	E	F
$\text{num}_{16}(x)$	8	9	10	11	12	13	14	15

Die Zuordnung von Wörtern zu Zahlen ist im Wesentlichen gegeben durch

$$\forall w \in Z_{16}^* \; \forall x \in Z_{16} : \text{Num}_{16}(wx) = 16 \cdot \text{Num}_{16}(w) + \text{num}_{16}(x)$$

Ein Problem ergibt sich dadurch, dass die Alphabete Z_2, Z_3, usw. nicht disjunkt sind. Daher ist z. B. die Zeichenfolge `111` mehrdeutig: Man muss wissen, eine Darstellung zu welcher Basis das ist, um sagen zu können, welche Zahl hier repräsentiert wird. Zum Beispiel ist

$\text{Num}_2(\texttt{111})$ die Zahl sieben,
$\text{Num}_8(\texttt{111})$ die Zahl dreiundsiebzig,
$\text{Num}_{10}(\texttt{111})$ die Zahl einhundertelf und
$\text{Num}_{16}(\texttt{111})$ die Zahl zweihundertdreiundsiebzig.

Deswegen ist es in manchen Programmiersprachen so, dass für Zahldarstellungen zu den Basen 2, 8 und 16 als Präfix respektive `0b`, `0o` und `0x` vorgeschrieben sind. Dann sieht man der Darstellung unmittelbar an, wie sie zu interpretieren ist. In anderen Programmiersprachen sind entsprechende Darstellungen gar nicht möglich.

10.1.3 Von Zahlen zu ihren Darstellungen

So wie man zu einem Wort die dadurch repräsentierte Zahl berechnen kann, kann man auch umgekehrt zu einer nichtnegativen ganzen Zahl $n \in \mathbb{N}_0$ eine sogenannte k-äre Darstellung berechnen.

Dazu benutzt man ein Alphabet Z_k mit k Ziffern, deren Bedeutungen die Zahlen in $\mathbb{G}_k$ sind. Für $i \in \mathbb{G}_k$ sei $\text{repr}_k(i)$ dieses Zeichen. Die Abbildung repr_k ist also gerade die Umkehrfunktion zu num_k.

Gesucht ist dann eine Repräsentation von $n \in \mathbb{N}_0$ als Wort $w \in Z_k^*$ mit der Eigenschaft $\text{Num}_k(w) = n$. Dabei nimmt man die naheliegende Definition von Num_k an.

Wie wir gleich sehen werden, gibt es immer solche Wörter. Und es sind dann auch immer gleich unendlich viele, denn wenn $\text{Num}_k(w) = n$ ist, dann auch $\text{Num}_k(\mathtt{0}w) = n$ (Beweis durch Induktion). Eine k-äre Darstellung einer Zahl kann man zum Beispiel so bestimmen:

// Eingabe: $n \in \mathbb{N}_0$

$y \leftarrow n$

$w \leftarrow \varepsilon$

$$m \leftarrow \begin{cases} 1 + \lfloor \log_k(n) \rfloor & \text{falls } n > 0 \\ 1 & \text{falls } n = 0 \end{cases}$$

for $i \leftarrow 0$ **to** $m - 1$ **do**

  $r \leftarrow y$ **mod** k

  $w \leftarrow \text{repr}_k(r) \cdot w$ // Konkatenation

  $y \leftarrow y$ **div** k

od

// am Ende: $n = \text{Num}_k(w)$

Am deutlichsten wird die Arbeitsweise dieses Algorithmus, wenn man ihn einmal für einen vertrauten Fall benutzt: Nehmen wir $k = 10$ und $n = 4711$. Dann ist $m = 4$ und für jedes $i \in \mathbb{G}_5$ haben die Variablen r, w und y nach i Schleifendurchläufen

die in der folgende Tabelle angegebenen Werte. Um die Darstellung besonders klar zu machen, haben wir die Fälle von $i = 0$ bis $i = 4$ von rechts nach links aufgeschrieben:

i	4	3	2	1	0
r	4	7	1	1	
w	`4711`	`711`	`11`	`1`	ε
y	0	4	47	471	4711

Die Schleifeninvariante $y \cdot 10^i + \mathrm{Num}_k(w) = n$ drängt sich förmlich auf. Und wenn man bewiesen hat, dass am Ende $y = 0$ ist, ist man fertig.

Der obige Algorithmus liefert das (es ist eindeutig) kürzeste Wort $w \in Z_k^*$ mit $\mathrm{Num}_k(w) = n$. Dieses Wort wollen wir auch mit $\mathrm{Repr}_k(n)$ bezeichnen. Es ist also stets

$$\mathrm{Num}_k(\mathrm{Repr}_k(n)) = n \ .$$

Man beachte, dass umgekehrt $\mathrm{Repr}_k(\mathrm{Num}_k(w))$ im allgemeinen nicht unbedingt wieder w ist, weil „überflüssige" führende Nullen wegfallen.

10.2 von einem alphabet zum anderen

10.2.1 Ein Beispiel: Übersetzung von Zahldarstellungen

Wir betrachten die Funktion $\mathrm{Trans}_{2,16} = \mathrm{Repr}_2 \circ \mathrm{Num}_{16}$ von Z_{16}^* nach Z_2^*. Sie bildet zum Beispiel das Wort `A3` ab auf

$$\mathrm{Repr}_2(\mathrm{Num}_{16}(\texttt{A3})) = \mathrm{Repr}_2(163) = \texttt{10100011} \ .$$

Der wesentliche Punkt ist, dass die beiden Wörter `A3` und `10100011` die gleiche Bedeutung haben: die Zahl einhundertdreiundsechzig.

Allgemein wollen wir folgende Sprechweisen vereinbaren. Sehr oft, wie zum Beispiel gesehen bei Zahldarstellungen, schreibt man Wörter einer formalen Sprache L über einem Alphabet und meint aber etwas anderes, ihre Bedeutung. Die Menge der Bedeutungen der Wörter aus L ist je nach Anwendungsfall sehr unterschiedlich. Es kann so etwas einfaches sein wie Zahlen, oder so etwas kompliziertes wie die Bedeutung der Ausführung eines Java-Programmes. Für so eine Menge von „Bedeutungen" schreiben wir im folgenden einfach Sem.

Wir gehen nun davon aus, dass zwei Alphabete A und B gegeben sind, und zwei Abbildungen $\mathrm{sem}_A : L_A \to \mathrm{Sem}$ und $\mathrm{sem}_B : L_B \to \mathrm{Sem}$ von formalen Sprachen $L_A \subseteq A^*$ und $L_B \subseteq B^*$ in die *gleiche* Menge Sem von Bedeutungen.

Übersetzung

Eine Abbildung $f : L_A \to L_B$ heiße eine *Übersetzung* bezüglich sem_A und sem_B, wenn f die Bedeutung erhält, d. h.

$$\forall w \in L_A : \text{sem}_A(w) = \text{sem}_B(f(w))$$

Betrachten wir noch einmal die Funktion $\text{Trans}_{2,16} = \text{Repr}_2 \circ \text{Num}_{16}$. Hier haben wir den einfachen Fall, dass $L_A = A^* = Z_{16}^*$ und $L_B = B^* = Z_2^*$. Die Bedeutungsfunktionen sind $\text{sem}_A = \text{Num}_{16}$ und $\text{sem}_B = \text{Num}_2$. Dass bezüglich dieser Abbildungen $\text{Trans}_{2,16}$ tatsächlich um eine Übersetzung handelt, kann man leicht nachrechnen:

$$\begin{aligned} \text{sem}_B(f(w)) &= \text{Num}_2(\text{Trans}_{2,16}(w)) \\ &= \text{Num}_2(\text{Repr}_2(\text{Num}_{16}(w))) \\ &= \text{Num}_{16}(w) \\ &= \text{sem}_A(w) \end{aligned}$$

Im allgemeinen kann die Menge der Bedeutungen recht kompliziert sein. Wenn es um die Übersetzung von Programmen aus einer Programmiersprache in eine andere Programmiersprache geht, dann ist die Menge Sem die Menge der Bedeutungen von Programmen. Als kleine Andeutung wollen hier nur erwähnen, dass man dann z. B. die Semantik einer Zuweisung $x \leftarrow 5$ definieren könnte als die Abbdildung, die aus einer Speicherbelegung m die Speicherbelegung $\text{memwrite}(m, x, 5)$ macht (siehe Kapitel 9). Eine grundlegende Einführung in solche Fragestellungen können Sie in Vorlesungen über die Semantik von Programmiersprachen bekommen.

Warum macht man Übersetzungen? Zumindest die folgenden Möglichkeiten fallen einem ein:

- *Lesbarkeit:* Übersetzungen können zu kürzeren und daher besser lesbaren Texten führen. `A3` ist leichter erfassbar als `10100011` (findet der Autor dieser Zeilen).
- *Verschlüsselung:* Im Gegensatz zu verbesserter Lesbarkeit übersetzt man mitunter gerade deshalb, um die Lesbarkeit möglichst unmöglich zu machen, jedenfalls für Außenstehende. Wie man das macht, ist Gegenstand von Vorlesungen über Kryptographie.
- *Kompression:* Manchmal führen Übersetzungen zu kürzeren Texten, die also weniger Platz benötigen. Und zwar *ohne* zu einem größeren Alphabet überzugehen. Wir werden im Abschnitt 10.3 über Huffman-Codes sehen, warum und wie das manchmal möglich ist.
- *Fehlererkennung* und *Fehlerkorrektur:* Manchmal kann man Texte durch Übersetzung auf eine Art länger machen, dass man selbst dann, wenn ein korrek-

ter Funktionswert $f(w)$ „zufällig" „kaputt" gemacht wird (z. B. durch Übertragungsfehler auf einer Leitung) und nicht zu viele Fehler passieren, man die Fehler korrigieren kann, oder zumindest erkennt, dass Fehler passiert sind. Ein typisches Beispiel sind lineare Codes, von denen Sie (vielleicht) in anderen Vorlesung hören werden.

Es gibt einen öfter anzutreffenden Spezialfall, in dem man sich um die Einhaltung der Forderung $\text{sem}_A(w) = \text{sem}_B(f(w))$ keine Gedanken machen muss. Zumindest bei Verschlüsselung, aber auch bei manchen Anwendungen von Kompression ist es so, dass man vom Übersetzten $f(x)$ eindeutig zurückkommen können möchte zum ursprünglichen x. M. a. w., f ist injektiv. In diesem Fall kann man dann die Bedeutung sem_B im wesentlichen *definieren* durch die Festlegung $\text{sem}_B(f(x)) = \text{sem}_A(x)$. Man mache sich klar, dass an dieser Stelle die Injektivität von f wichtig ist, damit sem_B wohldefiniert ist. Denn wäre für zwei $x \neq y$ zwar $\text{sem}_A(x) \neq \text{sem}_B(x)$ aber die Funktionswerte $f(x) = f(y) = z$, dann wäre nicht klar, was $\text{sem}_B(z)$ sein soll.

Wenn f injektiv ist, wollen wir das eine *Codierung* nennen. Für $w \in L_A$ heißt $f(w)$ ein *Codewort* und die Menge $\{f(w) \mid w \in L_A\}$ aller Codewörter heißt dann auch ein *Code*.

Codierung
Codewort
Code

Es stellt sich die Frage, wie man eine Übersetzung vollständig spezifiziert. Man kann ja nicht für im allgemeinen unendliche viele Wörter $w \in L_A$ einzeln erschöpfend aufzählen, was $f(w)$ ist.

Eine Möglichkeit bieten sogenannte Homomorphismen und Block-Codierungen, auf die wir im Folgenden noch genauer eingehen werden.

10.2.2 Homomorphismen

Es seien A und B zwei Alphabete und $h : A \to B^*$ eine Abbildung. Zu h kann man in der Ihnen inzwischen vertrauten Art eine Funktion $h^{**} : A^* \to B^*$ definieren vermöge

$$h^{**}(\varepsilon) = \varepsilon$$
$$\forall w \in A^* : \forall x \in A : h^{**}(wx) = h^{**}(w)h(x)$$

Eine solche Abbildung h^{**} nennt man einen *Homomorphismus*. Häufig erlaubt man sich, statt h^{**} einfach wieder h zu schreiben (weil für alle $x \in A$ gilt: $h^{**}(x) = h(x)$).

h^{**}
Homomorphismus

Ein Homomorphismus heißt *ε-frei*, wenn für alle $x \in A$ gilt: $h(x) \neq \varepsilon$.

ε-freier Homomorphismus

Dass eine Abbildung $h : A \to B^*$ eine Codierung, also eine injektive Abbildung $h^{**} : A^* \to B^*$ induziert, ist im allgemeinen nicht ganz einfach zu sehen. Es gibt aber einen Spezialfall, in dem das klar ist, nämlich dann, wenn h *präfixfrei* ist. Das

präfixfrei

bedeutet, dass für *keine* zwei verschiedenen Symbole $x_1, x_2 \in A$ gilt: $h(x_1)$ ist ein Präfix von $h(x_2)$.

Die Decodierung ist in diesem Fall relativ einfach. Allerdings hat man in allen nichttrivialen Fällen das Problem zu beachten, dass nicht alle Wörter aus B^* ein Codewort sind. Mit anderen Worten ist h^{**} im allgemeinen nicht surjektiv. Um die Decodierung trotzdem als totale Abbildung u definieren zu können, wollen wir hier als erstes festlegen, dass es sich um eine Abbildung $u : B^* \to (A \cup \{\perp\})^*$. Das zusätzliche Symbol $\perp$ wollen wir benutzen, wenn ein $w \in B^*$ gar kein Codewort ist und nicht decodiert werden kann. In diesem Fall soll $u(w)$ das Symbol $\perp$ enthalten.

Als Beispiel betrachten wir den Homomorphismus $h : \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^* \to \{\mathtt{0}, \mathtt{1}\}^*$ mit $h(\mathtt{a}) = \mathtt{1}$, $h(\mathtt{b}) = \mathtt{01}$ und $h(\mathtt{c}) = \mathtt{001}$. Dieser Homomorphismus ist präfixfrei.

Wir schreiben nun zunächst einmal Folgendes hin:

$$u(w) = \begin{cases} \varepsilon & \text{falls } w = \varepsilon \\ \mathtt{a}u(w') & \text{falls } w = \mathtt{1}w' \\ \mathtt{b}u(w') & \text{falls } w = \mathtt{01}w' \\ \mathtt{c}u(w') & \text{falls } w = \mathtt{001}w' \\ \perp & \text{sonst} \end{cases}$$

Sei w das Beispielcodewort $w = \mathtt{100101} = h(\mathtt{acb})$. Versuchen wir nachzurechnen, was die Abbildung u mit w „macht“:

$$\begin{aligned} u(\mathtt{100101}) &= \mathtt{a}u(\mathtt{00101}) \\ &= \mathtt{ac}u(\mathtt{01}) \\ &= \mathtt{acb}u(\varepsilon) \\ &= \mathtt{acb} \end{aligned}$$

Prima, das hat geklappt. Aber warum? In jedem Schritt war klar, welche Zeile der Definition von u anzuwenden war. Und warum war das klar? Im Wesentlichen deswegen, weil ein Wort w nicht gleichzeitig mit den Codierungen verschiedener Symbole anfangen kann; kein $h(x)$ ist Anfangsstück eines $h(y)$ für *verschiedene* $x, y \in A$. Das ist nichts anderes als die Präfixfreiheit von h.

wohldefiniert

Man spricht hier auch davon, dass die oben festgelegte Abbildung u *wohldefiniert* ist. Über Wohldefiniertheit muss man immer dann nachdenken, wenn ein Funktionswert potenziell „auf mehreren Wegen“ festgelegt wird. Dann muss man sich entweder klar machen, dass in Wirklichkeit wie im obigen Beispiel immer nur ein Weg „gangbar“ ist, oder dass auf den verschiedenen Wegen am Ende der gleiche Funktionswert herauskommt. Für diesen zweiten Fall werden wir später noch Beispiele sehen, z. B. in dem Kapitel über Äquivalenz- und Kongruenzrelationen.

Allgemein kann man bei einem präfixfreien Code also so decodieren:

$$u(w) = \begin{cases} \varepsilon & \text{falls } w = \varepsilon \\ x\, u(w') & \text{falls } w = h(x)w' \text{ für ein } x \in A \\ \perp & \text{sonst} \end{cases}$$

Man beachte, dass das *nicht* heißt, dass man nur präfixfreie Codes decodieren kann. Es heißt nur, dass man nur präfixfreie Codes „so einfach" decodieren kann.

Das praktische Beispiel schlechthin für einen Homomorphismus ist die Repräsentation des ASCII-Zeichensatzes (siehe Abschnitt 3.1.1) im Rechner. Das geht einfach so: Ein Zeichen x mit der Nummer n im ASCII-Code wird codiert durch dasjenige Byte $w \in \{0, 1\}^8$, für das $\text{Num}_2(w) = n$ ist. Und längere Texte werden übersetzt, indem man nacheinander jedes Zeichen einzeln so abbildet.

Dieser Homomorphismus hat sogar die Eigenschaft, dass alle Zeichen durch Wörter gleicher Länge codiert werden. Das muss aber im allgemeinen nicht so sein. Im nachfolgenden Unterabschnitt kommen wir kurz auf einen wichtigen Fall zu sprechen, bei dem das nicht so ist.

10.2.3 Beispiel Unicode: UTF-8 Codierung

Auch die Zeichen des Unicode-Zeichensatz kann man natürlich im Rechner speichern. Eine einfache Möglichkeit besteht darin, analog zu ASCII für alle Zeichen die jeweiligen Nummern (Code Points) als jeweils gleich lange Wörter darzustellen. Da es so viele Zeichen sind (und (unter anderem deswegen) manche Code Points große Zahlen sind), bräuchte man für jedes Zeichen vier Bytes.

Nun ist es aber so, dass zum Beispiel ein deutscher Text nur sehr wenige Zeichen benutzen wird, und diese haben auch noch kleine Nummern. Man kann daher auf die Idee kommen, nach platzsparenderen Codierungen zu suchen. Eine von ihnen ist *UTF-8*.

UTF-8

Nachfolgend ist die Definition dieses Homomorphismus in Ausschnitten wiedergegeben. Sie stammen aus *RFC 3629* (`http://tools.ietf.org/html/rfc3629`, 22.11.2010).

RFC 3629

> The table below summarizes the format of these different octet types. The letter x indicates bits available for encoding bits of the character number.

Char. number range (hexadecimal)	UTF-8 octet sequence (binary)
`0000 0000 - 0000 007F`	`0`$xxxxxxx$
`0000 0080 - 0000 07FF`	`110`$xxxxx$ `10`$xxxxxx$
`0000 0800 - 0000 FFFF`	`1110`$xxxx$ `10`$xxxxxx$ `10`$xxxxxx$
`0001 0000 - 0010 FFFF`	`11110`xxx `10`$xxxxxx$ `10`$xxxxxx$ `10`$xxxxxx$

Encoding a character to UTF-8 proceeds as follows:

- Determine the number of octets required from the character number and the first column of the table above. It is important to note that the rows of the table are mutually exclusive, i.e., there is only one valid way to encode a given character.
- Prepare the high-order bits of the octets as per the second column of the table.
- Fill in the bits marked x from the bits of the character number, expressed in binary. Start by putting the lowest-order bit of the character number in the lowest-order position of the last octet of the sequence, then put the next higher-order bit of the character number in the next higher-order position of that octet, etc. When the x bits of the last octet are filled in, move on to the next to last octet, then to the preceding one, etc. until all x bits are filled in.

Da die druckbaren Zeichen aus dem ASCII-Zeichensatz dort die gleichen Nummern haben wie bei Unicode, hat dieser Homomorphismus die Eigenschaft, dass Texte, die nur ASCII-Zeichen enthalten, bei der ASCII-Codierung und bei UTF-8 die gleiche Codierung besitzen.

10.3 HUFFMAN-CODIERUNG

Huffman-Codierung

Es sei ein Alphabet A und ein Wort $w \in A^*$ gegeben. Eine sogenannte *Huffman-Codierung* von w ist der Funktionswert $h(w)$ einer Abbildung $h : A^* \to Z_2^*$, die ε-freier Homomorphismus ist. Die Konstruktion von h wird dabei „auf w zugeschnitten“, damit $h(w)$ besonders „schön“, d. h. in diesem Zusammenhang besonders kurz, ist.

Jedes Symbol $x \in A$ kommt mit einer gewissen absoluten Häufigkeit $N_x(w)$ in w vor. Der wesentliche Punkt ist, dass Huffman-Codes häufigere Symbole durch kürzere Wörter codieren und seltener vorkommende Symbole durch längere.

Wir beschreiben als erstes, wie man die $h(x)$ bestimmt. Anschließend führen wir interessante und wichtige Eigenschaften von Huffman-Codierungen auf, die auch der Grund dafür sind, dass sie Bestandteil vieler Kompressionsverfahren sind. Zum Schluß erwähnen wir eine naheliegende Verallgemeinerung des Verfahrens.

10.3.1 Algorithmus zur Berechnung von Huffman-Codes

Gegeben sei ein $w \in A^*$ und folglich die Anzahlen $N_x(w)$ aller Symbole $x \in A$ in w. Da man Symbole, die in w überhaupt nicht vorkommen, auch nicht codieren muss, beschränken wir uns bei der folgenden Beschreibung auf den Fall, dass alle $N_x(w) > 0$ sind (w also eine surjektive Abbildung auf A ist).

Der Algorithmus zur Bestimmung eines Huffman-Codes arbeitet in zwei Phasen:

1. Zunächst konstruiert er Schritt für Schritt einen Baum. Die Blätter des Baumes entsprechen $x \in A$, innere Knoten, d. h. Nicht-Blätter entsprechen Mengen von Symbolen. Um Einheitlichkeit zu haben, wollen wir sagen, dass ein Blatt für eine Menge $\{x\}$ steht.

   An *jedem* Knoten wird eine Häufigkeit notiert. Steht ein Knoten für eine Knotenmenge $X \subseteq A$, dann wird als Häufigkeit gerade die Summe $\sum_{x \in X} N_x(w)$ der Häufigkeiten der Symbole in X aufgeschrieben. Bei einem Blatt ist das also einfach ein $N_x(w)$. Zusätzlich wird bei jedem Blatt das zugehörige Symbol x notiert.

   In dem konstruierten Baum hat jeder innere Knoten zwei Nachfolger, einen linken und einen rechten.
2. In der zweiten Phase werden alle Kanten des Baumes beschriftet, und zwar jede linke Kante mit `0` und jede rechte Kante mit `1`.

   Um die Codierung eines Zeichens x zu berechnen, geht man dann auf dem kürzesten Weg von der Wurzel des Baumes zu dem Blatt, das x entspricht, und konkateniert der Reihe nach alle Symbole, mit denen die Kanten auf diesem Weg beschriftet sind.

Wenn zum Beispiel das Wort w = `afebfecaffdeddccefbeff` gegeben ist, dann kann sich der Baum ergeben, der in Abbildung 10.2 dargestellt ist. In diesem Fall ist dann der Homomorphismus gegeben durch

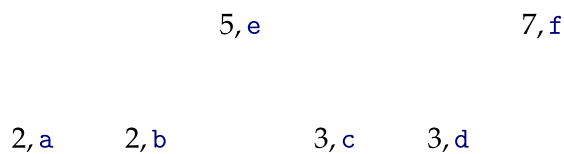
x	a	b	c	d	e	f
h(x)	000	001	100	101	01	11

Es bleibt zu beschreiben, wie man den Baum konstruiert. Zu jedem Zeitpunkt hat man eine Menge M von „noch zu betrachtenden Symbolmengen mit ihren

Häufigkeiten". Diese Menge ist initial die Menge aller $\{x\}$ für $x \in A$ mit den zugehörigen Symbolhäufigkeiten, die wir so aufschreiben wollen:

$$M_0 = \{\ (2, \{\mathtt{a}\})\ ,\ (2, \{\mathtt{b}\})\ ,\ (3, \{\mathtt{c}\})\ ,\ (3, \{\mathtt{d}\})\ ,\ (5, \{\mathtt{e}\})\ ,\ (7, \{\mathtt{f}\})\ \}$$

Als Anfang für die Konstruktion des Baumes zeichnet man für jedes Symbol einen Knoten mit Markierung $(x, N_x(w))$. Im Beispiel ergibt sich



Solange eine Menge M_i noch mindestens zwei Paare enthält, tut man folgendes:

- Man bestimmt man eine Menge M_{i+1} wie folgt:
  - Man wählt zwei Paare (X_1, k_1) und (X_2, k_2), deren Häufigkeiten zu den kleinsten noch vorkommenden gehören.
  - Man entfernt diese Paare aus M_i und fügt statt dessen das eine Paar $(X_1 \cup X_2, k_1 + k_2)$ hinzu. Das ergibt M_{i+1}.

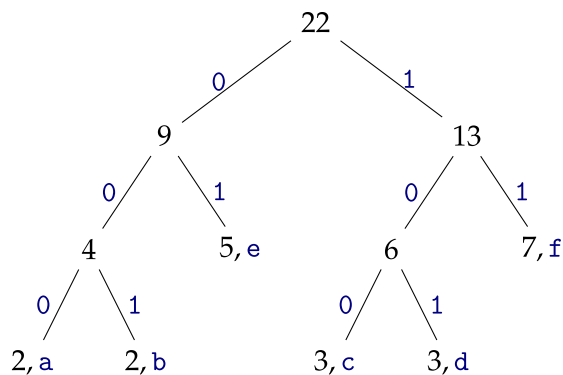
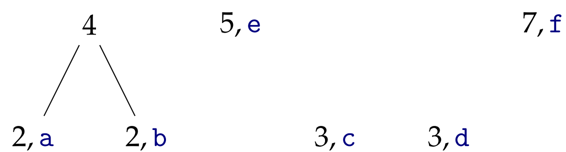


Abbildung 10.2: Ein Beispielbaum für die Berechnung eines Huffman-Codes.

Im Beispiel ergibt sich also

$$M_1 = \{\ (4, \{\mathtt{a}, \mathtt{b}\})\ ,\ (3, \{\mathtt{c}\})\ ,\ (3, \{\mathtt{d}\})\ ,\ (5, \{\mathtt{e}\})\ ,\ (7, \{\mathtt{f}\})\ \}$$

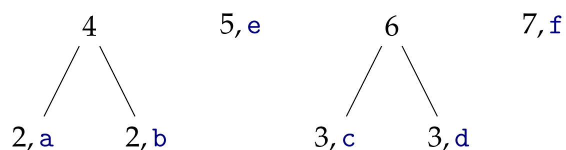
- Als zweites fügt man dem schon konstruierten Teil des Graphen einen weiteren Knoten hinzu, markiert mit der Häufigkeit $k_1 + k_2$ und Kanten zu den Knoten, die für (X_1, k_1) und (X_2, k_2) eingefügt worden waren



Da M_1 noch mehr als ein Element enthält, wiederholt man die beiden obigen Teilschritte:

$$M_2 = \{\ (4, \{\mathtt{a}, \mathtt{b}\})\ ,\ (6, \{\mathtt{c}, \mathtt{d}\})\ ,\ (5, \{\mathtt{e}\})\ ,\ (7, \{\mathtt{f}\})\ \}$$

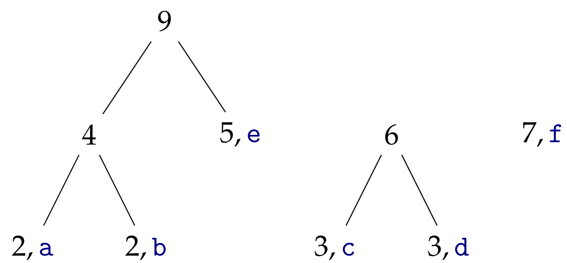
und der Graph sieht dann so aus:



Da M_2 noch mehr als ein Element enthält, wiederholt man die beiden obigen Teilschritte wieder:

$$M_3 = \{\ (9, \{\mathtt{a}, \mathtt{b}, \mathtt{e}\})\ ,\ (6, \{\mathtt{c}, \mathtt{d}\})\ ,\ (7, \{\mathtt{f}\})\ \}$$

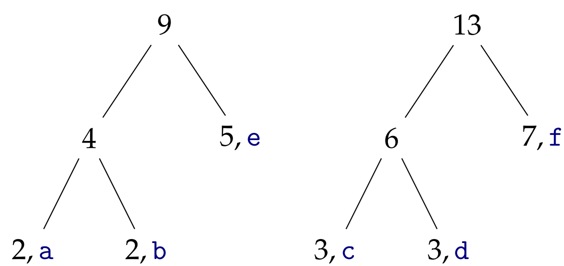
und der Graph sieht dann so aus:



Da M_3 noch mehr als ein Element enthält, wiederholt man die beiden obigen Teilschritte wieder:

$$M_4 = \{ \ (9, \{\texttt{a}, \texttt{b}, \texttt{e}\}) \ , \ (13, \{\texttt{c}, \texttt{d}, \texttt{f}\}) \ \}$$

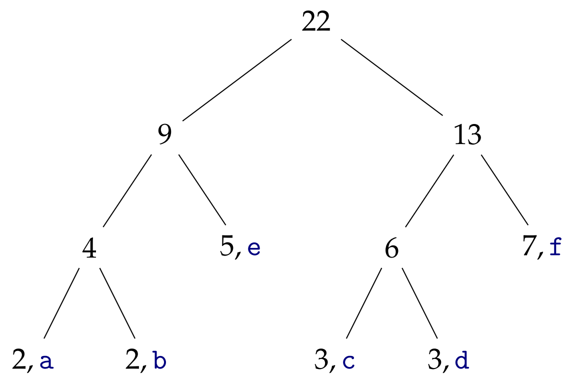
und der Graph sieht dann so aus:



Zum Schluss berechnet man noch

$$M_5 = \{ \ (22, \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}\}) \ \}$$

und es ergibt sich der Baum



Aus diesem ergibt sich durch die Beschriftung der Kanten die Darstellung in Abbildung 10.2.

10.3.2 Weiteres zu Huffman-Codes

Wir haben das obige Beispiel so gewählt, dass immer eindeutig klar war, welche zwei Knoten zu einem neuen zusammengefügt werden mussten. Im allgemeinen ist das nicht so: Betrachten Sie einfach den Fall, dass viele Zeichen alle gleichhäufig vorkommen. Außerdem ist nicht festgelegt, welcher Knoten linker Nachfolger und welcher rechter Nachfolger eines inneren Knotens wird.

Konsequenz dieser Mehrdeutigkeiten ist, dass ein Huffman-Code nicht eindeutig ist. Das macht aber nichts: alle, die sich für ein Wort w ergeben können, sind „gleich gut".

Dass Huffman-Codes gut sind, kann man so präzisieren: unter allen präfixfreien Codes führen Huffman-Codes zu kürzesten Codierungen. Einen Beweis findet man zum Beispiel unter `http://www.maths.abdn.ac.uk/~igc/tch/mx4002/notes/node59.html`, 22.11.2010).

Zum Schluss wollen wir noch auf eine (von mehreren) Verallgemeinerung des oben dargestellten Verfahrens hinweisen. Manchmal ist es nützlich, nicht von den Häufigkeiten einzelner Symbole auszugehen und für die Symbole einzeln Codes zu berechnen, sondern für Teilwörter einer festen Länge $b > 1$. Alles wird ganz analog durchgeführt; der einzige Unterschied besteht darin, dass an den Blättern des Huffman-Baumes eben Wörter stehen und nicht einzelne Symbole.

Eine solche Verallgemeinerung wird bei mehreren gängigen Kompressionsverfahren (z. B. gzip, bzip2) benutzt, die, zumindest als einen Bestandteil, Huffman-Codierung benutzen.

Allgemein gesprochen handelt es sich bei diesem Vorgehen um eine sogenannte *Block-Codierung*. Statt wie bei einem Homomorphsimus die Übersetzung $h(x)$

Block-Codierung

jedes einzelnen Zeichens $x \in A$ festzulegen, tut man das für alle Teilwörter, die sogenannten Blöcke, einer bestimmten festen Länge $b \in \mathbb{N}_+$. Man geht also von einer Funktion $h : A^b \to B^*$ aus, und erweitert diese zu einer Funktion $h : (A^b)^* \to B^*$.

10.4 AUSBLICK

Wir haben uns in dieser Einheit nur mit der Darstellung von nichtnegativen ganzen Zahlen beschäftigt. Wie man bei negativen und gebrochenen Zahlen z. B. in einem Prozessor vorgeht, wird in einer Vorlesung über technische Informatik behandelt werden.

Wer Genaueres über UTF-8 und damit zusammenhängende Begriffe bei Unicode wissen will, sei die detaillierte Darstellung im *Unicode Technical Report UTR-17* empfohlen, den man unter `http://www.unicode.org/reports/tr17/` (22.11.2010) im WWW findet.

Mehr über Bäume und andere Graphen werden wir in der Einheit dem überraschenden Titel „Graphen“ lernen.

11 GRAPHEN

In den bisherigen Einheiten kamen schon an mehreren Stellen Diagramme und Bilder vor, in denen irgendwelche „Gebilde" durch Linien oder Pfeile miteinander verbunden waren. Man erinnere sich etwa an die Ableitungsbäume wie in Abbildung 8.2 in der Einheit über Grammatiken, an Huffman-Bäume wie in Abbildung 10.2 der Einheit über Codierungen, oder an die Abbildung am Ende von Abschnitt 2.4.

Das sind alles Darstellungen sogenannter Graphen. Sie werden in dieser Einheit sozusagen vom Gebrauchsgegenstand zum Untersuchungsgegenstand. Dabei unterscheidet man üblicherweise gerichtete und ungerichtete Graphen, denen im folgenden getrennte Abschnitte gewidmet sind.

11.1 GERICHTETE GRAPHEN

11.1.1 Graphen und Teilgraphen

Ein *gerichteter Graph* ist festgelegt durch ein Paar $G = (V, E)$, wobei $E \subseteq V \times V$ ist. Die Elemente von V heißen *Knoten*, die Elemente von E heißen *Kanten*. Wir verlangen (wie es üblich ist), dass die Knotenmenge nicht leer ist. Und wir beschränken uns in dieser Vorlesung auf *endliche* Knotenmengen. Die Kantenmenge darf leer sein.

gerichteter Graph
Knoten
Kanten

Typischerweise stellt man Graphen graphisch dar. Statt zu schreiben

$$V = \{0, 1, 2, 3, 4, 5\}$$
$$E = \{(0, 1), (0, 3), (1, 2), (1, 3), (4, 5), (5, 4)\}$$

malt man lieber Abbildungen wie diese:

Abbildung 11.1: zweimal der gleiche Graph: links ohne Angabe der Knotenidentitäten, rechts mit

Ob man die Knoten als anonyme dicke Punkte oder Kringel darstellt wie auf der linken Seite in Abbildung 11.1, oder ob man jeweils das entsprechende Element von V mit notiert wie auf der rechten Seite in Abbildung 11.1, hängt von den

Umständen ab. Beides kommt vor. Eine Kante (x, y) wird durch einen Pfeil von dem Knoten x in Richtung zu dem Knoten y dargestellt. Wenn es eine Kante $(x, y) \in E$ gibt, dann sagt man auch, die Knoten x und y seien *adjazent*.

adjazente Knoten

Außerdem ist die Anordnung der Knoten in der Darstellung irrelevant. Abbildung 11.2 zeigt den gleichen Graphen wie Abbildung 11.1:

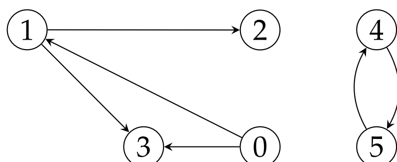


Abbildung 11.2: eine andere Zeichnung des Graphen aus Abbildung 11.1

Wir wollen noch zwei weitere Beispiele betrachten. Im ersten sei $G = (V, E)$ definiert durch

- $V = \{1\} \left(\bigcup_{i=0}^{2} \{0, 1\}^i\right)$ und
- $E = \{(w, wx) \mid x \in \{0, 1\} \land w \in V \land wx \in V\}$.

Schreibt man die beiden Mengen explizit auf, ergibt sich

- $V = \{1, 10, 11, 100, 101, 110, 111\}$
- $E = \{(1, 10), (1, 11), (10, 100), (10, 101), (11, 110), (11, 111)\}$

Im einem Bild kann man diesen Graphen so hinmalen:

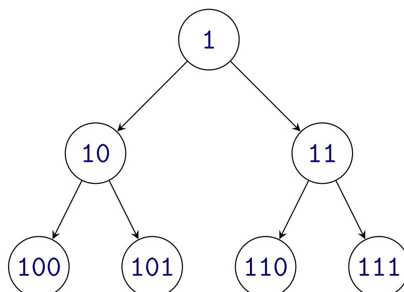


Abbildung 11.3: ein Graph, der ein sogenannter Baum ist

Als zweites Beispiel wollen wir den Graphen $G = (V, E)$ betrachten, für den gilt:

- $V = \{0, 1\}^3$
- $E = \{(xw, wy) \mid x, y \in \{0, 1\} \land w \in \{0, 1\}^2\}$

Die Knotenmenge ist also einfach $V = \{000, 001, 010, 011, 100, 101, 110, 111\}$. Die Kantenmenge wollen wir gar nicht mehr vollständig aufschreiben. Nur zwei Kanten schauen wir uns beispielhaft kurz an:

- Wählt man $x = y = \texttt{0}$ und $w = \texttt{00}$ dann ist laut Definition von E also $(\texttt{000}, \texttt{000})$ eine Kante.
- Wählt man $x = \texttt{0}$, $y = \texttt{1}$ und $w = \texttt{10}$ dann ist laut Definition von E also $(\texttt{010}, \texttt{101})$ eine Kante.

Graphisch kann man diesen Graphen z. B. wie in Abbildung 11.4 darstellen. Es handelt sich um einen sogenannten De Bruijn-Graphen. Wie man sieht, können

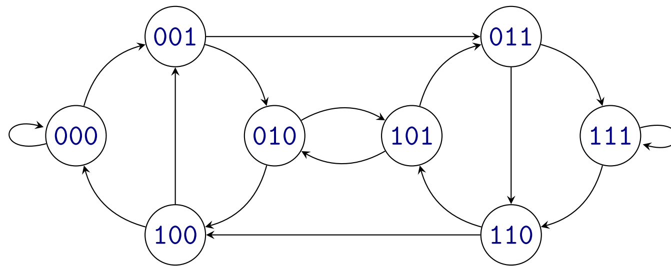


Abbildung 11.4: Der de Bruijn-Graphen mit 8 Knoten

bei einer Kante Start- und Zielknoten gleich sein. Eine solche Kante, die also von der Form $(x, x) \in E$ ist, heißt auch eine *Schlinge*. Manchmal ist es bequem, davon auszugehen, dass ein Graph keine Schlingen besitzt. Ein solcher Graph heißt *schlingenfrei*.

Schlinge

schlingenfrei

Ein ganz wichtiger Begriff ist der eines Teilgraphen eines gegebenen Graphen. Wir sagen, dass $G' = (V', E')$ ein *Teilgraph* von $G = (V, E)$ ist, wenn $V' \subseteq V$ ist und $E' \subseteq E \cap V' \times V'$. Knoten- und Kantenmenge von G' müssen also Teilmenge von V resp. E sein, und die Endpunkte jeder Kante von E' müssen auch zu V' gehören.

Teilgraph

Als Beispiel zeigen wir einen Teilgraphen des oben dargestellten de Bruijn-Graphen:

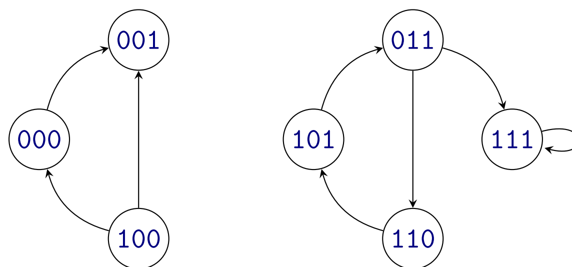


Abbildung 11.5: Ein Teilgraph des de Bruijn-Graphen aus Abbildung 11.4

 © worsch 2008–2010

11.1.2 Pfade und Erreichbarkeit

Im folgenden benutzen wir die Schreibweise $M^{(+)}$ für die Menge aller nichtleeren Listen, deren Elemente aus M stammen. Und solch eine Liste notieren wir in der Form $(m_1, m_2 \dots, m_k)$.

Pfad

Länge eines Pfades

Ein weiteres wichtiges Konzept sind Pfade. Wir wollen sagen, dass eine nichtleere Liste $p = (v_0, \dots, v_n) \in V^{(+)}$ von Knoten ein *Pfad* in einem gerichteten Graphen $G = (V, E)$ ist, wenn für alle $i \in \mathbb{G}_n$ gilt: $(v_i, v_{i+1}) \in E$. Die Anzahl $n = |p| - 1$ der Kanten (!) heißt die *Länge* des Pfades. Auch wenn wir Pfade als Knotenlisten definiert haben, wollen wir davon sprechen, dass „in einem Pfad" $(v_0, \dots, v_n)$ Kanten vorkommen; was wir damit meinen sind die Kanten (v_i, v_{i+1}) für $i \in \mathbb{G}_n$.

erreichbar

Wenn $p = (v_0, \dots, v_n)$ ein Pfad ist, sagt man auch, dass v_n von v_0 aus *erreichbar* ist.

wiederholungsfreier Pfad

Ein Pfad $(v_0, \dots, v_n)$ heißt *wiederholungsfrei*, wenn gilt: Die Knoten $v_0, \dots, v_{n-1}$ sind paarweise verschieden und die Knoten $v_1, \dots, v_n$ sind paarweise verschieden. Die beiden einzigen Knoten, die gleich sein dürfen, sind also v_0 und v_n.

geschlossener Pfad

Ein Pfad mit $v_0 = v_n$ heißt *geschlossen*.

Zyklus

einfacher Zyklus

Ein geschlossener Pfad $(v_0, \dots, v_n)$ heißt auch *Zyklus*. Er heißt ein *einfacher Zyklus*, wenn er außerdem wiederholungsfrei ist. Zum Beispiel ist in Abbildung 11.5 der Pfad (011, 110, 101, 011) ein einfacher Zyklus der Länge 3. Manchmal bezeichnet man auch den Teilgraphen, der aus den verschiedenen Knoten des Zyklus und den dazugehörigen Kanten besteht, als Zyklus.

Wie man in Abbildung 11.5 auch sieht, kann es unterschiedlich lange Pfade von einem Knoten zu einem anderen geben: von 100 nach 001 gibt es einen Pfad der Länge 1 und einen Pfad der Länge 2. Und es gibt in diesem Graphen auch Knotenpaare, bei denen gar kein Pfad von einem zum anderen existiert.

streng zusammenhängender Graph

Ein gerichteter Graph heißt *streng zusammenhängend*, wenn für jedes Knotenpaar $(x, y) \in V^2$ gilt: Es gibt in G einen Pfad von x nach y. Zum Beispiel ist der de Bruijn-Graph aus Abbildung 11.4 streng zusammenhängend (wie man durch Durchprobieren herausfindet), aber der Teilgraph aus Abbildung 11.5 eben nicht.

Baum

Wurzel

Sozusagen eine sehr viel einfachere Variante von Zusammenhang ist bei Graphen mit einer speziellen Struktur gegeben, die an ganz vielen Stellen in der Informatik immer wieder auftritt. Auch in dieser Vorlesung kam sie schon mehrfach vor: Bäume. Ein (gerichteter) *Baum* ist ein Graph $G = (V, E)$, in dem es einen Knoten $r \in V$ gibt mit der Eigenschaft: Für jeden Knoten $x \in V$ gibt es in G genau einen Pfad von r nach x. Wir werden uns gleich kurz überlegen, dass es nur *einen* Knoten r mit der genannten Eigenschaft geben kann. Er heißt die *Wurzel* des Baumes. In Abbildung 11.3 ist ein Baum dargestellt, dessen Wurzel 1 ist.

11.1 Lemma. Die Wurzel eines gerichteten Baumes ist eindeutig.

11.2 Beweis. Angenommen, r und r' wären verschiedene Wurzeln des gleichen Baumes. Dann gäbe es

- einen Pfad von r nach r', weil r Wurzel ist, und
- einen Pfad von r' nach r, weil r' Wurzel ist.

Wenn $r \neq r'$ ist, haben beide Pfade eine Länge > 0. Durch „Hintereinanderhängen" dieser Pfade ergäbe sich ein Pfad von r nach r, der vom Pfad (r) der Länge 0 verschieden wäre. Also wäre der Pfad von r nach r gar nicht eindeutig. ■

Es kommt immer wieder vor, dass man darüber reden will, wieviele Kanten in einem gerichteten Graphen $G = (V, E)$ zu einem Knoten hin oder von ihn weg führen. Der *Eingangsgrad* eines Knoten y wird mit $d^-(y)$ bezeichnet und ist definiert als

Eingangsgrad

$$d^-(y) = |\{x \mid (x, y) \in E\}|$$

Analog nennt man

$$d^+(x) = |\{y \mid (x, y) \in E\}|$$

den *Ausgangsgrad* eines Knotens x. Die Summe $d(x) = d^-(x) + d^+(x)$ heißt auch der *Grad* des Knotens x.

Ausgangsgrad

Grad

In einem gerichteten Baum gibt es immer Knoten mit Ausgangsgrad 0. Solche Knoten heißen *Blätter*.

Blatt eines Baums

11.1.3 Isomorphie von Graphen

Wir haben eingangs davon gesprochen, dann man von Graphen manchmal nur die „Struktur" darstellt, aber nicht, welcher Knoten wie heißt. Das liegt daran, dass manchmal eben nur die Struktur interessiert und man von allem weiteren abstrahieren will. Hier kommt der Begriff der Isomorphie von Graphen zu Hilfe. Ein Graph $G_1 = (V_1, E_1)$ heißt *isomorph* zu einem Graphen $G_2 = (V_2, E_2)$, wenn es eine bijektive Abbildung $f : V_1 \to V_2$ gibt mit der Eigenschaft:

isomorph

$$\forall x \in V_1 : \forall y \in V_1 : (x, y) \in E_1 \iff (f(x), f(y)) \in E_2$$

Mit anderen Worten ist f einfach eine „Umbenennung" der Knoten. Die Abbildung f heißt dann auch ein (Graph-)*Isomorphismus*.

Isomorphismus

Man kann sich überlegen:

- Wenn G_1 isomorph zu G_2 ist, dann ist auch G_2 isomorph zu G_1: Die Umkehrabbildung zu f leistet das Gewünschte.
- Jeder Graph ist isomorph zu sich selbst: Man wähle $f = I_V$.
- Wenn G_1 isomorph zu G_2 ist (dank f) und G_2 isomorph zu G_3 (dank g), dann ist auch G_1 isomorph zu G_3: Man betrachte die Abbildung $g \circ f$.

11.1.4 Ein Blick zurück auf Relationen

Vielleicht haben Sie bei der Definition von gerichteten Graphen unmittelbar gesehen, dass die Kantenmenge E ja nichts anderes ist als eine binäre Relation auf der Knotenmenge V (vergleiche Abschnitt 3.2). Für solche Relationen hatten wir in Abschnitt 8.3 Potenzen definiert. Im folgenden wollen wir uns klar machen, dass es eine enge Verbindung gibt zwischen den Relationen E^i für $i \in \mathbb{N}_0$ und Pfaden der Länge i im Graphen. Daraus wird sich dann auch eine einfache Interpretation von E^* ergeben.

Betrachten wir zunächst den Fall $i = 2$.

Ein Blick zurück in Abschnitt 8.3 zeigt, dass $E^2 = E \circ E^1 = E \circ E \circ \mathrm{I} = E \circ E$ ist. Nach Definition des Relationenproduktes ist

$$E \circ E = \{(x,z) \in V \times V \mid \exists y \in V : (x,y) \in E \wedge (y,z) \in E\}$$

Ein Pfad der Länge 2 ist eine Knotenliste $p = (v_0, v_1, v_2)$ mit der Eigenschaft, dass $(v_0, v_1) \in E$ ist und ebenso $(v_1, v_2) \in E$.

Wenn ein Pfad $p = (v_0, v_1, v_2)$ vorliegt, dann ist also gerade $(v_0, v_2) \in E^2$. Ist umgekehrt $(v_0, v_2) \in E^2$, dann gibt es einen Knoten v_1 mit $(v_0, v_1) \in E$ und $(v_1, v_2) \in E$. Und damit ist dann (v_0, v_1, v_2) ein Pfad im Graphen, der offensichtlich Länge 2 hat.

Also ist ein Paar von Knoten *genau dann* in der Relation E^2, *wenn* die beiden durch einen Pfad der Länge 2 miteinander verbunden sind.

Analog, aber noch einfacher, kann man sich überlegen, dass ein Paar von Knoten genau dann in der Relation $E^1 = E$, wenn die beiden durch einen Pfad der Länge 1 miteinander verbunden sind.

Und die entsprechende Aussage für $i = 0$ gilt auch: Sind zwei Knoten x und y in der Relation $E^0 = \mathrm{I}_V$, dann ist $x = y$ und folglich ist in der Tat (x) ein Pfad der Länge 0 von x nach $y = x$. Umgekehrt: Ein Pfad der Länge 0 von x nach y ist von der Form (z) und fängt mit $z = x$ an und hört mit $z = y$ auf, also ist $x = y$, und folglich $(x,y) = (x,x) \in \mathrm{I}_V = E^0$.

Damit haben wir uns explizit davon überzeugt, dass für alle $i \in \mathbb{G}_3$ gilt: Ein Paar von Knoten ist genau dann in der Relation E^i, wenn die beiden Knoten durch einen Pfad der Länge i miteinander verbunden sind. Und es ist wohl klar, dass man durch vollständige Induktion beweisen kann, dass diese Aussage sogar für alle $i \in \mathbb{N}_0$ gilt.

11.3 Lemma. Es sei $G = (V, E)$ ein gerichteter Graph. Für alle $i \in \mathbb{N}_0$ gilt: Ein Paar von Knoten (x,y) ist genau dann in der Relation E^i, wenn x und y in G durch einen Pfad der Länge i miteinander verbunden sind.

Damit gibt es nun auch eine anschauliche Intepretation von E^*, das wir ja definiert hatten als Vereinigung aller E^i für $i \in \mathbb{N}_0$:

11.4 Korollar. Es sei $G = (V, E)$ ein gerichteter Graph. Ein Paar von Knoten (x, y) ist genau dann in der Relation E^*, wenn x und y in G durch einen Pfad (evtl. der Länge 0) miteinander verbunden sind.

Folglich gilt auch:

11.5 Korollar. Ein gerichteter Graph $G = (V, E)$ ist genau dann streng zusammenhängend, wenn $E^* = V \times V$ ist.

11.2 UNGERICHTETE GRAPHEN

Manchmal hat man mit Graphen zu tun, bei denen für *jede* Kante $(x, y) \in E$ stets $(y, x) \in E$ auch eine Kante in E ist. In einem solchen Fall ist meist angebracht, üblich und übersichtlicher, in der graphischen Darstellung nicht einen Pfeil von x nach y und einen Pfeil von y nach x zu zeichnen, sondern die beiden Knoten einfach durch *einen* Strich (*ohne* Pfeilspitzen) miteinander zu verbinden. Und man spricht dann auch nur von *einer* Kante. Üblicherweise passt man dann auch die

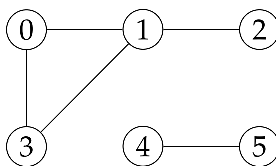


Abbildung 11.6: ein ungerichteter Graph

Formalisierung an und definiert: Ein *ungerichteter Graph* ist eine Struktur $U = (V, E)$ mit einer endlichen nichtleeren Menge V von Knoten und einer Menge E von Kanten, wobei $E \subseteq \{ \{x, y\} \mid x \in V \wedge y \in V\}$. Analog zum gerichteten Fall heißen zwei Knoten eines ungerichteten Graphen *adjazent*, wenn sie durch eine Kante miteinander verbunden sind.

ungerichteter Graph

adjazent

Eine Kante, bei der Start- und Zielknoten gleich sind, heißt wie bei gerichteten Graphen eine *Schlinge*. In der Formalisierung schlägt sich das so nieder, dass aus $\{x, y\}$ einfach $\{x\}$ wird. Wenn ein ungerichteter Graph keine Schlingen besitzt, heißt er auch wieder *schlingenfrei*.

Schlinge

schlingenfrei

Wir sagen, dass $U' = (V', E')$ ein *Teilgraph* eines ungerichteten Graphen $U =$

Teilgraph

(V, E) ist, wenn $V' \subseteq V$ ist und $E' \subseteq E \cap \{ \{x, y\} \mid x, y \in V'\}$. Knoten- und Kantenmenge von U' müssen also Teilmenge von V resp. E sein, und die Endpunkte jeder Kante von E' müssen auch zu V' gehören.

Bei gerichteten Graphen haben wir von Pfaden geredet. Etwas entsprechendes wollen wir auch bei ungerichteten Graphen können. Aber da Kanten anders formalisiert wurden, wird auch eine neue Formalisierung des Analogons zu Pfaden benötigt. Wir wollen sagen, dass eine nichtleere Liste $p = (v_0, \ldots, v_n) \in V^{(+)}$ von Knoten ein *Weg* in einem ungerichteten Graphen $G = (V, E)$ ist, wenn für alle $i \in \mathbb{G}_n$ gilt: $\{v_i, v_{i+1}\} \in E$. Die Anzahl $n = |p| - 1$ der Kanten (!) heißt die *Länge* des Weges.

Weg
Länge eines Weges

Bei gerichteten Graphen war E eine binäre Relation auf V und infolge dessen waren alle E^i definiert. Bei ungerichteten Graphen ist E nichts, was unter unsere Definition von binärer Relation fällt. Also ist auch E^i nicht definiert. Das ist schade und wir beheben diesen Mangel umgehend: Zur Kantenmenge E eines ungerichteten Graphen $U = (V, E)$ definieren wir die *Kantenrelation* $E_g \subseteq V \times V$ vermöge:

Kantenrelation

$$E_g = \{(x, y) \mid \{x, y\} \in E\} .$$

Damit haben wir eine Relation auf V. Und folglich auch einen gerichteten Graphen $G = (V, E_g)$ mit der gleichen Knotenmenge V wie U. Und wenn in U zwei Knoten x und y durch eine Kante miteinander verbunden sind, dann gibt es in G die (gerichtete) Kante von x nach y und umgekehrt auch die Kante von y nach x (denn $\{x, y\} = \{y, x\}$). Man sagt auch, dass (V, E_g) der zu (V, E) gehörige gerichtete Graph ist.

Man sagt, ein ungerichteter Graph (V, E) sei *zusammenhängend*, wenn der zugehörige gerichtete Graph (V, E_g) streng zusammenhängend ist.

zusammenhängender ungerichteter Graph

Der Übergang von ungerichteten zu gerichteten Graphen ist auch nützlich, um festzulegen, wann zwei ungerichtete Graphen die „gleiche Struktur" haben: $U_1 = (V_1, E_1)$ und $U_2 = (V_2, E_2)$ heißen *isomorph*, wenn die zugehörigen gerichteten Graphen U_{1g} und U_{2g} isomorph sind. Das ist äquivalent dazu, dass es eine bijektive Abbildung $f : V_1 \to V_2$ gibt mit der Eigenschaft:

isomorph

$$\forall x \in V_1 : \forall y \in V_1 : \{x, y\} \in E_1 \iff \{f(x), f(y)\} \in E_2$$

Auch für ungerichtete Graphen ist Isomorphie eine Äquivalenzrelation.

Eben war es bequem, von einem ungerichteten zu dem zugehörigen gerichteten Graphen überzugehen. Die umgekehrte Richtung ist manchmal auch ganz praktisch. Ist $G = (V, E)$ ein gerichteter Graph, dann definieren wir $E_u = \{ \{x, y\} \mid (x, y) \in E\}$ und nennen $U = (V, E_u)$ den zu G gehörigen ungerichteten Graphen. Er entsteht aus G also sozusagen dadurch, dass man in G alle Pfeilspitzen „entfernt" (oder „vergisst" oder wie auch immer Sie das nennen wollen).

Damit definieren wir nun, was wir im ungerichteten Fall als Baum bezeichnen wollen: Ein ungerichteter Graph $U = (V, E)$ heißt ein *Baum*, wenn es einen gerichteten Baum $G = (V, E')$ gibt mit $E = E'_u$. Abbildung 11.7 zeigt zwei ungerichtete Bäume.

ungerichteter Baum

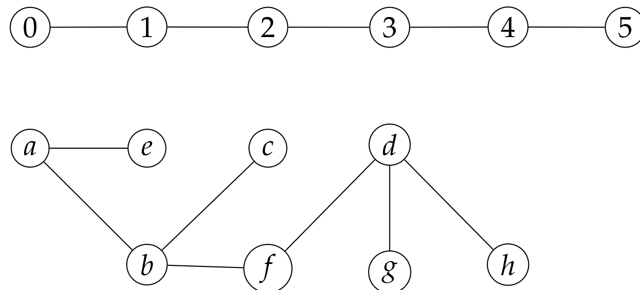


Abbildung 11.7: zwei ungerichtete Bäume

Man beachte einen Unterschied zwischen gerichteten und ungerichteten Bäumen. Im gerichteten Fall ist die Wurzel leicht zu identifizieren: Es ist der einzige Knoten, von dem Pfade zu den anderen Knoten führen. Im ungerichteten Fall ist das anders: Von jedem Knoten führt ein Weg zu jedem anderen Knoten. Nichtsdestotrotz ist manchmal „klar“, dass ein Knoten die ausgezeichnete Rolle als Wurzel spielt. Im Zweifelsfall sagt man es eben explizit dazu.

Auch für ungerichtete Graphen führt man den Grad eines Knotens ein (aber nicht getrennt Eingangs- und Ausgangsgrad). In der Literatur findet man zwei unterschiedliche Definitionen. Wir wollen in dieser Vorlesung die folgende benutzen: Der *Grad* eines Knotens $x \in V$ ist

Grad

$$d(x) = |\{y \mid y \neq x \land \{x, y\} \in E\}| + \begin{cases} 2 & \text{falls } \{x, x\} \in E \\ 0 & \text{sonst} \end{cases}$$

11.2.1 Anmerkung zu Relationen

Die Kantenrelation eines ungerichteten Graphen hat eine Eigenschaft, die auch in anderen Zusammenhängen immer wieder auftritt. Angenommen $(x, y) \in E_g$. Das kann nur daher kommen, dass $\{x, y\} \in E$ ist. Dann ist aber auch „automatsich“ $(y, x) \in E_g$. Also: Wenn $(x, y) \in E_g$, dann $(y, x) \in E_g$. Dieser Eigenschaft, die eine Relation haben kann, geben wir einen Namen:

Eine Relation $R \subseteq M \times M$ heißt *symmetrisch* wenn für alle $x \in M$ und $y \in M$ gilt:

symmetrische Relation

$$(x, y) \in R \Longrightarrow (y, x) \in R\ .$$

Und wir wollen an dieser Stelle schon einmal erwähnen, dass eine Relation, die reflexiv, transitiv und symmetrisch ist, eine sogenannte *Äquivalenzrelation* ist.

Äquivalenzrelation

Wir habe weiter vorne in dieser Einheit auch eine erste interessante Äquivalenzrelation kennengelernt: die Isomorphie von Graphen. Man lese noch einmal aufmerksam die drei Punkte der Aufzählung am Ende von Unterabschnitt 11.1.3.

11.3 GRAPHEN MIT KNOTEN- ODER KANTENMARKIERUNGEN

Häufig beinhaltet die Graphstruktur nicht die Gesamtheit an Informationen, die von Interesse sind. Zum Beispiel sind bei Ableitungsbäumen die Nichtterminalsymbole an den inneren Knoten und die Terminalsymbole und ε an den Blättern wesentlich. Bei Huffman-Bäumen haben wir Markierungen an Kanten benutzt, um am Ende die Codierungen von Symbolen herauszufinden.

Allgemein wollen wir davon sprechen, dass ein Graph mit Knotenmarkierungen oder *knotenmarkierter Graph* vorliegt, wenn zusätzlich zu $G = (V, E)$ auch noch eine Abbildung $m_V : V \to M_V$ gegeben ist, die für jeden Knoten v seine Markierung $m_V(v)$ festlegt. Die Wahl der Menge M_V der möglichen Knotenmarkierungen ist abhängig vom einzelnen Anwendungsfall. Bei Huffman-Bäumen hatten wir als Markierungen natürliche Zahlen (nämlich die Häufigkeiten von Symbolmengen); es war also $M_V = \mathbb{N}_+$.

knotenmarkierter Graph

Aus Landkarten, auf denen Länder mit ihren Grenzen eingezeichnet sind, kann man auf verschiedene Weise Graphen machen. Hier ist eine Möglichkeit: Jedes Land wird durch einen Knoten des (ungerichteten) Graphen repräsentiert. Eine Kante verbindet zwei Knoten genau dann, wenn die beiden repräsentierten Länder ein Stück gemeinsame Grenzen haben. Nun ist auf Landkarten üblicherweise das Gebiet jedes Landes in einer Farbe eingefärbt, und zwar so, dass benachbarte Länder verschiedene Farben haben (damit man sie gut unterscheiden kann). Die Zuordnung von Farben zu Knoten des Graphen ist eine Knotenmarkierung. (Man spricht auch davon, dass der Graph gefärbt sei.) Wofür man sich interessiert, sind „legale" Färbungen, bei denen adjazente Knoten verschiedene Farben haben: $\{x, y\} \in E \implies m_V(x) \neq m_V(y)$. Ein Optimierungsproblem besteht dann z. B. darin, herauszufinden, welches die minimale Anzahl von Farben ist, die ausreicht, um den Graphen legal zu färben. Solche Färbungsprobleme müssen nicht nur (vielleicht) von Verlagen für Atlanten gelöst werden, sondern sie werden auch etwa von modernen Compilern beim Übersetzen von Programmen bearbeitet.

kantenmarkierter Graph

Ein Graph mit Kantenmarkierungen oder *kantenmarkierter Graph* liegt vor, wenn zusätzlich zu $G = (V, E)$ auch noch eine Abbildung $m_E : E \to M_E$ gegeben ist,

die für jede Kante $e \in E$ ihre Markierung $m_E(e)$ festlegt. Die Wahl der Menge der Markierungen ist abhängig vom einzelnen Anwendungsfall. Bei Huffman-Bäumen hatten wir als Markierungen an den Kanten die Symbole `0` und `1`, es war also $M_E = \{0, 1\}$.

11.3.1 Gewichtete Graphen

Ein Spezialfall von markierten Graphen sind *gewichtete Graphen*. Bei ihnen sind die Markierungen z. B. Zahlen. Nur diesen Fall werden wir im folgenden noch ein wenig diskutieren. Im allgemeinen sind es vielleich auch mal Vektoren von Zahlen o. ä.; jedenfalls soll die Menge der Gewichte irgendeine Art von algebraischer Struktur aufweisen, so dass man „irgendwie rechnen“ kann.

gewichtete Graphen

Als Motiviation können Sie sich vorstellen, dass man z. B. einen Teil des Straßen- oder Eisenbahnnetzes modelliert. Streckenstücke ohne Abzweigungen werden als einzelne Kanten repräsentiert. Das Gewicht jeder Kante könnte dann z. B. die Länge des entsprechenden echten Streckenstückes sein oder die dafür benötigte Fahrzeit. Oder man stellt sich vor, man hat einen zusammenhängenden Graphen gegeben. Die Kante stellen mögliche Verbindungen dar und die Gewichte sind Baukosten. Die Aufgabe bestünde dann darin, einen Teilgraphen zu finden, der immer noch zusammenhängend ist, alle Knoten umfasst, aber möglichst wenige, geeignet gewählte Kanten, so dass die Gesamtkosten für den Bau minimal werden. Für den Fall eines Stromnetzes in der damaligen Tschechoslowakei war dies die tatsächliche Aufgabe, die in den Zwanziger Jahren O. Borůvka dazu brachte, seinen Algorithmus für minimale aufspannende Bäume zu entwickeln. Ihnen werden Graphalgorithmen noch an vielen Stellen im Studium begegnen.

Eine andere Interpretation von Kantengewichten kann man bei der Modellierung eines Rohrleitungsnetzes, sagen wir eines Wasserleitungsnetzes, benutzen: Das Gewicht einer Kante ist dann vielleicht der Querschnitt des entsprechenden Rohres; das sagt also etwas über Transportkapazitäten aus. Damit wird es sinnvoll zu fragen, welchen Fluss man maximal („über mehrere Kanten parallel“) erzielen kann, wenn Wasser von einem Startknoten s zu einem Zielknoten t transportiet werden soll.

12 ERSTE ALGORITHMEN IN GRAPHEN

In dieser Einheit wollen wir beginnen, Algorithmen auch unter quantitativen Gesichtspunkten zu betrachten.

Als „Aufhänger“ werden wir eine vereinfachte Problemstellung betrachten, die mit einer der am Ende der Einheit 11 über Graphen aufgezählten verwandt ist: Man finde heraus, ob es in einem gegebenen gerichteten Graphen einen Pfad von einem gegebenen Knoten i zu einem gegebenen Knoten j gibt.

Wir beginnen in Abschnitt 12.1 mit der Frage, wie man denn Graphen im Rechner repräsentiert. In 12.2 nähern wir uns dann langsam dem Erreichbarkeitsproblem, indem wir uns erst einmal nur für Pfade der Länge 2 interessieren. Das führt auch zu den Konzepten Matrixaddition und Matrixmultiplikation. Auf der Matrizenrechnung aufbauen beginnen wir dann in 12.3 mit einem ganz naiven Algorithmus und verbessern ihn in zwei Schritten. Einen der klassischen Algorithmen, den von Warshall, für das Problem, werden wir in Abschnitt 12.4 kennenlernen.

Nachdem wir uns in dieser Einheit beispielhaft auch mit dem Thema beschäftigt haben werden, wie man — in einem gewissen Sinne — die Güte eines Algorithmus quantitativ erfassen kann, werden wir das in der nachfolgenden Einheit 13 über quantitative Aspekte von Algorithmen an weiteren Beispielen aber auch allgemein etwas genauer beleuchten.

12.1 REPRÄSENTATION VON GRAPHEN IM RECHNER

In der Vorlesung über Programmieren haben Sie schon von Klassen, Objekten und Attributen gehört und Arrays kennengelernt. Das kann man auf verschiedene Arten nutzen, um z. B. Graphen im Rechner zu repräsentieren. Ein erster Ansatz in Java könnte z. B. so aussehen:

```
class Vertex {
   String name;
}

class Graph {
   Vertex[] vertices;
   Edge[] edges;
}
```

```
class Edge {
   Vertex start;
   Vertex end;
}
```

Dabei hat man aber mehr hingeschrieben als man „eigentlich" will, denn die Knoten (und auch die Kanten) sind durch die Nummerierung der Komponenten der Arrays total angeordnet worden. Das ist bei den Mengen der mathematischen Definition nicht der Fall.

Aber es schadet nicht. Da man die Nummern aber sowieso schon hat, macht man, zumindest wenn man besonders kurz und übersichtlich sein will, den Schritt und sagt, dass die Identitäten der Knoten einfach die Zahlen eines Anfangsstückes der natürlichen Zahlen sind. Solange man sich mit Problemen beschäftigt, die unter Isomorphie invariant sind, ergeben sich hierdurch keine Probleme. Deswegen ist für uns im folgenden bei allen Graphen $V = \mathbb{G}_n$ für ein $n \geq 1$.

```
class Vertex {
   int id;
}

class Edge {
   Vertex start;
   Vertex end;
}

class Graph {
   Vertex[] vertices;
   Edge[] edges;
}
```

Gelegentlich verwendet man als Knotennummern auch Anfangsstücke der positiven ganzen Zahlen (also ohne Null). Lassen Sie sich von solchen kleinen technischen Details nicht verunsichern. Man macht, was einem gerade am besten erscheint.

Wenn man Graphen in Java wie oben skizziert implementieren würde, dann könnte man bei einer gegebenen Kante leicht auf deren Anfangs- und Endknoten zugreifen. Wie Sie bald sehen werden, will man aber mitunter umgekehrt zu einem gegebenen Knoten v z. B. auf die ihn verlassenden Kanten zugreifen. Das wäre aber nur umständlich möglich: Man müsste systematisch alle Kanten darauf hin überprüfen, ob sie bei v starten.

Es gibt (neben anderen) zwei gängige Methoden, dieses Problem zu beseitigen. Die eine besteht darin, zu jedem Knoten eine Liste der ihn verlassenden Kanten oder der über solche Kanten erreichbaren Nachbarknoten mitzuführen. Wenn man diese Liste als Array implementiert, dann wäre

```
class Vertex {
   int id;
   Vertex[] neighbors;
}

class Graph {
   Vertex[] vertices;
}
```

Man spricht dann davon, dass für jeden Knoten die *Adjazenzliste* vorhanden ist. *Adjazenzliste*

Wenn man mit kantenmarkierten Graphen arbeiten muss, benutzt man statt dessen lieber die *Inzidenzlisten*. Das ist für einen Knoten die Liste der Kanten, die ihn als einen Endpunkt haben. *Inzidenzliste*

Wir wollen im folgenden aber eine andere Methode benutzen, um die Beziehungen zwischen Knoten zu speichern. Wenn man zu einem Knoten u womöglich oft und schnell herausfinden möchte, ob ein Knoten v Nachbar von u ist, dann ist es bequem, wenn man das immer leicht herausfinden kann. Man müsste dann (unter Umständen) nur noch die Klassen für einzelne Knoten und einen Graphen implementieren, z. B. so:

```
class Vertex {
   int id;
   boolean[] is_connected_to;
}
```

```
class Graph {
    Vertex[] vertices;
}
```

Für einen Knoten, also ein Objekt u der Klasse `Vertex`, wäre `is_connected_to` also ein Feld mit genau so vielen Komponenten wie es Knoten im Graphen gibt. Und u`.is_connected_to[`v`.id]` sei genau dann `true`, wenn eine Kante von u nach v existiert, und ansonsten `false` .

Betrachten wir als Beispiel den Graphen aus Abbildung 12.1:

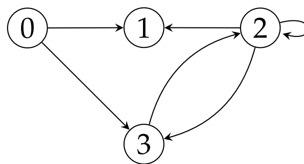


Abbildung 12.1: Ein kleiner Beispielgraph

Für das Objekt u der Klasse `Vertex`, das den Knoten 0 repräsentiert, würde z. B. gelten:

u.id	u.is_connected_to			
0	false	true	false	true
	0	1	2	3

Schreibt man das für alle vier Knoten untereinander, erhält man:

u.id	u.is_connected_to			
0	false	true	false	true
1	false	false	false	false
2	false	true	true	true
3	false	false	true	false
	0	1	2	3

Adjazenzmatrix

Wenn man in dieser zweidimensionalen Tabelle nun noch `false` durch 0 und `true` durch 1 ersetzt, erhält man die sogenannte *Adjazenzmatrix* des Graphen.

Manche haben Matrizen inzwischen in der Vorlesung „Lineare Algebra" kennengelernt, andere haben zumindest schon ein Beispiel gesehen, in dem ein Graph als zweidimensionale Tabelle repräsentiert wurde. Im allgemeinen können Zeilenzahl m und Spaltenzahl n einer Matrix A verschieden sein. Man spricht dann von einer $m \times n$-Matrix. Die einzelnen Einträge in Matrizen werden in dieser Vorlesung immer Zahlen sein. Für den Eintrag in Zeile i und Spalte j von A schreiben wir auch A_{ij} (oder manchmal $(A)_{ij}$ o. ä.).

Für die Menge aller $m \times n$-Matrizen, deren Einträge alle aus einer Menge M stammen, schreiben wir gelegentlich $M^{m \times n}$.

Typischerweise notiert man eine Matrix ohne die ganzen senkrechten und waagerechten Striche, aber mit großen runden (oder manchmal auch eckigen) Klammern außen herum. Wenn es hilfreich ist, notiert man außerhalb der eigentlichen Matrix auch die Nummerierung der Zeilen bzw Spalten, wie es z. B. in Abbildung 12.2 gemacht ist.

Die Adjazenzmatrix eines gerichteten Graphen $G = (V, E)$ mit n Knoten ist eine $n \times n$-Matrix A mit der Eigenschaft:

$$A_{ij} = \begin{cases} 1 & \text{falls } (i,j) \in E \\ 0 & \text{falls } (i,j) \notin E \end{cases}$$

Als Beispiel ist in Abbildung 12.2 noch einmal der Graph mit vier Knoten und nun auch die zugehörige Adjazenzmatrix angegeben.

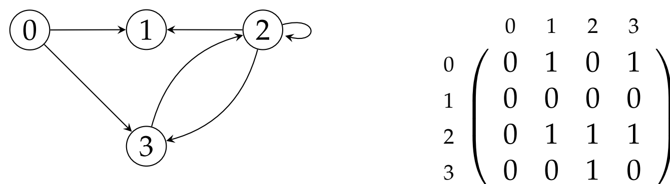


Abbildung 12.2: Ein Graph und seine Adjazenzmatrix

Im Falle eines ungerichteten Graphen $U = (V, E)$ versteht man unter seiner Adjazenzmatrix die des zugehörigen gerichteten Graphen $G = (V, E_g)$.

Allgemein kann man jede binäre Relation $R \subseteq M \times M$ auf einer endlichen Menge M mit n Elementen durch eine $n \times n$-Matrix $A(R)$ repräsentieren, indem man definiert:

$$(A(R))_{ij} = \begin{cases} 1 & \text{falls } (i,j) \in R \quad \text{d. h. also } iRj \\ 0 & \text{falls } (i,j) \notin R \quad \text{d. h. also } \neg(iRj) \end{cases}$$

(Dabei gehören zu verschiedenen Relationen auf der gleichen Menge M verschiedene Matrizen und umgekehrt.)

So, wie man die Kantenrelation E eines gerichteten Graphen als Adjazenzmatrix darstellen kann, kann man natürlich auch jede andere Relation durch eine entsprechende Matrix darstellen, z. B. die „Erreichbarkeitsrelation" E^*. Die zugehörige Matrix W eines Graphen wird üblicherweise *Wegematrix* genannt. Sie hat also die Eigenschaft:

Wegematrix

$$W_{ij} = \begin{cases} 1 & \text{falls } (i,j) \in E^* \\ 0 & \text{falls } (i,j) \notin E^* \end{cases}$$
$$= \begin{cases} 1 & \text{falls es in } G \text{ einen Pfad von } i \text{ nach } j \text{ gibt} \\ 0 & \text{falls es in } G \text{ keinen Pfad von } i \text{ nach } j \text{ gibt} \end{cases}$$

Im folgenden wollen wir uns mit dem algorithmischen Problem beschäftigen, zu gegebener Adjazenzmatrix A die zugehörige Wegematrix W zu berechnen.

12.2 Berechnung der 2-Erreichbarkeitsrelation und Rechnen mit Matrizen

Es sei A die Adjazenzmatrix eines Graphen $G = (V, E)$; Abbildung 12.3 zeigt ein Beispiel.
Wir interessieren uns nun zum Beispiel für die Pfade der Länge 2 von Knoten 2 zu Knoten 4. Wie man in dem Graphen sieht, gibt es genau zwei solche Pfade: den über Knoten 1 und den über Knoten 6.

Wie findet man „systematisch" alle solchen Pfade? Man überprüft *alle* Knoten $k \in V$ daraufhin, ob sie als „Zwischenknoten" für einen Pfad $(2, k, 4)$ in Frage kommen. Und $(2, k, 4)$ ist genau dann ein Pfad, wenn sowohl $(2, k) \in E$, also eine Kante, ist, als auch $(k, 4) \in E$, also eine Kante, ist. Und das ist genau dann der Fall, wenn in der Adjazenzmatrix A von G sowohl $A_{2k} = 1$ als auch $A_{k4} = 1$ ist. Das kann man auch so schreiben, dass $A_{2k} \cdot A_{k4} = 1$ ist.

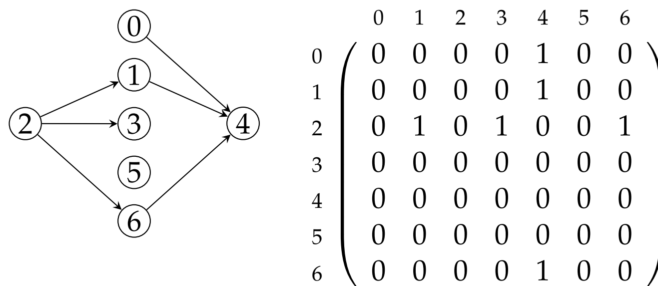


Abbildung 12.3: Beispielgraph für die Berechung von E^2

Wenn man nacheinander alle Elemente A_{2k} inspiziert, dann durchläuft man in A nacheinander die Elemente in der *Zeile* für Knoten 2. Und wenn man nacheinander alle Elemente A_{k4} inspiziert, dann durchläuft man in A nacheinander die Elemente in der *Spalte* für Knoten 4.

In Abbildung 12.4 haben wir schräg übereinander zweimal die Matrix A hingeschrieben, wobei wir der Deutlichkeit halber einmal nur die Zeile für Knoten 2 explizit notiert haben und das andere Mal nur die Spalte für Knoten 4. Außerdem haben wir im oberen linken Viertel alle Produkte $A_{2k} \cdot A_{k4}$ angegeben. Die Frage, ob es einen Pfad der Länge zwei $(2, k, 4)$ gibt, ist gleichbedeutend mit der Frage, ob eines dieser Produkte gleich 1 ist.

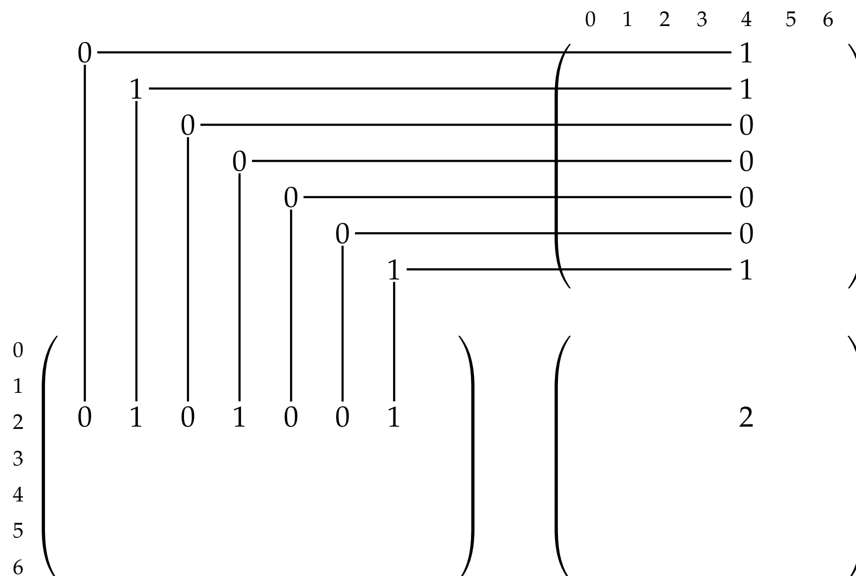


Abbildung 12.4: Der erste Schritt in Richtung Matrixmultiplikation

Wir tun jetzt aber noch etwas anderes. Wir addieren die Werte alle zu einer Zahl

$$P_{24} = \sum_{k=0}^{6} A_{2k} \cdot A_{k4}$$

und notieren Sie in einer dritten Matrix im unteren rechten Viertel der Darstellung. Dabei ist jetzt wichtig:

- Der Wert P_{24} gibt an, wieviele Pfade der Länge zwei es von Knoten 2 nach Knoten 4 gibt.
- Analog kann man für jedes andere Knotenpaar (i, j) die gleiche Berechnung durchführen:

  $$P_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot A_{kj}$$

  Dann ist P_{ij} die Anzahl der Pfade der Länge zwei von Knoten i zu Knoten j.

12.2.1 Matrixmultiplikation

Was wir eben aufgeschrieben haben ist nichts anderes als ein Spezialfall von *Matrixmultiplikation*. Ist A eine $\ell \times n$-Matrix und B eine $n \times m$-Matrix, so heißt die $\ell \times m$-Matrix C, bei der für alle i und alle j

Matrixmultiplikation

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot B_{kj}$$

gilt, das Produkt der Matrizen A und B und man schreibt auch $C = A \cdot B$. (Wichtig: Selbst wenn einmal $\ell = n = m$ ist, ist trotzdem im Allgemeinen $A \cdot B \neq B \cdot A$!)

Algorithmisch notiert sähe die naheliegende Berechung des Produktes zweier Matrizen so aus, wie in Algorithmus 12.1 dargestellt. Wir werden im nächsten Kapitel aber sehen, dass es durchaus andere Möglichkeiten gibt!

Von besonderer Wichtigkeit sind die sogenannten *Einheitsmatrizen*. Das sind diejenigen $n \times n$-Matrizen I, bei denen für alle i und j gilt:

Einheitsmatrix

$$\mathrm{I}_{ij} = \begin{cases} 1 & \text{falls } i = j \\ 0 & \text{falls } i \neq j \end{cases}$$

Zu beliebiger $m \times n$-Matrix A gilt für die jeweils passende Einheitsmatrizen:

$$\mathrm{I} \cdot A = A = A \cdot \mathrm{I}$$

Man beachte, dass die Einheitsmatrix auf der linken Seite Größe $m \times m$ hat und die auf der rechten Seite Größe $n \times n$.

Algorithmus 12.1: Einfachster Algorithmus, um zwei Matrizen zu multiplizieren

```
for i ← 0 to ℓ − 1 do
  for j ← 0 to m − 1 do
    C_ij ← 0
    for k ← 0 to n − 1 do
      C_ij ← C_ij + A_ik · B_kj
    od
  od
od
```

Ist eine Matrix quadratisch (wie z. B. die Adjazenzmatrix A eines Graphen), dann kann man A mit sich selbst multiplizieren. Dafür benutzt wie schon an mehreren anderen Stellen in dieser Vorlesung die Potenzschreibweise:

$$A^0 = I$$
$$\forall n \in \mathbb{N}_0 : A^{n+1} = A^n \cdot A$$

12.2.2 Matrixaddition

Matrixaddition

Für zwei Matrizen A und B der gleichen Größe $m \times n$ ist für uns in Kürze auch noch die Summe $A + B$ von Interesse. Die *Matrixaddition* von A und B liefert die $m \times n$-Matrix C, bei der für alle i und alle j gilt:

$$C_{ij} = A_{ij} + B_{ij} .$$

Nullmatrix

Das neutrale Element ist die sogenannte *Nullmatrix* (genauer gesagt die Nullmatrizen; je nach Größe). Sie enthält überall nur Nullen. Wir schreiben für Nullmatrizen einfach 0.

Zwei Matrizen zu addieren, ist leicht:

```
for i ← 0 to m − 1 do
  for j ← 0 to n − 1 do
      C_ij ← A_ij + B_ij
  od
od
```

12.3 Berechnung der Erreichbarkeitsrelation

Eine naheliegende Idee für die Berechnung von E^* ist natürlich, auf die Definition

$$E^* = \bigcup_{i=0}^{\infty} E^i$$

zurückzugreifen. Allerdings stellen sich sofort drei Probleme:

- Was kann man tun, um nicht unendlich viele Matrizen berechnen zu müssen? D. h., kann man das ∞ durch eine natürliche Zahl ersetzen?
- Woher bekommt man die Matrizen für die Relationen E^i, d. h. welcher Operation bei Matrizen entspricht das Berechnen von Potenzen bei Relationen?
- Wenn man die Matrizen hat, welcher Operation bei Matrizen entspricht die Vereinigung bei Relationen?

Beginnen wir mit dem ersten Punkt. Was ist bei Graphen spezieller als bei allgemeinen Relationen? Richtig: Es gibt nur *endlich* viele Knoten. Und das ist in der Tat eine große Hilfe: Wir interessieren uns für die Frage, ob für gegebene Knoten i und j ein Pfad in G von i nach j existiert. Sei $G = (V, E)$ mit $|V| = n$. Nehmen wir an, es existiert ein Pfad: $p = (i_0, i_1, \ldots, i_k)$ mit $i_0 = i$ und $i_k = j$. Was dann? Nun, wenn k „groß“ ist, genauer gesagt, $k \geq n$, dann kommen in der Liste p also $k + 1 \geq n + 1$ „Knotennamen“ vor. Aber G hat nur n verschiedene Knoten. Also muss mindestens ein Knoten x doppelt in der Liste p vorkommen. Das bedeutet, dass man auf dem Pfad von i nach j einen Zyklus von x nach x geht. Wenn man den weglässt, ergibt sich ein kürzerer Pfad, der immer noch von i nach j führt. Indem man dieses Verfahren wiederholt, solange im Pfad mindestens $n + 1$ Knoten vorkommen, gelangt man schließlich zu einem Pfad, in dem höchstens noch n Knoten, und damit höchstens $n - 1$ Kanten, vorkommen, und der auch immer noch von i nach j führt.

Mit anderen Worten: Was die Erreichbarkeit in einem endlichen Graphen mit n Knoten angeht, gilt:

$$E^* = \bigcup_{i=0}^{n-1} E^i$$

Aber höhere Potenzen schaden natürlich nicht. Das heißt, es gilt sogar:

12.1 Lemma. Für jeden gerichteten Graphen $G = (V, E)$ mit n Knoten gilt:

$$\forall k \geq n - 1 : E^* = \bigcup_{i=0}^{k} E^i$$

12.3.1 Potenzen der Adjazenzmatrix

Wenn man die Adjazenzmatrix A eines Graphen quadriert, erhält man als Eintrag in Zeile i und Spalte j

$$(A^2)_{ij} = \sum_{k=0}^{n-1} A_{ik}A_{kj} \; .$$

Jeder der Summanden ist genau dann 1, wenn $A_{ik} = A_{kj} = 1$ ist, also genau dann, wenn (i, k, j) ein Pfad der Länge 2 von i nach j ist. Und für verschiedene k sind das auch verschiedene Pfade. Also ist

$$(A^2)_{ij} = \sum_{k=0}^{n-1} A_{ik}A_{kj}$$

gleich der Anzahl der Pfade der Länge 2 von i nach j.

Überlegen Sie sich, dass analoge Aussagen für $(A^1)_{ij}$ und Pfade der Länge 1 von i nach j, sowie $(A^0)_{ij}$ und Pfade der Länge 0 von i nach j richtig sind. Tatsächlich gilt:

12.2 Lemma. Es sei G ein gerichteter Graph mit Adjazenzmatrix A. Für alle $k \in \mathbb{N}_0$ gilt: $(A^k)_{ij}$ ist die Anzahl der Pfade der Länge k in G von i nach j.

Der Beweis wäre eine recht einfache vollständige Induktion. Der einzige gegenüber dem Fall $k = 2$ zusätzlich zu beachtende Punkt besteht darin, dass die Verlängerung verschiedener Wege um die gleiche Kante (falls das überhaupt möglich ist), wieder zu verschiedenen Wegen führt.

Signum-Funktion

Wir bezeichnen nun mit sgn die sogenannte *Signum-Funktion*

$$\text{sgn} : \mathbb{R} \to \mathbb{R} : \text{sgn}(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases}$$

Für die Erweiterung auf Matrizen durch komponentenweise Anwendung schreiben wir auch wieder sgn:

$$\text{sgn} : \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times n} : (\text{sgn}(M))_{ij} = \text{sgn}(M_{ij})$$

Unter Verwendung dieser Hilfsfunktion ergibt sich aus Lemma 12.2:

12.3 Korollar. Es sei G ein gerichteter Graph mit Adjazenzmatrix A.

1. Für alle $k \in \mathbb{N}_0$ gilt:

$$\operatorname{sgn}((A^k)_{ij}) = \begin{cases} 1 & \text{falls in } G \text{ ein Pfad der Länge } k \\ & \quad \text{von } i \text{ nach } j \text{ existiert} \\ 0 & \text{falls in } G \text{ kein Pfad der Länge } k \\ & \quad \text{von } i \text{ nach } j \text{ existiert} \end{cases}$$

2. Für alle $k \in \mathbb{N}_0$ gilt: $\operatorname{sgn}(A^k)$ ist die Matrix, die die Relation E^k repräsentiert.

12.3.2 Erste Möglichkeit für die Berechnung der Wegematrix

Um zu einem ersten Algorithmus zur Berechnung der Wegematrix zu kommen, müssen wir als letztes noch klären, welche Operation auf Matrizen „zur Vereinigung von Relationen passt“. Seien dazu auf der gleichen Menge M zwei binäre Relationen $R \subseteq M \times M$ und $R' \subseteq M \times M$ gegeben, repräsentiert durch Matrizen A und A'. Dann gilt:

$$\begin{aligned} (i,j) \in R \cup R' &\iff (i,j) \in R \vee (i,j) \in R' \\ &\iff A_{ij} = 1 \vee A'_{ij} = 1 \\ &\iff A_{ij} + A'_{ij} \geq 1 \\ &\iff (A + A')_{ij} \geq 1 \\ &\iff \operatorname{sgn}(A + A')_{ij} = 1 \end{aligned}$$

Also wird die Relation $R \cup R'$ durch die Matrix $\operatorname{sgn}(A + A')$ repräsentiert.

Aus Lemma 12.1 und dem eben aufgeführten Korollar 12.3 folgt recht schnell eine erste Formel für die Berechnung der Wegematrix:

12.4 Lemma. Es sei G ein gerichteter Graph mit Adjazenzmatrix A. Dann gilt für alle $k \geq n - 1$:

- Die Matrix $\operatorname{sgn}(\sum_{i=0}^{k} A^i)$ repräsentiert die Relation E^*.
- Mit anderen Worten:

$$W = \operatorname{sgn}\left(\sum_{i=0}^{k} A^i\right)$$

  ist die Wegematrix des Graphen G.

12.5 Beweis. Angesichts der schon erarbeiteten Ergebnisse ist es ausreichend, sich noch die beiden folgenden eher technischen Dinge zu überlegen:

- Die Vereiniung $\bigcup_{i=0}^{n-1} E^i$ wird repräsentiert durch die Matrix $\operatorname{sgn}(\sum_{i=0}^{k} \operatorname{sgn}(A^i))$.

- In dieser Formel darf man die „inneren“ Anwendungen von sgn weglassen.

Das sieht man so:

- Zum ersten Punkt genügt es, die obige Überlegung zur Matrixrepräsentation von $R \cup R'$ per Induktion auf beliebig endliche viele Relationen zu übertragen.
- Mit einer ähnlichen Herangehensweise wie oben ergibt sich

$$\begin{aligned} \operatorname{sgn}(\operatorname{sgn}(A) + \operatorname{sgn}(A'))_{ij} = 1 &\iff (\operatorname{sgn}(A) + \operatorname{sgn}(A'))_{ij} \geq 1 \\ &\iff \operatorname{sgn}(A)_{ij} + \operatorname{sgn}(A')_{ij} \geq 1 \\ &\iff \operatorname{sgn}(A)_{ij} = 1 \vee \operatorname{sgn}(A')_{ij} = 1 \\ &\iff A_{ij} \geq 1 \vee A'_{ij} \geq 1 \\ &\iff A_{ij} + A'_{ij} \geq 1 \\ &\iff (A + A')_{ij} \geq 1 \\ &\iff \operatorname{sgn}(A + A')_{ij} = 1 \end{aligned}$$

  Dabei haben wir in beiden Punkten benutzt, dass die Einträge in den interessierenden Matrizen nie negativ sind.

■

Das sich ergebende Verfahren ist in Algorithmus 12.2 dargestellt.

Algorithmus 12.2: einfachster Algorithmus, um die Wegematrix zu berechnen

```
// MatrixAseidieAdjazenzmatrix
// MatrixWwirdamEndedieWegematrixenthalten
// MatrixMwirdbenutztumA^izuberechnen
W ← 0                  // Nullmatrix
for i ← 0 to n − 1 do
  M ← I                // Einheitsmatrix
  for j ← 1 to i do
    M ← M · A          // Matrixmultiplikation
  od
  W ← W + M            // Matrixaddition
od
W ← sgn(W)
```

12.3.3 Zählen durchzuführender arithmetischer Operationen

Wir stellen nun ein erstes Mal die Frage danach wie aufwändig es ist, die Wegematrix eines Graphen auszurechnen. Unter Aufwand wollen hier zunächst einmal der Einfachheit halber die Anzahl benötigter arithmetischer Operationen verstehen. (Wir werden das im Laufe weiterer Einheiten noch genauer diskutieren.)

Wir beginnen mit der wohl naivsten Herangehensweise, wollen aber schon einmal darauf hinweisen, dass Sie schon in diesem Fall sehen werden, dass man manchmal durch Nachdenken oder hübsche Ideen signifikante Verbesserungen erreichen kann.

Die einfachste Methode besteht darin, Algorithmus 12.2 zu benutzen und ohne viel Nachdenken zu implementieren. Das bedeutet, dass wir zur Berechung von $\operatorname{sgn}\left(\sum_{i=0}^{n-1} A^i\right)$ folgende Berechnungen (nicht unbedingt in dieser Reihenfolge) durchzuführen haben:

- n^2 Berechnungen der sgn Funktion;
- n Matrix-Additionen;
- $0 + 1 + \cdots + n - 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$ Matrix-Multiplikationen;

Alle Matrizen haben Größe $n \times n$.

Für jeden der n^2 Einträge in einer Summenmatrix muss man eine Addition durchführen, also benötigt eine Matrixaddition n^2 arithmetische Operationen.

Für jeden der n^2 Einträge in einer Produktmatrix besteht jedenfalls die naheliegende Methode darin, eine Formel der Form $\sum_{k=0}^{n-1} a_{ik}b_{kj}$ auszuwerten. (Tatsächlich gibt es andere Möglichkeiten, wie wir in nachfolgenden Abschnitten sehen werden!) Das sind jeweils n Multiplikationen und $n-1$ Additionen. Insgesamt ergeben sich so $2n^3 - n^2$ Operationen.

Für die Berechnung der Wegematrix nach dieser Methode kommt man so auf

$$\begin{aligned}
&n^2 + n^2 \cdot n + (2n^3 - n^2)n(n-1)/2 \\
=&n^2 + n^3 + (2n^3 - n^2)(n^2 - n)/2 \\
=&n^2 + n^3 + (2n^5 - 2n^4 - n^4 + n^3)/2 \\
=&n^5 - \frac{3}{2}n^4 + \frac{3}{2}n^3 + n^2
\end{aligned}$$

Operationen. Wenn z. B. $n = 1000$ ist, dann sind das immerhin 998 501 501 000 000 Operationen, also „fast“ 10^{15}.

12.3.4 Weitere Möglichkeiten für die Berechnung der Wegematrix

Kann man die Wegematrix auch mit „deutlich“ weniger Operationen berechnen? Vielleicht haben Sie eine Möglichkeit schon gesehen: Wir haben weiter vorne so

getan, als müsste man für die Berechnung von A^i immer $i-1$ Matrixmultiplikationen durchführen. Da aber der Reihe nach *alle* Potenzen A^i berechnet werden, ist das nicht so. Man merkt sich einfach immer das alte A^{i-1} und braucht dann nur *eine* Matrixmultiplikation, um zu A^i zu gelangen. Diese Vorgehensweise ist in Algorithmus 12.3 dargestellt. Damit ergeben sich insgesamt nur n Matrixmultiplikationen statt $n(n-1)/2$ und die Gesamtzahl arithmetischer Operationen sinkt von $n^5 - (3/2)n^4 + (3/2)n^3 + n^2$ auf $2n^4 + n^2$. Für $n = 1000$ sind das 2 000 001 000 000, also „ungefähr" 500 mal weniger als mit Algorithmus 12.2.

Algorithmus 12.3: verbesserter Algorithmus, um die Wegematrix zu berechnen

```
// MatrixAseidieAdjazenzmatrix
// MatrixWwirdamEndedieWegematrixenthalten
// MatrixMwirdbenutztumA^izuberechnen
W ← 0                  // Nullmatrix
M ← I                  // Einheitsmatrix
for i ← 0 to n − 1 do
   W ← W + M           // Matrixaddition
   M ← M · A           // Matrixmultiplikation
od
W ← sgn(W)
```

Wenn man einmal unterstellt, dass jede Operation gleich lange dauert, dann erhält man also Ergebnisse um etwa einen Faktor $n/2$ schneller.

Und es geht noch schneller: statt n Matrixmultiplikationen wollen wir nun mit $\log_2 n$ von ihnen auskommen. Hierfür nutzen wir die Beobachtung aus, deren explizite Erwähnung in Lemma 12.1 Sie vielleicht ein bisschen gewundert hat:

$$\forall k \geq n-1 : E^* = \bigcup_{i=0}^{k} E^i$$

Statt $n-1$ wählen wir nun die nächstgrößere Zweierpotenz $k = 2^{\lceil \log_2 n \rceil}$. Außerdem benutzen wir noch einen Trick, der es uns erlaubt, statt $\bigcup_{i=0}^{k} E^i$ etwas ohne viele Vereinigungen hinzuschreiben. Dazu betrachten wir $F = E^0 \cup E^1 = \mathrm{I}_V \cup E$. Unter Verwendung der Rechenregel (die Sie sich bitte klar machen) für Relationen

$$(A \cup B) \circ (C \cup D) = (A \circ C) \cup (A \circ D) \cup (B \circ C) \cup (B \circ D)$$

ergibt sich

$$F^2 = (E^0 \cup E^1) \circ (E^0 \cup E^1) = E^0 \cup E^1 \cup E^1 \cup E^2 = E^0 \cup E^1 \cup E^2$$

Daraus folgt

$$\begin{aligned} F^4 = (F^2)^2 &= (E^0 \cup E^1 \cup E^2) \circ (E^0 \cup E^1 \cup E^2) \\ &= \dots \\ &= E^0 \cup E^1 \cup E^2 \cup E^3 \cup E^4 \end{aligned}$$

und durch Induktion sieht man, dass für alle $m \in \mathbb{N}_0$ gilt:

$$F^{2^m} = \bigcup_{i=0}^{2^m} E^i$$

Wenn man einfach zu Beginn die Matrix für $F = E^0 + E$ berechnet und sie dann so oft quadriert, dass nach m-maligen Durchführen $2^m \geq n-1$ ist, hat man das gewünschte Ergebnis. Offensichtlich genügt $m = \lceil \log_2 n \rceil$.

Im Matrizenrechnung übersetzt ergeben sich

$$2n^2 + \lceil \log_2 n \rceil (2n^3 - n^2)$$

arithmetische Operationen, was gegenüber $2n^4 + n^2$ wieder eine beträchtliche Verbesserung ist, nämlich ungefähr um einen Faktor $2n / \lceil \log_2 n \rceil$.

12.4 ALGORITHMUS VON WARSHALL

Wir kommen nun zu einem Algorithmus zur Berechnung der Wegematrix eines Graphen, bei dem gegenüber der eben zuletzt behandelten Methode der Faktor $\log_2 n$ sogar auf eine (kleine) Konstante sinkt. Er stammt von Warshall (1962) und ist in Algorithmus 12.4 dargestellt.
Zum besseren Verständnis sei als erstes darauf hingewiesen, dass für zwei Bits x und y das Maximum $\max(x, y)$ dem logischen Oder entspricht, wenn man 1 als wahr und 0 als falsch interpretiert. Analog entspricht das Minimum $\min(x, y)$ dem logischen Und.

Den Aufwand dieses Algorithmus sieht man schnell. Für die Initialisierung der Matrix W im ersten Teil werden n^2 Operationen benötigt. Die weitere Rechnung besteht aus drei ineinander geschachtelten **for**-Schleifen, von denen jede jedes mal n mal durchlaufen wird. Das ergibt n^3-malige Ausführung des Schleifenrumpfes, bei der jedes Mal zwei Operationen durchgeführt werden.

Algorithmus 12.4: Berechnung der Wegematrix nach Warshall

```
for i ← 0 to n − 1 do
  for j ← 0 to n − 1 do
    W_ij ← { 1     falls i = j
           { A_ij  falls i ≠ j
  od
od
for k ← 0 to n − 1 do
  for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
      W_ij ← max( W_ij, min(W_ik, W_kj) )
    od
  od
od
```

Weitaus weniger klar dürfte es für Sie sein, einzusehen, warum der Algorithmus tatsächlich die Wegematrix berechnet. Es stellt sich hier mit anderen Worten wieder einmal die Frage nach der *Korrektheit* des Algorithmus.

Die algorithmische Idee, die hier im Algorithmus von Warshall benutzt wird, geht auf eine fundamentale Arbeit von Stephen Kleene (1956) zurück, der sie im Zusammenhang mit *endlichen Automaten* und *regulären Ausdrücken* benutzt hat. (Unter anderem wird in dieser Arbeit die Schreibweise mit dem hochgestellten Stern * für den Konkatenationsabschluss eingeführt, der deswegen auch Kleene-Stern heißt.) Auf diese Themen werden wir in einer späteren Einheit eingehen.

Für den Nachweis der Korrektheit des Algorithmus von Warshall besteht die Hauptaufgabe darin, eine Schleifeninvariante für

```
for k ← 0 to n − 1 do
  ...
od
```

zu finden. Für die Formulierung ist es hilfreich, bei einem Pfad $p = (v_0, v_1, \dots, v_{m-1}, v_m)$ der Länge $m \geq 2$ über die Knoten $v_1, \dots, v_{m-1}$ reden zu können. Wir nennen sie im folgenden die *Zwischenknoten* des Pfades. Pfade der Längen 0 und 1 besitzen keine Zwischenknoten. Hier ist nun die Schleifeninvariante:

12.6 Lemma. Für alle $i, j \in \mathbb{G}_n$ gilt: Nach k Durchläufen der äußeren Schleife des Algorithmus von Warshall ist $W[i, j]$ genau dann 1, wenn es einen wiederholungsfreien Pfad von i nach j gibt, bei dem alle Zwischenknoten Nummern in $\mathbb{G}_k$ haben (also Nummern, die echt kleiner als k sind).

Hat man erst einmal nachgewiesen, dass das tatsächlich Schleifeninvariante ist, ist man gleich fertig. Denn dann gilt insbesondere nach Beendigung der Schleife, also nach n Schleifendurchläufen:

- Für alle $i, j \in \mathbb{G}_n$ gilt: Nach n Schleifendurchläufen ist W_{ij} genau dann 1, wenn es einen wiederholungsfreien Pfad von i nach j gibt, bei dem alle Zwischenknoten Nummern in $\mathbb{G}_n$ haben, wenn also überhaupt ein Pfad existiert (denn andere Knotennummern gibt es gar nicht).

12.7 Beweis. (von Lemma 12.6)

Induktionsanfang: Dass die Behauptung im Fall $k = 0$ gilt, ergibt sich aus der Initialisierung der Matrix W: Knoten mit Nummern echt kleiner als 0 gibt es nicht; in den zur Rede stehenden Pfaden kommen als keine Knoten außer erstem und letztem vor. Das bedeutet aber, dass die Pfade von einer der Formen (x) oder (x, y) sein müssen.

Für den Induktionsschritt sei $k > 0$ beliebig aber fest und wir treffen die

Induktionsvoraussetzung: Für alle $i, j \in \mathbb{G}_n$ gilt: Nach $k - 1$ Durchläufen der äußeren Schleife des Algorithmus von Warshall ist W_{ij} genau dann 1, wenn es einen wiederholungsfreien Pfad von i nach j gibt, bei dem alle Zwischenknoten Nummern haben, die in $\mathbb{G}_{k-1}$ sind.

Induktionsschluss: Wir bezeichnen mit $W^{[k]}$ die Matrix, wie sie nach k Schleifendurchläufen berechnet wird, und analog mit $W^{[k-1]}$ die Matrix nach $k - 1$ Schleifendurchläufen. Die beiden Implikationen werden getrennt bewiesen:

$\Longrightarrow$: Es sei $W_{ij}^{[k]} = 1$. Dann hat also mindestens eine der folgenden Bedingungen zugetroffen:

- $W_{ij}^{[k-1]} = 1$: In diesem Fall existiert ein Pfad, dessen Zwischenknoten alle Nummern in $\mathbb{G}_{k-1}$ haben, und das ist auch einer, dessen Zwischenknoten alle Nummern in $\mathbb{G}_k$ haben.
- $W_{ik}^{[k-1]} = 1$ und $W_{kj}^{[k-1]} = 1$. Dann existieren Pfade von i nach k und von k nach j, deren Zwischenknoten alle Nummern in $\mathbb{G}_{k-1}$ sind. Wenn man die Pfade zusammensetzt, erhält man einen Pfad von i nach j, dessen Zwischenknoten alle Nummern in $\mathbb{G}_k$ haben. Durch Entfernen von Zyklen kann man auch einen solchen wiederholungsfreien Pfad konstruieren.

$\Longleftarrow$: Es gebe einen wiederholungsfreien Pfad p von i nach j, dessen Zwischenknoten alle Nummern in $\mathbb{G}_k$ haben. Dann sind zwei Fälle möglich:

- Ein Zwischenknoten in p hat Nummer $k-1$:
  Da p wiederholungsfrei ist, enthält das Anfangsstück von p, das von i nach $k-1$ führt, nicht $k-1$ als Zwischenknoten, also nur Knotennummern in $\mathbb{G}_{k-1}$. Das gleiche gilt für das Endstück von p, das von $k-1$ nach j führt. Nach Induktionsvoraussetzung sind also $W_{i,k-1}^{[k-1]} = 1$ und $W_{k-1,j}^{[k-1]} = 1$. Folglich wird im k-ten Durchlauf $W_{ij}^{[k]} = 1$ gesetzt.
- Kein Zwischenknoten in p hat Nummer $k-1$:
  Dann sind die Nummern der Zwischenknoten alle in $\mathbb{G}_{k-1}$. Nach Induktionsvoraussetzung ist folglich $W_{ij}^{[k-1]} = 1$ und daher auch $W_{ij}^{[k]} = 1$.

■

12.5 AUSBLICK

Wir sind in dieser Einheit davon ausgegangen, dass die Matrizen auf die naheliegende Weise miteinander multipliziert werden. In der nächsten Einheit werden wir sehen, dass es auch andere Möglichkeiten gibt, die in einem gewissen Sinne sogar besser sind. Dabei werden auch Möglichkeiten zur Quantifizierung von „gut“, „besser“, usw. Thema sein.

Effiziente Algorithmen für Problemstellungen bei Graphen sind nach wie vor Gegenstand intensiver Forschung. Erste weiterführende Aspekte werden Sie im kommenden Semester in der Vorlesung „Algorithmen 1“ zu sehen bekommen.

LITERATUR

Kleene, Stephen C. (1956). “Representation of Events in Nerve Nets and Finite Automata”. In: *Automata Studies*. Hrsg. von Claude E. Shannon und John McCarthy. Princeton University Press. Kap. 1, S. 3–40.
Eine Vorversion ist online verfügbar; siehe http://www.rand.org/pubs/research_memoranda/2008/RM704.pdf (8.12.08).

Warshall, Stephen (1962). “A Theorem on Boolean Matrices”. In: *Journal of the ACM* 9, S. 11–12.

13 QUANTITATIVE ASPEKTE VON ALGORITHMEN

13.1 RESSOURCENVERBRAUCH BEI BERECHNUNGEN

Wir haben in Einheit 12 mit ersten Graphalgorithmen damit begonnen, festzustellen, wieviele arithmetische Operationen bei der Ausführung eines Algorithmus für eine konkrete Probleminstanz ausgeführt werden. Zum Beispiel hatten wir angemerkt, dass bei der Addition zweier $n \times n$-Matrizen mittels zweier ineinander geschachtelten **for**-Schleifen n^2 Additionen notwendig sind. Das war auch als ein erster Schritt gedacht in Richtung der Abschätzung von Laufzeiten von Algorithmen.

Laufzeit, Rechenzeit
Ressource
Speicherplatzbedarf
Komplexitätsmaß

Rechenzeit ist wohl die am häufigsten untersuchte *Ressource*, die von Algorithmen „verbraucht" wird. Eine zweite ist der *Speicherplatzbedarf*. Man spricht in diesem Zusammenhang auch von *Komplexitätsmaßen*. Sie sind Untersuchungsgegenstand in Vorlesungen über Komplexitätstheorie (engl. *computational complexity*) und tauchen darüber hinaus in vielen Gebieten (und Vorlesungen) zur Beurteilung der Qualität von Algorithmen auf.

Hauptgegenstand dieser Einheit wird es sein, das wichtigste Handwerkszeug bereitzustellen, das beim Reden über und beim Ausrechnen von z. B. Laufzeiten hilfreich ist und in der Literatur immer wieder auftaucht, insbesondere die sogenannte Groß-O-Notation.

Dazu sei als erstes noch einmal explizit daran erinnert, dass wir in Einheit 6 zum informellen Algorithmusbegriff festgehalten haben, dass ein Algorithmus für Eingaben beliebiger Größe funktionieren sollte: Ein Algorithmus zur Multiplikation von Matrizen sollte nicht nur für 3×3-Matrizen oder 42×42-Matrizen funktionieren, sollte für Matrizen mit beliebiger Größe $n \times n$. Es ist aber „irgendwie klar", dass dann die Laufzeit keine Konstante sein kann, sondern eine Funktion ist, die zumindest von n abhängt. Und zum Beispiel bei Algorithmus 12.2 zur Bestimmung der Wegematrix eines Graphen hatte auch nichts anderes Einfluss auf die Laufzeit.

Aber betrachten wir als ein anderes Beispiel das Sortieren von Zahlen und z. B. den Insertionsort-Algorithmus aus der Programmieren-Vorlesung, den wir in Algorithmus 13.1 noch mal aufgeschrieben haben. Wie oft die **while**-Schleife in der Methode *insert* ausgeführt wird, hängt nicht einfach von der Problemgröße $n = a.length$ ab. Es hängt auch von der konkreten Probleminstanz ab. Selbst bei gleicher Zahl n kann die Schleife unterschiedlich oft durchlaufen werden: Ist das Array a von Anfang an sortiert, wird die **while**-Schleife überhaupt nicht ausgeführt. Ist es

genau in entgegengesetzter Richtung sortiert, wird ihr Schleifenrumpf insgesamt $\sum_{i=1}^{n-1} i = n(n-1)/2$ mal ausgeführt.

```
public class InsertionSort {
   public static void sort(long[ ] a) {
      for (int i ← 1;  i < a.length;  i + +) {
         insert(a, i);
      }
   }
   private static void insert(long[ ] a, int idx) {
      // Tausche a[idx] nach links bis es einsortiert ist
      for (int i ← idx; i > 0 and a[i − 1] > a[i]; i − −) {
         long tmp ← a[i − 1];
         a[i − 1] ← a[i];
         a[i] ← tmp;
      }
   }
}
```

Algorithmus 13.1: Insertionsort aus der Vorlesung „Programmieren“

Meistens ist es aber so, dass man nicht für jede Probleminstanz einzeln angeben will oder kann, wie lange ein Algorithmus zu seiner Lösung benötigt. Man beschränkt sich auf eine vergröbernde Sichtweise und beschreibt z. B. die Laufzeit nur in Abhängigkeit von der Problemgröße n. Es bleibt die Frage zu klären, was man dann angibt, wenn die Laufzeit für verschiedene Instanzen gleicher Größe variiert: Den Durchschnitt? Den schnellsten Fall? Den langsamsten Fall?

worst case

average case

Am weitesten verbreitet ist es, als Funktionswert für jede Problemgröße n den jeweils schlechtesten Fall (engl. *worst case*) zu nehmen. Eine entsprechende Analyse eines Algorithmus ist typischerweise deutlich einfacher als die Berechnung von Mittelwerten (engl. *average case*), und wir werden uns jedenfalls in dieser Vorlesung darauf beschränken.

13.2 GROSS-O-NOTATION

Die Aufgabe besteht also im allgemeinen darin, bei einem Algorithmus für jede mögliche Eingabegröße n genau anzugeben, wie lange der Algorithmus für Probleminstanzen der Größe n im schlimmsten Fall zur Berechung des Ergebnisses

benötigt. Leider ist es manchmal so, dass man die exakten Werte nicht bestimmen will oder kann.

Dass man es nicht *will*, muss nicht unbedingt Ausdruck von Faulheit sein. Vielleicht sind gewisse Ungenauigkeiten (die man noch im Griff hat) für das Verständnis nicht nötig. Oder die genauen Werte hängen von Umständen ab, die sich ohnehin „bald" ändern. Man denke etwa an die recht schnelle Entwicklung bei Prozessoren in den vergangenen Jahren. Dass ein Programm für gewisse Eingaben auf einem bestimmten Prozessor soundso lange braucht, ist unter Umständen schon nach wenigen Monaten uninteressant, weil ein neuer Prozessor viel schneller ist. Das muss nicht einfach an einer höheren Taktrate liegen (man könnte ja auch einfach Takte zählen statt Nanosekunden), sondern z. B. an Verbesserungen bei der Prozessorarchitektur.

Darüber hinaus *kann* man die Laufzeit eines Algorithmus mitunter gar nicht exakt abschätzen. Manchmal könnte man es vielleicht im Prinzip, aber man ist zu dumm. Oder die vergröbernde Darstellung nur der schlechtesten Fälle führt eben dazu, dass die Angabe für andere Fälle nur eine obere Schranke ist. Oder man erlaubt sich bei der Formulierung von Algorithmen gewisse Freiheiten bzw. Ungenauigkeiten, die man (vielleicht wieder je nach Prozessorarchitektur) unterschiedlich bereinigen kann, weil man eben an Aussagen interessiert ist, die von Spezifika der Prozessoren unabhängig sind.

Damit haben wir zweierlei angedeutet:

- Zum einen werden wir im weiteren Verlauf dieses Abschnittes eine Formulierungshilfe bereitstellen, die es erlaubt, in einem gewissen überschaubaren Rahmen ungenau über Funktionen zu reden.
- Zum anderen werden wir in einer späteren Einheit auf Modelle für Rechner zu sprechen kommen. Ziel wird es sein, z. B. über Laufzeit von Algorithmen in einer Weise reden zu können, die unabhängig von konkreten Ausprägungen von Hardware und trotzdem noch aussagekräftig ist.

13.2.1 Ignorieren konstanter Faktoren

Wie ungenau wollen wir über Funktionen reden?

Ein erster Aspekt wird dadurch motiviert, dass man das so tun möchte, dass z. B. Geschwindigkeitssteigerungen bei Prozessoren irrelevant sind. Etwas genauer gesagt sollen konstante Faktoren beim Wachstum von Funktionen keine Rolle spielen.

Wir bezeichnen im folgenden mit $\mathbb{R}_+$ die Menge der positiven reellen Zahlen (also *ausschließlich* 0) und mit $\mathbb{R}_0^+$ die Menge der nichtnegativen rellen Zahlen, also $\mathbb{R}_0^+ = \mathbb{R}_+ \cup \{0\}$. Wir betrachten Funktionen $f : \mathbb{N}_0 \to \mathbb{R}_0^+$.

asymptotisches Wachstum
größenordnungsmäßiges Wachstum

Wir werden im folgenden vom *asymptotischen Wachstum* oder auch *größenordnungsmäßigen Wachstum* von Funktionen sprechen (obwohl es das Wort „größenordnungsmäßig" im Deutschen gar nicht gibt — zumindest steht es nicht im Duden; betrachten wir es als Terminus technicus). Eine Funktion $g : \mathbb{N}_0 \to \mathbb{R}_0^+$ wächst *größenordnungsmäßig genauso schnell* wie eine Funktion $f : \mathbb{N}_0 \to \mathbb{R}_0^+$, wenn gilt:

$$\exists c, c' \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : cf(n) \leq g(n) \leq c'f(n) \ .$$

$f \asymp g$

Wir schreiben in diesem Fall auch $f \asymp g$ oder $f(n) \asymp g(n)$. Zum Beispiel gilt $3n^2 \asymp 10^{-2}n^2$. Denn einerseits gilt für $c = 10^{-3}$ und $n_0 = 0$:

$$\forall n \geq n_0 : cf(n) = 10^{-3} \cdot 3n^2 \leq 10^{-2}n^2 = g(n)$$

Andererseits gilt z. B. für $c' = 1$ und $n_0 = 0$:

$$\forall n \geq n_0 : g(n) = 10^{-2}n^2 \leq 3n^2 = c'f(n)$$

Die eben durchgeführte Rechnung lässt sich leicht etwas allgemeiner durchführen. Dann zeigt sich, dass man festhalten kann:

13.1 (Rechenregel) Für alle $f : \mathbb{N}_0 \to \mathbb{R}_0^+$ gilt:

$$\forall a, b \in \mathbb{R}_+ : af(n) \asymp bf(n)$$

Damit können wir uns als zweites Beispiel $f(n) = n^3 + 5n^2$ und $g(n) = 3n^3 - n$ ansehen und nun schon recht leicht einsehen, dass $f \asymp g$ ist. Denn einerseits ist für $n \geq 0$ offensichtlich $f(n) = n^3 + 5n^2 \leq n^3 + 5n^3 = 6n^3 = 9n^3 - 3n^3 \leq 9n^3 - 3n = 3(3n^3 - n) = 3g(n)$. Andererseits ist $g(n) = 3n^3 - n \leq 3n^3 \leq 3(n^3 + 5n^2) = 3f(n)$.

Es gibt auch Funktionen, für die $f \asymp g$ *nicht* gilt. Als einfaches Beispiel betrachten wir $f(n) = n^2$ und $g(n) = n^3$. Die Bedingung $g(n) \leq c'f(n)$ aus der Definition ist für $f(n) \neq 0$ gleichbedeutend mit $g(n)/f(n) \leq c'$. Damit $f \asymp g$ gilt, muss insbesondere $g(n)/f(n) \leq c'$ für ein $c' \in \mathbb{R}_+$ ab einem n_0 für alle n gelten. Es ist aber $g(n)/f(n) = n$, und das kann durch keine Konstante beschränkt werden.

Mitunter ist eine graphische Darstellung nützlich. In Abbildung 13.1 sieht man zweimal die gleiche Funktion $f(x)$ für den gleichen Argumentbereich geplottet. Auf der linken Seite sind beide Achsen linear skaliert. Auf der rechten Seite ist die y-Achse logarithmisch skaliert. In einer solchen Dartstellung erhält man den Graph für $cf(x)$ aus dem für $f(x)$ durch Verschieben um $\log(c)$ nach oben. Für eine Funktion $g(x)$ mit $g(x) \asymp f(x)$ muss gelten, dass, von endlich vielen Ausnahmen abgesehen, für „fast" alle $n \in \mathbb{N}_0$ gilt: $cf(n) \leq g(n) \leq c'f(n)$. Auf die graphische Darstellung übertragen bedeutet das, dass der Graph für g fast überall

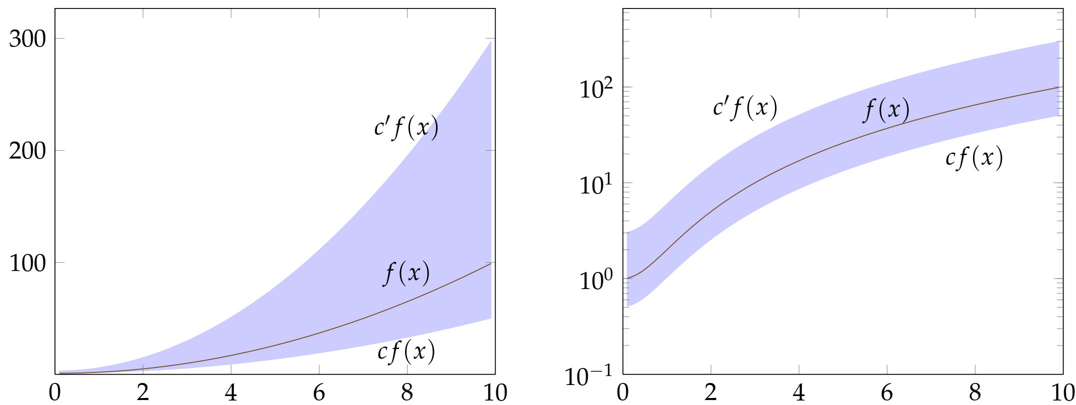


Abbildung 13.1: Zweimal die gleiche Funktion $f(x)$ und zwei Schranken $cf(x)$ und $c'f(x)$; links Darstellung mit linear skalierter y-Achse, rechts mit logarithmisch skalierter y-Achse

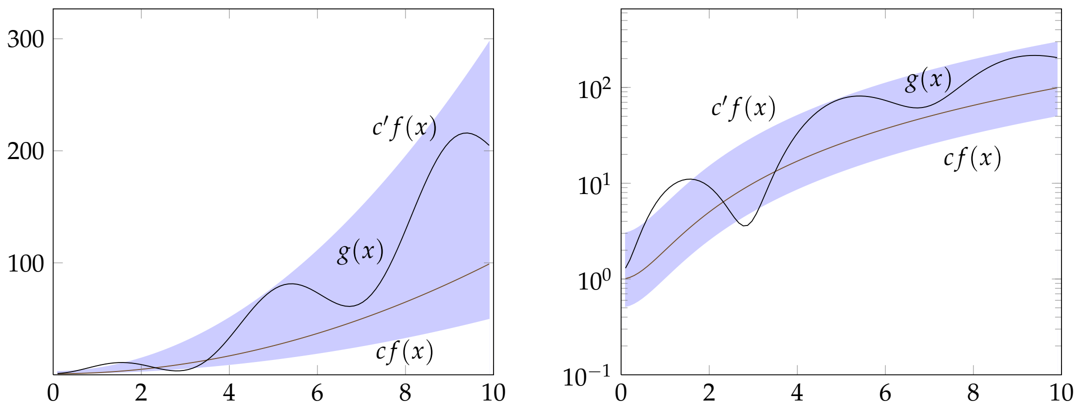


Abbildung 13.2: Zweimal die gleiche Funktion $g(x)$, die ab $n \geq 4$ durch $cf(n)$ und $c'f(n)$ beschränkt ist.

in dem farblich hinterlegten „Schlauch" um f liegen muss. In Abbildung 13.2 ist ein Beispiel dargestellt.

Wer mit Grenzwerten schon vertraut ist, dem wird auch klar sein, dass z. B. nicht $n^2 \asymp 2^n$ ist: Da $\lim_{n\to\infty} 2^n/n^2 = \infty$ ist, ist offensichtlich *nicht* für alle großen n die Ungleichung $2^n \leq c'n^2$ erfüllbar. Man kann sich das aber auch „zu Fuß" überlegen: Für die Folge von Zahlen $n_i = 2^{i+2}$ gilt:

$$\forall i \in \mathbb{N}_0 : 2^{n_i} \geq 4^i n_i^2$$

Der Induktionsanfang ist klar und es ist

$$2^{n_{i+1}} = 2^{2n_i} = 2^{n_i} \cdot 2^{n_i} \geq 4^i n_i^2 \cdot 4^{n_i/2} \geq 4^i n_i^2 \cdot 16 = 4^i \cdot 4 \cdot 4n_i^2 = 4^{i+1} \cdot n_{i+1}^2 \ .$$

Also kann nicht für alle großen n gelten: $2^n \leq cn^2$.

Das Zeichen $\asymp$ erinnert an das Gleichheitszeichen. Das ist auch bewusst so gemacht, denn die Relation $\asymp$ hat wichtige Eigenschaften, die auch auf Gleichheit zutreffen:

13.2 Lemma. Die Relation $\asymp$ ist eine Äquivalenzrelation.

13.3 Beweis. Wir überprüfen die drei definierenden Eigenschaften von Äquivalenzrelationen (siehe Abschnitt 11.2.1).

- *Reflexitvität*: Es ist stets $f \asymp f$, denn wenn man $c = c' = 1$ und $n_0 = 0$ wählt, dann gilt für $n \geq n_0$ offensichtlich $cf(n) \leq f(n) \leq c'f(n)$.
- *Symmetrie:* Wenn $f \asymp g$ gilt, dann auch $g \asymp f$: Denn wenn für positive Konstanten c, c' und alle $n \geq n_0$

  $$cf(n) \leq g(n) \leq c'f(n)$$

  gilt, dann gilt für die gleichen $n \geq n_0$ und die ebenfalls positiven Konstanten $d = 1/c$ und $d' = 1/c'$:

  $$d'g(n) \leq f(n) \leq dg(n)$$

- *Transitivität:* Wenn $f \asymp g$ ist, und $g \asymp h$, dann ist auch $f \asymp h$: Es gelte für Konstanten $c, c' \in \mathbb{R}_+$ und alle $n \geq n_0$

  $$cf(n) \leq g(n) \leq c'f(n)$$

  und analog für Konstanten $d, d' \in \mathbb{R}_+$ und alle $n \geq n_1$

  $$dg(n) \leq h(n) \leq d'g(n) \ .$$

  Dann gilt für alle $n \geq \max(n_0, n_1)$

  $$dcf(n) \leq dg(n) \leq h(n) \leq d'g(n) \leq d'c'f(n) \ ,$$

  wobei auch die Konstanten dc und $d'c'$ wieder positiv sind.

■

Es ist üblich, für die Menge aller Funktionen, die zu einer gegebenen Funktion $f(n)$ im Sinne von $\asymp$ äquivalent sind, $\Theta(f)$ bzw. $\Theta(f(n))$ zu schreiben. Also:

$\Theta(f)$

$$\begin{aligned}\Theta(f) &= \{g \mid f \asymp g\} \\ &= \{g \mid \exists c, c' \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : cf(n) \leq g(n) \leq c'f(n)\}\end{aligned}$$

Aus Rechenregel 13.1 wird bei Verwendung von Θ:

13.4 (Rechenregel) Für alle $f : \mathbb{N}_0 \to \mathbb{R}_0^+$ und alle Konstanten $a, b \in \mathbb{R}_+$ gilt: $\Theta(af(n)) = \Theta(bf(n))$.

13.2.2 Notation für obere und untere Schranken des Wachstums

In Fällen wie der unbekannten Anzahl von Durchläufen der **while**-Schleife in Algorithmus 13.1 genügt es nicht, wenn man konstante Faktoren ignorieren kann. Man kennt nur den schlimmsten Fall: $\sum_{i=1}^{n-1} i = n(n-1)/2$. Dementsprechend ist im schlimmsten Fall die Laufzeit in $\Theta(n^2)$; im allgemeinen kann sie aber auch kleiner sein. Um das bequem ausdrücken und weiterhin konstante Faktoren ignorieren zu können, definiert man:

$O(f)$, $\Omega(f)$

$$\begin{aligned}O(f(n)) &= \{g(n) \mid \exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \leq cf(n)\} \\ \Omega(f(n)) &= \{g(n) \mid \exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \geq cf(n)\}\end{aligned}$$

Man schreibt gelegentlich auch

$g \preceq f$, $g \succeq f$

$$\begin{aligned}g \preceq f &\text{ falls } g \in O(f) \\ g \succeq f &\text{ falls } g \in \Omega(f)\end{aligned}$$

und sagt, dass g *asymptotisch höchstens so schnell wie* f wächst (falls $g \preceq f$) bzw. dass g *asymptotisch mindestens so schnell wie* f wächst (falls $g \succeq f$).

Betrachten wir drei Beispiele.

- Es ist $10^{90}n^7 \in O(10^{-90}n^8)$, denn für $c = 10^{180}$ ist für alle $n \geq 0$: $10^{90}n^7 \leq c \cdot 10^{-90}n^8$.

  Dieses Beispiel soll noch einmal deutlich machen, dass man in $O(\cdot)$ usw. richtig große Konstanten „verstecken" kann. Ob ein hypthetischer Algorithmus mit Laufzeit in $O(n^8)$ in der Praxis wirklich tauglich ist, hängt durchaus davon ab, ob die Konstante c bei der oberen Schranke cn^8 eher im Bereich 10^{-90} oder im Bereich 10^{90} ist.
- Mitunter trifft man auch die Schreibweise $O(1)$ an. Was ist das? Die Definiti-

$O(1)$

on sagt, dass das alle Funktionen $g(n)$ sind, für die es eine Konstante $c \in \mathbb{R}_+$ gibt und ein $n_0 \in \mathbb{N}_0$, so dass für alle $n \geq n_0$ gilt:

$$g(n) \leq c \cdot 1 = c$$

Das sind also alle Funktionen, die man durch Konstanten beschränken kann. Dazu gehören etwa alle konstanten Funktionen, aber auch Funktionen wie $3 + \sin(n)$. (So etwas habe ich aber noch nie eine Rolle spielen sehen.)

- Weiter vorne hatten wir benutzt, dass der Quotient n^2/n nicht für alle hinreichend großen n durch eine Konstante beschränkt werden kann. Also gilt *nicht* $n^2 \preceq n$. Andererseits gilt (machen Sie sich das bitte kurz klar) $n \preceq n^2$. Die Relation $\preceq$ ist also *nicht* symmetrisch.

  Allgemein gilt für positive reelle Konstanten $0 < a < b$, dass $n^a \preceq n^b$ ist, aber *nicht* $n^b \preceq n^a$. Ebenso ist für reelle Konstanten a und b, die beide echt größer 1 sind, $n^a \preceq b^n$ aber *nicht* $b^n \preceq n^a$.

Man muss nur in der Ungleichung $g(n) \leq cf(n)$ die Konstante auf die andere Seite bringen und schon kann man sich davon überzeugen, dass gilt:

13.5 (Rechenregel) Für alle Funktionen $f : \mathbb{N}_0 \to \mathbb{R}_0^+$ und $g : \mathbb{N}_0 \to \mathbb{R}_0^+$ gilt:

$$g(n) \in \mathrm{O}(f(n)) \iff f(n) \in \Omega(g(n)), \quad \text{also} \quad g \preceq f \iff f \succeq g$$

Man kann auch zeigen:

$$\Theta(f(n)) = \mathrm{O}(f(n)) \cap \Omega(f(n))$$
$$\text{also} \qquad g \asymp f \iff g \preceq f \land g \succeq f$$

13.2.3 Die furchtbare Schreibweise

Damit Sie bei Lektüre von Büchern und Aufsätzen alles verstehen, was dort mit Hilfe von $\Theta(\cdot)$, $\mathrm{O}(\cdot)$ und $\Omega(\cdot)$ aufgeschrieben steht, müssen wir Ihnen nun leider noch etwas mitteilen. Man benutzt eine (unserer Meinung nach) sehr unschöne (um nicht zu sagen irreführende) Variante der eben eingeführten Notation. Aber weil sie so verbreitet ist, muten wir sie Ihnen zu. Man schreibt nämlich

$$\begin{aligned} g(n) &= \mathrm{O}(f(n)) \quad \text{statt} \quad g(n) \in \mathrm{O}(f(n)) \ , \\ g(n) &= \Theta(f(n)) \quad \text{statt} \quad g(n) \in \Theta(f(n)) \ , \\ g(n) &= \Omega(f(n)) \quad \text{statt} \quad g(n) \in \Omega(f(n)) \ . \end{aligned}$$

Die Ausdrücke auf der linken Seite sehen zwar aus wie Gleichungen, *aber es sind keine!* Lassen Sie daher bitte immer *große Vorsicht* walten:

- Es ist *falsch*, aus $g(n) = O(f_1(n))$ und $g(n) = O(f_2(n))$ zu folgern, dass $O(f_1(n)) = O(f_2(n))$ ist.
- Es ist *falsch*, aus $g_1(n) = O(f(n))$ und $g_2(n) = O(f(n))$ zu folgern, dass $g_1(n) = g_2(n)$ ist.

Noch furchtbarer ist, dass manchmal etwas von der Art $O(g) = O(f)$ geschrieben wird, *aber nur die Inklusion* $O(g) \subseteq O(f)$ *gemeint ist.*

Auch Ronald Graham, Donald Knuth und Oren Patashnik sind nicht begeistert, wie man den Ausführungen auf den Seiten 432 und 433 ihres Buches *Concrete Mathematics* (Graham, Knuth und Patashnik 1989) entnehmen kann. Sie geben vier Gründe an, warum man das doch so macht. Der erste ist Tradition; der zweite ist Tradition; der dritte ist Tradition. Der vierte ist, dass die Gefahr, etwas falsch zu machen, oft eher klein ist. Also dann: toi toi toi.

13.2.4 Rechnen im O-Kalkül

Ist $g_1 \preceq f_1$ und $g_2 \preceq f_2$, dann ist auch $g_1 + g_2 \preceq f_1 + f_2$. Ist umgekehrt $g \preceq f_1 + f_2$, dann kann man g in der Form $g = g_1 + g_2$ schreiben mit $g_1 \preceq f_1$ und $g_2 \preceq f_2$. Das schreiben wir auch so:

13.6 Lemma. Für alle Funktionen $f_1, f_2 : \mathbb{N}_0 \to \mathbb{R}_0^+$ gilt:

$$O(f_1) + O(f_2) = O(f_1 + f_2)$$

Dabei muss allerdings erst noch etwas definiert werden: die „Summe“ von Mengen (von Funktionen). So etwas nennt man manchmal *Komplexoperationen* und definiert sie so: Sind M_1 und M_2 Mengen von Elementen, die man addieren bzw. multiplizieren kann, dann sei

Komplexoperationen

$$M_1 + M_2 = \{g_1 + g_2 \mid g_1 \in M_1 \land g_2 \in M_2\}$$
$$M_1 \cdot M_2 = \{g_1 \cdot g_2 \mid g_1 \in M_1 \land g_2 \in M_2\}$$

Für Funktionen sei Addition und Multiplikation (natürlich?) argumentweise definiert. Dann ist z. B.

$$O\left(n^3\right) + O\left(n^3\right) = \{g_1(n) + g_2(n) \mid g_1 \in O\left(n^3\right) \land g_2 \in O\left(n^3\right)\}$$

z. B.

$$(2n^3 - n^2) + 7n^2 = 2n^3 + 6n^2 \in O\left(n^3\right) + O\left(n^3\right)$$

Wenn eine der Mengen M_i einelementig ist, lässt man manchmal die Mengenklammern darum weg und schreibt zum Beispiel bei Zahlenmengen

statt $\{3\} \cdot \mathbb{N}_0 + \{1\}$ kürzer $3\mathbb{N}_0 + 1$

oder bei Funktionenmengen

$$\text{statt} \qquad \{n^3\} + \mathrm{O}\left(n^2\right) \qquad \text{kürzer} \qquad n^3 + \mathrm{O}\left(n^2\right)$$

Solche Komplexoperationen sind übrigens nichts Neues für Sie. Die Definition des Produkts formaler Sprachen passt genau in dieses Schema (siehe Unterabschnitt 5.2.1).

13.7 **Beweis. (von Lemma 13.6)** Wir beweisen die beiden Inklusionen getrennt.

„⊆“: Wenn $g_1 \in \mathrm{O}(f_1)$, dann existiert ein $c_1 \in \mathbb{R}_+$ und ein n_{01}, so dass für alle $n \geq n_{01}$ gilt: $g_1(n) \leq c_1 f_1(n)$. Und wenn $g_2 \in \mathrm{O}(f_2)$, dann existiert ein $c_2 \in \mathbb{R}_+$ und ein n_{02}, so dass für alle $n \geq n_{02}$ gilt: $g_2(n) \leq c_2 f_2(n)$.

Folglich gilt für alle $n \geq n_0 = \max(n_{01}, n_{02})$ und für $c = \max(c_1, c_2) \in \mathbb{R}_0^+$:

$$\begin{aligned} g_1(n) + g_2(n) &\leq c_1 f_1(n) + c_2 f_2(n) \\ &\leq c f_1(n) + c f_2(n) \\ &= c(f_1(n) + f_2(n)) \end{aligned}$$

„⊇“: Wenn $g \in \mathrm{O}(f_1 + f_2)$ ist, dann gibt es $c \in \mathbb{R}_+$ und ein n_0, so dass für alle $n \geq n_0$ gilt: $g(n) \leq c(f_1(n) + f_2(n))$.

Man definiere nun eine Funktion $g_1 : \mathbb{N}_0 \to \mathbb{R}_0^+$ vermöge

$$g_1(n) = \begin{cases} g(n) & \text{falls } g(n) \leq c f_1(n) \\ c f_1(n) & \text{falls } g(n) > c f_1(n) \end{cases}$$

Dann ist offensichtlich $g_1 \in \mathrm{O}(f_1)$.

Außerdem ist $g_1(n) \leq g(n)$ und folglich $g_2(n) = g(n) - g_1(n)$ stets größer gleich 0. Behauptung: $g_2 \in \mathrm{O}(f_2)$. Sei $n \geq n_0$. Dann ist

$$\begin{aligned} g_2(n) &= g(n) - g_1(n) \\ &= \begin{cases} 0 & \text{falls } g(n) \leq c f_1(n) \\ g(n) - c f_1(n) & \text{falls } g(n) > c f_1(n) \end{cases} \\ &\leq \begin{cases} 0 & \text{falls } g(n) \leq c f_1(n) \\ c(f_1(n) + f_2(n)) - c f_1(n) & \text{falls } g(n) > c f_1(n) \end{cases} \\ &= \begin{cases} 0 & \text{falls } g(n) \leq c f_1(n) \\ c f_2(n) & \text{falls } g(n) > c f_1(n) \end{cases} \\ &\leq c f_2(n)\,, \end{aligned}$$

also $g_2 \in \mathrm{O}(f_2)$. Also ist $g = g_1 + g_2 \in \mathrm{O}(f_1) + \mathrm{O}(f_2)$.

13.8 (Rechenregel) Wenn $g_1 \preceq f_1$ ist, und wenn $g_1 \asymp g_2$ und $f_1 \asymp f_2$, dann gilt auch $g_2 \preceq f_2$.

13.9 (Rechenregel) Wenn $g \preceq f$ ist, also $g \in \mathrm{O}(f)$, dann ist auch $\mathrm{O}(g) \subseteq \mathrm{O}(f)$ und $\mathrm{O}(g+f) = \mathrm{O}(f)$.

Es gibt noch eine Reihe weiterer Rechenregeln für $\mathrm{O}(\cdot)$ und außerdem ähnliche für $\Theta(\cdot)$ und $\Omega(\cdot)$ (zum Beispiel Analoga zu Lemma 13.6). Wir verzichten hier darauf, sie alle aufzuzählen.

13.3 MATRIXMULTIPLIKATION

Wir wollen uns nun noch einmal ein bisschen genauer mit der Multiplikation von $n \times n$-Matrizen beschäftigen, und uns dabei insbesondere für

- die Anzahl $N_{add}(n)$ elementarer Additionen ist und
- die Anzahl $N_{mult}(n)$ elementarer Multiplikationen

interessieren. Deren Summe bestimmt im wesentlichen (d. h. bis auf konstante Faktoren) die Laufzeit.

13.3.1 Rückblick auf die Schulmethode

Die „Schulmethode" für die Multiplikation von 2×2-Matrizen geht so:

		b_{11}	b_{12}
		b_{21}	b_{22}
a_{11}	a_{12}	$a_{11}b_{11} + a_{12}b_{21}$	$a_{11}b_{12} + a_{12}b_{22}$
a_{21}	a_{22}	$a_{21}b_{11} + a_{22}b_{21}$	$a_{21}b_{12} + a_{22}b_{22}$

Wie man sieht ist dabei

- $N_{mult}(2) = 2^2 \cdot 2 = 8$ und
- $N_{add}(2) = 2^2 \cdot (2-1) = 4$.

Wenn n gerade ist (auf diesen Fall wollen uns im folgenden der einfacheren Argumentation wegen beschränken), dann ist die Schulmethode für $n \times n$ Matrizen äquivalent zum Fall, dass man 2×2 Blockmatrizen mit Blöcken der Größe $n/2$ vorliegen hat, die man nach dem gleichen Schema wie oben multiplizieren kann:

		B_{11}	B_{12}
		B_{21}	B_{22}
A_{11}	A_{12}	$A_{11}B_{11} + A_{12}B_{21}$	$A_{11}B_{12} + A_{12}B_{22}$
A_{21}	A_{22}	$A_{21}B_{11} + A_{22}B_{21}$	$A_{21}B_{12} + A_{22}B_{22}$

Das sind 4 Additionen von Blockmatrizen und 8 Multiplikationen von Blockmatrizen. Die Anzahl elementarer Operationen ist also

- $N_{mult}(n) = 8 \cdot N_{mult}(n/2)$ und
- $N_{add}(n) = 8 \cdot N_{add}(n/2) + 4 \cdot (n/2)^2 = 8 \cdot N_{add}(n/2) + n^2$.

Wir betrachten den Fall $n = 2^k$ (die anderen Fälle gehen im Prinzip ähnlich). Dann ergibt sich aus $N_{mult}(n) = 8 \cdot N_{mult}(n/2)$:

$$\begin{aligned} N_{mult}(2^k) &= 8 \cdot N_{mult}(2^{k-1}) = 8 \cdot 8 \cdot N_{mult}(2^{k-2}) = \cdots = 8^k \cdot N_{mult}(1) \\ &= 8^k = 8^{\log_2(n)} = 2^{3\log_2(n)} = 2^{\log_2(n)\cdot 3} = n^3 \end{aligned}$$

Dass man statt der Pünktchen einen Induktionsbeweis führen kann, ist Ihnen inzwischen klar, richtig?

Aus $N_{add}(n) = 8 \cdot N_{add}(n/2) + n^2$ ergibt sich analog:

$$\begin{aligned} N_{add}(2^k) &= 8 \cdot N_{add}(2^{k-1}) + 4^k \\ &= 8 \cdot 8 \cdot N_{add}(2^{k-2}) + 8 \cdot 4^{k-1} + 4^k = \cdots \\ &= 8 \cdot 8 \cdot N_{add}(2^{k-2}) + 2 \cdot 4^k + 4^k = \cdots \\ &= 8^k N_{add}(2^0) + (2^{k-1} + \cdots 1) \cdot 4^k = \\ &= 2^k \cdot 4^k \cdot 0 + (2^k - 1) \cdot 4^k = \\ &= 2^k \cdot 4^k - 4^k = n^3 - n^2 \end{aligned}$$

13.3.2 Algorithmus von Strassen

Nun kommen wir zu der Idee von Strassen (1969). Er hat bemerkt, dass man die Blockmatrizen C_{ij} des Matrixproduktes auch wie folgt berechnen kann:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

und dann

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

Das sieht erst einmal umständlicher aus, denn es sind 18 Additionen von Blockmatrizen statt nur 4 bei der Schulmethode. Aber es sind nur 7 Multiplikationen von Blockmatrizen statt 8! Und das zahlt sich aus, denn im Gegensatz zum skalaren Fall sind Multiplikationen aufweniger als Additionen. Für die Anzahl elementarer Operationen ergibt sich:

- $N_{mult}(n) = 7 \cdot N_{mult}(n/2)$
- $N_{add}(n) = 7 \cdot N_{add}(n/2) + 18 \cdot (n/2)^2 = 7 \cdot N_{add}(n/2) + 4.5 \cdot n^2$

Für den Fall $n = 2^k$ ergibt sich:

$$\begin{aligned}
N_{mult}(2^k) &= 7 \cdot N_{mult}(2^{k-1}) = 7 \cdot 7 \cdot N_{mult}(2^{k-2}) = \cdots = 7^k \cdot N_{mult}(1) \\
&= 7^k = 7^{\log_2(n)} = 2^{\log_2 7 \cdot \log_2(n)} = n^{\log_2 7} \approx n^{2.807\ldots}
\end{aligned}$$

Analog erhält man auch für die Anzahl der Additionen $N_{add}(n) \in \Theta(n^{\log_2 7})$. Die Gesamtzahl elementarer arithmetischer Operationen ist also in $\Theta(n^{\log_2 7}) + \Theta(n^{\log_2 7}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807\ldots})$.

Es gibt sogar Algorithmen, die asymptotisch noch weniger Operationen benötigen. Das in dieser Hinsicht beste Ergebnis stammt von Coppersmith und Winograd (1990), die mit $O\left(n^{2.376\ldots}\right)$ elementaren arithmetischen Operationen auskommen. Auch dieses Verfahren benutzt wie das von Strasse eine Vorgehensweise, die man in vielen Algorithmen wiederfindet: Man teilt die Probleminstanz in kleinere Teile auf, die man wie man sagt rekursiv nach dem gleichen Verfahren bearbeitet und die Teilergebnisse dann benutzt, um das Resultat für die ursprüngliche Eingabe zu berechnen. Man spricht von *„teile und herrsche“* (engl. *divide and conquer*).

13.4 ASYMPTOTISCHES VERHALTEN „IMPLIZIT“ DEFINIERTER FUNKTIONEN

Sie werden im Laufe der kommenden Semester viele Algorithmen kennenlernen, bei denen wie bei Strassens Algorithmus für Matrixmultiplikation das Prinzip „Teile und Herrsche“ benutzt wird. In den einfacheren Fällen muss man zur Bearbeitung eines Problems der Größe n eine konstante Anzahl a von Teilprobleme gleicher Größe n/b lösen. Die zusätzlichen Kosten zur Berechnung des eigentlichen Ergebnisses mögen zusätzlich einen Aufwand $f(n)$ kosten. Das beinhaltet

auch den unter Umständen erforderlichen Aufwand zum Erzeugen der Teilprobleme.

Dann ergibt sich für Abschätzung (z. B.) der Laufzeit $T(n)$ eine Rekursionsformel, die grob gesagt von der Form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

ist. Dabei ist sinnvollerweise $a \geq 1$ und $b > 1$.

Obige Rekursionsformel ist unpräzise, denn Problemgrößen sind immer ganzzahlig, n/b im allgemeinen aber nicht. Es zeigt sich aber, dass sich jedenfalls in den nachfolgend aufgeführten Fällen diese Ungenauigkeit im folgenden Sinne nicht auswirkt: Wenn man in der Rekursionsformel n/b durch $\lfloor n/b \rfloor$ oder durch $\lceil n/b \rceil$ ersetzt oder gar durch $\lfloor n/b + c \rfloor$ oder durch $\lceil n/b + c \rceil$ für eine Konstante c, dann behalten die folgenden Aussagen ihre Gültigkeit.

Wenn zwischen den Konstanten a und b und der Funktion $f(n)$ gewisse Zusammenhänge bestehen, dann kann man ohne viel Rechnen (das schon mal jemand anders für uns erledigt hat) eine Aussage darüber machen, wie stark $T(n)$ wächst.

Es gibt drei wichtige Fälle, in denen jeweils die Zahl $\log_b a$ eine Rolle spielt:

Fall 1: Wenn $f(n) \in \mathrm{O}\left(n^{\log_b a - \varepsilon}\right)$ für ein $\varepsilon > 0$ ist, dann ist $T(n) \in \Theta\left(n^{\log_b a}\right)$.

Fall 2: Wenn $f(n) \in \Theta\left(n^{\log_b a}\right)$ ist, dann ist $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$.

Fall 3: Wenn $f(n) \in \Omega\left(n^{\log_b a + \varepsilon}\right)$ für ein $\varepsilon > 0$ ist, und wenn es eine Konstante d gibt mit $0 < d < 1$, so dass für alle hinreichend großen n gilt $af(n/b) \leq df(n)$, dann ist $T(n) \in \Theta(f(n))$.

Dass die Aussagen in diesen drei Fällen richtig sind, bezeichnet man manchmal als *Mastertheorem*, obwohl es sich sicherlich um keine sehr tiefschürfenden Erkenntnisse handelt.

Mastertheorem

Betrachten wir als Beispiele noch einmal die Matrixmultiplikation. Als „Problemgröße" n benutzen wir die Zeilen- bzw. Spaltenzahl. Der Fall von $n \times n$-Matrizen wird auf den kleineren Fall von $n/2 \times n/2$-Matrizen zurückgeführt.

Bei der Schulmethode haben wir $a = 8$ Multiplikationen kleinerer Matrizen der Größe $n/2$ durchzuführen; es ist also $b = 2$. In diesem Fall ist $\log_b a = \log_2 8 = 3$. Der zusätzliche Aufwand besteht in 4 kleinen Matrixadditionen, so dass $f(n) = 4 \cdot n^2/4 = n^2$. Damit ist $f(n) \in \mathrm{O}\left(n^{3-\varepsilon}\right)$ (z. B. für $\varepsilon = 1/2$) und der erste Fall des Mastertheorems besagt, dass folglich $T(n) \in \Theta\left(n^3\right)$. (Und das hatten wir uns weiter vorne tatsächlich auch klar gemacht.)

Bei Strassens geschickterer Methode sind nur $a = 7$ Multiplikationen kleinerer Matrizen der Größe $n/2$ durchzuführen (es ist also wieder $b = 2$). In diesem Fall ist $\log_b a = \log_2 7 \approx 2.807\ldots$. Der zusätzliche Aufwand besteht in 18 kleinen

Matrixadditionen, so dass $f(n) = 18 \cdot n^2/4 \in \Theta(n^2)$. Auch hier gilt für ein geeignetes ε wieder $f(n) \in O\left(n^{\log_b a - \varepsilon}\right) = O\left(n^{\log_2 7 - \varepsilon}\right)$. Folglich benötigt Strassens Algorithmus $T(n) \in \Theta\left(n^{\log_2 7}\right) = \Theta\left(n^{2.807\cdots}\right)$ Zeit.

13.5 UNTERSCHIEDLICHES WACHSTUM EINIGER FUNKTIONEN

Wir hatten schon darauf hingewiesen, dass gilt:

1. Für positive reelle Konstanten $0 < a < b$ ist $n^a \preceq n^b$, aber *nicht* $n^b \preceq n^a$.
2. Für reelle Konstanten a und b, die beide echt größer 1 sind, gilt $n^a \preceq b^n$ aber *nicht* $b^n \preceq n^a$.

Zur Veranschaulichung des ersten Punktes sind in Abbildung 13.3 die Funktionen $f(x) = x$, $f(x) = x^2$ und $f(x) = x^3$ geplottet. Allerdings fällt in der gewählten Darstellung $f(x) = x$ nahezu mit der x-Achse zusammen. Wie man sieht, wird in doppelt-logarithmischen Plots jede Funktion x^d durch eine Gerade repräsentiert (deren Steigung d ist, wenn die „Einheiten" auf beiden Achsen gleich sind). Allgemeine Polynomfunktionen werden durch Linien repräsentiert, die sich für große n einer Geraden anschmiegen.

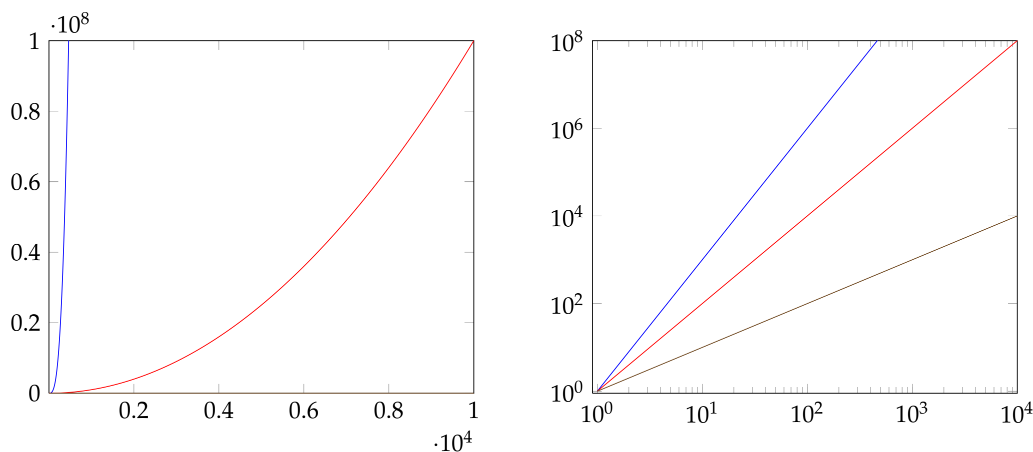


Abbildung 13.3: Die Funktionen $f(x) = x$, $f(x) = x^2$ und $f(x) = x^3$ in Plots mit linear skalierten Achsen (links) und in doppelt logarithmischer Darstellung (rechts); $f(x) = x$ ist praktisch nicht von der x-Achse zu unterscheiden.

Abbildung 13.4 zeigt in doppelt-logarithmischer Darstellung zum Vergleich zwei Polynom- und zwei Exponentialfunktionen. Der Unterschied sollte klar sein.

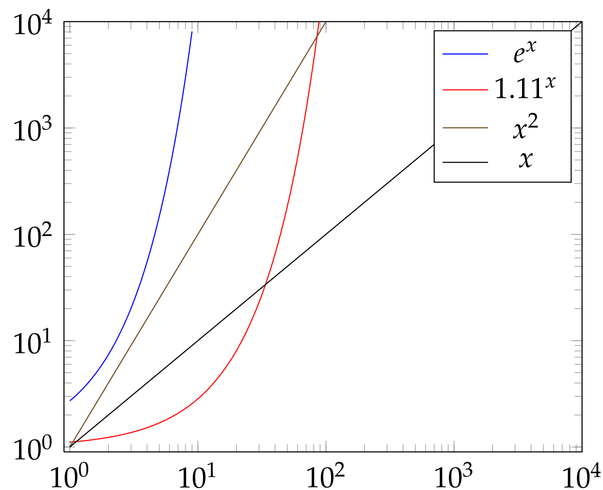


Abbildung 13.4: Zwei Polynom- und zwei Exponentialfunktionen im Vergleich; doppelt-logarithmische Darstellung.

13.6 AUSBLICK

Algorithmen, bei denen die anderen beiden Fälle des Mastertheorems zum Tragen kommen, werden Sie im kommenden Semester in der Vorlesung „Algorithmen 1“ kennenlernen.

Manchmal wird „Teile und Herrsche“ auch in etwas komplizierterer Form angewendet (zum Beispiel mit deutlich unterschiedlich großen Teilproblemen). Für solche Situationen gibt Verallgemeinerungen obiger Aussagen (Satz von Akra und Bazzi).

LITERATUR

Coppersmith, Don und Shmuel Winograd (1990). "Matrix Multiplication via Arithmetic Progressions". In: *Journal of Symbolic Computation* 9, S. 251–280.

Graham, Ronald L., Donald E. Knuth und Oren Patashnik (1989). *Concrete Mathematics*. Addison-Wesley.

Strassen, Volker (1969). "Gaussian Elimination Is Not Optimal". In: *Numerische Mathematik* 14, S. 354–356.

14 ENDLICHE AUTOMATEN

14.1 ERSTES BEISPIEL: EIN GETRÄNKEAUTOMAT

Als erstes Beispiel betrachten wir den folgenden primitiven Getränkeautomaten (siehe Abbildung 14.1). Man kann nur 1-Euro-Stücke einwerfen und vier Tasten drücken: Es gibt zwei Auswahltasten für Mineralwasser rein und Zitronensprudel zitro, eine Abbruch-Taste C und eine OK-Taste.

- Jede Flasche Sprudel kostet 1 Euro.
- Es kann ein Guthaben von 1 Euro gespeichert werden. Wirft man weitere Euro-Stücke ein, werden sie sofort wieder ausgegeben.
- Wenn man mehrfach Auswahltasten drückt, wird der letzte Wunsch gespeichert.
- Bei Drücken der Abbruch-Taste wird alles bereits eingeworfenen Geld wieder zurückgegeben und kein Getränkewunsch mehr gespeichert.
- Drücken der OK-Taste wird ignoriert, solange noch kein Euro eingeworfen wurde oder keine Getränkesorte ausgewählt wurde.

  Andernfalls wird das gewünschte Getränk ausgeworfen.

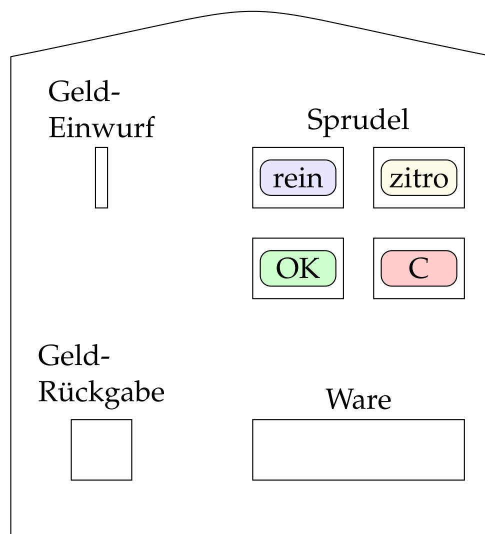


Abbildung 14.1: Ein primitiver Getränkeautomat

Dieser Getränke*automat* im umgangssprachlichen Sinne ist auch ein *endlicher Automat* wie sie in der Informatik an vielen Stellen eine Rolle spielen.

Offensichtlich muss der Automat zwischen den vielen Eingaben, die sein Verhalten beeinflussen können (Geldeinwürfe und Getränkewahl), gewisse Nachrichten (im Sinne von Abschnitt 2.3) speichern. Und zwar

- zum einen, ob schon ein 1-Euro-Stück eingeworfen wurde, und
- zum anderen, ob schon ein Getränk ausgewählt wurde und wenn ja, welches.

Man kann das zum Beispiel modellieren durch Paare (x, y), bei denen die Komponente $x \in \{0, 1\}$ den schon eingeworfenen Geldbetrag angibt und Komponente $y \in \{\texttt{-}, \texttt{R}, \texttt{Z}\}$ die Getränkewahl repräsentiert. Wir wollen $Z = \{0, 1\} \times \{\texttt{-}, \texttt{R}, \texttt{Z}\}$ die Menge der möglichen Zustände des Automaten nennen.

Der erste wesentliche Aspekt jedes Automaten ist, dass Einflüsse von außen, die wir *Eingaben* nennen, zu *Zustandsänderungen* führen. Bei dem Getränkeautomaten sind mögliche Eingaben der Einwurf eines 1-Euro-Stückes und das Drücken einer der Tasten (wir wollen davon absehen, dass jemand vielleicht mehrere Tasten gleichzeitig drückt). Wir modellieren die möglichen Eingaben durch Symbole `1`, `R`, `Z`, `C` und `0`, die zusammen das sogenannte *Eingabealphabet* X bilden. Ein aktueller Zustand $z \in Z$ und ein Eingabesymbol $x \in X$ legen — jedenfalls bei dem Getränkeautomaten — eindeutig den neuen Zustand fest. Dieser Aspekt eines endlichen Automaten kann also durch eine endliche Funktion $f : Z \times X \to Z$ formalisiert werden. In vielen Fällen ist es hilfreich, diese Funktion nicht durch eine Tabelle zu spezifizieren, sondern durch eine Darstellung als Graph wie in Abbildung 14.2.

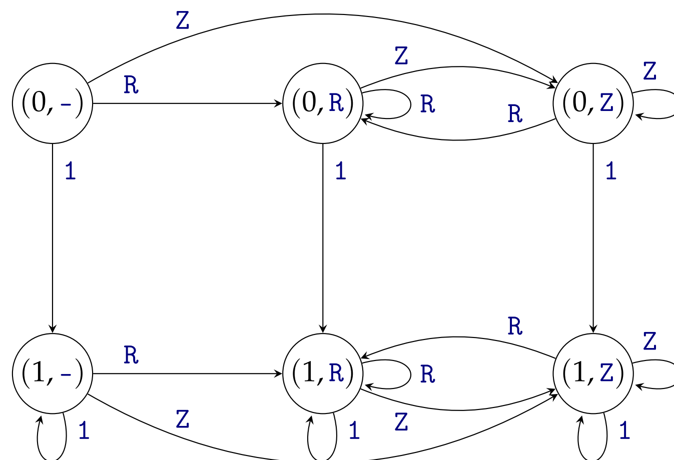


Abbildung 14.2: Graphische Darstellung der Zustandsübergänge des Getränkeautomaten für die drei Eingabesymbole `1`, `R` und `Z`.

Die Zustände sind die Knoten des Graphen, und es gibt gerichtete Kanten, die mit Eingabesymbolen beschriftet sind. Für jedes $z \in Z$ und jedes $x \in X$ führt eine mit

x beschriftete Kante von z nach $f(z, x)$.

Aus Gründen der Übersichtlichkeit sind in Abbildung 14.2 zunächst einmal nur die Zustandsübergänge für die Eingabesymbole 1, R und Z dargestellt. Hinzu kommen noch die aus Abbildung 14.3 für die Eingaben C und O. Wenn bei einem Zustand für mehrere Eingabesymbole der Nachfolgezustand der gleiche ist, dann zeichnet man oft nur einen Pfeil und beschriftet ihn mit allen Eingabesymbolen, durch Kommata getrennt. In Abbildung 14.3 betrifft das den Übergang von Zustand (1, R) nach Zustand (0, -) für die Eingaben O und C (und analog von den Zuständen (1, Z) und (0, -)).

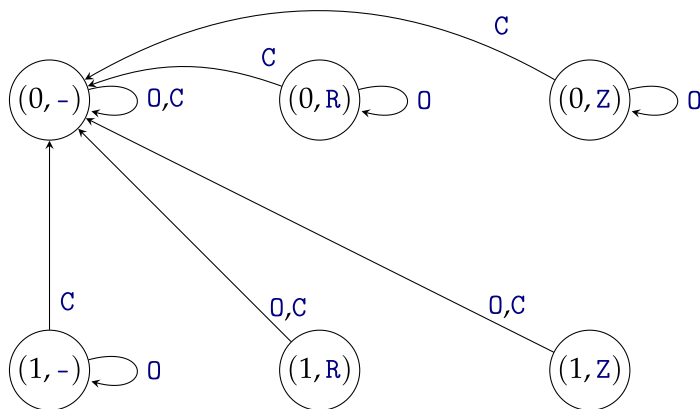


Abbildung 14.3: Graphische Darstellung der Zustandsübergänge des Getränkeautomaten für die Eingabesymbole C und O.

Stellt man alle Übergänge in einem Diagramm dar, ergibt sich Abbildung 14.4.
Der zweite wichtige Aspekt jedes Automaten ist, dass sich seine Arbeit, im vorliegenden Fall also die Zustandsübergänge, zumindest von Zeit zu Zeit in irgendeiner Weise auf seine Umwelt auswirken (warum sollte man ihn sonst arbeiten lassen). Beim Getränkeautomaten zeigt sich das in der Ausgabe von Geldstücken und Getränkeflaschen. Dazu sehen wir eine Menge $Y = \{1, R, Z\}$ von Ausgabesymbolen vor, deren Bedeutung klar sein sollte. Beim Getränkeautomaten ist es plausibel zu sagen, dass jedes Paar (z, x) von aktuellem Zustand z und aktueller Eingabe x eindeutig einen neuen Zustand festlegt, es ebenso eindeutig eine Ausgabe festlegt. Wir formalisieren das als eine Funktion $g : Z \times X \to Y^*$. Als Funktionswerte sind also Wörter von Symbolen aus Y erlaubt, einschließlich des leeren Wortes, das man zur Modellierung von „keine Ausgabe“ verwenden kann.

Auch die Funktion g wird üblicherweise in den Zustandsübergangsdiagrammen mit angegeben, und zwar an der jeweiligen Kante neben dem Eingabesym-

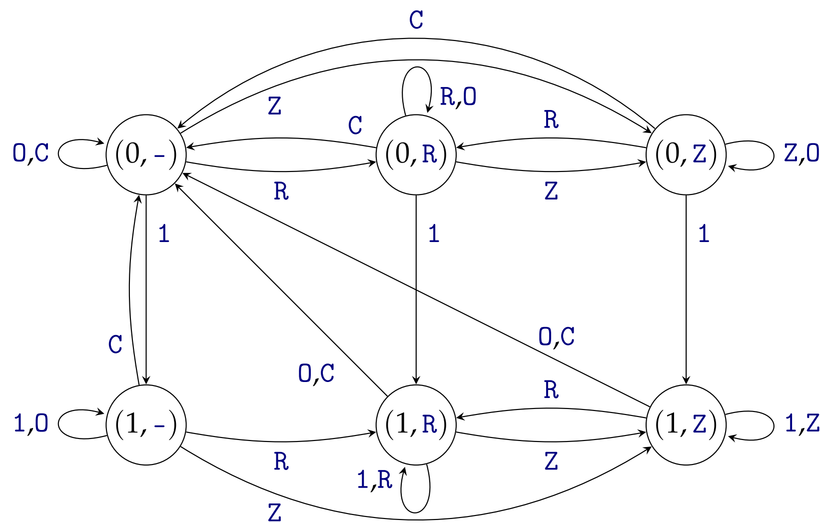


Abbildung 14.4: Graphische Darstellung der Zustandsübergänge des Getränkeautomaten für alle Eingabesymbole.

bol, von diesem durch einen senkrechten Strich getrennt (manche nehmen auch ein Komma). Aus Abbildung 14.4 ergibt sich Abbildung 14.5.

14.2 Mealy-Automaten

Mealy-Automat

Ein *(endlicher) Mealy-Automat* $A = (Z, z_0, X, f, Y, g)$ ist festgelegt durch

- eine endliche Zustandsmenge Z,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X,
- eine Zustandsüberführungsfunktion $f : Z \times X \to Z$,
- ein Ausgabealphabet Y,
- eine Ausgabefunktion $g : Z \times X \to Y^*$

Für einen Zustand $z \in Z$ und ein Eingabesymbol $x \in X$ ist $f(z, x)$ der Zustand nach Eingabe dieses einzelnen Symbols ausgehend von Zustand z. Gleichzeitig mit jedem Zustandsübergang wird eine Ausgabe produziert. Wir modellieren das als Wort $g(z, x) \in Y^*$. In graphischen Darstellungen von Automaten wird der Anfangszustand üblicherweise dadurch gekennzeichnet, dass man einen kleinen Pfeil auf ihn zeigen lässt, der *nicht* bei einem anderen Zustand anfängt.

Manchmal möchte man auch über den nach Eingabe eines ganzen Wortes $w \in X^*$ erreichten Zustand oder über alle dabei durchlaufenen Zustände (ein-

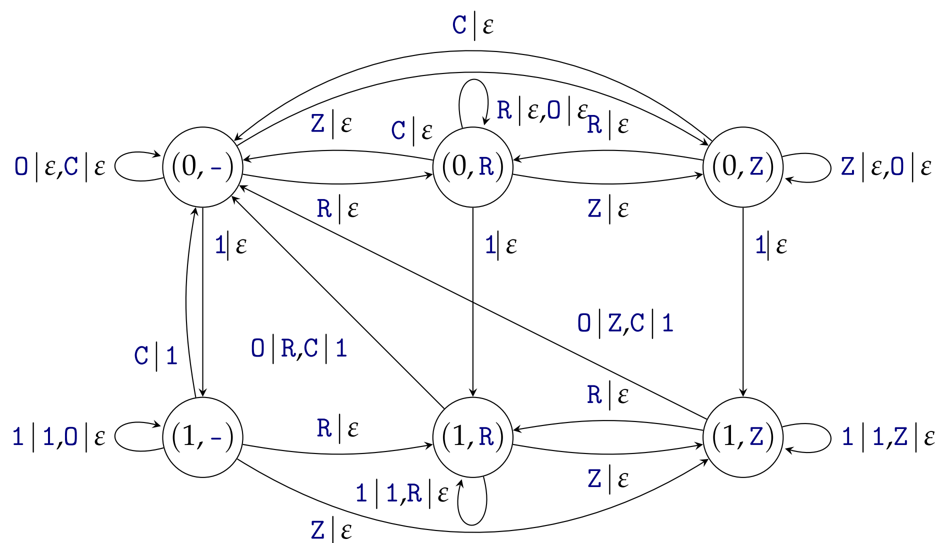


Abbildung 14.5: Graphische Darstellung der Zustandsübergänge und Ausgaben des Getränkeautomaten für alle Eingabesymbole.

schließlich des Anfangszustands) reden. Und manchmal will man auch bei den Ausgaben über allgemeinere Aspekte sprechen.

Um das bequem hinzuschreiben zu können, definieren wir Abbildungen f^* und f^{**} und analog g^* und g^{**}. Dabei soll der erste Stern andeuten, dass zweites Argument nicht ein einzelnes Eingabesymbol sondern ein ganzes Wort von Eingabesymbolen ist; und der zweite Stern soll gegebenenfalls andeuten, dass wir uns nicht für einen einzelnen Funktionswert (von f bzw. g) interessieren, sondern wiederum für ein ganzes Wort von ihnen. Als erstes legen wir $f^* : Z \times X^* \to Z$ fest:

f^*

$$\begin{aligned} f^*(z, \varepsilon) &= z \\ \forall w \in X^* : \forall x \in X : \quad f^*(z, wx) &= f(f^*(z, w), x) \end{aligned}$$

Alternativ hätte man auch definieren können:

$$\begin{aligned} \bar{f}^*(z, \varepsilon) &= z \\ \forall w \in X^* : \forall x \in X : \quad \bar{f}^*(z, xw) &= \bar{f}^*(f(z, x), w) \end{aligned}$$

Machen Sie sich bitte klar, dass beide Definitionen die gleiche Funktion liefern (also $f^* = \bar{f}^*$): Für Argumente $z \in Z$ und $w \in X^*$ ist $f^*(z, w)$ der Zustand, in dem der Automat sich am Ende befindet, wenn er in z startet und der Reihe

nach die Eingabesymbole von w eingegeben werden. Je nachdem, was für einen Beweis bequem ist, können Sie die eine oder die andere Definitionsvariante zu Grunde legen. Das gleiche gilt für die folgenden Funktionen. (Sie dürfen sich aber natürlich nicht irgendeine Definition aussuchen, sondern nur eine, die zur explizit angegebenen äquivalent ist.)

f^{**}

Da wir vielleicht auch einmal nicht nur über den am Ende erreichten Zustand, sondern bequem über alle der Reihe nach durchlaufenen (einschließlich des Zustands, in dem man anfängt) reden wollen, legen wir nun $f^{**} : Z \times X^* \to Z^*$ für alle $z \in Z$ wie folgt fest:

$$\begin{aligned} & f^{**}(z, \varepsilon) = z \\ \forall w \in X^* : \forall x \in X : \qquad & f^{**}(z, wx) = f^{**}(z, w) \cdot f(f^*(z, w), x) \end{aligned}$$

Auch hier gibt es wieder eine alternative Definitionsmöglichkeit, indem man nicht das letzte, sondern das erste Symbol des Eingabewortes separat betrachtet.

g^*

Nun zu den verallgemeinerten Ausgabefunktionen. Zuerst definieren wir die Funktion $g^* : Z \times X^* \to Y^*$, deren Funktionswert die zum letzten Eingabesymbol gehörende Ausgabe sein soll. Das geht für alle $z \in Z$ so:

$$\begin{aligned} & g^*(z, \varepsilon) = \varepsilon \\ \forall w \in X^* : \forall x \in X : \qquad & g^*(z, wx) = g(f^*(z, w), x) \end{aligned}$$

g^{**}

Um auch über die Konkatenation der zu allen Eingabesymbolen gehörenden Ausgaben reden zu können, definieren wir die Funktion $g^{**} : Z \times X^* \to Y^*$ für alle $z \in Z$ wie folgt:

$$\begin{aligned} & g^{**}(z, \varepsilon) = \varepsilon \\ \forall w \in X^* : \forall x \in X : \qquad & g^{**}(z, wx) = g^{**}(z, w) \cdot g^*(z, wx) \end{aligned}$$

14.3 MOORE-AUTOMATEN

Moore-Automat

Manchmal ist es näherliegend, sich vorzustellen, dass ein Automat „in jedem Zustand“ eine Ausgabe produziert, und nicht bei jedem Zustandsübergang. Dementsprechend ist ein *Moore-Automat* $A = (Z, z_0, X, f, Y, h)$ festgelegt durch

- eine endliche Zustandsmenge Z,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X,
- eine Zustandsüberführungsfunktion $f : Z \times X \to Z$,
- ein Ausgabealphabet Y,

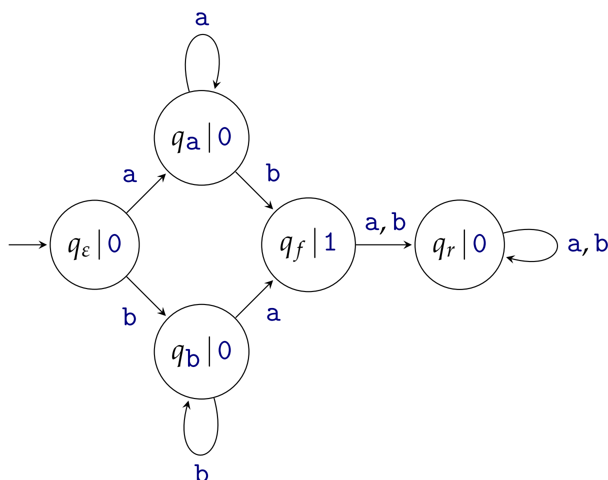


Abbildung 14.6: Ein einfacher Moore-Automat (aus der Dokumentation des LaTeX-Pakets `tikz`; modifiziert)

- eine Ausgabefunktion $h : Z \to Y^*$

Als einfaches Beispiel betrachten wir den Automaten in Abbildung 14.6 mit 5 Zuständen, Eingabealphabet $X = \{a, b\}$ und Ausgabealphabet $Y = \{0, 1\}$.

In jedem Knoten des Graphen sind jeweils ein Zustand z und, wieder durch einen senkrechten Strich getrennt, die zugehörige Ausgabe $h(z)$ notiert.

Die Definitionen für f^* und f^{**} kann man ohne Änderung von Mealy- zu Moore-Automaten übernehmen. Zum Beispiel ist im obigen Beispiel $f^*(q_\varepsilon, \mathtt{aaaba}) = q_r$, denn bei Eingabe aaaba durchläuft der Automat ausgehend von q_ε nacheinander die Zustände

f^*

f^{**}

$$q_\varepsilon \xrightarrow{\mathtt{a}} q_\mathtt{a} \xrightarrow{\mathtt{a}} q_\mathtt{a} \xrightarrow{\mathtt{a}} q_\mathtt{a} \xrightarrow{\mathtt{b}} q_f \xrightarrow{\mathtt{a}} q_r$$

Und folglich ist auch $f^{**}(q_\varepsilon, \mathtt{aaaba}) = q_\varepsilon q_\mathtt{a} q_\mathtt{a} q_\mathtt{a} q_f q_r$.

Bei Mealy-Automaten hatten wir zu g die Verallgemeinerungen g^* und g^{**} definiert, die als Argumente einen Startzustand $z \in Z$ und ein Eingabewort $w \in X^*$ erhielten und deren Funktionswerte „die letzte Ausgabe“ bzw. „die Konkatenation aller Ausgaben“ waren.

Entsprechendes kann man natürlich auch bei Moore-Automaten festlegen. Die Definitionen fallen etwas einfacher aus als bei Mealy-Automaten. Zum Beispiel ist $g^* : Z \times X^* \to Y^*$ einfach hinzuschreiben als $g^*(z, w) = h(f^*(z, w))$ (für alle $(z, w) \in Z \times X^*$). Also kurz: $g^* = h \circ f^*$.

g^*

Im obigen Beispielautomaten ist etwa

$$g^*(q_\varepsilon, \mathtt{aaaba}) = h(f^*(q_\varepsilon, \mathtt{aaaba})) = h(q_r) = \mathtt{0}$$

das zuletzt ausgegebene Bit, wenn man vom Startzustand ausgehend aaaba eingibt.

g^{**}

Auch $g^{**} : Z \times X^* \to Y^*$ für die Konkatenation aller Ausgaben ist leicht hinzuschreiben, wenn man sich des Begriffes des Homomorphismus erinnert, den wir in Unterabschnitt 10.2.2 kennengelernt haben. Die Ausgabeabbildung $h : Z \to Y^*$ induziert einen Homomorphismus $h^{**} : Z^* \to Y^*$ (indem man einfach h auf jeden Zustand einzeln anwendet). Damit ist für alle $(z, w) \in Z \times X^*$ einfach $g^{**}(z, w) = h^{**}(f^{**}(z, w))$, also $g^{**} = h^{**} \circ f^{**}$.

In unserem Beispiel ist

$$\begin{aligned} g^{**}(q_\varepsilon, \texttt{aaaba}) &= h^{**}(f^{**}(q_\varepsilon, \texttt{aaaba})) \\ &= h^{**}(q_\varepsilon q_\texttt{a} q_\texttt{a} q_\texttt{a} q_f q_r) \\ &= h(q_\varepsilon)h(q_\texttt{a})h(q_\texttt{a})h(q_\texttt{a})h(q_f)h(q_r) \\ &= \texttt{000010} \end{aligned}$$

14.4 ENDLICHE AKZEPTOREN

Ein besonders wichtiger Sonderfall endlicher Moore-Automaten sind sogenannte endliche Akzeptoren. Unser Beispiel im vorangegangenen Abschnitt war bereits einer.

Die Ausgabe ist bei einem Akzeptor immer nur ein Bit, das man interpretiert als die Mitteilung, dass die Eingabe „gut“ oder „schlecht“ war, oder mit anderen Worten „syntaktisch korrekt“ oder „syntaktisch falsch“ (für eine gerade interessierende Syntax). Formal ist bei einem endlichen Akzeptor also $Y = \{0, 1\}$ und $\forall z : h(z) \in Y$. Man macht es sich dann üblicherweise noch etwas einfacher, und schreibt statt der Funktion h einfach die Teilmenge der sogenannten *akzeptierenden Zustände* auf. Damit ist $F = \{z \mid h(z) = 1\} \subseteq Z$ gemeint. Zustände, die nicht akzeptierend sind, heißen auch *ablehnend*.

akzeptierender Zustand
ablehnender Zustand
endlicher Akzeptor

Ein *endlicher Akzeptor* $A = (Z, z_0, X, f, F)$ ist also festgelegt durch

- eine endliche Zustandsmenge Z,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X,
- eine Zustandsüberführungsfunktion $f : Z \times X \to Z$,
- eine Menge $F \subseteq Z$ akzeptierender Zustände

In graphischen Darstellungen werden die akzeptierenden Zustände üblicherweise durch doppelte Kringel statt einfacher gekennzeichnet. Abbildung 14.7 zeigt „den gleichen“ Automaten wie Abbildung 14.6, nur in der eben beschriebenen Form dargestellt. Es ist $F = \{q_f\}$, weil q_f der einzige Zustand mit Ausgabe 1 ist.

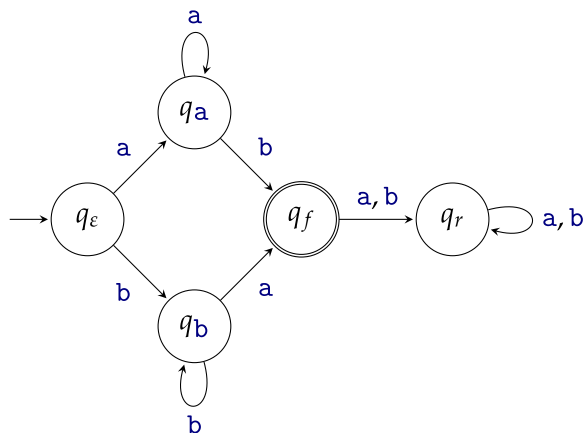


Abbildung 14.7: Ein einfacher Akzeptor (aus der Dokumentation des LaTeX-Pakets `tikz`; modifiziert)

14.4.1 Beispiele formaler Sprachen, die von endlichen Akzeptoren akzeptiert werden können

Man sagt, ein Wort $w \in X^*$ werde *akzeptiert*, falls $f^*(z_0, w) \in F$ ist, d. h. wenn man ausgehend vom Anfangszustand bei Eingabe von w in einem akzeptierenden Zustand endet. Wird ein Wort nicht akzeptiert, dann sagt man, dass es *abgelehnt* wird. Das schon mehrfach betrachtete Wort `aaaba` wird also abgelehnt, weil $f^*(z_0, \texttt{aaaba}) = q_r \notin F$ ist. Aber z. B. das Wort `aaab` wird akzeptiert. Das gilt auch für alle anderen Wörter, die mit einer Folge von mindestens einem `a` beginnen, auf das genau ein `b` folgt, also alle Wörter der Form $\texttt{a}^k\texttt{b}$ für ein $k \in \mathbb{N}_+$. Und es werden auch alle Wörter akzeptiert, die von der Form $\texttt{b}^k\texttt{a}$ sind ($k \in \mathbb{N}_+$).

akzeptiertes Wort

abgelehntes Wort

Die von einem Akzeptor A *akzeptierte formale Sprache* $L(A)$ ist die Menge aller von ihm akzeptierten Wörter:

akzeptierte formale Sprache

$$L(A) = \{w \in X^* \mid f^*(z_0, w) \in F\}$$

In unserem Beispiel ist also

$$L(A) = \{\texttt{a}\}^+\{\texttt{b}\} \cup \{\texttt{b}\}^+\{\texttt{a}\} ,$$

denn außer den oben genannten Wörtern werden keine anderen akzeptiert. Das kann man sich klar machen, in dem man überlegt,

- dass Wörter ohne ein `b` oder ohne ein `a` abgelehnt werden
- dass Wörter, die sowohl mindestens zwei `a` als auch mindestens zwei `b` enthalten, abgelehnt werden, und

- dass Wörter abgelehnt werden, die z. B. nur genau ein a enthalten, aber sowohl davor als auch dahinter mindestens ein b, bzw. umgekehrt.

Eine im Alltag öfters vorkommende Aufgabe besteht darin, aus einer Textdatei diejenigen Zeilen zu extrahieren und z. B. auszugeben, in denen ein gewisses Wort vorkommt (und alle anderen Zeilen zu ignorieren). Jede Zeile der Textdatei ist eine Zeichenkette w, die darauf hin untersucht werden muss, ob ein gewisses Textmuster m darin vorkommt. So etwas kann ein endlicher Akzeptor durchführen.

Als Beispiel betrachten wir Eingabealphabet $X = \{\mathtt{a}, \mathtt{b}\}$ und Textmuster $m = \mathtt{ababb}$. Ziel ist es, einen endlichen Akzeptor A zu konstruieren, der genau diejenigen Wörter akzeptiert, in denen irgendwo m als Teilwort vorkommt. Die erkannte Sprache soll also $L(A) = \{w_1\mathtt{ababb}w_2 \mid w_1, w_2 \in \{\mathtt{a}, \mathtt{b}\}^*\}$ sein.

Man kann diese Aufgabe natürlich ganz unterschiedlich angehen. Eine Möglichkeit, besteht darin, erst einmal einen Teil des Akzeptors hinzumalen, der „offensichtlich" oder jedenfalls (hoffentlich) plausibel ist.

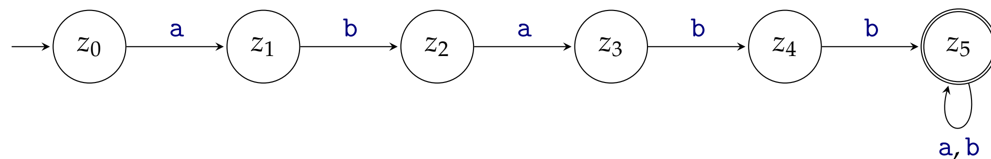


Abbildung 14.8: Teil eines Akzeptors für Wörter der Form $w_1\mathtt{ababb}w_2$

Damit sind wir aber noch nicht fertig. Denn erstens werden noch nicht alle gewünschten Wörter akzeptiert (z. B. abababb), und zweitens verlangt unsere Definition endlicher Akzeptoren, dass für *alle* Paare (z, x) der nächste Zustand $f(z, x)$ festgelegt wird.

Zum Beispiel die genauere Betrachtung des Wortes abababb gibt weitere Hinweise. Nach Eingabe von abab ist der Automat in z_4. Wenn nun wieder ein a kommt, dann darf man nicht nach Zustand z_5 gehen, aber man hat zuletzt wieder aba gesehen. Das lässt es sinnvoll erscheinen, A wieder nach z_3 übergehen zu lassen. Durch weitere Überlegungen kann man schließlich zu dem Automaten aus Abbildung 14.9

Wir unterlassen es hier, im Detail zu beweisen, dass der Akzeptor aus Abbildung 14.9 tatsächlich die gewünschte Sprache erkennt. Man mache sich aber klar, dass für $0 \leq i \leq 4$ die folgende Aussage richtig ist:

> A ist genau dann in Zustand z_i, wenn das längste Suffix der bisher gelesenen Eingabe, das Präfix von ababb ist, gerade Länge i hat.

Für z_5 ist die Aussage etwas anders; überlegen Sie sich eine passende Formulierung!

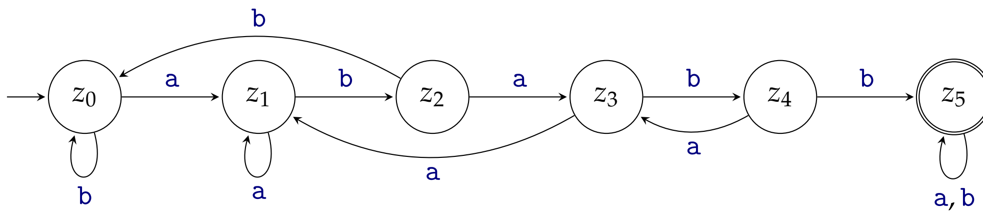


Abbildung 14.9: Der vollständige Akzeptor für alle Wörter der Form $w_1$ababbw_2

14.4.2 Ein formale Sprache, die von keinem endlichen Akzeptoren akzeptiert werden kann

Wir haben gesehen, dass es formale Sprachen gibt, die man mit endlichen Akzeptoren erkennen kann. Es gibt aber auch formale Sprachen, die man mit endlichen Akzeptoren *nicht* erkennen kann. Ein klassisches Beispiel ist die Sprache

$$L = \{\mathtt{a}^k\mathtt{b}^k \mid k \in \mathbb{N}_0\} .$$

Bevor wir diese Behauptung beweisen, sollten Sie sich (falls noch nötig) klar machen, dass man natürlich ein (Java-)Programm schreiben kann, dass von einer beliebigen Zeichenkette überprüft, ob sie die eben angegebene Form hat. Man kann also mit „allgemeinen" Algorithmen *echt mehr* Probleme lösen als mit endlichen Automaten.

14.1 Lemma. Es gibt keinen endlichen Akzeptor A mit

$$L(A) = \{\mathtt{a}^k\mathtt{b}^k \mid k \in \mathbb{N}_0\} .$$

Können Sie sich klar machen, was diese Sprache „zu schwer" macht für endliche Akzeptoren? Informell gesprochen „muss" man zählen und sich „genau" merken, wieviele a am Anfang eines Wortes vorkommen, damit ihre Zahl mit der der b am Ende „vergleichen" kann.

14.2 Beweis. Machen wir diese Idee präzise. Wir führen den Beweis indirekt und nehmen an: Es gibt einen endlichen Akzeptor A, der genau $L = \{\mathtt{a}^k\mathtt{b}^k \mid k \in \mathbb{N}_0\}$ erkennt. Diese Annahme müssen wir zu einem Widerspruch führen.

A hat eine gewisse Anzahl Zustände, sagen wir $|Z| = m$. Betrachten wir ein spezielles Eingabewort, nämlich $w = \mathtt{a}^m\mathtt{b}^m$.

1. Offensichtlich ist $w \in L$. Wenn also $L(A) = L$ ist, dann muss A bei Eingabe von w in einen akzeptierenden Zustand z_f gelangen: $f^*(z_0, w) = z_f \in F$.

2. Betrachten wir die Zustände, die A bei Eingabe der ersten Hälfte des Wortes durchläuft: $f^*(z_0, \varepsilon) = z_0$, $f^*(z_0, \mathtt{a})$, $f^*(z_0, \mathtt{aa})$, ..., $f^*(z_0, \mathtt{a}^m)$. Nennen wir diese Zustände allgemein z_i, d. h. für $0 \leq i \leq m$ ist $z_i = f^*(z_0, \mathtt{a}^i)$. Mit anderen Worten: $f^{**}(z_0, \mathtt{a}^m) = z_0 z_1 \cdots z_m$.

   Offensichtlich gilt dann: $f^*(z_m, \mathtt{b}^m) = z_f$.

   Andererseits besteht die Liste $z_0 z_1 \cdots z_m$ aus $m + 1$ Werten. Aber A hat nur m verschiedene Zustände. Also kommt mindestens ein Zustand doppelt vor.

   D. h. der Automat befindet sich in einer Schleife. Sei etwa $z_i = z_j$ für gewisse $i < j$. Genauer sei z_i das erste Auftreten irgendeines mehrfach auftretenden Zustandes und z_j das zweite Auftreten des gleichen Zustandes. Dann gibt es eine „Schleife" der Länge $\ell = j - i > 0$. Und ist der Automat erst einmal in der Schleife, dann bleibt er natürlich darin, solange er weitere $\mathtt{a}$ als Eingabe erhält. Also ist auch $z_{m-\ell} = z_m$.
3. Nun entfernen wir einige der $\mathtt{a}$ in der Eingabe, so dass die Schleife einmal weniger durchlaufen wird, d. h. wir betrachten die Eingabe $w' = \mathtt{a}^{m-\ell}\mathtt{b}^m$. Wie verhält sich der Akzeptor bei dieser Eingabe? Nachdem er das Präfix $\mathtt{a}^{m-\ell}$ gelesen hat, ist er in Zustand $z_{m-\ell}$. Dieser ist aber gleich dem Zustand z_m, d. h. A ist in dem Zustand in dem er auch nach der Eingabe $\mathtt{a}^m$ ist. Und wir wissen: $f^*(z_m, \mathtt{b}^m) = z_f \in F$. Also ist $f^*(z_0, \mathtt{a}^{m-\ell}\mathtt{b}^m) = f^*(f^*(z_0, \mathtt{a}^{m-\ell}), \mathtt{b}^m) = f^*(z_{m-\ell}, \mathtt{b}^m) = f^*(z_m, \mathtt{b}^m) = z_f$, d. h. A akzeptiert die Eingabe $w' = \mathtt{a}^{m-\ell}\mathtt{b}^m$. Aber das Wort w' gehört nicht zu L, da es verschieden viele $\mathtt{a}$ und $\mathtt{b}$ enthält! Also ist $L(A) \neq L$. Widerspruch!

Also war die Annahme falsch und es gibt gar keinen endlichen Akzeptor, der L erkennt. ■

14.5 AUSBLICK

Wir haben nur endliche Automaten betrachtet, bei denen $f : Z \times X \to Z$ eine Funktion, also linkstotal und rechtseindeutig, ist. Auch Verallgemeinerungen, bei denen f eine beliebige Relation sein darf, sind als sogenannte *nichtdeterministische endliche Automaten* ausgiebig untersucht und spielen an vielen Stellen in der Informatik eine Rolle (zum Beispiel bei Compilerbau-Werkzeugen).

Nichtdeterminismus

Deterministischen Automaten, also die, die wir in dieser Einheit betrachtet haben, werden Sie in Vorlesungen über Betriebssysteme und Kommunikation wiederbegegnen, z. B. im Zusammenhang mit der Beschreibung von sogenannten *Protokollen*.

15 REGULÄRE AUSDRÜCKE UND RECHTSLINEARE GRAMMATIKEN

Am Ende von Einheit 14 über endliche Automaten haben wir gesehen, dass manche formale Sprachen zwar von kontextfreien Grammatiken erzeugt, aber nicht von endlichen Akzeptoren erkannt werden können. Damit stellt sich für Sie vielleicht zum ersten Mal die Frage nach einer *Charakterisierung*, nämlich der der mit endlichen Akzeptoren erkennbaren Sprachen. Damit ist eine präzise Beschreibung dieser formalen Sprachen gemeint, die nicht (oder jedenfalls nicht offensichtlich) Automaten benutzt.

In den beiden Abschnitten dieser Einheit werden Sie zwei solche Charakterisierungen kennenlernen, die über reguläre Ausdrücke und die über rechtslineare Grammatiken. In Abschnitt 15.3 wird es unter anderem um eine bequeme Verallgemeinerung vollständiger Induktion gehen.

15.1 REGULÄRE AUSDRÜCKE

Der Begriff *regulärer Ausdruck* geht ursprünglich auf Stephen Kleene (1956) zurück und wird heute in unterschiedlichen Bedeutungen genutzt. In dieser Einheit führen wir kurz regulären Ausdrücken nach der „klassischen" Definition ein.

Etwas anderes (nämlich allgemeineres) sind die Varianten der *Regular Expressions*, von denen Sie möglicherweise schon im Zusammenhang mit dem ein oder anderen Programm (emacs, grep, sed, ...) oder der ein oder anderen Programmiersprache (Java, Python, ...) gelesen haben. Für Java gibt es das Paket java.util.regex. Regular expressions sind eine deutliche Verallgemeinerung regulärer Ausdrücke, auf die wir in dieser Vorlesung nicht eingehen werden. Alles was wir im folgenden über reguläre Ausdrücke sagen, ist aber auch bei regular expressions anwendbar.

Wir kommen direkt zur Definition regulärer Ausdrücke. Sie wird sie hoffentlich an das ein oder andere aus der Einheit 5 über formale Sprachen und die zugehörigen Übungsaufgaben erinnern.

Es sei A ein Alphabet, das keines der fünf Zeichen aus $Z = \{|, (,), *, \emptyset\}$ enthält. Ein *regulärer Ausdruck* über A ist eine Zeichenfolge über dem Alphabet $A \cup Z$, die gewissen Vorschriften genügt. Die Menge der regulären Ausdrücke ist wie folgt festgelegt:

regulärer Ausdruck

- $\emptyset$ ist ein regulärer Ausdruck.
- Für jedes $x \in A$ ist x ein regulärer Ausdruck.
- Wenn R_1 und R_2 reguläre Ausdrücke sind, dann sind auch $(R_1|R_2)$ und (R_1R_2) reguläre Ausdrücke.
- Wenn R ein regulärer Ausdruck ist, dann auch $(R*)$.

- Nichts anderes sind reguläre Ausdrücke.

Um sich das Schreiben zu vereinfachen, darf man Klammern auch weglassen. Im Zweifelsfall gilt „Stern- vor Punkt- und Punkt- vor Strichrechnung“, d. h. $R_1|R_2R_3*$ ist z. B. als $(R_1|(R_2(R_3*)))$ zu verstehen. Bei mehreren gleichen binären Operatoren gilt das als links geklammert; zum Beispiel ist $R_1|R_2|R_3$ als $((R_1|R_2)|R_3)$ zu verstehen.

Man kann die korrekte Syntax regulärer Ausdrücke auch mit Hilfe einer kontextfreien Grammatik beschreiben: Zu gegebenem Alphabet A sind die legalen regulären Ausdrücke gerade die Wörter, die von der Grammatik

$$\begin{aligned} & G = (\{R\}, \{|, (,), *, \emptyset\} \cup A, R, P) \\ \text{mit} \quad & P = \{R \to \emptyset, R \to (R|R), R \to (RR), R \to (R*)\} \\ & \quad \cup \{R \to x \mid x \in A\} \end{aligned}$$

erzeugt werden.

Die folgenden Zeichenketten sind alle reguläre Ausdrücke über dem Alphabet {a, b}:

- ∅
- a
- b
- (ab)
- ((ab)a)
- (((ab)a)a)
- ((ab)(aa))
- (∅|b)
- (a|b)
- ((a(a|b))|b)
- (a|(b|(a|a)))
- (∅*)
- (a*)
- ((ba)(b*))
- (((ba)b)*)
- ((a*)*)
- ((((ab)b)*)*)|(∅*))

Wendet man die Klammereinsparungsregeln an, so ergibt sich aus den Beispielen mit Klammern:

- ab
- aba
- abaa
- ab(aa)
- ∅|b
- a|b
- a(a|b)|b
- a|(b|(a|a))
- ∅*
- a*
- bab*
- (bab)*
- a**
- (abb)**|∅*

Die folgenden Zeichenketten sind dagegen auch bei Berücksichtigung der Klammereinsparungsregeln *keine* regulären Ausdrücke über {a, b}:

(|b)	vor | fehlt ein regulärer Ausdruck
|∅|	vor und hinter | fehlt je ein regulärer Ausdruck
()ab	zwischen (und) fehlt ein regulärer Ausdruck
((ab)	Klammern müssen „gepaart“ auftreten
*(ab)	vor * fehlt ein regulärer Ausdruck
c*	c ist nicht Zeichen des Alphabetes

Reguläre Ausdrücke werden benutzt, um formale Sprachen zu spezifizieren. Auch dafür bedient man sich wieder einer induktiven Vorgehensweise; man spricht auch von einer induktiven Definition:

Die *von einem regulären Ausdruck R beschriebene formale Sprache* $\langle R \rangle$ ist wie folgt definiert:

durch R beschriebene Sprache $\langle R \rangle$

- $\langle \emptyset \rangle = \{\}$ (d. h. die leere Menge).
- Für $x \in A$ ist $\langle x \rangle = \{x\}$.
- Sind R_1 und R_2 reguläre Ausdrücke, so ist $\langle R_1 | R_2 \rangle = \langle R_1 \rangle \cup \langle R_2 \rangle$.
- Sind R_1 und R_2 reguläre Ausdrücke, so ist $\langle R_1 R_2 \rangle = \langle R_1 \rangle \cdot \langle R_2 \rangle$.
- Ist R ein regulärer Ausdruck, so ist $\langle R* \rangle = \langle R \rangle^*$.

Betrachten wir drei einfache Beispiele:

- $R = \mathtt{a|b}$: Dann ist $\langle R \rangle = \langle \mathtt{a|b} \rangle = \langle \mathtt{a} \rangle \cup \langle \mathtt{b} \rangle = \{\mathtt{a}\} \cup \{\mathtt{b}\} = \{\mathtt{a}, \mathtt{b}\}$.
- $R = \mathtt{(a|b)*}$: Dann ist $\langle R \rangle = \langle \mathtt{(a|b)*} \rangle = \langle \mathtt{a|b} \rangle^* = \{\mathtt{a}, \mathtt{b}\}^*$.
- $R = \mathtt{(a*b*)*}$: Dann ist $\langle R \rangle = \langle \mathtt{(a*b*)*} \rangle = \langle \mathtt{a*b*} \rangle^* = (\langle \mathtt{a*} \rangle \langle \mathtt{b*} \rangle)^* = (\langle \mathtt{a} \rangle^* \langle \mathtt{b} \rangle^*)^* = (\{\mathtt{a}\}^* \{\mathtt{b}\}^*)^*$.

Mehr oder weniger kurzes Überlegen zeigt übrigens, dass für die Sprachen des zweiten und dritten Beispiels gilt: $(\{\mathtt{a}\}^* \{\mathtt{b}\}^*)^* = \{\mathtt{a}, \mathtt{b}\}^*$. Man kann also die gleiche formale Sprache durch verschiedene reguläre Ausdrücke beschreiben — wenn sie denn überhaupt so beschreibbar ist.

Damit klingen (mindestens) die beiden folgenden Fragen an:

1. Kann man allgemein algorithmisch von zwei beliebigen regulären Ausdrücken R_1, R_2 feststellen, ob sie die gleiche formale Sprache beschreiben, d. h. ob $\langle R_1 \rangle = \langle R_2 \rangle$ ist?
2. Welche formalen Sprachen sind denn durch reguläre Ausdrücke beschreibbar?

Die Antwort auf die erste Frage ist *ja*. Allerdings hat das Problem, die Äquivalenz zweier regulärer Ausdrücke zu überprüfen, die Eigenschaft PSPACE-vollständig zu sein wie man in der Komplexitätstheorie sagt. Was das ist, werden wir in der Einheit über Turingmaschinen kurz anreißen. Es bedeutet unter anderem, dass alle *bisher bekannten* Algorithmen im allgemeinen *sehr sehr langsam* sind: die Rechenzeit wächst „stark exponentiell“ mit der Länge der regulären Ausdrücke (z. B. wie 2^{n^2} o.ä.). Es sei noch einmal betont, dass dies für alle bisher bekannten Algorithmen gilt. Man weiß nicht, ob es vielleicht doch signifikant schnellere Algorithmen für das Problem gibt, aber man sie „nur noch nicht gefunden“ hat.

Nun zur Antwort auf die zweite Frage. (Was rechtslineare Grammatiken sind, werden wir in nachfolgenden Abschnitt 15.2 gleich noch beschreiben. Es handelt sich um einen Spezialfall kontextfreier Grammatiken.)

15.1 Satz. *Für jede formale Sprache L sind die folgenden drei Aussagen äquivalent:*

1. *L kann von einem endlichen Akzeptor erkannt werden.*

2. *L kann durch einen regulären Ausdruck beschrieben werden.*
3. *L kann von einer rechtslinearen Grammatik erzeugt werden.*

reguläre Sprache

Eine formale Sprache, die die Eigenschaften aus Satz 15.1 hat, heißt *reguläre Sprache*. Da jede rechtslineare Grammatik eine kontextfreie Grammatik ist, ist jede reguläre Sprache eine kontextfreie Sprache.

Zwar werden wir Satz 15.1 nicht im Detail beweisen, aber wir wollen zumindest einige Dinge andeuten, insbesondere auch eine grundlegende Vorgehensweise.

Satz 15.1 hat folgende prinzipielle Struktur:

- Es werden drei Aussagen $\mathcal{A}$, $\mathcal{B}$ und $\mathcal{C}$ formuliert.
- Es wird behauptet:
  - $\mathcal{A} \iff \mathcal{B}$
  - $\mathcal{B} \iff \mathcal{C}$
  - $\mathcal{C} \iff \mathcal{A}$

Man kann nun natürlich einfach alle sechs Implikationen einzeln beweisen. Aber das muss man gar nicht! Dann wenn man zum Beispiel schon gezeigt hat, dass $\mathcal{A} \Longrightarrow \mathcal{B}$ gilt und dass $\mathcal{B} \Longrightarrow \mathcal{C}$, dann folgt $\mathcal{A} \Longrightarrow \mathcal{C}$ automatisch. Das sieht man anhand der folgenden Tabelle:

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{A} \Longrightarrow \mathcal{B}$	$\mathcal{B} \Longrightarrow \mathcal{C}$	$\mathcal{A} \Longrightarrow \mathcal{C}$
1				W	W	W
2			W	W	W	W
3		W		W		W
4		W	W	W	W	W
5	W				W	
6	W		W		W	W
7	W	W		W		
8	W	W	W	W	W	W

In allen Zeilen 1, 2, 4 und 8, in denen sowohl für $\mathcal{A} \Longrightarrow \mathcal{B}$ als auch für $\mathcal{B} \Longrightarrow \mathcal{C}$ ein W (für *wahr*) eingetragen ist, ist das auch für $\mathcal{A} \Longrightarrow \mathcal{C}$ der Fall. Statt *falsch* haben wir der besseren Übersicht wegen die entsprechenden Felder freigelassen.

Wenn man $\mathcal{A} \Longrightarrow \mathcal{B}$ und $\mathcal{B} \Longrightarrow \mathcal{C}$ schon bewiesen hat, dann muss man also $\mathcal{A} \Longrightarrow \mathcal{C}$ gar nicht mehr beweisen. Und beweist man nun zusätzlich noch $\mathcal{C} \Longrightarrow \mathcal{A}$, dann

- folgt mit $\mathcal{A} \Longrightarrow \mathcal{B}$ sofort $\mathcal{C} \Longrightarrow \mathcal{B}$ und
- mit $\mathcal{B} \Longrightarrow \mathcal{C}$ folgt sofort $\mathcal{B} \Longrightarrow \mathcal{A}$,

und man ist fertig.

Statt sechs Implikationen zu beweisen zu müssen, reichen also drei. Für einen Beweis von Satz 15.1 genügen daher folgende Konstruktionen:

- zu gegebenem endlichen Akzeptor A ein regulärer Ausdruck R mit $\langle R \rangle = L(A)$:

  Diese Konstruktion ist „mittel schwer". Man kann z. B. einen Algorithmus benutzen, dessen Struktur und Idee denen des Algorithmus von Warshall ähneln.
- zu gegebenem regulären Ausdruck R eine rechtslineare Grammatik G mit $L(G) = \langle R \rangle$:

  Diese Konstruktion ist „relativ leicht". Wir werden im nächsten Abschnitt noch etwas genauer darauf eingehen.
- zu gegebener rechtslinearer Grammatik G ein endlicher Akzeptor A mit $L(A) = L(G)$:

  Diese Konstruktion ist die schwierigste.

Wie wertvoll Charakterisierungen wie Satz 15.1 sein können, sieht man an folgendem Beispiel: Es sei L eine reguläre Sprache, z. B. die Sprache aller Wörter, in denen irgendwo das Teilwort `abbab` vorkommt. Aufgabe: Man zeige, dass auch das Komplement $L' = \{\mathtt{a}, \mathtt{b}\}^* \setminus L$, also die Menge aller Wörter, in denen nirgends das Teilwort `abbab` vorkommt, regulär ist.

Wüssten wir nur, dass reguläre Sprachen die durch reguläre Ausdrücke beschreibbaren sind, und hätten wir nur einen solchen für L, dann stünden wir vor einem Problem. Damit Sie das auch merken, sollten Sie einmal versuchen, einen regulären Ausdruck für L' hinzuschreiben.

Aber wir wissen, dass wir uns auch endlicher Akzeptoren bedienen dürfen. Und dann ist alles *ganz* einfach: Denn wenn A ein endlicher Akzeptor ist, der L erkennt, dann bekommt man daraus den für L', indem man einfach akzeptierende und ablehnende Zustände vertauscht.

15.2 RECHTSLINEARE GRAMMATIKEN

Mit beliebigen kontextfreien Grammatiken kann man jedenfalls zum Teil andere formale Sprachen erzeugen, als man mit endlichen Akzepteren erkennen kann. Denn die Grammatik $G = (\{X\}, \{\mathtt{a}, \mathtt{b}\}, X, \{X \to \mathtt{a}X\mathtt{b} \mid \varepsilon\})$ erzeugt $\{\mathtt{a}^k\mathtt{b}^k \mid k \in \mathbb{N}_0\}$ und diese Sprache ist nicht regulär.

Aber die folgende einfache Einschränkung tut „das Gewünschte". Eine *rechtslineare Grammatik* ist eine kontextfreie Grammatik $G = (N, T, S, P)$, die der folgenden Einschränkung genügt: Jede Produktion ist entweder von der Form $X \to w$ oder von der Form $X \to wY$ mit $w \in T^*$ und $X, Y \in N$. Auf der rechten Seite einer

rechtslineare Grammatik

Produktion darf also höchstens ein Nichterminalsymbol vorkommen, und wenn dann nur als letztes Symbol.

Die oben erwähnte Grammatik $G = (\{X\}, \{\mathtt{a}, \mathtt{b}\}, X, \{X \to \mathtt{a}X\mathtt{b} \mid \varepsilon\})$ ist also *nicht* rechtslinear, denn in der Produktion $X \to \mathtt{a}X\mathtt{b}$ steht das Nichtterminalsymbol X nicht am rechten Ende.

Und da wir uns überlegt hatten, dass die erzeugte formale Sprache nicht regulär ist, kann es auch gar keine rechtslineare Grammatik geben, die $\{\mathtt{a}^k\mathtt{b}^k \mid k \in \mathbb{N}_0\}$ erzeugt.

Typ-3-Grammatiken

Typ-2-Grammatiken

Es sei auch noch die folgende Sprechweise eingeführt: Rechtslineare Grammatiken heißen auch *Typ-3-Grammatiken* und die schon eingeführten kontextfreien Grammatiken nennt man auch *Typ-2-Grammatiken*. Hier ahnt man schon, dass es noch weiter geht. Es gibt auch noch *Typ-1-Grammatiken* und *Typ-0-Grammatiken*.

Wenn für ein $i \in \{0, 1, 2, 3\}$ eine formale Sprache L von einer Typ-i-Grammatik erzeugt wird, dann sagt man auch, L sei eine *Typ-i-Sprache* oder kurz *vom Typ i*.

Zumindest einer der Vorteile rechtslinearer Grammatiken gegenüber deterministischen endlichen Akzeptoren, wie wir sie im vorangegangenen Kapitel eingeführt haben, ist, dass sie manchmal deutlich kürzer und übersichtlicher hinzuschreiben sind. Ein genaueres Verständnis dafür, warum das so ist, werden Sie bekommen, wenn Sie im dritten Semester auch etwas über sogenannte nichtdeterministische endliche Akzeptoren gelernt haben.

15.3 KANTOROWITSCH-BÄUME UND STRUKTURELLE INDUKTION

Kantorowitsch-Baum

Reguläre Ausdrücke kann man auch als sogenannte *Kantorowitsch-Bäume* darstellen. Für den regulären Ausdruck `((b|∅)a)(b*)` ergibt sich zum Beispiel der Graph aus Abbildung 15.1

Hier handelt es sich *nicht* um den Ableitungsbaum gemäß der Grammatik aus Abschnitt 15.1. Aber die Struktur des regulären Ausdruckes wird offensichtlich ebenso gut wiedergegeben und die Darstellung ist sogar noch kompakter.

Solche Bäume wollen wir im folgenden der Einfachheit halber Regex-Bäume nennen. Es sei A irgendein Alphabet. Dann ist ein Baum ein Regex-Baum, wenn gilt:

- Entweder ist es Baum dessen Wurzel zugleich Blatt ist und das ist mit einem $x \in A$ oder `∅` beschriftet,
- oder es ist ein Baum, dessen Wurzel mit `*` beschriftet ist und die genau einen Nachfolgeknoten hat, der Wurzel eines Regex-Baumes ist

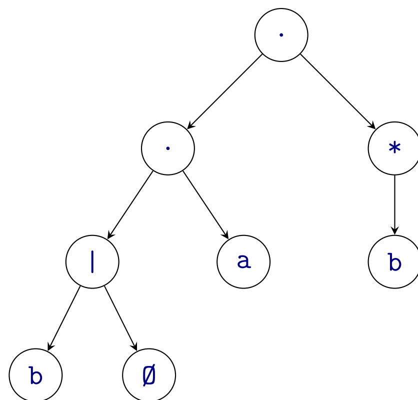


Abbildung 15.1: Der Kantorowitsch-Baum für den regulären Ausdruck `((b|∅)a)(b*)`

- oder es ist ein Baum, dessen Wurzel mit · oder mit | beschriftet ist und die genau zwei Nachfolgeknoten hat, die Wurzeln zweier Regex-Bäume sind.

Man beachte, dass linker und rechter Unter-Regex-Baum unterhalb der Wurzel unterschiedliche Höhe haben können.

Wichtig ist für uns im folgenden, dass erstens größere Bäume „aus kleineren zusammengesetzt" werden, und dass zweitens diese Zusammensetzung immer eindeutig ist. Außerdem kommt man bijektiv von regulären Ausdrücken zu Regex-Bäumen und umgekehrt.

Für das weitere definieren wir noch ein oft auftretendes Maß für Bäume, die sogenannte *Höhe* $h(T)$ eines Baumes T. Das geht so:

Höhe eines Baumes

$$h(T) = \begin{cases} 0 & \text{falls die Wurzel Blatt ist} \\ 1 + \max_i h(U_i) & \text{falls die } U_i \text{ alle Unterbäume von } T \text{ sind} \end{cases}$$

Wenn man beweisen möchte, dass eine Aussage für alle regulären Ausdrücke gilt, dann kann man das dadurch tun, dass man die entsprechende Aussage für alle Regex-Bäume beweist. Und den Beweis für alle Regex-Bäume kann man durch vollständige Induktion über ihre Höhe führen. Bei naiver Herangehensweise tritt aber ein Problem auf: Beim Schritt zu Bäumen der Höhe $n + 1$ darf man nur auf Bäume der Höhe n zurückgreifen. Auftretende Unterbäume können aber alle Höhen $i \leq n$ haben, und man möchte gerne für alle die Induktionsvoraussetzung benutzen. Das darf man auch! Wir wollen uns zunächst klar machen, warum das so ist.

Dazu sehen wir uns eine einfache Verallgemeinerung vollständiger Induktion an. Es sei $\mathcal{B}(n)$ eine Aussage, die von einer Zahl $n \in \mathbb{N}_0$ abhängt. Wir wollen

beweisen: $\forall n \in \mathbb{N}_0 : \mathcal{B}(n)$. Dazu definieren wir eine Aussage $\mathcal{A}(n)$ wie folgt: $\forall i \leq n : \mathcal{B}(i)$. Wenn wir beweisen können, dass $\forall n \in \mathbb{N}_0 : \mathcal{A}(n)$ gilt, dann sind wir fertig, denn aus $\mathcal{A}(n)$ folgt stets $\mathcal{B}(n)$.

Wie sieht ein Induktionsbeweis für $\forall n \in \mathbb{N}_0 : \mathcal{A}(n)$ aus?

Induktionsanfang: Es ist zu zeigen, dass $\mathcal{A}(0)$ gilt, also die Aussage $\forall i \leq 0 : \mathcal{B}(i)$. Das ist offensichtlich äquivalent zu $\mathcal{B}(0)$. Das ist zu zeigen.

Induktionsvoraussetzung: für beliebiges aber festes $n \in \mathbb{N}_0$ gilt: $\mathcal{A}(n)$, also die Aussage $\forall i \leq n : \mathcal{B}(i)$.

Induktionsschluss: Es ist zu zeigen, dass auch $\mathcal{A}(n+1)$ gilt, also die Aussage $\forall i \leq n+1 : \mathcal{B}(i)$. Diese Aussage ist äquivalent zu $\big(\forall i \leq n : \mathcal{B}(i)\big) \wedge \mathcal{B}(n+1)$. Wie zeigt man das?

- Der erste Teil $\forall i \leq n : \mathcal{B}(i)$ ist trivial, denn das ist ja gerade die Induktionsvoraussetzung.
- Daher bleibt nur zu zeigen: $\mathcal{B}(n+1)$. Zum Beweis dafür kann man aber ebenfalls auf die Induktionsvoraussetzung zurückgreifen, also auf *alle* Aussagen $\mathcal{B}(0), \mathcal{B}(1), \dots, \mathcal{B}(n)$ und nicht nur die letzte von ihnen.

Diese Vorgehensweise wollen wir nun anwenden, um zu einen Beweis dafür zu skizzieren, dass es für jeden regulären Ausdruck R eine rechtslineare Grammatik G gibt mit $\langle R \rangle = L(G)$. Bei regulären Ausdrücken denkt man nun vorteilhafterweise an Regex-Bäume und wir machen nun zunächst eine „normale" vollständige Induktion über die Höhe der Regex-Bäume. Im Schema von eben ist also $\mathcal{B}(n)$ die Aussage:

> Für jeden Regex-Baum R der Höhe n gibt es eine rechtslineare Grammatik G gibt mit $\langle R \rangle = L(G)$.

Wir wollen zeigen, dass $\forall n \in \mathbb{N}_0 : \mathcal{B}(n)$ gilt.

Induktionsanfang: Es ist $\mathcal{B}(0)$ zu zeigen. Man muss also rechtslineare Grammatiken angeben, die die formalen Sprachen $\{x\} = \langle x \rangle$ für $x \in A$ und die leere Menge $\{\} = \langle \emptyset \rangle$ erzeugen. Das ist eine leichte Übung.

Induktionsvoraussetzung: für beliebiges aber festes $n \in \mathbb{N}_0$ gelte die Aussage $\forall i \leq n : \mathcal{B}(i)$, d. h. für jeden Regex-Baum R' mit einer Höhe $i \leq n$ gibt es eine rechtslineare Grammatik G gibt mit $\langle R' \rangle = L(G)$.

Induktionsschluss: Es bleibt zu zeigen, dass auch $\mathcal{B}(n+1)$ gilt, dass also für jeden Regex-Baum R der Höhe $n+1$ eine rechtslineare Grammatik G mit $\langle R \rangle = L(G)$ existiert.

Sei daher R ein beliebiger Regex-Baum der Höhe $n+1$. Dann gibt es drei mögliche Fälle:

1. Die Wurzel von R ist ein $*$-Knoten und hat genau einen Unterbaum R' der Höhe n.
2. Die Wurzel von R ist ein $|$-Knoten und hat genau zwei Unterbäume R_1 und R_2. Da R Höhe $n+1$ hat, hat einer der beiden Unterbäume Höhe n, der andere hat eine Höhe $i \leq n$.
3. Die Wurzel von R ist ein „Konkatenations-Knoten" und hat genau zwei Unterbäume R_1 und R_2. Da R Höhe $n+1$ hat, hat einer der beiden Unterbäume Höhe n, der andere hat eine Höhe $i \leq n$.

Der entscheidende Punkt ist nun: In den Fällen 2 und 3 darf man nach Induktionsvoraussetzung annehmen, dass für *beide* Unterbäume rechtslineare Grammatiken der gewünschten Art existieren.

In allen drei Fällen kann man dann aus den Grammatiken für den Unterbaum bzw. die Unterbäume die Grammatik für den Regex-Baum, also regulären Ausdruck, R konstruieren. Das wollen wir hier nicht allen Details durchführen und beschränken uns auf den einfachsten Fall, nämlich Fall 2: Seien also $G_1 = (N_1, A, S_1, P_1)$ und $G_2 = (N_2, A, S_2, P_2)$ Typ-3-Grammatiken, die $L(G_1) = \langle R_1 \rangle$ bzw. $L(G_2) = \langle R_2 \rangle$ erzeugen. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $N_1 \cap N_2 = \emptyset$ ist. Wir wählen ein „neues" Nichtterminalsymbol $S \notin N_1 \cup N_2$ Damit können wir eine Typ-3-Grammatik G mit $L(G) = \langle R_1 | R_2 \rangle$ ganz leicht hinschreiben:

$$G = (\{S\} \cup N_1 \cup N_2, A, S, \{S \to S_1 \mid S_2\} \cup P_1 \cup P_2)$$

Als erstes muss man sich klar machen, dass auch die Grammatik rechtslinear ist. Tun Sie das; es ist nicht schwer.

Etwas Arbeit würde es machen, zu beweisen, dass $L(G) = L(G_1) \cup L(G_2)$ ist. Das wollen wir uns an dieser Stellen sparen. Sie können das aber ruhig selbst einmal versuchen.

Und für die anderen beiden Fälle können Sie das auch einmal versuchen, geeignete (rechtslineare!) Grammatiken zu konstruieren, oder einfach glauben, dass es geht.

strukturelle Induktion

Um zu einer manchmal sogenannten *strukturellen Induktion* zu kommen, muss man nun nur noch das Korsett der vollständigen Induktion über die Höhe der Bäume „vergessen". Was bleibt ist folgende prinzipielle Situation:

1. Man beschäfigt sich mit irgendwelchen „Gebilden" (eben waren das reguläre Ausdrücke bzw. Bäume). Dabei gibt es kleinste „atomare" oder „elementare" Gebilde (eben waren das die regulären Ausdrücke x für $x \in A$ und $\emptyset$) und eine oder mehrere Konstruktionsvorschriften, nach denen man aus kleineren Gebilden größere zusammensetzen kann (eben waren das $*$, $|$ und Konkatenation).

2. Man möchte beweisen, dass alle Gebilde eine gewisse Eigenschaft haben. Dazu macht man dann eine strukturelle Induktion:
   - Im Induktionsanfang zeigt man zunächst für *alle* „atomaren“ Gebilde, dass sie eine gewünschte Eigenschaft haben und
   - im Induktionsschritt zeigt man, wie sich bei einem „großen“ Gebilde die Eigenschaft daraus folgt, dass schon alle Untergebilde die Eigenschaft haben, gleich nach welcher Konstruktionsvorschrift das große Gebilde gebaut ist.

15.4 AUSBLICK

Beweise für die Behauptungen aus Satz 15.1 werden Sie vielleicht in der Vorlesung „Theoretische Grundlagen der Informatik“ oder in „Formale Systeme“ sehen. Insbesondere ist es dafür nützlich, sich mit nichtdeterministischen endlichen Automaten zu beschäftigen, auf die wir am Ende von Einheit 14 schon hingewiesen haben.

Sie werden sehen, dass reguläre Ausdrücke bei der Verarbeitung von Textdateien des öfteren nützlich sind. Dabei kommen zu dem, was wir in Abschnitt 15.1 definiert haben, zum einen noch bequeme Abkürzungen hinzu, denn wer will schon z. B.

$$\mathtt{a}|\mathtt{b}|\mathtt{c}|\mathtt{d}|\mathtt{e}|\mathtt{f}|\mathtt{g}|\mathtt{h}|\mathtt{i}|\mathtt{j}|\mathtt{k}|\mathtt{l}|\mathtt{m}|\mathtt{n}|\mathtt{o}|\mathtt{p}|\mathtt{q}|\mathtt{r}|\mathtt{s}|\mathtt{t}|\mathtt{u}|\mathtt{v}|\mathtt{w}|\mathtt{x}|\mathtt{y}|\mathtt{z}$$

schreiben müssen als regulären Ausdruck für einen einzelnen Kleinbuchstaben. Zum anderen gibt es aber auch noch Erweiterungen, die dazu führen, dass die resultierenden *regular expressions* mächtiger sind als reguläre Ausdrücke. Wer sich dafür (jetzt schon) genauer interessiert, dem sei das Buch von Friedl (2006) empfohlen.

LITERATUR

Friedl, Jeffrey (2006). *Mastering Regular Expressions*. 3rd edition. O’Reilly Media, Inc.

Kleene, Stephen C. (1956). “Representation of Events in Nerve Nets and Finite Automata”. In: *Automata Studies*. Hrsg. von Claude E. Shannon und John McCarthy. Princeton University Press. Kap. 1, S. 3–40.
Eine Vorversion ist online verfügbar; siehe http://www.rand.org/pubs/research_memoranda/2008/RM704.pdf (8.12.08).

16 TURINGMASCHINEN

Turingmaschinen sind eine und waren im wesentlichen die erste mathematische Präzisierung des Begriffes des *Algorithmus* so wie er klassisch verstanden wird: Zu jeder endlichen Eingabe wird in endlich vielen Schritten eine endliche Ausgabe berechnet.

Algorithmus

Einen technischen Hinweis wollen wir an dieser Stelle auch noch geben: In dieser Einheit werden an verschiedenen Stellen partielle Funktionen vorkommen. Das sind rechtseindeutige Relationen, die nicht notwendig linkstotal sind (siehe Abschnitt 3.2). Um deutlich zu machen, dass eine partielle Funktion vorliegt, schreiben wir im folgenden $f : M \dashrightarrow M'$. Das bedeutet dann also, dass für ein $x \in M$ entweder eindeutig ein Funktionswert $f(x) \in M'$ definiert ist, oder dass *kein* Funktionswert $f(x)$ definiert ist. Man sagt auch, f sei an der Stelle x undefiniert.

16.1 ALAN MATHISON TURING

Alan Mathison Turing wurde am 23.6.1912 geboren.

Mitte der Dreißiger Jahre beschäftigte sich Turing mit Gödels Unvollständigkeitssätzen und Hilberts Frage, ob man für jede mathematische Aussage algorithmisch entscheiden könne, ob sie wahr sei oder nicht. Das führte zu der bahnbrechenden Arbeit *"On computable numbers, with an application to the Entscheidungsproblem"* von 1936.

Von 1939 bis 1942 arbeitete Turing in Bletchly Park an der Dechiffrierung der verschlüsselten Texte der Deutschen. Für den Rest des zweiten Weltkriegs beschäftigte er sich in den USA mit Ver- und Entschlüsselungsfragen. Mehr zu diesem Thema (und verwandten) können Sie in Vorlesungen zum Thema *Kryptographie* erfahren.

Nach dem Krieg widmete sich Turing unter anderem dem Problem der *Morphogenese* in der Biologie. Ein kleines bisschen dazu findet sich in der Vorlesung „Algorithmen in Zellularautomaten".

Alan Turing starb am 7.6.1954 an einer Zyankalivergiftung.

Eine Art „Homepage" von Alan Turing findet sich unter `http://www.turing.org.uk/turing/index.html` (19.1.2011).

16.2 TURINGMASCHINEN

Als *Turingmaschine* bezeichnet man heute etwas, was Turing (1936) in seiner Arbeit eingeführt hat. Die Bezeichnung selbst geht wohl auf eine Besprechung von

Turings Arbeit durch Alonzo Church zurück (laut einer WWW-Seite von J. Miller; `http://jeff560.tripod.com/t.html`, 14.1.2010).

Eine Turingmaschine kann man als eine Verallgemeinerung endlicher Automaten auffassen, bei der die Maschine nicht mehr darauf beschränkt ist, nur feste konstante Zahl von Bits zu speichern. Im Laufe der Jahrzehnte wurden viele Varianten definiert und untersucht. Wir führen im folgenden nur die einfachste Variante ein.

Zur Veranschaulichung betrachte man Abbildung 16.1. Im oberen Teil sieht man die *Steuereinheit*, die im wesentlichen ein endlicher *Mealy-Automat* ist. Zusätzlich gibt es ein *Speicherband*, das in einzelne *Felder* aufgeteilt ist, die mit jeweils einem Symbol beschriftet sind. Die Steuereinheit besitzt einen *Schreib-Lese-Kopf*, mit dem sie zu jedem Zeitpunkt von einem Feld ein Symbol als Eingabe lesen kann. Als Ausgabe produziert sie die Turingmaschine ein Symbol, das auf das gerade besuchte Feld geschrieben wird und sie kann den Kopf um ein Feld auf dem Band nach links oder rechts bewegen. Ausgabesymbol und Kopfbewegung ergeben sich ebenso eindeutig aus aktuellem Zustand der Steuereinheit und gelesenem Symbol wie der neue Zustand der Steuereinheit.

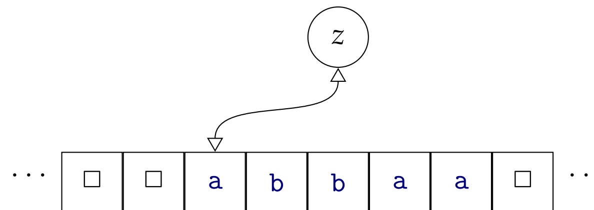


Abbildung 16.1: schematische Darstellung einer (einfachen) Turingmaschine

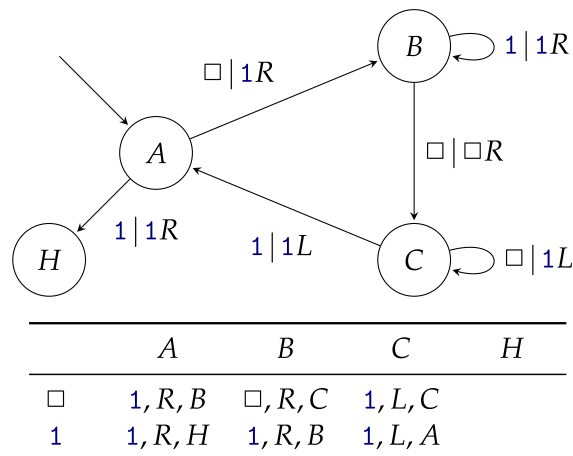
Turingmaschine

Formal kann man sich also eine *Turingmaschine* $T = (Z, z_0, X, f, g, m)$ festgelegt vorstellen durch

- eine Zustandsmenge Z
- einen Anfangszustand $z_0 \in Z$
- ein Bandalphabet X
- eine partielle Zustandsüberführungsfunktion $f : Z \times X \dashrightarrow Z$
- eine partielle Ausgabefunktion $g : Z \times X \dashrightarrow X$ und
- eine partielle Bewegungsfunktion $m : Z \times X \dashrightarrow \{-1, 0, 1\}$

Wir verlangen, dass die drei Funktionen f, g und m für die gleichen Paare $(z, x) \in Z \times X$ definiert bzw. nicht definiert sind. Warum wir im Gegensatz zu z. B. endlichen Akzeptoren erlauben, dass die Abbildungen nur partiell sind, werden wir später noch erläutern.

Es gibt verschiedene Möglichkeiten, die Festlegungen für eine konkrete Turingmaschine darzustellen. Manchmal schreibt man die drei Abbildungen f, g und m in Tabellenform auf, manchmal macht man es graphisch, ähnlich wie bei Mealy-Automaten. In Abbildung 16.2 ist die gleiche Turingmaschine auf beide Arten definiert. Die Bewegungsrichtung notiert man oft auch mit L (für links) statt -1 und mit R (für rechts) statt 1.



	A	B	C	H
$\square$	$1, R, B$	$\square, R, C$	$1, L, C$	
1	$1, R, H$	$1, R, B$	$1, L, A$	

Abbildung 16.2: Zwei Spezifikationsmöglichkeiten der gleichen Turingmaschine; sie heißt BB3.

Konfiguration

Eine Turingmaschine befindet sich zu jedem Zeitpunkt in einem „Gesamtzustand“, den wir eine *Konfiguration* nennen wollen. Sie ist vollständig beschrieben durch

- den aktuellen Zustand $z \in Z$ der Steuereinheit,
- die aktuelle Beschriftung des gesamten Bandes, die man als Abbildung $b : \mathbb{Z} \to X$ formalisieren kann, und
- die aktuelle Position $p \in \mathbb{Z}$ des Kopfes.

Blanksymbol

Eine Bandbeschriftung ist also ein potenziell unendliches „Gebilde“. Wie aber schon in Abschnitt 6.2 erwähnt und zu Beginn dieser Einheit noch einmal betont, interessieren in weiten Teilen der Informatik endliche Berechnungen, die aus endlichen Eingaben endliche Ausgaben berechnen. Um das adäquat zu formalisieren, ist es üblich, davon auszugehen, dass das Bandalphabet ein sogenanntes *Blanksymbol* enthält, für das wir $\square \in X$ schreiben. Bandfelder, die „mit $\square$ beschriftet“ sind, wollen wir als „leer“ ansehen; und so stellen wir sie dann gelegentlich auch dar, oder lassen sie ganz weg. Jedenfalls in dieser Vorlesung (und in vielen anderen auch) sind alle tatsächlich vorkommenden Bandbeschriftungen von der Art, dass nur endlich viele Felder nicht mit $\square$ beschriftet sind.

16.2.1 Berechnungen

Schritt einer Turingmaschine

Wenn $c = (z, b, p)$ die aktuelle Konfiguration einer Turingmaschine T ist, dann kann es sein, dass sie einen *Schritt* durchführen kann. Das geht genau dann, wenn für das Paar $(z, b(p))$ aus aktuellem Zustand und aktuell gelesenem Bandsymbol die Funktionen f, g und m definiert sind. Gegebenenfalls führt das dann dazu, dass T in die Konfiguration $c' = (z', b', p')$ übergeht, die wie folgt definiert ist:

- $z' = f(z, b(p))$
- $\forall i \in \mathbb{Z} : b'(i) = \begin{cases} b(i) & \text{falls } i \neq p \\ g(z, b(p)) & \text{falls } i = p \end{cases}$
- $p' = p + m(z, b(p))$

Wir schreiben $c' = \Delta_1(c)$. Bezeichnet $\mathcal{C}_T$ die Menge aller Konfigurationen einer Turingmaschine T, dann ist das also die partielle Abbildung $\Delta_1 : \mathcal{C}_T \dashrightarrow \mathcal{C}_T$, die als Funktionswert $\Delta_1(c)$ gegebenenfalls die ausgehend von c nach einem Schritt erreichte Konfiguration bezeichnet.

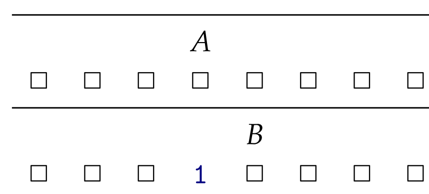
Endkonfiguration
Halten

Falls für eine Konfiguration c die Nachfolgekonfiguration $\Delta_1(c)$ nicht definiert ist, heißt c auch eine *Endkonfiguration* und man sagt, die Turingmaschine habe *gehalten*.

Die Turingmaschine aus Abbildung 16.2 wollen wir BB3 nennen. Wenn BB3 im Anfangszustand A auf einem vollständig leeren Band gestartet wird, dann macht sie wegen

- $f(A, \square) = B$,
- $g(A, \square) = 1$ und
- $m(A, \square) = R$

folgenden Schritt:



Dabei haben wir den Zustand der Turingmaschine jeweils über dem gerade besuchten Bandfeld notiert. In der entstandenen Konfiguration kann BB3 einen weiteren Schritt machen, und noch einen und noch einen .... Es ergibt sich folgender Ablauf.

			A				
□	□	□	□	□	□	□	□
				B			
□	□	□	1	□	□	□	□
					C		
□	□	□	1	□	□	□	□
				C			
□	□	□	1	□	1	□	□
			C				
□	□	□	1	1	1	□	□
		A					
□	□	□	1	1	1	□	□
			B				
□	□	1	1	1	1	□	□
				B			
□	□	1	1	1	1	□	□
					B		
□	□	1	1	1	1	□	□
						B	
□	□	1	1	1	1	□	□
							C
□	□	1	1	1	1	□	□
						C	
□	□	1	1	1	1	□	1
					C		
□	□	1	1	1	1	1	1
				A			
□	□	1	1	1	1	1	1
					H		
□	□	1	1	1	1	1	1

In Zustand H ist kein Schritt mehr möglich; es ist eine Endkonfiguration erreicht und BB3 hält.

endliche Berechnung

Eine *endliche Berechnung* ist eine endliche Folge von Konfigurationen $(c_0, c_1, c_2, \dots, c_t)$ mit der Eigenschaft, dass für alle $0 < i \leq t$ gilt: $c_i = \Delta_1(c_{i-1})$. Eine Berechnung ist *haltend*, wenn es eine endliche Berechung ist und ihre letzte Konfiguration eine Endkonfiguration ist.

haltende Berechnung

unendliche Berechnung

Eine *unendliche Berechnung* ist eine unendliche Folge von Konfigurationen $(c_0, c_1, c_2, \dots)$ mit der Eigenschaft, dass für alle $0 < i$ gilt: $c_i = \Delta_1(c_{i-1})$. Eine unendliche Berechnung heißt auch *nicht haltend*.

nicht haltende Berechung

Eine nicht haltende Berechungen würden wir zum Beispiel bekommen, wenn wir BB3 dahingehend abändern, dass $f(A, \square) = A$ und $g(A, \square) = \square$ ist. Wenn man dann BB3 auf dem vollständig leeren Band startet, dann bewegt sie ihren Kopf immer weiter nach rechts, lässt das Band leer und bleibt immer im Zustand A.

Analog zu Δ_1 liefere generell für $t \in \mathbb{N}_0$ die Abbildung Δ_t als Funktionswert $\Delta_t(c)$ gegebenenfalls die ausgehend von c nach t Schritten erreichte Konfiguration. Also

$$\Delta_0 = \mathrm{I}$$
$$\forall t \in \mathbb{N}_+ : \Delta_{t+1} = \Delta_1 \circ \Delta_t$$

Zu jeder Konfiguration c gibt es genau eine Berechnung, die mit c startet, und wenn diese Berechnung hält, dann ist der Zeitpunkt zu dem das geschieht natürlich auch eindeutig. Wir schreiben Δ_* für die partielle Abbildung $\mathcal{C}_T \dashrightarrow \mathcal{C}_T$ mit

$$\Delta_*(c) = \begin{cases} \Delta_t(c) & \text{falls } \Delta_t(c) \text{ definiert und} \\ & \text{Endkonfiguration ist} \\ \text{undefiniert} & \text{falls } \Delta_t(c) \text{ für alle } t \in \mathbb{N}_0 \text{ definiert ist} \end{cases}$$

16.2.2 Eingaben für Turingmaschinen

Informell (und etwas ungenau) gesprochen werden Turingmaschinen für „zwei Arten von Aufgaben“ eingesetzt: Zum einen wie endliche Akzeptoren zur Entscheidung der Frage, ob ein Eingabewort zu einer bestimmten formalen Sprache gehört. Man spricht in diesem Zusammenhang auch von *Entscheidungsproblemen*. Zum anderen betrachtet man allgemeiner den Fall der „Berechung von Funktionen“, bei denen der Funktionswert aus einem größeren Bereich als nur $\{0, 1\}$ kommt.

Entscheidungsproblem

In beiden Fällen muss aber jedenfalls der Turingmaschine die Eingabe zur Verfügung gestellt werden. Dazu fordern wir, dass stets ein *Eingabealphabet* $A \subset X \setminus \{\square\}$ spezifiziert ist. (Das Blanksymbol gehört also nie zum Eingabealphabet.)

Eingabealphabet

Und die Eingabe eines Wortes $w \in A^*$ wird bewerkstelligt, indem die Turingmaschine im Anfangszustand z_0 mit dem Kopf auf Feld 0 gestartet wird mit der Bandbeschriftung

$$b_w : \mathbb{Z} \to X$$
$$b_w(i) = \begin{cases} \Box & \text{falls } i < 0 \vee i \geq |w| \\ w(i) & \text{falls } 0 \leq i \wedge i < |w| \end{cases}$$

Für die so definierte *zur Eingabe w gehörende Anfangskonfiguration* schreiben wir auch $c_0(w)$.

zu w gehörende Anfangskonfiguration

Interessiert man sich z. B. für die Berechnung von Funktionen der Form $f : \mathbb{N}_0 \to \mathbb{N}_0$, dann wählt man üblicherweise die naheliegende Binärdarstellung des Argumentes x für f als Eingabewort für die Turingmaschine und vereinbart z. B., dass in der Endkonfiguration das Band wieder vollständig leer ist bis auf die Binärdarstellung von $f(x)$. Die ganzen technischen Details hierzu wollen wir uns sparen. Sie mögen aber bitte glauben, dass man das alles ordentlich definieren kann.

16.2.3 Ergebnisse von Turingmaschinen

Man wählt verschiedene Arten, wie eine Turingmaschine ein Ergebnis „mitteilt“, wenn sie hält. Im Falle von Entscheidungsproblemen wollen wir wie bei endlichen Akzeptoren davon ausgehen, dass eine Teilmenge $F \subset Z$ von *akzpetierenden Zuständen* definiert ist. Ein Wort w gilt als *akzeptiert*, wenn die Turingmaschine für Eingabe w hält und der Zustand der Endkonfiguration $\Delta_*(c_0(w))$ ein akzepierender ist. Die Menge aller von einer Turingmaschine T akzeptierten Wörter heißt wieder *akzeptierte formale Sprache* $L(T)$. Wir sprechen in diesem Zusammenhang gelegentlich auch von *Turingmaschinenakzeptoren*.

akzeptierender Zustand
akzeptiertes Wort
akzeptierte formale Sprache
Turingmaschinenakzeptor

Wenn ein Wort w von einer Turingmaschine *nicht* akzeptiert wird, dann gibt es dafür zwei mögliche Ursachen:

- Die Turingmaschine hält für Eingabe w in einem nicht akzeptierenden Zustand.
- Die Turingmaschine hält für Eingabe w nicht.

Im ersten Fall bekommt man sozusagen die Mitteilung „Ich bin fertig und lehne die Eingabe ab.“ Im zweiten Fall weiß man nach jedem Schritt nur, dass die Turingmaschine noch arbeitet. Ob sie irgendwann anhält, und ob sie die Eingabe dann akzeptiert oder ablehnt, ist im allgemeinen unbekannt. Eine formale Sprache, die von einer Turingmaschine akzeptiert werden kann, heißt auf *aufzählbare Sprache*.

aufzählbare Sprache

Wenn es eine Turingmaschine T gibt, die L akzeptiert und für jede Eingabe hält, dann sagt man auch, dass T die Sprache L *entscheide* und dass L *entscheidbar*

entscheiden
entscheidbare Sprache

ist. Dass das eine echt stärkere Eigenschaft ist als Aufzählbarkeit werden wir in Abschnitt 16.4 ansprechen.

Als Beispiel betrachten wir die Aufgabe, für jedes Eingabewort $w \in L = \{\mathtt{a}, \mathtt{b}\}^*$ festzustellen, ob es ein Palindrom ist oder nicht. Es gilt also einen Turingmaschinenakzeptor zu finden, der genau L entscheidet. In Abbildung 16.3 ist eine solche Turingmaschine angegeben. Ihr Anfangszustand ist r und einziger akzeptierender Zustand ist f_+. Der Algorithmus beruht auf der Idee, dass ein Wort genau dann Palindrom ist, wenn erstes und letztes Symbol übereinstimmen und das Teilwort dazwischen auch ein Palindrom ist.

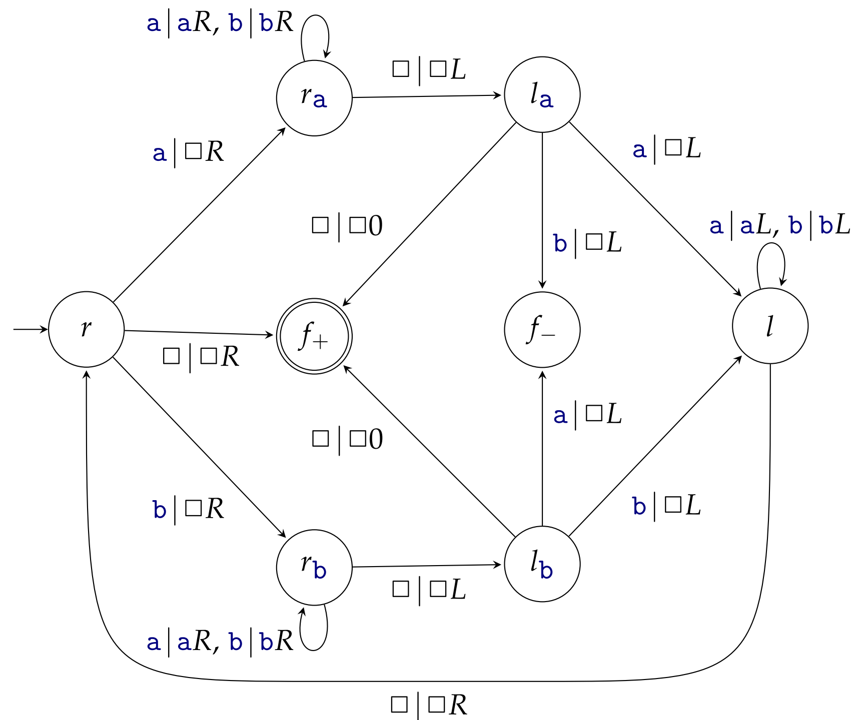


Abbildung 16.3: Eine Turingmaschine zur Palindromerkennung; f_+ sei der einzige akzeptierende Zustand

Zur Erläuterung der Arbeitsweise der Turingmaschine ist in Abbildung 16.4 beispielhaft die Berechnung für Eingabe `abba` angegeben. Man kann sich klar machen (tun Sie das auch), dass die Turingmaschine alle Palindrome und nur die akzeptiert und für jede Eingabe hält. Sie entscheidet die Sprache der Palindrome also sogar.

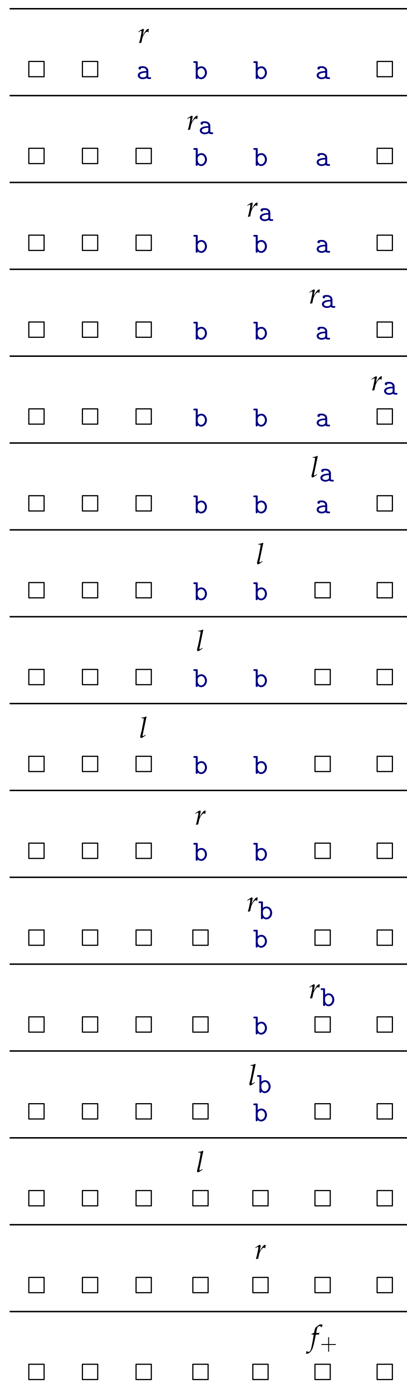


Abbildung 16.4: Akzeptierende Berechung der Turingmaschine aus Abbildung 16.3 für Eingabe abba

16.3 Berechnungskomplexität

Wir beginnen mit einem wichtigen Hinweis: Der Einfachheit halber wollen wir in diesem Abschnitt davon ausgehen, dass wir ausschließlich mit Turingmaschinen zu tun haben, die für jede Eingabe halten. Die Definitionen sind dann leichter hinzuschreiben. Und für die Fragestellungen, die uns in diesem und im folgenden Abschnitt interessieren ist in „in Ordnung“. Warum dem so ist, erfahren Sie vielleicht einmal in einer Vorlesung über Komplexitätstheorie.

In Abschnitt 16.4 werden wir dann aber wieder gerade von dem allgemeinen Fall ausgehen, dass eine Turingmaschine für manche Eingaben *nicht* hält. Warum das wichtig ist, werden Sie dann schnell einsehen. Viel mehr zu diesem Thema werden Sie vielleicht einmal in einer Vorlesung über Berechenbarkeit oder/und Rekursionstheorie hören.

16.3.1 Komplexitätsmaße

Komplexitätsmaß

Bei Turingmaschinen kann man leicht zwei sogenannte *Komplexitätsmaße* definieren, die Rechenzeit und Speicherplatzbedarf charakterisieren.

Für die Beurteilung des Zeitbedarfs definiert man zwei Funktionen $\text{time}_T : A^+ \to \mathbb{N}_+$ und $\text{Time}_T : \mathbb{N}_+ \to \mathbb{N}_+$ wie folgt:

$$\text{time}_T(w) = \text{dasjenige } t \text{ mit } \Delta_t(c_0(w)) = \Delta_*(c_0(w))$$
$$\text{Time}_T(n) = \max\{\text{time}_T(w) \mid w \in A^n\}$$

Da wir im Moment davon ausgehen, dass die betrachteten Turingmaschinen immer halten, sind diese Abbildungen total. Der Einfachheit halber lassen wir das leere Wort bei diesen Betrachtungen weg.

Zeitkomplexität

Üblicherweise heißt die Abbildung Time_T die *Zeitkomplexität* der Turingmaschine T. Man beschränkt sich also darauf, den Zeitbedarf in Abhängigkeit von der Länge der Eingabe (und nicht für jede Eingabe einzeln) anzugeben (nach oben beschränkt). Man sagt, dass die Zeitkomplexität einer Turingmaschine *polynomiell* ist, wenn ein Polynom $p(n)$ existiert, so dass $\text{Time}_T(n) \in \mathrm{O}(p(n))$.

pollynomielle Zeitkomplexität

Welche Zeitkomplexität hat unsere Turingmaschine zur Palindromerkennung? Für eine Eingabe der Länge $n \geq 2$ muss sie schlimmstenfalls

- erstes und letztes Symbol miteinander vergleichen, stellt dabei fest, dass sie übereinstimmen, und muss dann
- zurücklaufen an Anfang des Restwortes der Länge $n-2$ ohne die Randsymbole und
- dafür wieder einen Palindromtest machen.

Der erste Teil erfordert $2n+1$ Schritte. Für den Zeitbedarf $\text{Time}(n)$ gilt also jedenfalls:

$$\text{Time}(n) \leq n + 1 + n + \text{Time}(n-2)$$

Der Zeitaufwand für Wörter der Länge 1 ist ebenfalls gerade $2n+1$. Eine kurze Überlegung zeigt, dass daher die Zeitkomplexität $\text{Time}(n) \in O(n^2)$, also polynomiell ist.

Für die Beurteilung des Speicherplatzbedarfs definiert man zwei Funktionen $\text{space}_T(w) : A^+ \to \mathbb{N}_+$ $\text{Space}_T(n) : \mathbb{N}_+ \to \mathbb{N}_+$ wie folgt:

$$\begin{aligned} \text{space}_T(w) &= \text{die Anzahl der Felder, die während der} \\ &\qquad \text{Berechnung für Eingabe } w \text{ benötigt werden} \\ \text{Space}_T(n) &= \max\{\text{space}_T(w) \mid w \in A^n\} \end{aligned}$$

Raumkomplexität
Platzkomplexität

Üblicherweise heißt die Abbildung Space_T die *Raumkomplexität* oder *Platzkomplexität* der Turingmaschine T. Dabei gelte ein Feld als „benötigt", wenn es zu Beginn ein Eingabesymbol enthält oder irgendwann vom Kopf der Turingmaschine besucht wird. Man sagt, dass die Raumkomplexität einer Turingmaschine *polynomiell* ist, wenn ein Polynom $p(n)$ existiert, so dass $\text{Space}_T(n) \in O(p(n))$.

polynomielle Raumkomplexität

Unsere Beispielmaschine zur Palindromerkennung hat Platzbedarf $\text{Space}(n) = n+1 \in \Theta(n)$, weil außer den n Feldern mit den Eingabesymbolen nur noch ein weiteres Feld rechts davon besucht wird.

Welche Zusammenhänge bestehen zwischen der Zeit- und der Raumkomplexität einer Turingmaschine? Wenn eine Turingmaschine für eine Eingabe w genau $\text{time}(w)$ viele Schritte macht, dann kann sie nicht mehr als $1 + \text{time}(w)$ verschiedene Felder mit ihrem Kopf besuchen. Folglich ist sicher immer

$$\text{space}(w) \leq \max(|w|, 1 + \text{time}(w)) .$$

Also hat eine Turingmaschine mit polynomieller Laufzeit auch nur polynomiellen Platzbedarf.

Umgekehrt kann man aber auf k Feldern des Bandes $|X|^k$ verschieden Inschriften speichern. Daraus folgt, dass es sehr wohl Turingmaschinen gibt, die zwar polynomielle Raumkomplexität, aber exponentielle Zeitkomplexität haben.

16.3.2 Komplexitätsklassen

Komplexitätsklasse

Eine *Komplexitätsklasse* ist eine Menge von Problemen. Wir beschränken uns im folgenden wieder auf Entscheidungsprobleme, also formale Sprachen. Charakterisiert werden Komplexitätsklassen oft durch Beschränkung der zur Verfügung

stehen Ressourcen, also Schranken für Zeitkomplexität oder/und Raumkomplexität (oder andere Maße). Zum Beispiel könnte man die Menge aller Entscheidungsprobleme betrachten, die von Turingmaschinen entschieden werden können, bei denen gleichzeitig die Zeitkomplexität in $O\left(n^2\right)$ und die Raumkomplexität in $O\left(n^{3/2}\log n\right)$ ist, wobei hier n wieder für die Größe der Probleminstanz, also die Länge des Eingabewortes, steht.

Es hat sich herausgestellt, dass unter anderem die beiden folgenden Komplexitätsklassen interessant sind:

- **P** ist die Menge aller Entscheidungsprobleme, die von Turingmaschinen entschieden werden können, deren Zeitkomplexität polynomiell ist.
- **PSPACE** ist die Menge aller Entscheidungsprobleme, die von Turingmaschinen entschieden werden können, deren Raumkomplexität polynomiell ist.

Zu **P** gehört zum Beispiel das Problem der Palindromerkennung. Denn wie wir uns überlegt haben, benötigt die Turingmaschine aus Abbildung 16.3 für Wörter der Länge n stets $O\left(n^2\right)$ Schritte, um festzustellen, ob w Palindrom ist.

Ein Beispiel eines Entscheidungsproblemes aus **PSPACE** haben wir schon in Einheit 15 erwähnt, nämlich zu entscheiden, ob zwei reguläre Ausdrücke die gleiche formale Sprache beschreiben. In diesem Fall besteht also jede Probleminstanz aus zwei regulären Ausdrücken. Als das (jeweils eine) Eingabewort für die Turingmaschine würde man in diesem Fall die Konkatenation der beiden regulären Ausdrücke mit einem dazwischen gesetzten Trennsymbol, das sich von allen anderen Symbolen unterscheidet, wählen.

Welche Zusammenhänge bestehen zwischen **P** und **PSPACE**? Wir haben schon erwähnt, dass eine Turingmaschine mit polynomieller Laufzeit auch nur polynomiell viele Felder besuchen kann. Also ist eine Turingmaschine, die belegt, dass ein Problem in **P** liegt, auch gleich ein Beleg dafür, dass das Problem in **PSPACE** liegt. Folglich ist

$$\mathbf{P} \subseteq \mathbf{PSPACE}\ .$$

Und wie ist es umgekehrt? Wir haben auch erwähnt, dass eine Turingmaschine mit polynomiellem Platzbedarf exponentiell viele Schritte machen kann. Und solche Turingmaschinen gibt es auch. Aber Vorsicht: Das heißt *nicht*, dass **PSPACE** eine echte Obermenge von **P** ist. Bei diesen Mengen geht es um Probleme, nicht um Turingmaschinen! Es könnte ja sein, dass es zu jeder Turingmaschine mit polynomiellem Platzbedarf auch dann, wenn sie exponentielle Laufzeit hat, eine andere Turingmaschine mit nur polynomiellem Zeitbedarf gibt, die genau das gleiche Problem entscheidet. Ob das so ist, weiß man nicht. Es ist eines der großen offenen wissenschaftlichen Probleme, herauszufinden, ob $\mathbf{P} = \mathbf{PSPACE}$ ist oder $\mathbf{P} \neq \mathbf{PSPACE}$.

16.4 UNENTSCHEIDBARE PROBLEME

In gewisser Weise noch schlimmer als Probleme, die exorbitanten Ressourcenbedarf zur Lösung erfordern, sind Probleme, die man algorithmisch, also z. B. mit Turingmaschinen oder Java-Programmen, überhaupt nicht lösen kann.

In diesem Abschnitt wollen wir zumindest andeutungsweise sehen, dass es solche Probleme tatsächlich gibt.

16.4.1 Codierungen von Turingmaschinen

Zunächst soll eine Möglichkeit beschrieben werden, wie man jede Turingmaschine durch ein Wort über dem festen Alphabet $A = \{\texttt{[},\texttt{]},\texttt{0},\texttt{1}\}$ beschreiben kann. Natürlich kann man diese Symbole durch Wörter über $\{\texttt{0},\texttt{1}\}$ codieren und so die Alphabetgröße auf einfache Weise noch reduzieren.

Eine Turingmaschine $T = (Z, z_0, X, \square, f, g, m)$ kann man zum Beispiel wie folgt codieren.

- Die Zustände von T werden ab 0 durchnummeriert.
- Der Anfangszustand bekommt Nummer 0.
- Alle Zustände werden durch gleich lange Binärdarstellungen ihrer Nummern, umgeben von einfachen eckigen Klammern, repräsentiert.
- Wir schreiben $\mathrm{cod}_Z(z)$ für die Codierung von Zustand z.
- Die Bandsymbole werden ab 0 durchnummeriert.
- Das Blanksymbol bekommt Nummer 0.
- Alle Bandsymbole werden durch gleich lange Binärdarstellungen ihrer Nummern, umgeben von einfachen eckigen Klammern, repräsentiert.
- Wir schreiben $\mathrm{cod}_X(x)$ für die Codierung von Bandsymbol x.
- Die möglichen Bewegungsrichtungen des Kopfes werden durch die Wörter `[10]`, `[00]` und `[01]` (für -1, 0 und 1) repräsentiert.
- Wir schreiben $\mathrm{cod}_M(r)$ für die Codierung der Bewegungsrichtung r.
- Die partiellen Funktionen f, g und m werden wie folgt codiert:
  - Wenn sie für ein Argumentpaar (z, x) nicht definiert sind, wird das codiert durch das Wort $\mathrm{cod}_{fgm}(z, x) = \texttt{[}\,\mathrm{cod}_Z(z)\,\mathrm{cod}_X(x)\,\texttt{[]}\,\texttt{[]}\,\texttt{[]}\texttt{]}$.
  - Wenn sie für ein Argumentpaar (z, x) definiert sind, wird das codiert durch das Wort $\mathrm{cod}_{fgm}(z, x) =$ $\texttt{[}\,\mathrm{cod}_Z(z)\,\mathrm{cod}_X(x)\,\mathrm{cod}_Z(f(z, x))\,\mathrm{cod}_X(g(z, x))\,\mathrm{cod}_M(m(z, x))\texttt{]}$.
  - Die Codierung der gesamten Funktionen ist die Konkatenation aller $\mathrm{cod}_{fgm}(z, x)$ für alle $z \in Z$ und alle $x \in X$.
- Die gesamte Turingmaschine wird codiert als Konkatenation der Codierung des Zustands mit der größten Nummer, des Bandsymbols mit der größten Nummer und der Codierung der gesamten Funktionen f, g und m.

- Wir schreiben auch T_w für die Turingmaschine mit Codierung w.

Auch ohne dass wir das hier im Detail ausführen, können Sie hoffentlich zumindest glauben, dass man eine Turingmaschine konstruieren kann, die für jedes beliebiges Wort aus A^* feststellt, ob es die Codierung einer Turingmaschine ist. Mehr wird für das Verständnis des folgenden Abschnittes nicht benötigt.

universelle Turingmaschine

Tatsächlich kann man sogar noch mehr: Es gibt sogenannte *universelle Turingmaschinen*. Eine universelle Turingmaschine U

- erhält als Eingabe zwei Argumente, etwa als Wort `[`w_1`][`w_2`]`,
- überprüft, ob w_1 Codierung einer Turingmaschine T ist, und
- falls ja, simuliert Schritt für Schritt die Arbeit, die T für Eingabe w_2 durchführen würde,
- und falls T endet, liefert U am Ende als Ergebnis das, was T geliefert hat.

16.4.2 Das Halteproblem

Diagonalisierung

Der Kern des Nachweises der Unentscheidbarkeit des Halteproblems ist *Diagonalisierung*. Die Idee geht auf Georg Ferdinand Ludwig Philipp Cantor (1845–1918, siehe z. B. http://www-history.mcs.st-and.ac.uk/Biographies/Cantor.html, 14.1.2010), der sie benutzte um zu zeigen, dass die Menge der reellen Zahlen nicht abzählbar unendlich ist. Dazu sei eine „zweidimensionale unendliche Tabelle" gegeben, deren Zeilen mit Funktionen f_i ($i \in \mathbb{N}_0$) indiziert sind, und die Spalten mit Argumenten x_j ($j \in \mathbb{N}_0$). Eintrag in Zeile i und Spalte j der Tabelle sei gerade der Funktionswert $f_i(x_j)$. Die f_i mögen als Funktionswerte zumindest 0 und 1 annehmen können.

	x_0	x_1	x_2	x_3	x_4	$\cdots$
f_0	$f_0(x_0)$	$f_0(x_1)$	$f_0(x_2)$	$f_0(x_3)$	$f_0(x_4)$	$\cdots$
f_1	$f_1(x_0)$	$f_1(x_1)$	$f_1(x_2)$	$f_1(x_3)$	$f_1(x_4)$	$\cdots$
f_2	$f_2(x_0)$	$f_2(x_1)$	$f_2(x_2)$	$f_2(x_3)$	$f_2(x_4)$	$\cdots$
f_3	$f_3(x_0)$	$f_3(x_1)$	$f_3(x_2)$	$f_3(x_3)$	$f_3(x_4)$	$\cdots$
f_4	$f_4(x_0)$	$f_4(x_1)$	$f_4(x_2)$	$f_4(x_3)$	$f_4(x_4)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Cantors Beobachtung war (im Kern) die folgende: Wenn man die Diagonale der Tabelle nimmt, also die Abbildung d mit $d(x_i) = f_i(x_i)$ und dann *alle Einträge ändert*, also

$$\bar{d}(x_i) = \overline{f_i(x_i)} = \begin{cases} 1 & \text{falls } f_i(x_i) = 0 \\ 0 & \text{sonst} \end{cases}$$

dann erhält man eine Abbildung („Zeile") $\overline{d}$,

	x_0	x_1	x_2	x_3	x_4	$\cdots$
d	$f_0(x_0)$	$f_1(x_1)$	$f_2(x_2)$	$f_3(x_3)$	$f_4(x_4)$	$\cdots$
$\overline{d}$	$\overline{f_0(x_0)}$	$\overline{f_1(x_1)}$	$\overline{f_2(x_2)}$	$\overline{f_3(x_3)}$	$\overline{f_4(x_4)}$	$\cdots$

die sich von jeder Abbildung („Zeile") der gegebenen Tabelle unterscheidet, denn für alle i ist

$$\overline{d}(x_i) = \overline{f_i(x_i)} \neq f_i(x_i) .$$

Die Ausnutzung dieser Tatsache ist von Anwendung zu Anwendung (und es gibt in der Informatik mehrere, z. B. in der Komplexitätstheorie) verschieden.

Im folgenden wollen wir mit Hilfe dieser Idee nun beweisen, dass das Halteproblem unentscheidbar ist. Es ist keine Beschränkung der Allgemeinheit, wenn wir uns auf ein Alphabet A festlegen, über dem wir bequem Codierungen von Turingmaschinen aufschreiben können. Statt „Turingmaschine T hält für Eingabe w" sagen wir im folgenden kürzer „$T(w)$ hält".

Das *Halteproblem* ist die formale Sprache

Halteproblem

$$H = \{w \in A^* \mid w \text{ ist eine TM-Codierung und } T_w(w) \text{ hält.}\}$$

Wir hatten weiter vorne erwähnt, dass es kein Problem darstellt, für ein Wort festzustellen, ob es z. B. gemäß der dort beschriebenen Codierung eine Turingmaschine beschreibt. Das wesentliche Problem beim Halteproblem ist also tatsächlich das Halten.

16.1 Theorem *Das Halteproblem ist unentscheidbar, d. h. es gibt keine Turingmaschine, die H entscheidet.*

16.2 Beweis. Wir benutzen (eine Variante der) Diagonalisierung. In der obigen großen Tabelle seien nun die x_i alle Codierungen von Turingmaschinen in irgendeiner Reihenfolge. Und f_i sei die Funktion, die von der Turingmaschine mit Codierung x_i, also T_{x_i}, berechnet wird.

Da wir es mit Turingmaschinen zu tun haben, werden manche der $f_i(x_j)$ nicht definiert sein, da T_{x_i} für Eingabe x_j nicht hält. Da die x_i *alle* Codierungen von Turingmaschinen sein sollen, ist in der Tabelle für *jede* Turingmaschine eine Zeile vorhanden.

Nun nehmen wir an, dass es doch eine Turingmaschine T_h gäbe, die das Halteproblem entscheidet, d. h. *für jede Eingabe x_i hält* und als Ergebnis mitteilt, ob T_{x_i} für Eingabe x_i hält.

Wir führen diese Annahme nun zu einem Widerspruch, indem wir zeigen: Es gibt eine Art „verdorbene Diagonale" $\overline{d}$ ähnlich wie oben mit den beiden folgenden sich widersprechenden Eigenschaften.

- Einerseits unterscheidet sich $\overline{d}$ von jeder Zeile der Tabelle, also von jeder von einer Turingmaschine berechneten Funktion.
- Andererseits kann auch $\overline{d}$ von einer Turingmaschine berechnet werden.

Und das ist ganz einfach: Wenn die Turingmaschine T_h existiert, dann kann man auch den folgenden Algorithmus in einer Turingmaschine $T_{\overline{d}}$ realisieren:

- Für eine Eingabe x_i berechnet $T_{\overline{d}}$ zunächst, welches Ergebnis T_h für diese Eingabe liefern würde.
- Dann arbeitet $T_{\overline{d}}$ wie folgt weiter:
  - Wenn T_h mitteilt, dass $T_{x_i}(x_i)$ hält,
    dann geht $T_{\overline{d}}$ in eine Endlosschleife.
  - Wenn T_h mitteilt, dass $T_{x_i}(x_i)$ nicht hält,
    dann hält $T_{\overline{d}}$ (und liefert irgendein Ergebnis, etwa 0).
- Andere Möglichkeiten gibt es nicht, und in beiden Fällen ist das Verhalten von $T_{\overline{d}}$ für Eingabe x_i anders als das von T_{x_i} für die gleiche Eingabe.

Also: Wenn Turingmaschine T_h existiert, dann existiert auch Turingmaschine $T_{\overline{d}}$, aber jede Turingmaschine (T_{x_i}) verhält sich für mindestens eine Eingabe (x_i) anders als $T_{\overline{d}}$.

Das ist ein Widerspruch. Folglich war die Annahme, dass es die Turingmaschine T_h gibt, die H entscheidet, falsch. ■

16.4.3 Die Busy-Beaver-Funktion

In Abschnitt 16.2 hatten wir BB3 als erstes Beispiel einer Turingmaschine gesehen. Diese Turingmaschine hat folgende Eigenschaften:

- Bandalphabet ist $X = \{\Box, 1\}$.
- Die Turingmaschine hat $3 + 1$ Zustände, wobei
  - in 3 Zuständen für jedes Bandsymbol der nächste Schritt definiert ist,
  - einer dieser 3 Zustände der Anfangszustand ist und
  - in einem weiteren Zustand für kein Bandsymbol der nächste Schritt definiert ist („Haltezustand").
- Wenn man die Turingmaschine auf dem leeren Band startet, dann hält sie nach endlich vielen Schritten.

Wir interessieren uns nun allgemein für Turingmaschinen mit den Eigenschaften:

- Bandalphabet ist $X = \{\Box, 1\}$.
- Die Turingmaschine hat $n + 1$ Zustände, wobei

  - in n Zuständen für jedes Bandsymbol der nächste Schritt definiert ist,
  - einer dieser n Zustände der Anfangszustand ist und
  - in einem weiteren Zustand für kein Bandsymbol der nächste Schritt definiert ist („Haltezustand“).
- Wenn man die Turingmaschine auf dem leeren Band startet, dann hält sie nach endlich vielen Schritten.

Solche Turingmaschinen wollen wir der Einfachheit halber *Bibermaschinen* nennen. Genauer wollen wir von einer n-Bibermaschine reden, wenn sie, einschließlich des Haltezustands $n+1$ Zustände hat. Wann immer es im folgenden explizit oder implizit um Berechnungen von Bibermaschinen geht, ist immer die gemeint, bei der sie auf dem vollständig leeren Band startet. Bei BB3 haben wir gesehen, dass sie 6 Einsen auf dem Band erzeugt.

Bibermaschine

Abbildung 16.5: Europäischer Biber (castor fiber), Bildquelle: http://upload.wikimedia.org/wikipedia/commons/d/d4/Beaver_2.jpg (14.1.2010)

Eine Bibermaschine heiße *fleißiger Biber*, wenn sie am Ende die maximale Anzahl Einsen auf dem Band hinterlässt unter allen Bibermaschinen mit gleicher Zustandszahl.

fleißiger Biber

Die *Busy-Beaver-Funktion* ist wie folgt definiert:

Busy-Beaver-Funktion

$$\begin{aligned} \text{bb} &: \mathbb{N}_+ \to \mathbb{N}_+ \\ \text{bb}(n) &= \text{die Anzahl von Einsen, die ein fleißiger Biber mit} \\ &\qquad n+1 \text{ Zuständen am Ende auf dem Band hinterlässt} \end{aligned}$$

Diese Funktion wird auch *Radó-Funktion* genannt nach dem ungarischen Mathematiker Tibor Radó, der sich als erster mit dieser Funktion beschäftigte. Statt $\text{bb}(n)$ wird manchmal auch $\Sigma(n)$ geschrieben.

Radó-Funktion

Da BB3 am Ende 6 Einsen auf dem Band hinterlässt, ist also jedenfalls $\text{bb}(3) \geq 6$. Radó fand heraus, dass sogar $\text{bb}(3) = 6$ ist. Die Turingmaschine BB3 ist also ein fleißiger Biber.

Brady hat 1983 gezeigt, dass $\text{bb}(4) = 13$ ist. Ein entsprechender fleißiger Biber ist

	A	B	C	D	H
$\square$	$1, R, B$	$1, L, A$	$1, R, H$	$1, R, D$	
1	$1, L, B$	$\square, L, C$	$1, L, D$	$\square, R, A$	

H. Marxen (http://www.drb.insel.de/~heiner/BB/, 14.1.2010) und J. Buntrock haben 1990 eine 5-Bibermaschine gefunden, die 4098 Einsen produziert und nach 47 176 870 Schritten hält. Also ist $\text{bb}(5) \geq 4098$. Man weiß nicht, ob es eine 5-Bibermaschine gibt, die mehr als 4098 Einsen schreibt.

Im Dezember 2007 haben Terry and Shawn Ligocki eine 6-Bibermaschine gefunden, die mehr als $4.6 \cdot 10^{1439}$ Einsen schreibt und nach mehr als $2.5 \cdot 10^{2879}$ Schritten hält. Also ist $\text{bb}(6) \geq 4.6 \cdot 10^{1439}$. Nein, hier liegt kein Schreibfehler vor!

Man hat den Eindruck, dass die Funktion bb sehr schnell wachsend ist. Das stimmt. Ohne den Beweis hier wiedergeben zu wollen (Interessierte finden ihn z. B. in einem Aufsatz von J. Shallit (http://grail.cba.csuohio.edu/~somos/beaver.ps, 14.1.10) teilen wir noch den folgenden Satz mit. In ihm und dem unmittelbaren Korollar wird von der Berechenbarkeit von Funktionen $\mathbb{N}_+ \to \mathbb{N}_+$ gesprochen. Damit ist gemeint, dass es eine Turingmaschine gibt, die

- als Eingabe das Argument (zum Beispiel) in binärer Darstellung auf einem ansonsten leeren Band erhält, und
- als Ausgabe den Funktionswert (zum Beispiel) in binärer Darstellung auf einem ansonsten leeren Band liefert.

16.3 Theorem *Für jede berechenbare Funktion $f : \mathbb{N}_+ \to \mathbb{N}_+$ gibt es ein n_0, so dass für alle $n \geq n_0$ gilt:* $\text{bb}(n) > f(n)$.

16.4 Korollar. Die Busy-Beaver-Funktion $\text{bb}(n)$ ist nicht berechenbar.

16.5 AUSBLICK

Sie werden an anderer Stelle lernen, dass es eine weitere wichtige Komplexitätsklasse gibt, die **NP** heißt. Man weiß, dass $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$ gilt, aber für keine der beiden Inklusionen weiß man, ob sie echt ist. Gewisse Probleme aus **NP** und **PSPACE** (sogenannte *vollständige* Probleme) spielen an vielen Stellen (auch in der Praxis) eine ganz wichtige Rolle. Leider ist es in allen Fällen so dass alle

bekannten Algorithmen exponentielle Laufzeit haben, aber man nicht beweisen kann, dass das so sein muss.

LITERATUR

Turing, A. M. (1936). "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society*. 2. Ser. 42, S. 230–265.
Die Pdf-Version einer HTML-Version ist online verfügbar; siehe http://web.comlab.ox.ac.uk/oucl/research/areas/ieg/e-library/sources/tp2-ie.pdf (15.1.10).

17 RELATIONEN

17.1 ÄQUIVALENZRELATIONEN

17.1.1 Definition

In Abschnitt 11.2.1 hatten wir schon einmal erwähnt, dass eine Relation $R \subseteq M \times M$ auf einer Menge M, die

- reflexiv,
- symmetrisch und
- transitiv

ist, *Äquivalenzrelation* heißt. Das bedeutet also, dass

Äquivalenzrelation

- für alle $x \in M$ gilt: $(x, x) \in R$
- für alle $x, y \in M$ gilt: wenn $(x, y) \in R$, dann auch $(y, x) \in R$
- für alle $x, y, z \in M$ gilt: wenn $(x, y) \in R$ und $(y, z) \in R$, dann auch $(x, z) \in R$.

Für Äquivalenzrelationen benutzt man oft Symbole wie $\equiv$, $\sim$ oder $\approx$, die mehr oder weniger deutlich an das Gleichheitszeichen erinnern, sowie Infixschreibweise. Dann liest sich die Definition so: Eine Relation $\equiv$ ist Äquivalenzrelation auf einer Menge M, wenn gilt:

- $\forall x \in M : x \equiv x$,
- $\forall x \in M : \forall y \in M : x \equiv y \Longrightarrow y \equiv x$
- $\forall x \in M : \forall y \in M : \forall z \in M : x \equiv y \wedge y \equiv z \Longrightarrow x \equiv z$

Der Anlass für Symbole, die an das „=“ erinnnern, ist natürlich der, dass Gleichheit, also die Relation $\mathrm{I} = \{(x, x) \mid x \in M\}$, auf jeder Menge eine Äquivalenzrelation ist, denn offensichtlich gilt:

- $\forall x \in M : x = x$,
- $\forall x \in M : \forall y \in M : x = y \Longrightarrow y = x$
- $\forall x \in M : \forall y \in M : \forall z \in M : x = y \wedge y = z \Longrightarrow x = z$

Ein klassisches Beispiel sind die „Kongruenzen modulo n“ auf den ganzen Zahlen. Es sei $n \in \mathbb{N}_+$. Zwei Zahlen $x, y \in \mathbb{Z}$ heißen *kongruent modulo n*, wenn die Differenz $x - y$ durch n teilbar, also ein ganzzahliges Vielfaches von n, ist. Man schreibt typischerweise $x \equiv y \pmod{n}$. Dass dies tatsächlich Äquivalenzrelationen sind, haben Sie in Mathematikvorlesungen mehr oder weniger explizit gesehen.

kongruent modulo n

- Die Reflexivität ergibt sich aus der Tatsache, dass $x - x = 0$ Vielfaches von n ist.
- Die Symmetrie gilt, weil mit $x - y$ auch $y - x = -(x - y)$ Vielfaches von n ist.
- Transitivität: Wenn $x - y = k_1 n$ und $y - z = k_2 n$ (mit $k_1, k_2 \in \mathbb{Z}$), dann ist auch $x - z = (x - y) + (y - z) = (k_1 + k_2)n$ ein ganzzahliges Vielfaches von n.

17.1.2 Äquivalenzrelationen von Nerode

Als ein durchgehendes Beispiel in diesem und weiteren Abschnitten betrachten wir eine Äquivalenzrelation $\equiv_L$, die durch eine formale Sprache $L \subseteq A^*$ auf der Menge A^* aller Wörter induziert wird. Sie heißt die *Äquivalenzrelation von Nerode*. Die Relation ist wie folgt definiert. Für alle $w_1, w_2 \in A^*$ ist

Äquivalenzrelation von Nerode

$$w_1 \equiv_L w_2 \iff \left(\forall w \in A^* : w_1 w \in L \iff w_2 w \in L\right)$$

Wenn man das erste Mal mit dieser Definition konfrontiert ist, muss man sie erfahrungsgemäß mehrfach lesen. Zwei Wörter w_1 und w_2 sind dann und nur dann äquivalent, wenn gilt: Gleich, welches Wort $w \in A^*$ man an die beiden anhängt, immer sind entweder beide Produkte $w_1 w$ und $w_2 w$ in L, oder keines von beiden. Anders gesagt: Zwei Wörter sind genau dann *nicht* $\equiv_L$-äquivalent, wenn es ein Wort $w \in A^*$ gibt, so dass genau eines der Wörter $w_1 w$ und $w_2 w$ in L liegt, aber das andere nicht.

Betrachtet man das leere Wort $w = \varepsilon$, dann ergibt sich insbesondere, dass $w_1 \equiv_L w_2$ höchstens dann gelten kann, wenn beide Wörter in L liegen, oder beide nicht in L.

Nehmen wir als Beispiel das Alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ und die formale Sprache $L = \langle \mathtt{a*b*} \rangle \subset A^*$ aller Wörter, in denen nirgends das Teilwort `ba` vorkommt. Versuchen wir anhand einiger Beispielpaare von Wörtern, ein Gefühl dafür zu bekommen, welche Wörter $\equiv_L$-äquivalent sind und welche nicht.

1. $w_1 =$ `aaa` und $w_2 =$ `a`:
   - Hängt man an beide Wörter ein $w \in \langle \mathtt{a*} \rangle$ an, dann sind sowohl $w_1 w$ als auch $w_2 w$ in L.
   - Hängt man an beide Wörter ein $w \in \langle \mathtt{a*bb*} \rangle$ an, dann sind ebenfalls wieder sowohl $w_1 w$ als auch $w_2 w$ in L.
   - Hängt man an beide Wörter ein w an, das das Teilwort `ba` enthält, dann enthalten $w_1 w$ und $w_2 w$ beide `ba`, sind also beide nicht in L.
   - Andere Möglichkeiten für ein Suffix w gibt es nicht, also sind die beiden Wörter $\equiv_L$-äquivalent.
2. $w_1 =$ `aaab` und $w_2 =$ `abb`:
   - Hängt man an beide Wörter ein $w \in \langle \mathtt{b*} \rangle$ an, dann sind sowohl $w_1 w$ als auch $w_2 w$ in L.
   - Hängt man an beide Wörter ein w an, das ein `a` enthält, dann enthalten $w_1 w$ und $w_2 w$ beide das Teilwort `ba`, sind also beide nicht in L.
   - Andere Möglichkeiten gibt es nicht, also sind die beiden Wörter $\equiv_L$-äquivalent.
3. $w_1 =$ `aa` und $w_2 =$ `abb`:

- Hängt man an beide Wörter $w = \texttt{a}$ an, dann ist zwar $w_1 w = \texttt{aaa} \in L$, aber $w_2 w = \texttt{abba} \notin L$.
- Also sind die beiden Wörter *nicht* $\equiv_L$-äquivalent.

4. $w_1 = \texttt{aba}$ und $w_2 = \texttt{babb}$:
   - Beide Wörter enthalten $\texttt{ba}$. Egal welches $w \in A^*$ man anhängt, bleibt das so, d. h. immer sind $w_1 w \notin L$ und $w_2 w \notin L$.
   - Also sind die beiden Wörter $\equiv_L$-äquivalent.
5. $w_1 = \texttt{ab}$ und $w_2 = \texttt{ba}$:
   - Da $w_1 \in L$, aber $w_2 \notin L$, zeigt schon das Suffix $w = \varepsilon$, dass die beiden Wörter nicht $\equiv_L$-äquivalent sind.

Die wesentliche Behauptung ist nun:

17.1 Lemma. Für jede formale Sprache L ist $\equiv_L$ eine Äquivalenzrelation.

17.2 Beweis. Man prüft nach, dass die drei definierenden Eigenschaften erfüllt sind.

- Reflexivität: Ist $w_1 \in A^*$, dann gilt für jedes $w \in A^*$ offensichtlich: $w_1 w \in L \iff w_1 w \in L$.
- Symmetrie: Für $w_1, w_2 \in A^*$ und alle $w \in A^*$ gelte: $w_1 w \in L \iff w_2 w \in L$. Dann gilt offensichtlich auch immer $w_2 w \in L \iff w_1 w \in L$.
- Transitivität: Es seien $w_1, w_2, w_3 \in A^*$ und es möge gelten

$$\forall w \in A^* : w_1 w \in L \iff w_2 w \in L \tag{17.1}$$

$$\forall w \in A^* : w_2 w \in L \iff w_3 w \in L \tag{17.2}$$

  Wir müssen zeigen: $\forall w \in A^* : w_1 w \in L \iff w_3 w \in L$. Sei dazu ein beliebiges $w \in A^*$ gegeben. Falls $w_1 w \in L$ ist, dann ist wegen (17.1) auch $w_2 w \in L$ und daher wegen (17.2) auch $w_3 w \in L$. Analog folgt aus $w_1 w \notin L$ der Reihe nach $w_2 w \notin L$ und $w_3 w \notin L$. Also gilt $w_1 w \in L \iff w_3 w \in L$.

■

17.1.3 Äquivalenzklassen und Faktormengen

Für $x \in M$ heißt $\{y \in M \mid x \equiv y\}$ die *Äquivalenzklasse von x*. Man schreibt für die Äquivalenzklasse von x mitunter $[x]_\equiv$ oder einfach $[x]$, falls klar ist, welche Äquivalenzrelation gemeint ist.

Äquivalenzklasse

Für die Menge aller Äquivalenzklassen schreibt man $M_{/\equiv}$ und nennt das manchmal auch die *Faktormenge* oder *Faserung von M nach* $\equiv$, also $M_{/\equiv} = \{[x]_\equiv \mid x \in M\}$.

Faktormenge
Faserung

Ist konkret $\equiv$ die Äquivalenzrelation „modulo n" auf den ganzen Zahlen, dann schreibt man für die Faktormenge auch $\mathbb{Z}_n$.

Mitunter ist es nützlich, sich anzusehen aus wievielen Äquivalenzklassen eine Faserung besteht. Nehmen wir als Beispiel wieder die durch $L = \langle \mathtt{a*b*} \rangle$ induzierte Nerode-Äquivalenz $\equiv_L$. Schaut man sich noch einmal die Argumentationen im vorangegangenen Abschnitt an, dann merkt man, dass jedes Wort zu genau einem der drei Wörter ε, $\mathtt{b}$ und $\mathtt{ba}$ äquivalent ist. Mit anderen Worten besteht $A^*_{/\equiv_L}$ aus drei Äquivalenzklassen:

- $[\varepsilon] = \langle \mathtt{a*} \rangle$
- $[\mathtt{b}] = \langle \mathtt{a*bb*} \rangle$
- $[\mathtt{ba}] = \langle \mathtt{a*bb*a(a|b)*} \rangle$

Die Wahl der Repräsentanten in dieser Aufzählung ist natürlich willkürlich. Wir hätten genauso gut schreiben können:

- $[\mathtt{aaaaa}] = \langle \mathtt{a*} \rangle$
- $[\mathtt{aabbbbb}] = \langle \mathtt{a*bb*} \rangle$
- $[\mathtt{aabbaabbba}] = \langle \mathtt{a*bb*a(a|b)*} \rangle$

Die durch eine formale Sprache L induzierte Nerode-Äquivalenz hat aber nicht immer nur endlich viele Äquivalenzklassen. Als Beispiel betrachte man das schon in Abschnitt 14.4 diskutierte

$$L = \{\mathtt{a}^k\mathtt{b}^k \mid k \in \mathbb{N}_0\} .$$

Für diese Sprache besteht $A^*_{/\equiv_L}$ aus unendlich vielen Äquivalenzklassen. Ist nämlich $k \neq m$, dann sind $w_1 = \mathtt{a}^k$ und $w_2 = \mathtt{a}^m$ nicht äquivalent, wie man durch Anhängen von $w = \mathtt{b}^k$ sieht:

- $w_1 w = \mathtt{a}^k\mathtt{b}^k \in L$, aber
- $w_2 w = \mathtt{a}^m\mathtt{b}^k \notin L$.

Also ist zumindest jedes Wort $\mathtt{a}^k$, $k \in \mathbb{N}_0$ in einer anderen Äquivalenzklasse. Und es sind auch jeweils keine anderen Wörter in diesen Äquivalenzklassen.

Die Wörter der Form $\mathtt{a}^k\mathtt{b}$, $k \in \mathbb{N}_+$ sind ebenfalls alle in paarweise verschiedenen Äquivalenzklassen. Jede von diesen ist aber unendlich groß, denn für jedes k sind jeweils alle Wörter der Form $\mathtt{a}^{k+m}\mathtt{b}^{1+m}$ für beliebiges $m \in \mathbb{N}_0$ äquivalent.

Vielleicht lässt die Tatsache, dass es für die reguläre Sprache $\langle \mathtt{a*b*} \rangle$ endlich viele Äquivalenzklassen gibt, aber für die nicht reguläre Sprache $L = \{\mathtt{a}^k\mathtt{b}^k \mid k \in \mathbb{N}_0\}$ unendlich viele, Sie schon etwas ahnen.

17.2 KONGRUENZRELATIONEN

Mitunter hat eine Menge M, auf der eine Äquivalenzrelation definiert ist, zusätzliche „Struktur“, bzw. auf M sind eine oder mehrere Operationen definiert. Als Beispiel denke man etwa an die ganzen Zahlen $\mathbb{Z}$ mit der Addition. Man kann

sich dann z. B. fragen, wie sich Funktionswerte ändern, wenn man Argumente durch andere, aber äquivalente ersetzt.

17.2.1 Verträglichkeit von Relationen mit Operationen

Um den Formalismus nicht zu sehr aufzublähen, beschränken wir uns in diesem Unterabschnitt auf die zwei am häufigsten vorkommenden einfachen Fälle.

Es sei $\equiv$ eine Äquivalenzrelation auf einer Menge M und $f : M \to M$ eine Abbildung. Man sagt, dass $\equiv$ mit f *verträglich* ist, wenn für alle $x_1, x_2 \in M$ gilt:

verträglich

$$x_1 \equiv x_2 \Longrightarrow f(x_1) \equiv f(x_2) .$$

Ist $\square$ eine binäre Operation auf einer Menge M, dann heißen $\equiv$ und $\square$ *verträglich* , wenn für alle $x_1, x_2 \in M$ und alle $y_1, y_2 \in M$ gilt:

verträglich

$$x_1 \equiv x_2 \land y_1 \equiv y_2 \Longrightarrow x_1 \square y_1 \equiv x_2 \square y_2 .$$

Ein typisches Beispiel sind wieder die Äquivalenzrelationen „modulo n“. Diese Relationen sind mit Addition, Subtraktion und Multiplikation verträglich. Ist etwa

$$\begin{aligned} & x_1 \equiv x_2 \pmod n && \text{also} \quad x_1 - x_2 = kn \\ \text{und} \quad & y_1 \equiv y_2 \pmod n && \text{also} \quad y_1 - y_2 = mn \end{aligned}$$

dann ist zum Beispiel

$$(x_1 + y_1) - (x_2 + y_2) = (x_1 - x_2) + (y_1 - y_2) = (k + m)n .$$

Mit anderen Worten ist dann auch

$$x_1 + y_1 \equiv x_2 + y_2 \pmod n .$$

Eine Äquivalenzrelation, die mit allen gerade interessierenden Funktionen oder/und Operationen verträglich ist, nennt man auch eine *Kongruenzrelation*.

Kongruenzrelation

Auch die Nerode-Äquivalenzen haben eine solche Eigenschaft. Sei $w' \in A^*$ ein beliebiges Wort und sei $f_{w'} : A^* \to A^*$ die Abbildung, die w' an ihr Argument anhängt, also $f_{w'}(v) = vw'$. Wir behaupten, dass $\equiv_L$ mit allen $f_{w'}$ verträglich ist. d. h.:

$$\forall w_1, w_2 \in A^* : w_1 \equiv_L w_2 \Longrightarrow w_1 w' \equiv_L w_2 w'$$

Wir müssen zeigen: Wenn $w_1 \equiv_L w_2$ ist, dann ist auch $w_1 w' \equiv_L w_2 w'$. Gehen wir also davon aus, dass für *alle* $w \in A^*$ gilt: $w_1 w \in L \iff w_2 w \in L$. Wir müssen

zeigen, dass für alle $v \in A^*$ gilt: $(w_1w')v \in L \iff (w_2w')v \in L$. Das geht ganz einfach. Sei $v \in A^*$ beliebig; dann gilt

$$\begin{aligned} (w_1w')v \in L &\iff w_1(w'v) \in L \\ &\iff w_2(w'v) \in L \qquad \text{weil } w_1 \equiv_L w_2 \\ &\iff (w_2w')v \in L\,. \end{aligned}$$

17.2.2 Wohldefiniertheit von Operationen mit Äquivalenzklassen

induzierte Operation

Wann immer man eine Kongruenzrelation vorliegen hat, also z. B. eine Äquivalenzrelation $\equiv$ auf M die mit einer binären Operation $\square$ auf M verträglich ist, *induziert* diese Operation auf M eine Operation auf $M_{/\equiv}$. Analoges gilt für Abbildungen $f : M \to M$.

Betrachten wir wieder die Nerode-Äquivalenzen. L sei wie immer eine beliebige formale Sprache $L \subseteq A^*$. Eben hatten wir uns überlegt, dass insbesondere für jedes $x \in A$ die Abbildung $f_x : A^* \to A^* : w \mapsto wx$ mit $\equiv_L$ verträglich ist.

Wir schreiben nun einmal hin:

$$f'_x : A^*_{/\equiv_L} \to A^*_{/\equiv_L} : [w] \mapsto [wx]$$

Der ganz entscheidende Punkt ist: Dies ist eine vernünftige Definition. Wenn Sie so etwas zum ersten Mal sehen, fragen Sie sich vielleicht, warum es überhaupt Unsinn sein könnte. Nun: Es wird hier versucht eine Abbildung zu definieren, die jede Äquivalenzklasse auf eine Äquivalenzklasse abbildet. Aber die durch $[w]$ beschriebene Klasse enthält ja im allgemeinen nicht nur w, sondern noch viele andere Wörter. Zum Beispiel hatten wir uns weiter vorne überlegt, dass im Fall $L = \langle \texttt{a*b*} \rangle$ die Wörter ε, $\texttt{a}$, $\texttt{a}^2$, $\texttt{a}^3$, usw. alle in einer Äquivalenzklasse liegen. Es ist also $[\varepsilon] = [\texttt{a}] = [\texttt{a}^2] = \cdots$. D. h., damit das, was wir eben für f'_x hingeschrieben haben, wirklich eine Definition ist, die für jedes Argument *eindeutig* einen Funktionswert festlegt, sollte dann bitte auch $[\varepsilon x] = [\texttt{a}x] = [\texttt{a}^2x] = \cdots$ sein. Und das ist so, denn hier hinter steckt nichts anderes als die Forderung

$$\begin{aligned} & w_1 \equiv_L w_2 \implies w_1x \equiv_L w_2x \\ \text{also}\quad & w_1 \equiv_L w_2 \implies f_x(w_1) \equiv_L f_x(w_2) \end{aligned}$$

wohldefiniert

Und weil wir gesehen hatte, dass das gilt, sind wie man auch sagt, die Abbildungen $f'_x : A^*_{/\equiv_L} \to A^*_{/\equiv_L}$ *wohldefiniert*. Die Abbildungsvorschrift ist unabhängig von der Wahl des Repräsentanten der Äquivalenzklasse, die als Argument verwendet wird.

Allgemein gilt: Wenn $\equiv$ mit $f : M \to M$ verträglich ist, dann ist $f' : M_{/\equiv} \to M_{/\equiv} : f'([x]) = [f(x)]$ wohldefiniert.

Zum Abschluss werfen wir einen letzten Blick auf die Nerode-Äquivalenzen. Sei nun L eine formale Sprache, für die $\equiv_L$ nur endlich viele Äquivalenzklassen hat. Wir schreiben zur Abkürzung $Z = A^*_{/\equiv_L}$ und definieren

$$f : Z \times A \to Z : f([w], x) = [wx]$$

Diese Abbildung ist nach dem oben Gesagten wohldefiniert. Und sie erinnert Sie hoffentlich an endliche Automaten. Das ist Absicht. Legt man nämlich noch fest

- $z_0 = [\varepsilon]$ und
- $F = \{[w] \mid w \in L\}$

dann hat man einen endlichen Akzeptor, der genau die formale Sprache L erkennt. Überlegen Sie sich das!

Ohne Beweis teilen wir Ihnen noch die folgenden schönen Tatsachen mit: Für jede formale Sprache, die von einem endlchen Akzeptor erkannt wird, hat $\equiv_L$ nur endlich viele Äquivalenzklassen. Und der gerade konstruierte Akzeptor ist unter allen, die L erkennen, einer mit minimaler Zustandszahl. Und dieser endliche Akzeptor ist bis auf Isomorphie (also Umbenenung von Zuständen) sogar eindeutig.

17.3 HALBORDNUNGEN

Eine Ihnen wohlvertraute Halbordnung ist die Mengeninklusion $\subseteq$. Entsprechende Beispiele tauchen daher im folgenden immer wieder auf, zumal Sie am Ende dieses Abschnittes sehen werden, dass man zum Beispiel durch den Fixpunktsatz von Knaster-Tarski für sogenannte vollständige Halbordnungen noch einmal einen neuen Blick auf kontextfreie Grammatiken bekommt.

17.3.1 Grundlegende Definitionen

Eine Relation $R \subseteq M \times M$ heißt *antisymmetrisch*, wenn für alle $x, y \in M$ gilt:

Antisymmetrie

$$xRy \land yRx \Longrightarrow x = y$$

Eine Relation $R \subseteq M \times M$ heißt *Halbordnung*, wenn sie

Halbordnung

- reflexiv,
- antisymmetrisch und
- transitiv

ist. Wenn R eine Halbordnung auf einer Menge M ist, sagt man auch, die Menge sei *halbgeordnet*.

halbgeordnete Menge

Auf der Menge aller Wörter über einem Alphabet A ist die Relation $\sqsubseteq_p$ ein einfaches Beispiel, die definiert sei vermöge der Festlegung $w_1 \sqsubseteq_p w_2 \iff \exists u \in A^* : w_1 u = w_2$. Machen Sie sich zur Übung klar, dass sie tatsächlich die drei definierenden Eigenschaften einer Halbordnung hat.

Es sei M' eine Menge und $M = 2^{M'}$ die Potenzmenge von M'. Dann ist die Mengeninklusion $\subseteq$ eine Halbordnung auf M. Auch hier sollten Sie noch einmal aufschreiben, was die Aussagen der drei definierenden Eigenschaften einer Halbordnung sind. Sie werden merken, dass die Antisymmetrie eine Möglichkeit an die Hand gibt, die Gleichheit zweier Mengen zu beweisen (wir haben das auch schon ausgenutzt).

Wenn R Halbordnung auf einer *endlichen* Menge M ist, dann stellt man sie manchmal graphisch dar. Wir hatten schon in Unterabschnitt 11.1.4 darauf hingewiesen, dass Relationen und gerichtete Graphen sich formal nicht unterscheiden. Betrachten wir als Beispiel die halbgeordnete Menge $(2^{\{a,b,c\}}, \subseteq)$. Im zugehörigen Graphen führt eine Kante von M_1 zu M_2, wenn $M_1 \subseteq M_2$ ist. Es ergibt sich also die Darstellung aus Abbildung 17.1. Wie man sieht wird das ganze recht schnell

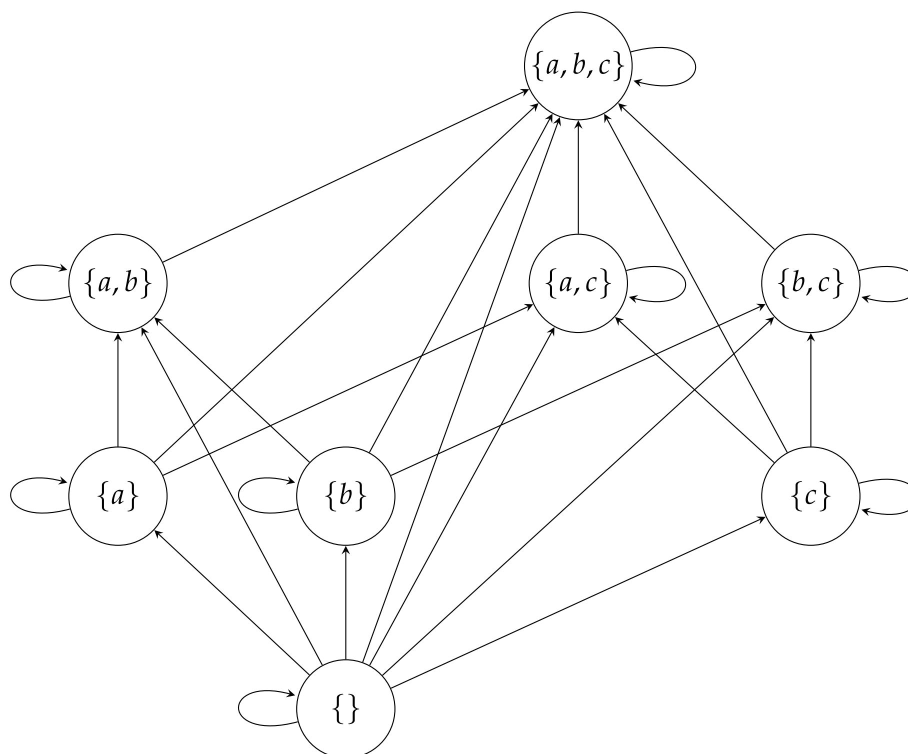


Abbildung 17.1: Die Halbordnung $(2^{\{a,b,c\}}, \subseteq)$ als Graph

relativ unübersichtlich. Dabei ist ein Teil der Kanten nicht ganz so wichtig, weil deren Existenz ohnehin klar ist (wegen der Reflexivität) oder aus anderen Kanten gefolgert werden kann (wegen der Transitivität). Deswegen wählt man meist die Darstellung als sogenanntes *Hasse-Diagramm* dar. Das ist eine Art „Skelett" der Halbordnung, bei dem die eben angesprochenen Kanten fehlen. Genauer gesagt ist es der Graph der Relation $H_R = (R \setminus \mathrm{I}) \setminus (R \setminus \mathrm{I})^2$. In unserem Beispiel ergibt sich aus Abbildung 17.1 durch Weglassen der Kanten Abbildung 17.2.

Hasse-Diagramm

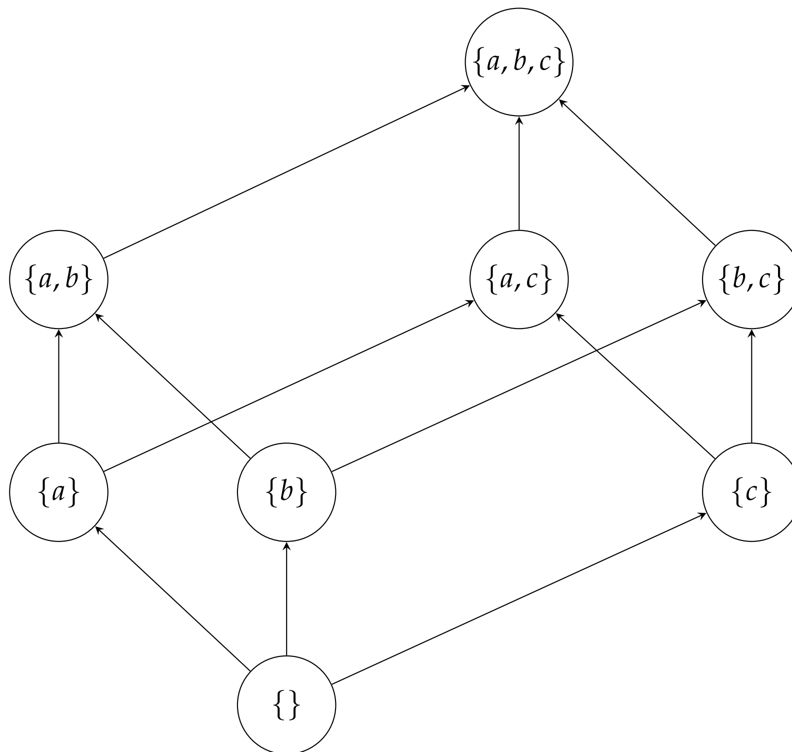


Abbildung 17.2: Hassediagramm der Halbordnung $(2^{\{a,b,c\}}, \subseteq)$

Vom Hassediagramm kommt man „ganz leicht" wieder zur ursprünglichen Halbordnung: Man muss nur die reflexiv-transitive Hülle bilden.

17.3 Lemma. Wenn R eine Halbordnung auf einer endlichen Menge M ist und H_R das zugehörige Hassediagramm, dann ist $H_R^* = R$.

17.4 Beweis. R und H_R sind beides Relationen über der gleichen Grundmenge M (also ist $R^0 = H_R^0$) und offensichtlich ist $H_R \subseteq R$. Eine ganz leichte Induktion (machen Sie sie) zeigt, dass für alle $i \in \mathbb{N}_0$ gilt: $H_R^i \subseteq R^i$ und folglich $H_R^* \subseteq R^* = R$.

Nun wollen wir zeigen, dass umgekehrt auch gilt: $R \subseteq H_R^*$. Sei dazu $(x,y) \in R$. Falls $x = y$ ist, ist auch $(x,y) \in \mathrm{I} \subseteq H_R^*$.

Sei daher im folgenden $x \neq y$, also $(x,y) \in R \setminus \mathrm{I}$, und sein $(x_0, x_1, \dots, x_m)$ eine Folge von Elementen mit folgenden Eigenschaften:

- $x_0 = x$ und $x_m = y$
- für alle $0 \leq i < m$ ist $x_i \sqsubseteq x_{i+1}$
- für alle $0 \leq i < m$ ist $x_i \neq x_{i+1}$

In einer solchen Folge kann kein Element $z \in M$ zweimal auftauchen. Wäre nämlich $x_i = z$ und $x_k = z$ mit $k > i$, dann wäre jedenfalls $k \geq i+2$ und $x_{i+1} \neq z$. Folglich wäre einerseits $z = x_i \sqsubseteq x_{i+1}$ und andererseits $x_{i+1} \sqsubseteq \cdots \sqsubseteq x_k$, also wegen Transitivität $x_{i+1} \sqsubseteq x_k = z$. Aus der Antisymmetrie von $\sqsubseteq$ würde $x_{i+1} = z$ folgen im Widerspruch zum eben Festgehaltenen.

Da in einer Folge $(x_0, x_1, \dots, x_m)$ der beschriebenen Art kein Element zweimal vorkommen kann und M endlich ist, gibt es auch maximal lange solche Folgen. Sei im folgenden $(x_0, x_1, \dots, x_m)$ maximal lang. Dann gilt also für alle $0 \leq i < m$, dass man zwischen zwei Elemente x_i und x_{i+1} kein weiteres Element einfügen kann, dass also gilt: $\neg \exists z \in M : x_i \sqsubseteq z \sqsubseteq x_{i+1} \land x_i \neq z \land x_{i+1} \neq z$.

Dafür kann man auch schreiben: $\neg \exists z \in M : (x_i, z) \in R \setminus \mathrm{I} \land (z, x_{i+1}) \in R \setminus \mathrm{I}$, d. h. $(x_i, x_{i+1}) \notin (R \setminus \mathrm{I})^2$.

Also gilt für alle $0 \leq i < m$: $(x_i, x_{i+1}) \in (R \setminus \mathrm{I}) \setminus (R \setminus \mathrm{I})^2 = H_R$. Daher ist $(x,y) = (x_0, x_m) \in H_R^m \subseteq H_R^*$. ■

gerichtete azyklische Graphen
Dag

Graphen, die das Hassediagramm einer endlichen Halbordnung sind, heißen auch *gerichtete azyklische Graphen* (im Englischen *directed acyclic graph* oder kurz *Dag*), weil sie keine Zyklen mit mindestens einer Kante enthalten. Denn andernfalls hätte man eine Schlinge oder (fast die gleiche Argumentation wie eben im Beweis 17.4) des Lemmas verschiedene Elemente x und y mit $x \sqsubseteq y$ und $y \sqsubseteq x$.

Gerichtete azyklische Graphen tauchen an vielen Stellen in der Informatik auf, nicht nru natürlich bei Problemstellungen im Zusammenhang mit Graphen, sondern z. B. auch bei der Darstellung von Datenflüssen, im Compilerbau, bei sogenannten *binary decision diagrams* zur Darstellung logischer Funktionen usw.

17.3.2 „Extreme“ Elemente

Es sei $(M, \sqsubseteq)$ eine halbgeordnete Menge und T eine beliebige Teilmenge von M.

minimales Element
maximales Element

Ein Element $x \in T$ heißt *minimales Element* von T, wenn es kein $y \in T$ gibt mit $y \sqsubseteq x$ und $y \neq x$. Ein Element $x \in T$ heißt *maximales Element* von T, wenn es kein $y \in T$ gibt mit $x \sqsubseteq y$ und $x \neq y$.

größtes Element

Ein Element $x \in T$ heißt *größtes Element* von T, wenn für alle $y \in T$ gilt: $y \sqsubseteq x$.

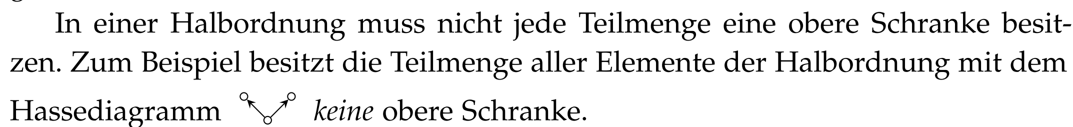
Ein Element $x \in T$ heißt *kleinstes Element* von T, wenn für alle $y \in T$ gilt: $x \sqsubseteq y$.

kleinstes Element

Eine Teilmenge T kann mehrere minimale (bzw. maximale) Elemente besitzen, aber nur ein kleinstes (bzw. größtes). Als Beispiel betrachte man die Teilmenge $T \subseteq 2^{\{a,b,c\}}$ aller Teilmengen von $\{a, b, c\}$, die nichtleer sind. Diese Teilmenge besitzt die drei minimalen Elemente $\{a\}$, $\{b\}$ und $\{c\}$. Und sie besitzt ein größtes Element, nämlich $\{a, b, c\}$.

Ein Element $x \in M$ heißt *obere Schranke* von T, wenn für alle $y \in T$ gilt: $y \sqsubseteq x$. Ein $x \in M$ heißt *untere Schranke* von T, wenn für alle $y \in T$ gilt: $x \sqsubseteq y$. In der Halbordnung $(2^{\{a,b,c\}}, \subseteq)$ besitzt zum Beispiel $T = \{\{\}, \{a\}, \{b\}\}$ zwei obere Schranken: $\{a, b\}$ und $\{a, b, c\}$. Die Teilmenge $T = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$ besitzt die gleichen oberen Schranken.

obere Schranke
untere Schranke

In einer Halbordnung muss nicht jede Teilmenge eine obere Schranke besitzen. Zum Beispiel besitzt die Teilmenge aller Elemente der Halbordnung mit dem Hassediagramm  *keine* obere Schranke.

Besitzt die Menge der oberen Schranken einer Teilmenge T ein kleinstes Element, so heißt dies das *Supremum* von T und wir schreiben dafür $\bigsqcup T$ (oder $\sup(T)$).

Supremum
$\bigsqcup T$, $\sup(T)$

Besitzt die Menge der unteren Schranken einer Teilmenge T ein größtes Element, so heißt dies das *Infimum* von T. Das werden wir in dieser Vorlesung aber nicht benötigen.

Infimum

Wenn eine Teilmenge kein Supremum besitzt, dann kann das daran liegen, dass sie gar keine oberen Schranken besitzt, oder daran, dass die Menge der oberen Schranken kein kleinstes Element hat. Liegt eine Halbordnung der Form $(2^M, \subseteq)$, dann besitzt aber jede Teilmenge $T \subseteq 2^M$ ein Supremum. $\bigsqcup T$ ist dann nämlich die Vereinigung aller Elemente von T (die Teilmengen von M sind).

17.3.3 Vollständige Halbordnungen

Eine *aufsteigende Kette* ist eine abzählbar unendliche Folge $(x_0, x_1, x_2, \dots)$ von Elementen einer Halbordnung mit der Eigenschaft: $\forall i \in \mathbb{N}_0 : x_i \sqsubseteq x_{i+1}$.

aufsteigende Kette

Eine Halbordnung heißt *vollständig*, wenn sie ein kleinstes Element besitzt und jede aufsteigende Kette ein Supremum besitzt. Für das kleinste Element schreiben wir im folgenden $\bot$. Für das das Supremum einer aufsteigenden Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ schreiben wir $\bigsqcup_i x_i$.

vollständige Halbordnung

Ein ganz wichtiges Beispiel für eine vollständige Halbordnung ist die schon mehrfach erwähnte Potenzmenge $2^{M'}$ einer Menge M' mit Mengeninklusion $\subseteq$ als Relation. Das kleinste Element ist die leere Menge $\emptyset$. Und das Supremum einer aufsteigenden Kette $T_0 \subseteq T_1 \subseteq T_2 \subseteq \cdots$ ist $\bigsqcup_i T_i = \bigcup T_i$.

Andererseits ist $(\mathbb{N}_0, \leq)$ *keine* vollständige Halbordung, denn unbeschränkt

wachsende aufsteigende Ketten wie z. B. $0 \leq 1 \leq 2 \leq \cdots$ besitzen kein Supremum in $\mathbb{N}_0$. Wenn man aber noch ein weiteres Element u „über" allen Zahlen hinzufügt, dann ist die Ordnung vollständig. Man setzt also $N = \mathbb{N}_0 \cup \{u\}$ und definiert

$$x \sqsubseteq y \iff (x, y \in \mathbb{N}_0 \land x \leq y) \lor (y = u)$$

Weil wir es später noch brauchen können, definieren wir auch noch $N' = \mathbb{N}_0 \cup \{u_1, u_2\}$ mit der totalen Ordnung

$$x \sqsubseteq y \iff \big(x, y \in \mathbb{N}_0 \land x \leq y\big) \lor \big(x \in \mathbb{N}_0 \cup \{u_1\} \land y = u_1\big) \lor y = u_2$$

also sozusagen

$$0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq 3 \sqsubseteq \cdots \sqsubseteq u_1 \sqsubseteq u_2$$

Ein anderes Beispiel einer (sogar totalen) Ordnung, die *nicht* vollständig ist, werden wir am Ende von Abschnitt 17.4 sehen.

17.3.4 Stetige Abbildungen auf vollständigen Halbordnungen

monotone Abbildung

Es sei $\sqsubseteq$ eine Halbordnung auf einer Menge M. Eine Abbildung $f : M \to M$ heißt *monoton*, wenn für alle $x, y \in M$ gilt: $x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$.

Zum Beispiel ist die Abbildung $f(x) = x + 1$ auf der Halbordnung $(\mathbb{N}_0, \leq)$ monoton. Die Abbildung $f(x) = x \bmod 5$ ist auf der gleichen Halbordnung dagegen nicht monoton, denn es ist zwar $3 \leq 10$, aber $f(3) = 3 \not\leq 0 = f(10)$.

stetige Abbildung

Eine Abbildung $f : D \to D$ auf einer vollständigen Halbordnung $(D, \sqsubseteq)$ heißt *stetig*, wenn für jede aufsteigende Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ gilt: $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$.

Betrachten wir als erstes Beispiel noch einmal die vollständige Halbordnung $N' = \mathbb{N}_0 \cup \{u_1, u_2\}$ von eben. Die Abbildung $f : N' \to N'$ mit

$$f(x) = \begin{cases} x+1 & \text{falls } x \in \mathbb{N}_0 \\ u_1 & \text{falls } x = u_1 \\ u_2 & \text{falls } x = u_2 \end{cases}$$

ist stetig. Denn für jede aufsteigende Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ gibt es nur zwei Möglichkeiten:

- Die Kette wird konstant. Es gibt also ein $n' \in N'$ und ein $i \in \mathbb{N}_0$ so dass gilt: $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_i = x_{i+1} = x_{i+2} = \cdots = n'$. Dann ist jedenfalls $\bigsqcup_i x_i = n'$. Es gibt nun drei Unterfälle zu betrachten:
  - Wenn $n' = u_2$ ist, dann ist wegen $f(u_2) = u_2$ ist auch $\bigsqcup_i f(x_i) = u_2$, also ist $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$.

- Wenn $n' = u_1$, gilt eine analoge Überlegung.
- Wenn $n' \in \mathbb{N}_0$ ist, dann ist $f(\bigsqcup_i x_i) = f(n') = n' + 1$. Andererseits ist die Kette der Funktionswerte $f(x_0) \sqsubseteq f(x_1) \sqsubseteq f(x_2) \sqsubseteq \cdots \sqsubseteq f(x_i) = f(x_{i+1}) = f(x_{i+2}) = \cdots = f(n') = n' + 1$. Also ist $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$.

- Der einzige andere Fall ist: die Kette wird nicht konstant. Dann müssen alle $x_i \in \mathbb{N}_0$ sein, und die Kette wächst unbeschränkt. Das gleiche gilt dann auch für die Kette der Funktionswerte. Also haben beide als Supremum u_1 und wegen $f(u_1) = u_1$ ist $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$.

Der letzte Fall zeigt einem auch gleich schon, dass dagegen die folgende Funktion $g : N' \to N'$ *nicht* stetig ist:

$$g(x) = \begin{cases} x+1 & \text{falls } x \in \mathbb{N}_0 \\ u_2 & \text{falls } x = u_1 \\ u_2 & \text{falls } x = u_2 \end{cases}$$

Der einzige Unterschied zu f ist, dass nun $g(u_1) = u_2$. Eine unbeschränkt wachsende Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ nichtnegativer ganzer Zahlen hat Supremem u_1, so dass $g(\bigsqcup_i x_i) = u_2$ ist. Aber die Kette der Funktionswerte $g(x_0) \sqsubseteq g(x_1) \sqsubseteq g(x_2) \sqsubseteq \cdots$ hat Supremem $\bigsqcup_i g(x_i) = u_1 \neq g(\bigsqcup_i x_i)$.

Der folgende Satz ist eine abgeschwächte Version des sogenannten Fixpunktsatzes von Knaster und Tarski.

17.5 Satz. *Es sei $f : D \to D$ eine monotone und stetige Abbildung auf einer vollständigen Halbordnung $(D, \sqsubseteq)$ mit kleinstem Element $\bot$. Elemente $x_i \in D$ seien wie folgt definiert:*

$$x_0 = \bot$$
$$\forall i \in \mathbb{N}_0 : x_{i+1} = f(x_i)$$

Dann gilt:

1. *Die x_i bilden eine Kette: $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$.*
2. *Das Supremum $x_f = \bigsqcup_i x_i$ dieser Kette ist Fixpunkt von f, also $f(x_f) = x_f$.*
3. *x_f ist der kleinste Fixpunkt von f: Wenn $f(y_f) = y_f$ ist, dann ist $x_f \sqsubseteq y_f$.*

17.6 Beweis. Mit den Bezeichnungen wie im Satz gilt:

1. Dass für alle $i \in \mathbb{N}_0$ gilt $x_i \sqsubseteq x_{i+1}$, sieht man durch vollständige Induktion: $x_0 \sqsubseteq x_1$ gilt, weil $x_0 = \bot$ das kleinste Element der Halbordnung ist. Und wenn man schon weiß, dass $x_i \sqsubseteq x_{i+1}$ ist, dann folgt wegen der Monotonie von f sofort $f(x_i) \sqsubseteq f(x_{i+1})$, also $x_{i+1} \sqsubseteq x_{i+2}$.

2. Wegen der Stetigkeit von f ist $f(x_f) = f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i) = \bigsqcup_i x_{i+1}$. Die Folge der x_{i+1} unterscheidet sich von der Folge der x_i nur durch das fehlende erste Element $\perp$. Also haben „natürlich" beide Folgen das gleiche Supremum x_f; also ist $f(x_f) = x_f$.

   Falls Sie das nicht ganz „natürlich" fanden, hier eine ganz genaue Begründung:

   - Einerseits ist für alle $i \geq 1$: $x_i \sqsubseteq \bigsqcup_i x_{i+1}$. Außerdem ist $\perp = x_0 \sqsubseteq \bigsqcup_i x_{i+1}$. Also ist $\bigsqcup_i x_{i+1}$ eine obere Schranke für alle x_i, $i \in \mathbb{N}_0$, also ist $\bigsqcup_i x_i \sqsubseteq \bigsqcup_i x_{i+1}$.
   - Andererseits ist für alle $i \geq 1$: $x_i \sqsubseteq \bigsqcup_i x_i$. Also ist $\bigsqcup_i x_i$ eine obere Schranke für alle x_{i+1}, $i \in \mathbb{N}_0$, also ist $\bigsqcup_i x_{i+1} \sqsubseteq \bigsqcup_i x_i$.
   - Aus $\bigsqcup_i x_i \sqsubseteq \bigsqcup_i x_{i+1}$ und $\bigsqcup_i x_{i+1} \sqsubseteq \bigsqcup_i x_i$ folgt mit der Antisymmetrie von $\sqsubseteq$ sofort die Gleichheit der beiden Ausdrücke.

3. Durch Induktion sieht man zunächst einmal: $\forall i \in \mathbb{N}_0 : x_i \sqsubseteq y_f$. Denn $x_0 \sqsubseteq y_f$ gilt, weil $x_0 = \perp$ das kleinste Element der Halbordnung ist. Und wenn man schon weiß, dass $x_i \sqsubseteq y_f$ ist, dann folgt wegen der Monotonie von f sofort $f(x_i) \sqsubseteq f(y_f)$, also $x_{i+1} \sqsubseteq y_f$. Also ist y_f eine obere Schranke der Kette, also ist gilt für die kleinste obere Schranke: $x_f = \bigsqcup_i x_i \sqsubseteq y_f$.

■

Dieser Fixpunktsatz (und ähnliche) finden in der Informatik an mehreren Stellen Anwendung. Zum Beispiel kann er Ihnen in Vorlesungen über Sematik von Programmiersprachen wieder begegnen.

17.4 ORDNUNGEN

Ordnung
totale Ordnung

Eine Relation $R \subseteq M \times M$ ist eine *Ordnung*, oder auch genauer *totale Ordnung*, wenn R Halbordnung ist und außerdem gilt:

$$\forall x, y \in M : xRy \lor yRx$$

Wie kann man aus der weiter vorne definierten Halbordnung $\sqsubseteq_p$ auf A^* eine totale Ordnung machen? Dafür gibt es natürlich verschiedene Möglichkeiten. Auf jeden Fall muss aber z. B. festgelegt werden, ob `a` $\sqsubseteq$ `b` oder `b` $\sqsubseteq$ `a`.

Es ist also auf jeden Fall eine totale Ordnung $\sqsubseteq_A$ auf den Symbolen des Alphabetes erforderlich. Nehmen wir an, wir haben das: also z. B. `a` $\sqsubseteq_A$ `b`.

lexikographische Ordnung

Dann betrachtet man des öfteren zwei sogenannte *lexikographische Ordnungen*. Die eine ist die naheliegende Verallgemeinerung dessen, was man aus Wörterbüchern kennt. Die andere ist für algorithmische Zwecke besser geeignet.

- Die lexikographische Ordnung $\sqsubseteq_1$, nach der Wörter im Lexika usw. sortiert sind, kann man wie folgt definieren. Seien $w_1, w_2 \in A^*$. Dann gibt es das eindeutig bestimmte maximal lange gemeinsame Präfix von w_1 und w_2, also das maximal lange Wort $v \in A^*$, so dass es $u_1, u_2 \in A^*$ gibt mit $w_1 = v\,u_1$ und $w_2 = v\,u_2$. Drei Fälle sind möglich:
  1. Falls $v = w_1$ ist, gilt $w_1 \sqsubseteq_1 w_2$.
  2. Falls $v = w_2$ ist, gilt $w_2 \sqsubseteq_1 w_1$.
  3. Falls $w_1 \neq v \neq w_2$, gibt es $x, y \in A$ und $u'_1, u'_2 \in A^*$ mit
     - $x \neq y$ und
     - $w_1 = v\,x\,u'_1$ und $w_2 = v\,y\,u'_2$.

     Dann gilt $w_1 \sqsubseteq_1 w_2 \iff x \sqsubseteq_A y$.

  Muss man wie bei einem Wörterbuch nur endlich viele Wörter ordnen, dann ergibt sich zum Beispiel

$$\begin{aligned} \mathtt{a} &\sqsubseteq_1 \mathtt{aa} \sqsubseteq_1 \mathtt{aaa} \sqsubseteq_1 \mathtt{aaaa} \\ &\sqsubseteq_1 \mathtt{ab} \sqsubseteq_1 \mathtt{aba} \sqsubseteq_1 \mathtt{abbb} \\ \sqsubseteq_1 \mathtt{b} &\sqsubseteq_1 \mathtt{baaaaaa} \sqsubseteq_1 \mathtt{baab} \\ &\sqsubseteq_1 \mathtt{bbbbb} \end{aligned}$$

  Allgemein auf der Menge aller Wörter ist diese Ordnung aber nicht ganz so „harmlos“. Wir gehen gleich noch darauf ein.
- Eine andere lexikographische Ordnung $\sqsubseteq_2$ auf A^* kann man definieren vermöge der Festlegungen: $w_1 \sqsubseteq_2 w_2$ gilt genau dann, wenn
  - entweder $|w_1| < |w_2|$
  - oder $|w_1| = |w_2|$ und $w_1 \sqsubseteq_1 w_2$ gilt.

  Diese Ordnung beginnt also z. B. im Falle $A = \{\mathtt{a}, \mathtt{b}\}$ mit der naheliegenden Ordnung $\sqsubseteq_A$ so:

$$\begin{aligned} \varepsilon &\sqsubseteq_2 \mathtt{a} \sqsubseteq_2 \mathtt{b} \\ &\sqsubseteq_2 \mathtt{aa} \sqsubseteq_2 \mathtt{ab} \sqsubseteq_2 \mathtt{ba} \sqsubseteq_2 \mathtt{bb} \\ &\sqsubseteq_2 \mathtt{aaa} \sqsubseteq_2 \cdots \sqsubseteq_2 \mathtt{bbb} \\ &\sqsubseteq_2 \mathtt{aaaa} \quad \sqsubseteq_2 \cdots \sqsubseteq_2 \mathtt{bbbb} \\ &\cdots \end{aligned}$$

Wir wollen noch darauf hinweisen, dass die lexikographische Ordnung $\sqsubseteq_1$ als Relation auf der Menge aller Wörter einige Eigenschaften hat, an die man als Anfänger vermutlich nicht gewöhnt ist. Zunächst einmal merkt man, dass die Ordnung nicht vollständig ist. Die aufsteigende Kette

$$\varepsilon \sqsubseteq_1 \mathtt{a} \sqsubseteq_1 \mathtt{aa} \sqsubseteq_1 \mathtt{aaa} \sqsubseteq_1 \mathtt{aaaa} \sqsubseteq_1 \cdots$$

besitzt kein Supremum. Zwar ist jedes Wort, das mindestens ein b enthält, obere Schranke, aber es gibt keine kleinste. Das merkt man, wenn man die absteigende Kette

$$\mathtt{b} \sqsupseteq_1 \mathtt{ab} \sqsupseteq_1 \mathtt{aab} \sqsupseteq_1 \mathtt{aaab} \sqsupseteq_1 \mathtt{aaaab} \sqsupseteq_1 \cdots$$

betrachtet. Jede obere Schranke der aufsteigenden Kette muss ein b enthalten. Aber gleich, welche obere Schranke w man betrachtet, das Wort $\mathtt{a}^{|w|}\mathtt{b}$ ist eine echt kleine obere Schranke. Also gibt es keine kleinste.

Dass es sich bei obigen Relationen überhaupt um totale Ordnungen handelt, ist auch unterschiedlich schwer zu sehen. Als erstes sollte man sich klar machen, dass $\sqsubseteq_1$ auf der Menge A^n aller Wörter einer festen Länge n eine totale Ordnung ist. Das liegt daran, dass für verschiedene Wörter gleicher Länge niemals Punkt 1 oder Punkt 2 zutrifft. Und da $\sqsubseteq_A$ als totale Ordnung vorausgesetzt wird, ist in Punkt 3 stets $x \sqsubseteq_A y$ oder $y \sqsubseteq_A x$ und folglich $w_1 \sqsubseteq_1 w_2$ oder $w_2 \sqsubseteq_1 w_1$.

Daraus folgt schon einmal das auch $\sqsubseteq_2$ auf der Menge A^n aller Wörter einer festen Länge n eine totale Ordnung ist, und damit überhaupt eine totale Ordnung.

Für $\sqsubseteq_1$ muss man dafür noch einmal genauer Wörter unterschiedlicher Länge in Betracht ziehen. Wie bei der Formulierung der Definition schon suggeriert, decken die drei Punkte alle Möglichkeiten ab.

17.5 AUSBLICK

Vollständige Halbordnungen spielen zum Beispiel eine wichtige Rolle, wenn man sich mit sogenannter denotationaler Semantik von Programmiersprachen beschäftigt und die Bedeutung von while-Schleifen und Programmen mit rekursiven Funktionsaufrufen präzisieren will. Den erwähnten Fixpunktsatz (oder verwandte Ergebnisse) kann man auch zum Beispiel bei der automatischen statischen Datenflussanalyse von Programmen ausnutzen. Diese und andere Anwendungen werden ihnen in weiteren Vorlesungen begegnen.

INDEX

$f \asymp g$, 122
Äquivalenzklasse, 179
Äquivalenzrelation, 98, 177
Äquivalenzrelation von Nerode, 178
Übersetzung, 78
äquivalente Aussagen, 14

A^n, 19
A^*, 18
Abbildung, 13
    bijektiv, 13
    injektiv, 13
    monotone, 188
    partielle, 13
    stetige, 188
    surjektiv, 13
    wohldefinierte, 182
abgelehntes Wort, 143
ablehnender Zustand, 142
Ableitung, 60
Ableitungsbaum, 63
Ableitungsfolge, 61
Ableitungsschritt, 60
Addition
    von Matrizen, 108
adjazente Knoten, 90, 95
Adjazenzliste, 103
Adjazenzmatrix, 104
akzeptierender Zustand
    bei endlichen Automaten, 142
    bei Turingmaschinen, 163
akzeptierte formale Sprache
    bei Turingmaschinen, 163
akzeptierte formale Sprache
    bei endlichen Automaten, 143
akzeptiertes Wort
    bei Turingmaschinen, 163
    bei endlichen Automaten, 143
Akzeptor, endlicher, 142
al-Khwarizmi, Muhammad ibn Musa, 35
Algebra, 35
Algorithmus, 36, 157
    informell, 37
Algorithmus von Warshall, 115
Allquantor, 15
Alphabet, 10
Anfangskonfiguration, 163
antisymmetrische Relation, 183
assoziative Operation, 27
asymptotisches Wachstum, 122
aufschreiben
    früh, 49
aufsteigende Kette, 187
aufzählbare Sprache, 163
Ausgabefunktion, 138, 141
Ausgangsgrad, 93
Aussagen
    äquivalente, 14
aussagenlogische Formel, 14
Auszeichnungssprache, 50
Automat
    endlicher, 138
    Mealy-, 138
    Moore-, 140
average case, 120

Bachelorarbeit, 49
Baum
    gerichtet, 92
    Höhe, 153
    Kantorowitsch-Baum, 152
    ungerichtet, 97
Berechnung

endliche, 162
haltend, 162
nicht haltend, 162
unendliche, 162
Biber
fleißiger, 173
Bibermaschine, 173
bijektive Abbildung, 13
Binärdarstellung, 74
binäre Relation, 12
Bit, 67
Blanksymbol, 159
Block-Codierung, 87
Busy-Beaver-Funktion, 173
Byte, 67

$c_0(w)$, 163
Champollion, Jean-Francois, 9
Code, 79
präfixfreier, 79
Codewort, 79
Codierung, 79
computational complexity, 119
Coppersmith, Don, 131

Dag, 186
Datum, 8
Definitionsbereich, 13
Diagonalisierung, 170
Dokument, 49

ε-freier Konkatenationsabschluss, 32
E-Mail, 23
einfacher Zyklus, 92
Eingabealphabet, 162
Eingangsgrad, 93
Endkonfiguration, 160
endliche Berechnung, 162
endlicher Akzeptor, 142
endlicher Automat, 138
entscheidbare Sprache, 163
Entscheidungsproblem, 162
erreichbarer Knoten, 92
Erscheinungsbild, 49
erzeugte formale Sprache, 61
Existenzquantor, 15

f^*
bei Mealy-Automaten, 139
bei Moore-Automaten, 141
f^{**}
bei Mealy-Automaten, 140
bei Moore-Automaten, 141
Faktormenge, 179
Faserung, 179
fleißiger Biber, 173
Form, 49
formale Sprache, 29
akzeptierte , bei Turingmaschinen, 163
akzeptierte , bei endlichen Automaten, 143
erzeugte, 61
kontextfrei, 61
Potenzen, 30
Formel
aussagenlogische, 14
früh aufschreiben, 49
Funktion, 13
bijektiv, 13
injektiv, 13
partielle, 13
surjektiv, 13

g^*
bei Mealy-Automaten, 140
bei Moore-Automaten, 141
g^{**}
bei Mealy-Automaten, 140

 © worsch 2008–2010

bei Moore-Automaten, 142
gerichteter azyklischer Graph, 186
gerichteter Graph, 89
geschlossener Pfad, 92
gewichteter Graph, 99
größenordnungsmäßiges Wachstum, 122
größtes Element, 186
Grad, 93, 97
Graham, Ronald, 127
Grammatik
kontextfreie, 60
rechtslinear, 151
Typ-2, 152
Typ-3, 152
Graph
gerichtet, 89
gerichteter azyklischer, 186
gewichtet, 99
isomorphe, 93, 96
kantenmarkiert, 98
knotenmarkiert, 98
schlingenfrei, 91, 95
streng zusammenhängend, 92
ungerichtet, 95
Graphisomorphismus, 93
Groß-O-Notation, 120

Höhe eines Baumes, 153
Hülle
reflexiv-transitive, 65
h^{**}, 79
halbgeordnete Menge, 183
Halbordnung, 183
vollständige, 187
Halten
einer Turingmaschine, 160
haltende Berechnung, 162
Halteproblem, 171
Hasse-Diagramm, 185
Hexadezimaldarstellung, 75
Homomorphismus, 79
ε-freier, 79
Huffman-Codierung, 82

I_M, 64
IETF, 67
Induktion
strukturelle, 155
vollständige, 25–27
induzierte Operation, 182
Infimum, 187
Infixschreibweise, 60
Informatik, 5
Information, 7
Inhalt, 49
injektive Abbildung, 13
Inschrift, 6
Internet Engineering Task Force, 67
Inzidenzlisten, 103
isomorphe Graphen, 93, 96
Isomorphismus
von Graphen, 93

kantenmarkierter Graph, 98
Kantenrelation, 96
Kantorowitsch-Baum, 152
kartesisches Produkt, 12
Klammerausdruck
korrekter, 63
wohlgeformter, 63
Kleene, Stephen Cole, 116, 147
kleinstes Element, 187
Knoten
adjazente, 90, 95
erreichbarer, 92
knotenmarkierter Graph, 98
Knuth, Donald, 50, 127
kommutative Operation, 27

Komplexitätsklasse, 167
Komplexitätsmaß, 119, 166
Komplexoperationen, 127
Konfiguration, 159
Kongruenz modulo n, 177
Kongruenzrelation, 181
Konkatenation
 von Wörtern, 20
Konkatenationsabschluss, 32
 ε-freier, 32
kontextfreie Grammatik, 60
kontextfreie Sprache, 61
korrekter Klammerausdruck, 63
Kryptographie, 157

Länge eines Pfades, 92
Länge eines Weges, 96
Länge eines Wortes, 17
LaTeX, 50
leeres Wort, 18
lexikographische Ordnung, 190
Linksableitung, 63
linkseindeutige Relation, 13
linkstotale Relation, 12

markup language, 50
Masterarbeit, 49
Mastertheorem, 132
Matrixaddition, 108
Matrixmultiplikation, 107
maximales Element, 186
Mealy-Automat, 138
Menge aller Wörter, 18
minimales Element, 186
modulo n, 177
monotone Abbildung, 188
Moore-Automat, 140
Morphogenese, 157

Nachricht, 7
Nerode
 Äquivalenzrelation, 178
nicht haltende Berechnung, 162
nichtdeterministische Automaten, 146
Nichtterminalsymbol, 60
Nullmatrix, 108

$\mathrm{O}(f)$, 125
$\mathrm{O}(1)$, 125
$\Omega(f)$, 125
obere Schranke, 187
Octet, 67
Operation
 assoziative, 27
 induzierte, 182
 kommutative, 27
Ordnung, 190
 lexikographische, 190
 total, 190

P, 168
partielle Abbildung, 13
Patashnik, Oren, 127
Pfad, 92
 geschlossener, 92
 wiederholungsfrei, 92
Pfadlänge, 92
Platzkomplexität, 167
polynomiell
 Raumkomplexität, 167
 Zeitkomplexität, 166
Potenzen
 einer formalen Sprache, 30
 einer Relation, 64
Potenzen eines Wortes, 24
präfixfreier Code, 79
Produkt
 formaler Sprachen, 30
 kartesisches, 12

Produkt der Relationen, 64
Produktion, 60
PSPACE, 168

Quantor, 15

$\langle R \rangle$, 149
Radó, Tibor, 173
Rado-Funktion, 173
Rat fürs Studium, 49
Raumkomplexität, 167
    polynomiell, 167
Rechenzeit, 119
rechtseindeutige Relation, 12
rechtslineare Grammatik, 151
rechtstotale Relation, 13
reflexiv-transitive Hülle, 65
reflexive Relation, 65
reguläre Sprache, 150
regulärer Ausdruck, 147
Relation, 12
    antisymmetrisch, 183
    binäre, 12
    linkseindeutig, 13
    linkstotal, 12
    Potenz, 64
    Produkt, 64
    rechtseindeutig, 12
    rechtstotal, 13
    reflexiv-transitive Hülle, 65
    reflexive, 65
    symmetrisch, 97
    transitive, 65
Reques for Comments, 67
Ressource, 119
RFC, 67
    2822 (E-Mail), 23
    3629, 81
    E-Mail (2822), 23
Schleifeninvariante, 45
Schlinge, 91, 95
schlingenfreier Graph, 91, 95
Schranke
    obere, 187
    untere, 187
Schritt einer Turingmaschine, 160
Semianrarbeit, 49
Signal, 6
Signum-Funktion, 110
Speicher, 67
Speicherplatzbedarf, 119
Speicherung, 6
Sprache
    aufzählbare, 163
    entscheidbare, 163
    formale, 29
    Produkt, 30
    reguläre, 150
Startsymbol, 60
stetige Abbildung, 188
Strassen, Volker, 130
streng zusammenhängender Graph, 92
Struktur, 49
strukturelle Induktion, 155
Studium
    ein guter Rat, 49
Supremum, 187
surjektive Abbildung, 13
symmetrische Relation, 97

Tantau, Till, 49
Teilgraph, 91, 95
Terminalsymbol, 60
$\Theta(f)$, 125
totale Ordnung, 190
transitive Relation, 65
Turing, Alan, 157
Turingmaschine, 158

Akzeptor, 163
Halten, 160
Schritt, 160
universelle, 170
Twain, Mark, 1
Typ-2-Grammatiken, 64, 152
Typ-3-Grammatiken, 152

unendliche Berechnung, 162
ungerichteter Baum, 97
ungerichteter Graph, 95
universelle Turingmaschine, 170
untere Schranke, 187
UTF-8, 81

Verträglichkeit, 181
vollständige Halbordnung, 187
vollständige Induktion, 25–27

Wachstum
asymptotisches, 122
größenordnungsmäßig, 122
größenordnungsmäßig gleich, 122
Warshall-Algorithmus, 115
Watson, Thomas, 1
Weg, 96
Wegematrix, 105
Weglänge, 96
wiederholungsfreier Pfad, 92
Winograd, Shmuel, 131
wohldefinierte Abbildung, 182
Wohldefiniertheit, 80
wohlgeformter Klammerausdruck, 63
worst case, 120
Wort, 17
abgelehntes, 143
akzeptiertes bei endlichen Automaten, 143
akzeptiertes, bei Turingmaschinen, 163
Länge, 17
leeres, 18
Potenzen, 24
Wurzel, 92

Zeitkomplexität, 166
polynomiell, 166
Zellularautomaten, 157
Zielbereich, 13
zusammenhängender ungerichteter Graph, 96
Zusicherung, 38
Zustand
ablehnender, 142
akzeptierender, bei endlichen Automaten, 142
akzeptierender, bei Turingmaschinen, 163
Zustandsüberführungsfunktion, 138, 140
Zyklus, 92
einfacher, 92

LITERATUR

Abeck, Sebastian (2005). *Kursbuch Informatik, Band 1*. Universitätsverlag Karlsruhe.

Coppersmith, Don und Shmuel Winograd (1990). "Matrix Multiplication via Arithmetic Progressions". In: *Journal of Symbolic Computation* 9, S. 251–280.

Friedl, Jeffrey (2006). *Mastering Regular Expressions*. 3rd edition. O'Reilly Media, Inc.

Goos, Gerhard (2006). *Vorlesungen über Informatik: Band 1: Grundlagen und funktionales Programmieren*. Springer-Verlag.

Graham, Ronald L., Donald E. Knuth und Oren Patashnik (1989). *Concrete Mathematics*. Addison-Wesley.

Kleene, Stephen C. (1956). "Representation of Events in Nerve Nets and Finite Automata". In: *Automata Studies*. Hrsg. von Claude E. Shannon und John McCarthy. Princeton University Press. Kap. 1, S. 3–40.
Eine Vorversion ist online verfügbar; siehe `http://www.rand.org/pubs/research_memoranda/2008/RM704.pdf` (8.12.08).

Spitzer, Manfred (2002). *Lernen: Gehirnforschung und Schule des Lebens*. Spektrum Akademischer Verlag.

Strassen, Volker (1969). "Gaussian Elimination Is Not Optimal". In: *Numerische Mathematik* 14, S. 354–356.

Turing, A. M. (1936). "On computable numbers, with an application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society*. 2. Ser. 42, S. 230–265.
Die Pdf-Version einer HTML-Version ist online verfügbar; siehe `http://web.comlab.ox.ac.uk/oucl/research/areas/ieg/e-library/sources/tp2-ie.pdf` (15.1.10).

Warshall, Stephen (1962). "A Theorem on Boolean Matrices". In: *Journal of the ACM* 9, S. 11–12.

COLOPHON

These lecture notes were prepared using *GNU Emacs*
(http://www.gnu.org/software/emacs/) and in particular the *AucTeX* package
(http://www.gnu.org/software/auctex/).
The notes are typeset with LaTeX 2_ε (more precisely pdfelatex) with Peter
Wilson's memoir document class using Hermann Zapf's *Palatino* and *Euler* type
faces. The layout was inspired by Robert Bringhurst's book *The Elements of
Typographic Style*.