

GRUNDBEGRIFFE DER INFORMATIK

THOMAS WORSCH

2008/2009

INHALTSVERZEICHNIS

1	Prolog	1
1.1	Aufbau der Vorlesung und Ziele	1
1.2	Quellen	2
2	Signale, Nachrichten, Informationen, Daten	4
2.1	Signal	4
2.2	Übertragung und Speicherung	5
2.3	Nachricht	5
2.4	Information	6
2.5	Datum	6
2.6	Zusammenfassung	7
3	Alphabete	8
3.1	Alphabete	8
3.2	Relationen und Abbildungen	11
3.3	Logisches	12
3.4	Zusammenfassung und Ausblick	14
4	Wörter	16
4.1	Wörter	16
4.2	Das leere Wort	17
4.3	Mehr zu Wörtern	18
4.4	Konkatenation von Wörtern	19
4.5	Binäre Operationen	24
4.6	Zusammenfassung	24
5	Der Begriff des Algorithmus	25
5.1	Lösen einer Sorte quadratischer Gleichungen	26
5.2	Zum informellen Algorithmusbegriff	26
5.3	Zur Korrektheit des Algorithmus zur Lösung einer Sorte quadratischer Gleichungen	27
5.4	Wie geht es weiter?	28
5.5	Ein Algorithmus zur Multiplikation nichtnegativer ganzer Zahlen	29
5.6	Der Algorithmus zur Multiplikation nichtnegativer ganzer Zahlen mit einer Schleife	32
6	Formale Sprachen	36
6.1	Formale Sprachen	36
6.2	Operationen auf formalen Sprachen	36
6.3	Zusammenfassung	39
7	Dokumente	40
7.1	Dokumente	40

7.2	Struktur von Dokumenten	41
7.3	Zusammenfassung	44
8	Kontextfreie Grammatiken	45
8.1	Rekursive Definition syntaktischer Strukturen	45
8.2	Kontextfreie Grammatiken	49
8.3	Relationen (Teil 2)	53
8.4	Ein Nachtrag zu Wörtern	54
8.5	Ausblick	55
9	Speicher	56
9.1	Bit und Byte	56
9.2	Speicher als Tabellen und Abbildungen	57
9.3	Binäre und dezimale Größenpräfixe	59
9.4	Ausblick	60
10	Übersetzungen und Codierungen	61
10.1	Von Wörtern zu Zahlen und zurück	61
10.2	Von einem Alphabet zum anderen	65
10.3	Huffman-Codierung	68
10.4	Ausblick	73
11	Graphen	74
11.1	Gerichtete Graphen	74
11.2	Ungerichtete Graphen	79
11.3	Graphen mit Knoten- oder Kantenmarkierungen	82
12	Erste Algorithmen in Graphen	84
12.1	Repräsentation von Graphen im Rechner	84
12.2	Berechnung der Erreichbarkeitsrelation	86
12.3	Algorithmus von Warshall	92
12.4	Ausblick	94
13	Quantitative Aspekte von Algorithmen	96
13.1	Ressourcenverbrauch bei Berechnungen	96
13.2	Groß-O-Notation	97
13.3	Matrixmultiplikation	103
13.4	Abschätzung des Wachstums rekursiv definierter Funktionen	106
13.5	Ausblick	107
14	Endliche Automaten	108
14.1	Erstes Beispiel: ein Getränkeautomat	108
14.2	Mealy-Automaten	111
14.3	Moore-Automaten	113
14.4	Endliche Akzeptoren	114
14.5	Ausblick	117

15	Reguläre Ausdrücke und rechtslineare Grammatiken	118
15.1	Reguläre Ausdrücke	118
15.2	Rechtslineare Grammatiken	122
15.3	Ausblick	123
16	Turingmaschinen	124
16.1	Alan Mathison Turing	124
16.2	Turingmaschinen	124
16.3	Berechnungskomplexität	130
16.4	Schwere Probleme	134
16.5	Unentscheidbare Probleme	135
16.6	Ausblick	140
17	Relationen	141
17.1	Äquivalenzrelationen	141
17.2	Kongruenzrelationen	144
17.3	Halbordnungen	146
17.4	Ordnungen	152
17.5	Ausblick	154
18	Logik	156
18.1	Formeln in Prädikatenlogik erster Stufe	156
18.2	Theorien und Beweisbarkeit	157
18.3	Interpretationen und Modelle für geschlossene Formeln	159
18.4	Beispiele für Modelle für geschlossene Formeln	160
18.5	Grenzen von Prädikatenlogik erster Stufe	161
18.6	Ausblick	162
	Index	163
	Literatur	172

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 zum Wintersemester 2008/2009 einschreiben konnte.

Der vorliegende Text ist ein Vorlesungsskript zu dieser Veranstaltung, wie ich sie in jenem Semester (halten will bzw. inzwischen) gehalten habe.

1.1 AUFBAU DER VORLESUNG UND ZIELE

Für diese Vorlesung stehen 15 Termine zu je 90 Minuten zur Verfügung. Ich habe versucht, den Vorlesungsinhalt auf eine Reihe überschaubarer inhaltlicher Einheiten aufzuteilen. Vermutlich wird ihre Zahl am Ende zwischen 15 und 30 liegen. Man wird sehen ... am Ende waren es 19.

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 *verständlich (!) und präzise* zu formulieren, und
- wie man ordentlich und ethisch einwandfrei andere Quellen benutzt und zitiert.

Ich habe versucht, der Versuchung zu widerstehen, prinzipiell wie in einem Nachschlagewerk 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 Februar 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 demformiert wird. In allen Fällen sind es also physi-



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

kalische Vorgänge, die ablaufen und 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“ bedeuten soll). Die Veränderung einer (oder mehrerer) physikalischer Größen (zum Beispiel Schalldruck) um etwas mitzuteilen nennt man ein *Signal*.

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.

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“.

Inschrift

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 überträgt 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	$\xrightarrow{\text{Programm-}} \\ \text{ausführung}$	59
	Interpretation↓	Interpre↓tation		Interpre↓tation
Mensch:	zweiund- vierzig	siebzehn	$\xrightarrow{\text{Rechnen}} \\ \text{Addition}$	neunund- fünfzig

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

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.

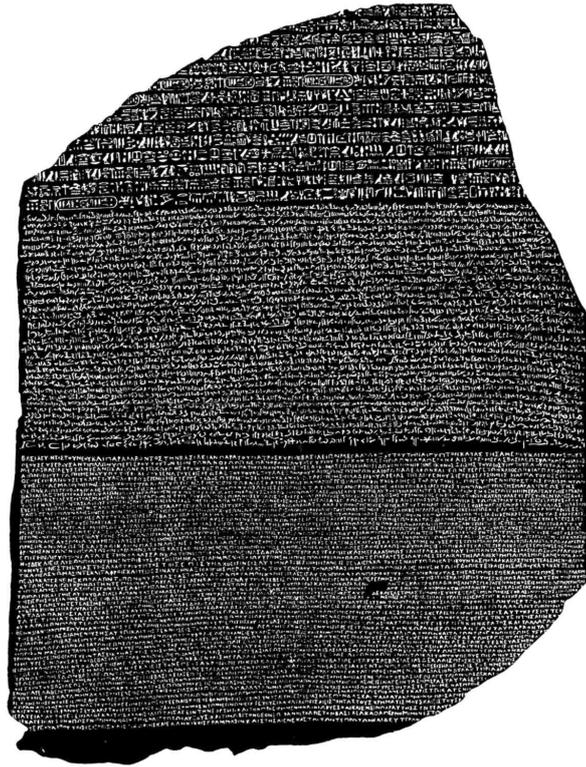


Abbildung 3.1: Der Rosetta-Stein, heute im Britischen Museum, London. Bildquelle: <http://www.gutenberg.org/files/16352/16352-h/images/p1.jpg> (19.10.08)

Wir sind gewohnt, lange Inschriften aus (Kopien der) immer wieder gleichen Zeichen zusammzusetzen. 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

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

Alphabet

seien einfach die elementaren Bausteine, aus denen Inschriften zusammengesetzt sind. Hier sind einfache Beispiele:

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

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 1. Wie man dort sieht, fehlen in diesem Alphabet etliche Buchstaben aus nichtenglischen Alphabeten, wie zum Beispiel ä, ç, è, ğ, ñ, œ, ß, û usw., von Kyrillisch, Japanisch und vielen anderen außereuropäischen Schriften ganz zu schweigen.

Auf ein Zeichen in Tabelle 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 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.

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

Der Unicode-Standard spezifiziert weitaus mehr als nur einen Zeichensatz. Für uns sind hier zunächst nur die beiden folgenden Aspekte wichtig¹:

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

Aber 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 eine Abbildung $f : A_U \rightarrow \mathbb{N}_0$ vor. Sie ist

- eine Abbildung, weil jedem Zeichen nur *eine* Nummer zugewiesen wird,

¹Hinzu 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.

- 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 nicht-negativen 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 $U \subseteq A_U \times \mathbb{N}_0$.

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

kartesches Produkt

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

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 “.

Relation

binäre Relation

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 $(a, n) \in U$.

Allgemein nennt man eine Relation $R \subseteq A \times B$ *linkstotal*, wenn für jedes $a \in A$ ein $b \in B$ existiert mit $(a, b) \in R$.

linkstotal

2. Für kein Zeichen $a \in A_U$ gibt es mehrere $n \in \mathbb{N}_0$ mit der Eigenschaft $(a, n) \in U$.

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.

rechtseindeutig

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 $R : A \rightarrow B$.

Abbildung

Funktion

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

linkseindeutig

$(a_1, b_1) \in R$ und alle $(a_2, b_2) \in R$ gilt:

wenn $a_1 \neq a_2$, 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 $U : A_U \rightarrow \mathbb{N}_0$ ist injektiv.“

Das ist eine Aussage. Sie ist *wahr*.

„Die Abbildung $U : A_U \rightarrow \mathbb{N}_0$ 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 alle 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 \mathcal{A} und \mathcal{B} genannt, 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: A oder B .

Dafür schreiben wir auch kurz $A \vee B$.

LOGISCHE IMPLIKATION: Wenn A , dann B .

Dafür schreiben wir auch kurz $A \Rightarrow B$.

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 A und 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 „inklusive“ (und nicht „exklusiv“): Wenn A und B beide wahr sind, dann auch $A \vee B$.
- 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 $\neg(A \vee B)$ und $(\neg A) \wedge (\neg B)$ 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.

Man kann sich das so noch etwas klarer machen, dass man sich 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.

Ein bisschen *Aussagenlogik* haben wir auch gemacht. Wir werden in einer späteren Einheit darauf zurück kommen, weil sich selbst hier schon schwierige algorithmische Problem verbergen.

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 nicht! Aber wie der Name sagt, fassen wir auch das Leerzeichen als ein Zeichen auf. Damit man es sieht, schreiben wir manchmal explizit \sqcup 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 \sqcup 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 $w : \mathbb{G}_n \rightarrow A$ 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 Unterabschnitt](#) gleich noch zu sprechen.)

*Wort, formalistisch
definiert
Länge eines Wortes*

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

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 ermöglicht 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.

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 $w : \mathbb{G}_n \rightarrow B$ mit $n \in \mathbb{N}_0$ und $B \subseteq A$. Es ist aber völlig in Ordnung, wenn Sie sich einfach Zeichenketten vorstellen.

Menge aller Wörter

A^*

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 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 $w : \{\} \rightarrow \{\}$.

Wichtig:

- Wundern Sie sich nicht, wenn Sie sich über $w : \{\} \rightarrow \{\}$ 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 $w : \{\} \rightarrow \{\}$ 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.

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 $A = \{a, b\}$ ist, dann ist:

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

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

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

Das unschöne daran sind die „ \dots “: 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, \dots (schon wieder die Pünktchen \dots) 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.

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

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:

Konkatenation

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

oder

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

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**.

Nun wollen wir die *Konkatenation zweier Wörter formal* definieren. Nehmen wir also an, es seien zwei Wörter $w_1 : \mathbb{G}_m \rightarrow A_1$ und $w_2 : \mathbb{G}_n \rightarrow A_2$ gegeben. Dann ist

*Konkatenation
zweier Wörter
formal*

$$w_1 \cdot w_2 : \mathbb{G}_{m+n} \rightarrow A_1 \cup A_2$$

$$i \mapsto \begin{cases} w_1(i) & \text{falls } 0 \leq i < m \\ w_2(i-m) & \text{falls } m \leq i < m+n \end{cases}$$

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 $w_1 \cdot w_2 : \mathbb{G}_{m+n} \rightarrow A_1 \cup A_2$ 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 $(w_1 w_2)(m + i_2) = w_2(i_2) = a$.

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

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

4.4.1 Konkatenation mit dem leeren Wort

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

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

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.

4.2 Lemma. Für jedes Alphabet A gilt:

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

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 Herr Gauß gemacht ...), sondern um an einem einfachen Beispiel etwas zu üben, was Sie durch Ihr ganzes Studium begleiten wird: Beweisen.

4.3 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* Annahmen macht und zeigt, dass die Aussage für dieses A gilt.

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 $w \in A^*$, d. h. eine surjektive Abbildung $w : \mathbb{G}_m \rightarrow B$ mit $B \subseteq A$.

Außerdem ist $\varepsilon : \mathbb{G}_0 \rightarrow \{\}$.

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 $w' : \mathbb{G}_{m+0} \rightarrow B \cup \{\}$, also schlicht $w' :$

$\mathbb{G}_m \rightarrow B$. 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 4.5).

4.4.3 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

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

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?

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

*induktive Definition
Potenzen von
Wörtern*

$$\begin{aligned} w^0 &= \varepsilon \\ \forall k \in \mathbb{N}_0 : w^{k+1} &= w^k \cdot w \end{aligned}$$

Man definiert also

- explizit, was ein Exponent 0 bedeuten soll: $w^0 = \varepsilon$. (Es hat sich als nützlich erwiesen, das so festzulegen.)

- 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.

4.4 Lemma. Für jedes Alphabet A , jedes Wort $w \in A^*$ und jedes $n \in \mathbb{N}_0$ gilt:

$$|w^n| = n|w|.$$

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 Fälle 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:

$$\begin{aligned} |w^1| &= |w^{0+1}| = |w^0 \cdot w| \\ &= |w^0| + |w| && \text{siehe Punkt 4.1} \\ &= 0|w| + |w| && \text{siehe Fall } n = 0 \\ &= 1|w| \end{aligned}$$

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:

$$\begin{aligned} |w^2| &= |w^{1+1}| = |w^1 \cdot w| \\ &= |w^1| + |w| && \text{siehe Punkt 4.1} \\ &= 1|w| + |w| && \text{siehe Fall } n = 1 \\ &= 2|w| \end{aligned}$$

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 aus der Richtigkeit der Behauptung für $|w^n|$ die für $|w^{n+1}|$ 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 oben 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

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

INDUKTIONSANFANG $n = 0$: Zu zeigen ist: $|w^0| = 0 \cdot |w|$. Das geht ganz ausführlich aufgeschrieben so:

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

INDUKTIONSSCHRITT $n \rightarrow n + 1$: 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, 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 für ein beliebiges aber festes n die

INDUKTIONSANNAHME ODER INDUKTIONSVORAUSSETZUNG:

$$|w^n| = n|w|.$$

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:

$$\begin{aligned} |w^{n+1}| &= |w^n \cdot w| \\ &= |w^n| + |w| \\ &= n|w| + |w| && \text{nach Induktionsannahme} \\ &= (n + 1)|w| \end{aligned}$$

■

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

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

$$\begin{array}{ll} \text{es gilt} & \mathcal{A}(0) \\ \text{und es gilt} & \forall n \in \mathbb{N}_0 : (\mathcal{A}(n) \Rightarrow \mathcal{A}(n + 1)) \end{array}$$

dann gilt auch:

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

4.5 BINÄRE OPERATIONEN

Unter einer binären Operation auf einer Menge M versteht man eine Abbildung $f : M \times M \rightarrow M$. Üblicherweise benutzt man aber ein „Operationssymbol“ 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 $\diamond : M \times M \rightarrow M$ genau dann *kommutativ*, wenn gilt:

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

*kommutative
Operation*

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

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

*assoziative
Operation*

4.6 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 Definitionen* gesehen und das Beweisprinzip der *vollständigen Induktion*.

5 DER BEGRIFF DES ALGORITHMUS

MUHAMMAD IBN MŪSĀ AL-KHWĀRIZMĪ lebte ungefähr von 780 bis 850. Abbildung 5.1 zeigt eine (relativ beliebte weil copyright-freie) Darstellung auf einer russischen Briefmarke anlässlich seines 1200. Geburtstages (jedenfalls in etwa). Im Jahr 830 (oder we-



Abbildung 5.1: Muhammad ibn Mūsā al-Khwārizmī; Bildquelle: http://commons.wikimedia.org/wiki/Image:Abu_Abdullah_Muhammad_bin_Musa_al-Khwarizmi.jpg (4.11.2008)

nig früher) schrieb al-Khwārizī ein wichtiges Buch mit dem Titel „Al-Kitāb al-mukhtaṣar fī ḥisāb 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 5.2 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 *Algorithmus* angesehen.

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

*Herkunft des Wortes
Algorithmus*

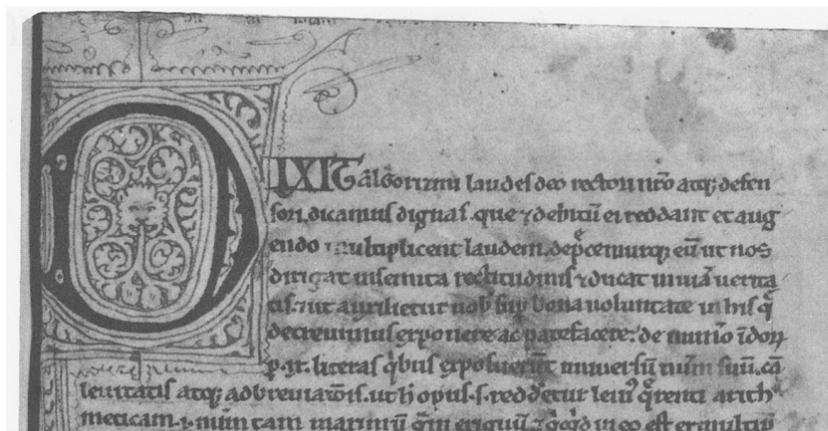


Abbildung 5.2: Ausschnitt einer Seite einer lateinischen Übersetzung der Arbeit von al-Khwārizmī über das „Rechnen mit indischen Ziffern“; Bildquelle: http://en.wikipedia.org/wiki/Image:Dixit_algorizmi.png (27.5.2008)

5.1 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:

$$h \leftarrow b/2 \quad (1)$$

$$q \leftarrow h^2 \quad (2)$$

$$s \leftarrow c + q \quad (3)$$

$$w \leftarrow \sqrt{s} \quad (4)$$

$$x \leftarrow w - h \quad (5)$$

(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 Eingangsgrößen 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.

5.2 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 Algorithmusbegriffes zu betrachten. Dazu gehören zum Beispiel
 - randomisierte Algorithmen, bei denen in manchen Situationen 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, und
 - Verfahren, die nicht terminieren (z. B. Ampelsteuerung).

5.3 ZUR KORREKTHEIT DES ALGORITHMUS ZUR LÖSUNG EINER SORTE QUADRATISCHER GLEICHUNGEN

Al-Khwarizmi gibt einen sehr schönen Beweis dafür an, dass die Rechnungen in (1)-(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>, 3.11.2008).

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:

```
// b > 0 ∧ c > 0
h ← b/2
// h = b/2
q ← h2
// q = b2/4
s ← c + q
// s = c + b2/4
w ← √s
// w = √(c + b2/4)
x ← w - h
// x = √(c + b2/4) - b/2
// x2 + bx = (√(c + b2/4) - b/2)2 + b(√(c + b2/4) - b/2)
// x2 + bx = c + b2/4 - b√(c + b2/4) + b2/4 + b√(c + b2/4) - b2/2
// x2 + bx = c
```

5.4 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 Unterabschnitt 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, aber auch) auf syntaktische Korrektheit
 - Beweise (etwa der Korrektheit von Algorithmen bezüglich spezifizierter Anforderungen).

5.5 EIN ALGORITHMUS ZUR MULTIPLIKATION NICHTNEGATIVER GANZER ZAHLEN

Betrachten wir als erstes den in Abbildung 5.3 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 $x \bmod y$ den Rest der ganzzahligen Division von x durch y . Und die Operation **div** liefere für zwei Argumente x und y als Funktionswert $x \mathbf{div} y$ den Wert der ganzzahligen Division von x durch y . Mit anderen Worten gilt für nichtnegative ganze Zahlen x und y stets:

$$x = y \cdot (x \mathbf{div} y) + (x \mathbf{mod} y) \quad (6)$$

Machen wir eine Beispielrechnung für den Fall $a = 6$ und $b = 9$. Für die in Abbildung 5.3 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.

```

// Eingaben:  $a \in \mathbb{G}_8, b \in \mathbb{N}_0$ 
 $P_0 \leftarrow 0$ 
 $X_0 \leftarrow a$ 
 $Y_0 \leftarrow b$ 
 $x_0 \leftarrow X_0 \bmod 2$ 
// — Algorithmusstelle —  $i = 0$ 
 $P_1 \leftarrow P_0 + x_0 \cdot Y_0$ 
 $X_1 \leftarrow X_0 \text{ div } 2$ 
 $Y_1 \leftarrow 2 \cdot Y_0$ 
 $x_1 \leftarrow X_1 \bmod 2$ 
// — Algorithmusstelle —  $i = 1$ 
 $P_2 \leftarrow P_1 + x_1 \cdot Y_1$ 
 $X_2 \leftarrow X_1 \text{ div } 2$ 
 $Y_2 \leftarrow 2 \cdot Y_1$ 
 $x_2 \leftarrow X_2 \bmod 2$ 
// — Algorithmusstelle —  $i = 2$ 
 $P_3 \leftarrow P_2 + x_2 \cdot Y_2$ 
 $X_3 \leftarrow X_2 \text{ div } 2$ 
 $Y_3 \leftarrow 2 \cdot Y_2$ 
 $x_3 \leftarrow X_3 \bmod 2$ 
// — Algorithmusstelle —  $i = 3$ 

```

Abbildung 5.3: Ein einfacher Algorithmus für die Multiplikation einer kleinen Zahl a mit einer anderen Zahl b .

	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 hoffentlich: 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 führt. Passende Aussagen zu finden ist nicht immer ganz einfach und Übung ist sehr(!) hilfreich. Hinweise kann 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:

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

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:

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

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

INDUKTIONSSCHRITT $i \rightarrow i + 1$:

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

INDUKTIONSSCHLUSS: zu zeigen:

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

Falls $i + 1 \geq 4$ ist, ist die Aussage ohnehin wahr.

Sei also $i + 1 < 4$. 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 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, dass jedenfalls in

der Beispielrechnung aber $X_3 = 0$. 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 $X_i \text{ div } 2 \leq X_i/2$. 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: $\forall i \in \mathbb{N}_0 : i < 4 \implies X_i \leq a/2^i$. 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 $X_3 = X_2 \text{ div } 2 = 0$.

5.6 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, 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 5.4.

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:

Schleife

for $\langle \text{Schleifenvariable} \rangle \leftarrow \langle \text{Startwert} \rangle$ **to** $\langle \text{Endwert} \rangle$ **do**
 $\langle \text{sogenannter Schleifenrumpf, der} \rangle$
 $\langle \text{aus mehreren Anweisungen bestehen darf} \rangle$
od

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

Abbildung 5.5 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.

```

// Eingaben:  $a \in \mathbb{G}_8, b \in \mathbb{N}_0$ 
 $X \leftarrow a$ 
 $Y \leftarrow b$ 
 $P \leftarrow 0$ 
 $x \leftarrow X \bmod 2$ 
// — Algorithmusstelle —  $i = 0$ 
 $P \leftarrow P + x \cdot Y$ 
 $X \leftarrow X \operatorname{div} 2$ 
 $Y \leftarrow 2 \cdot Y$ 
 $x \leftarrow X \bmod 2$ 
// — Algorithmusstelle —  $i = 1$ 
 $P \leftarrow P + x \cdot Y$ 
 $X \leftarrow X \operatorname{div} 2$ 
 $Y \leftarrow 2 \cdot Y$ 
 $x \leftarrow X \bmod 2$ 
// — Algorithmusstelle —  $i = 2$ 
 $P \leftarrow P + x \cdot Y$ 
 $X \leftarrow X \operatorname{div} 2$ 
 $Y \leftarrow 2 \cdot Y$ 
 $x \leftarrow X \bmod 2$ 
// — Algorithmusstelle —  $i = 3$ 

```

Abbildung 5.4: Vereinfachte Darstellung des Algorithmus aus Abbildung 5.3.

Interessant ist es nun, sich noch einmal klar zu machen, was nach dem Entfernen der Indizes aus den Aussagen $\mathcal{A}_i \equiv P_i + X_i Y_i = ab$ 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.

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

Schleifeninvariante

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.

```

// Eingaben:  $a \in \mathbb{G}_8, b \in \mathbb{N}_0$ 
 $X \leftarrow a$ 
 $Y \leftarrow b$ 
 $P \leftarrow 0$ 
 $x \leftarrow X \bmod 2$ 
for  $i \leftarrow 0$  to 2 do
    // — Algorithmusstelle —  $i$ 
     $P \leftarrow P + x \cdot Y$ 
     $X \leftarrow X \operatorname{div} 2$ 
     $Y \leftarrow 2 \cdot Y$ 
     $x \leftarrow X \bmod 2$ 
    // — Algorithmusstelle —  $i + 1$ 
od
// Ergebnis:  $P = a \cdot b$ 

```

Abbildung 5.5: Multiplikationsalgorithmus mit einer Schleife für größenbeschränkten Faktor a .

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.

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 5.6 dargestellte Algorithmus.

```

// Eingaben:  $a \in \mathbb{N}_0, b \in \mathbb{N}_0$ 
 $X \leftarrow a$ 
 $Y \leftarrow b$ 
 $P \leftarrow 0$ 
 $x \leftarrow X \bmod 2$ 
 $n \leftarrow 1 + \lceil \log_2 a \rceil$ 
for  $i \leftarrow 0$  to  $n - 1$  do
     $P \leftarrow P + x \cdot Y$ 
     $X \leftarrow X \operatorname{div} 2$ 
     $Y \leftarrow 2 \cdot Y$ 
     $x \leftarrow X \bmod 2$ 
od
// Ergebnis:  $P = a \cdot b$ 

```

Abbildung 5.6: Algorithmus zu Multiplikation beliebiger nichtnegativer ganzer Zahlen.

6 FORMALE SPRACHEN

6.1 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 *syntaktische 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 $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -\}$. 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).

6.2 OPERATIONEN AUF FORMALEN SPRACHEN

6.2.1 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

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

das *Produkt der Sprachen L_1 und L_2* .

6.1 Lemma. Für jede formale Sprache L ist

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

Produkt von Sprachen

6.2 Beweis. Einfaches Nachrechnen:

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

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

Wie bei Wörtern will man *Potenzen L^k* definieren. Der „Trick“ besteht darin, für den Fall $k = 0$ etwas Sinnvolles zu finden — Lemma 6.1 gibt einen Hinweis. Die Definition geht (wer hätte es gedacht?) wieder induktiv:

Potenzen L^k

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

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 $L = \{aa, b\}$. Dann ist

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

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

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

es ist also (angedeutet)

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

Welche Wörter sind in L^2 ? Die Definition 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)

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

Mit anderen Worten ist

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

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.

6.2.2 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:

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

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

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

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

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

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

Analog ist

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

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

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

Und für L^+ könnte man vielleicht notieren:

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

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:

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

6.3 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.

7 DOKUMENTE

7.1 DOKUMENTE

Die Idee zu dieser Einheit geht auf eine Vorlesung „Informatik A“ von Till Tantau zurück (siehe <http://www.itheoi.mu-luebeck.de/lehre/2008-ws/info-a/wiki/Vorlesung>, 10.11.08).

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

*ein Rat für Ihr
weiteres Studium*

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

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.

Auszeichnungssprache
markup language

7.2.1 \LaTeX

\LaTeX , ausgesprochen „Latech“, ist (etwas ungenau, aber für unsere Belange ausreichend formuliert) eine Erweiterung eines Textsatz-Programmes ($\text{\textit{iniTeX}}$), das von Donald Knuth entwickelt wurde (<http://www-cs-faculty.stanford.edu/~knuth/>, 12.11.08).

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:

```
\subsection{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 zB die Auswahl 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
<code>\begin{itemize}</code>	
<code>\item Inhalt</code>	• Inhalt
<code>\item Struktur</code>	• Struktur
<code>\item Form</code>	• Form
<code>\end{itemize}</code>	

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 $L_{\text{itemize}} = \{\text{\begin{itemize}}\} \left(\{\text{\item}\}_{L_{\text{item}}} \right)^+ \{\text{\end{itemize}}\}$. 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 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:

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

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

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

das heißt, eine Tabelle ist (*table*) 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 oben).

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 Klammern. Bei einer syntaktisch korrekten Klammerung gibt es zu jeder Klammer auf „weiter hinten“ die „zugehörige“ Klammer zu. Insbesondere gilt:

- Man kann beliebig viele korrekte Klammern 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 $\{ \} \cdot L_{\text{Klammer}} \cdot \{ \}$ 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 _{opt} }
2	BlockStatements:
	BlockStatement
	BlockStatements BlockStatement
3	BlockStatement:
	Statement

4	Statement:
	StatementWithoutTrailingSubstatement

5	StatementWithoutTrailingSubstatement:
	Block

Tabelle 2: 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 $\langle Block \rangle$, um das Lesen zu vereinfachen (und um schon mal deutlich zu machen, dass es sich z. B. bei $\langle Block \rangle$ 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 $\langle Block \rangle$ hat die Struktur: Ein Zeichen { , eventuell gefolgt von $\langle BlockStatements \rangle$ (das tiefgestellte Suffix „opt“ besagt, dass das optional ist), gefolgt von einem Zeichen } .
2. $\langle BlockStatements \rangle$ sind von einer der Strukturen
 - ein einzelnes $\langle BlockStatement \rangle$ oder
 - $\langle BlockStatements \rangle$ gefolgt von einem $\langle BlockStatement \rangle$

3. Ein $\langle \text{BlockStatement} \rangle$ kann ein $\langle \text{Statement} \rangle$ sein (oder anderes ...).
4. Ein $\langle \text{Statement} \rangle$ kann ein $\langle \text{StatementWithoutTrailingSubstatement} \rangle$ sein (oder anderes, zum Beispiel etwas „ganz einfaches“ ...).
5. Ein $\langle \text{StatementWithoutTrailingSubstatement} \rangle$ kann ein $\langle \text{Block} \rangle$ sein (oder anderes ...).

Diese Formulierungen beinhalten Rekursion: Bei der Beschreibung der Struktur von $\langle \text{BlockStatements} \rangle$ wird direkt auf $\langle \text{BlockStatements} \rangle$ Bezug genommen.

Außerdem wird bei der Definition von $\langle \text{Block} \rangle$ (indirekt) auf $\langle \text{Statement} \rangle$ verwiesen und bei der Definition von $\langle \text{Statement} \rangle$ (indirekt) wieder auf $\langle \text{Block} \rangle$.

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 $\langle \text{Block} \rangle$, $\langle \text{Statement} \rangle$ o.ä., und schreiben wir lieber runde Klammern $($ und $)$ statt der geschweiften, weil wir die die ganze Zeit als Mengenkammern 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:

$$L = \{\varepsilon\} \cup LL \cup \{()L()\} \quad (7)$$

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 7 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 7 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 7 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 $\{(\cdot)\}$ enthält, und
- zweitens dann zeigen, dass die Vereinigung aller dieser L_i eine Lösung von Gleichung 7 ist.

Also:

- Der Anfang ist ganz leicht: $L_0 = \{\varepsilon\}$.
- und weiter machen kann man so: für $i \in \mathbb{N}_0$ sei $L_{i+1} = L_i L_i \cup \{(\cdot)L_i\}$.

Wir behaupten, dass $L = \bigcup_{i=0}^{\infty} L_i$ Gleichung 7 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 $i \in \mathbb{N}_0$: $L_i \subseteq L_{i+1}$.

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

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

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

Damit haben wir gezeigt, dass Gleichung 7 (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:

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

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 (kürsicherweise 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.

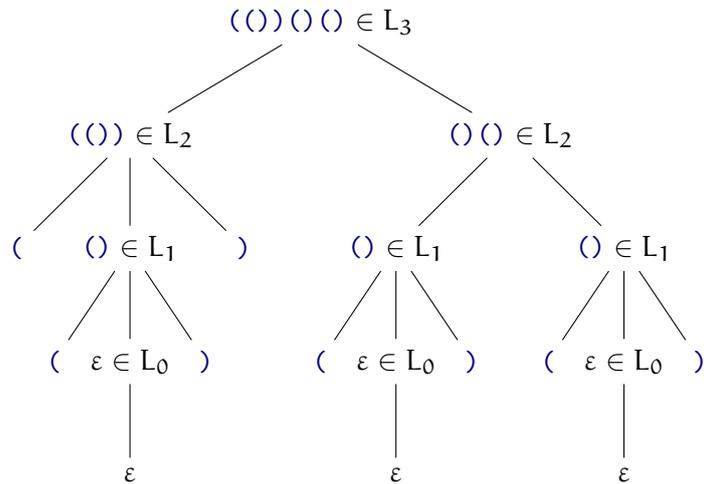


Abbildung 8.1: Eine „Begründung“, warum $()()() \in L_3$ ist.

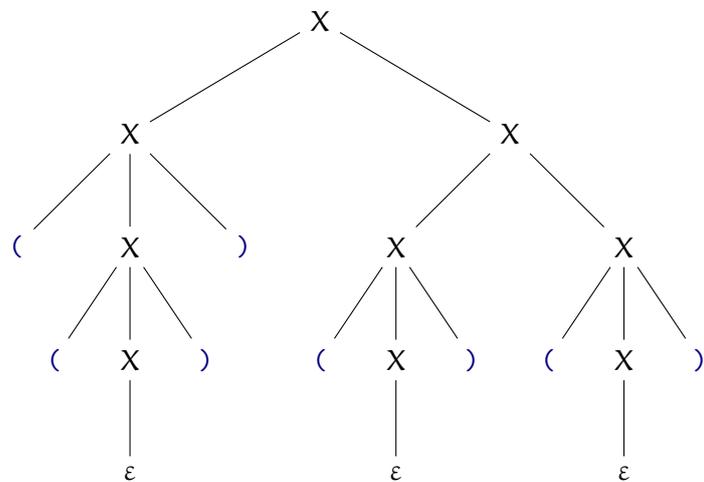


Abbildung 8.2: Eine abstraktere Darstellung der „Begründung“, warum $()()() \in L_3$ ist.

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

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

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

kontextfreie Grammatik

Nichtterminalsymbole

Terminalsymbole

- $S \in N$ ist das sogenannte *Startsymbol* (also ein Nichtterminalsymbol). Startsymbol
- $P \subseteq N \times V^*$ ist eine *endliche* Menge sogenannter *Produktionen*. Dabei ist $V = N \cup T$ die Menge aller Symbole überhaupt. Produktionen

Gilt für ein Nichtterminalsymbol X und ein Wort w , dass $(X, w) \in P$ ist, dann schreibt man diese Produktion üblicherweise in der Form $X \rightarrow w$ 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. $X \rightarrow w$

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 $X \rightarrow w$ in P gibt, so dass $u = w_1 X w_2$ und $v = w_1 w w_2$. Ableitungsschritt

Betrachten wir als Beispiel die Grammatik $G = (\{X\}, \{a, b\}, X, P)$ mit der Produktionenmenge $P = \{X \rightarrow \varepsilon, X \rightarrow aXb\}$. Dann gilt zum Beispiel $abaXbaXXXX \Rightarrow abaXbaaXbXXX$, wie man an der folgenden Aufteilung sieht:

$$\underbrace{abaXba}_{w_1} X \underbrace{XXXX}_{w_2} \Rightarrow \underbrace{abaXba}_{w_1} aXb \underbrace{XXX}_{w_2}$$

Ebenso gilt $abaXbaXXXX \Rightarrow abaaXbbaXXXX$:

$$\underbrace{aba}_{w_1} X \underbrace{baXXXX}_{w_2} \Rightarrow \underbrace{aba}_{w_1} aXb \underbrace{baXXXX}_{w_2}$$

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

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. Infixschreibweise

Im allgemeinen ist \Rightarrow weder links- noch rechtstotal und weder links- noch rechtseindeutig.

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 *Terminalsymbol*). 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 Ableitung

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

Bei unserer Beispielgrammatik gilt zum Beispiel

$$X \Rightarrow aXb \Rightarrow aaXbb \Rightarrow aaaXbbb \Rightarrow aaabbb$$

Also hat man z. B. die Beziehungen $X \Rightarrow^* aaXbb$, $aXb \Rightarrow^* aaaXbbb$, $X \Rightarrow^* aaabbb$ 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

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

von einer
Grammatik erzeugte
formale Sprache

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

kontextfreie Sprache

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

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

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

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

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

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

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

$$P = \{ X \rightarrow aXb \mid \varepsilon \}$$

Und damit können wir nun auch die richtige Interpretation für die Fragmente aus Tabelle 2 angeben: Es handelt sich um Produktionen einer kontextfreien Grammatik.

- Jedes der Wörter „Block“ ist *ein* Nichtterminalsymbol. Deswegen haben wir Schreibweisen wie $\langle \text{Block} \rangle$ vorgezogen, um deutlicher zu machen, dass wir so etwas als *ein* nicht weiter zerlegbares Objekt betrachten wollen.
- Der Doppelpunkt entspricht unserem Pfeil \rightarrow 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 | getrennt denken.

Der zweite Teil

2	BlockStatements:
	BlockStatement
	BlockStatements BlockStatement

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

$$\langle \text{BlockStatements} \rangle \rightarrow \langle \text{BlockStatement} \rangle$$

$$| \langle \text{BlockStatements} \rangle \langle \text{BlockStatement} \rangle$$

Und der erste Teil

1	Block:
	{ BlockStatements _{opt} }

der Tabelle repräsentiert die Produktionen

$$\langle \text{Block} \rangle \rightarrow \{ \langle \text{BlockStatements} \rangle \} | \{ \}$$

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: $(\{X\}, \{(,)\}, X, \{X \rightarrow XX | (X) | \epsilon\})$. Aber

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

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

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:

$$\begin{aligned} X &\Rightarrow XX \Rightarrow (X)X \Rightarrow ((X))X \Rightarrow ((\))X \Rightarrow ((\))XX \\ &\Rightarrow ((\))(X)X \Rightarrow ((\))(\))X \Rightarrow ((\))(\))(X) \Rightarrow ((\))(\))(\)) \end{aligned}$$

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

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

Typ-2-Grammatiken

- 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

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

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.

Produkt von Relationen

Mit Id_M bezeichnen wir die Relation

$$\text{Id}_M = \{(x, x) \mid x \in M\}$$

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:

$$R \circ \text{Id}_M = R = \text{Id}_M \circ R$$

Ist $R \subseteq M \times M$ binäre Relation auf einer Menge M , dann definiert man *Potenzen* R^i wie folgt:

$$R^0 = \text{Id}_M$$

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

Potenzen einer Relation

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

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

reflexiv-transitive Hülle

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 $\text{Id}_M \subseteq R$ ist.

Eine Relation heißt *transitiv*, wenn gilt:

$$\forall x \in M : \forall y \in M : \forall z \in M : xRy \wedge yRz \implies xRz$$

reflexive Relation
transitive Relation

Bitte bringen Sie den logischen Implikationspfeil \implies 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 $\text{Id}_M = R^0 \subseteq R^*$ 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 $N_x : A^* \rightarrow \mathbb{N}_0$, 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

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

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 Zeile 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 Halbleitermaterialien 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	000	10110101
Adresse 2	Wert 2	001	10101101
Adresse 3	Wert 3	010	10011101
Adresse 4	Wert 4	011	01110110
		100	00111110
		101	10101101
		110	00101011
Adresse n	Wert n	111	10101001

(a) allgemein

(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:

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

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

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

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:

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

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 $f : A \rightarrow B$ von einer Menge A in eine Menge B notieren wollen. Hierfür benutzen wir die Schreibweise mit 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:

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

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:

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

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:

$$\text{memread}(\text{memwrite}(m, a, v), a) = v \quad (8)$$

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:

$$\text{memread}(\text{memwrite}(m, a', v'), a) = \text{memread}(m, a) \quad (9)$$

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, sogenannte Caches bzw. allgemeiner Assoziativspeicher. Für eine adäquate Formalisierung des Schreibens in einen solchen Speicher müsste man also die Definition 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 Namen 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 8 und 9 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 die 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 (μ s), 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“ analog 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> (24.11.08) 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,
2. die die schöne Eigenschaft hat, dass jedes Ausgangswort und seine Übersetzung die gleiche Bedeutung haben.

Als erstes schauen wir uns ein einfaches und zu verstehendes 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 5 über den Algorithmusbegriff):

- Verwendet werden die Ziffern des Alphabetes $Z_{10} = \{0, \dots, 9\}$.
- 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 $x_{k-1} \dots x_0 \in Z_{10}^*$ von Ziffern wollen wir $\text{Num}_{10}(x_{k-1} \dots x_0)$ schreiben. In der Schule hat man gelernt, dass das gleich

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

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

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

ist, definiert man:

$$\text{Num}_{10}(\varepsilon) = 0$$

$$\forall w \in Z_{10}^* \quad \forall x \in Z_{10} : \text{Num}_{10}(wx) = 10 \cdot \text{Num}_{10}(w) + \text{num}_{10}(x)$$

10.1.2 Andere Zahldarstellungen

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.



Schon Leibniz hat erkannt, dass man die natürlichen Zahlen nicht nur mit den Ziffern $0, \dots, 9$ 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 (http://www.hs-augsburg.de/~harsch/germanica/Chronologie/17Jh/Leibniz/lei_bina.html, 9.10.2008) beschrieben und im Jahre 1703 in einer Zeitschrift veröffentlicht. Abbildung 10.1 zeigt Beispielrechnungen mit Zahlen in Binärdarstellung aus dieser Veröffentlichung.

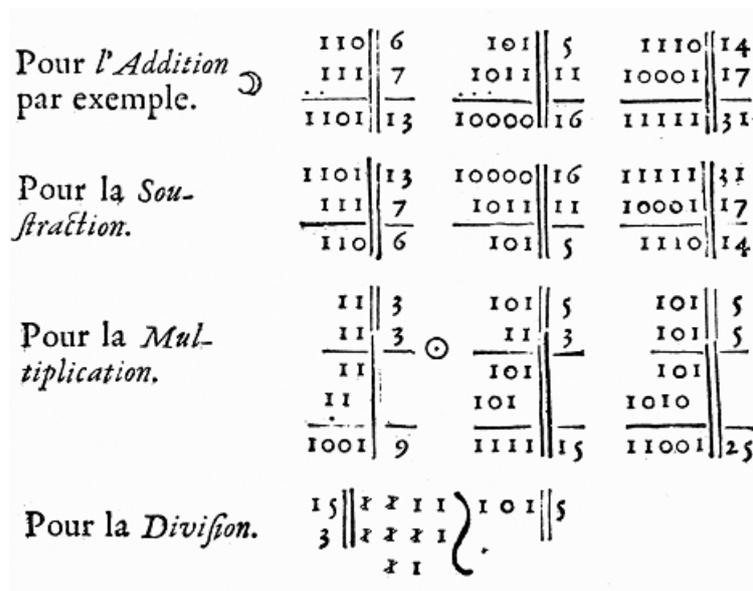


Abbildung 10.1: Ausschnitt aus dem Aufsatz „Explication de l'Arithmétique Binaire“ von Leibniz, Quelle: http://commons.wikimedia.org/wiki/Image:Leibniz_binary_system_1703.png (3.9.2008)

Bei der Binärdarstellung von nichtnegativen ganzen Zahlen geht man analog zur Dezimaldarstellung vor. Als Ziffernmeng

Binärdarstellung

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

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

Damit ist dann z. B.

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

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

Hexadezimaldarstellung

x	0	1	2	3	4	5	6	7
num ₁₆ (x)	0	1	2	3	4	5	6	7
x	8	9	A	B	C	D	E	F
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

- Num₂(**111**) die Zahl sieben,
- Num₈(**111**) die Zahl dreiundsiebzig,
- Num₁₀(**111**) die Zahl einhundertelf und
- Num₁₆(**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 $\text{Num}_k(w) = n$, dann auch $\text{Num}_k(0w) = n$ (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
     $r \leftarrow y \bmod k$ 
     $w \leftarrow \text{repr}_k(r) \cdot w$ 
     $y \leftarrow y \text{ div } k$ 
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 $y \cdot 10^i + \text{Num}_k(w) = n$ 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 $\text{Num}_k(w) = n$. Dieses Wort wollen wir auch mit $\text{Repr}_k(n)$ bezeichnen. Es ist also stets

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

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

10.2.1 Ein Beispiel: Übersetzung von Zahldarstellungen

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

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

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 $\text{sem}_A : L_A \rightarrow \text{Sem}$ und $\text{sem}_B : L_B \rightarrow \text{Sem}$ von formalen Sprachen $L_A \subseteq A^*$ und $L_B \subseteq B^*$ in die gleiche Menge Sem von Bedeutungen.

Eine Abbildung $f : L_A \rightarrow L_B$ heie eine *Übersetzung* bezüglich sem_A und sem_B , wenn f die Bedeutung erhält, d. h.

Übersetzung

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

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 Abbildung die aus einer Speicherbelegung m die Speicherbelegung $\text{memwrite}(m, x, 5)$ macht. Eine grundlegende Einführung in solche Fragestellungen können Sie in Vorlesungen über die Semantik von Programmiersprachen bekommen.

Warum macht man Übersetzungen? Zumindest die folgenden vier 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).
- *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.
- *Verschlüsselung*: Im Gegensatz zu verbesserter Lesbarkeit übersetzt man manchmal gerade deshalb, um die Lesbarkeit möglichst unmöglich zu machen, jedenfalls für Außenstehende. Wie man das macht, ist Gegenstand von Vorlesungen über Kryptographie.
- *Fehlererkennung und Fehlerkorrektur*: Manchmal kann man Texte durch Übersetzung auf eine Art länger machen, dass man selbst dann, wenn ein korrekter 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 Vorlesungen 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 $\text{sem}_A(x) \neq \text{sem}_A(y)$ 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. Und die Menge $\{f(w) \mid w \in L_A\}$ heißt dann auch ein *Code*.

Codierung
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 gleich bzw. am Ende dieser Einheit eingehen werden.

10.2.2 Homomorphismen

Es seien A und B zwei Alphabete und $h : A \rightarrow B^*$ eine Abbildung. Zu h kann man in der Ihnen inzwischen vertrauten Art eine Funktion $h^{**} : A^* \rightarrow B^*$ 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: $h^{**}(x) = h(x)$).

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

Dass eine Abbildung $h : A \rightarrow B^*$ eine Codierung, also eine injektive Abbildung $h : A^* \rightarrow B^*$ 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 bedeutet, dass für *keine* zwei verschiedenen Symbole $x_1, x_2 \in A$ gilt: $h(x_1)$ ist ein Präfix von $h(x_2)$.

Das praktische Beispiel schlechthin für einen Homomorphismus ist die Repräsentation des ASCII-Zeichensatzes (siehe Unterabschnitt 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 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, wo das nicht so ist.

10.2.3 Beispiel Unicode: UTF-8

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. Das es so viele Zeichen sind (und 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*.

Nachfolgend ist die Definition dieses Homomorphismus in Ausschnitten wiedergegeben. Sie stammen aus dem RFC 3629 (<http://www.rfc-editor.org/rfc/rfc3629.txt> (1.10.08))

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

h^{**}
Homomorphismus

ε -freier
Homomorphismus

präfixfrei

UTF-8

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

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 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

Es sei ein Alphabet A und ein Wort $w \in A^*$ gegeben. Eine sogenannte *Huffman-Codierung* von w ist eine Abbildung $h : A^* \rightarrow Z_2^*$, die ein ε -freier Homomorphismus ist. Es ist also h eindeutig festgelegt durch die Funktionswerte $h(x)$ für alle $x \in A$.

Huffman-Codierung

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.

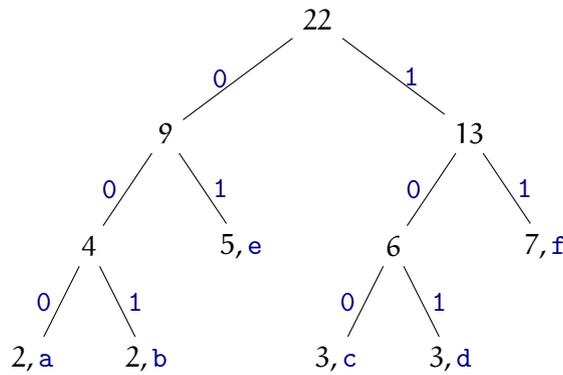


Abbildung 10.2: Ein Beispielbaum für die Berechnung eines Huffman-Codes.

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 $w = \text{afebfecaaffdeddccefbfeff}$ gegeben ist, dann kann sich der folgende Baum ergeben:

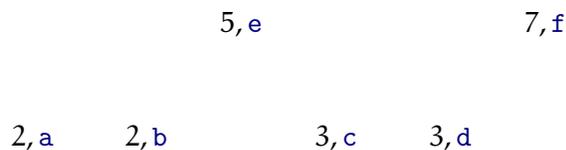
In diesem Fall ist dann der Homomorphismus gegeben durch

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:

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

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



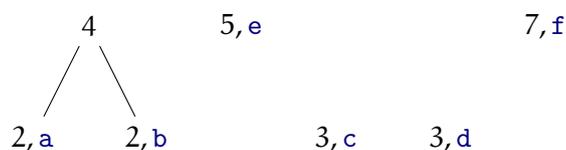
Solange eine Menge M_i noch mindestens zwei Paare enthält, tut man folgendes:

- 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} .

Im Beispiel ergibt sich also

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

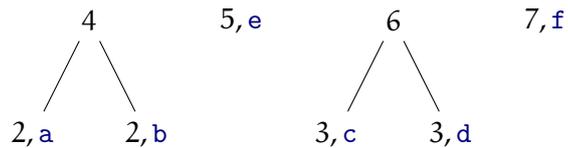
- Als zweites fügt man dem schon konstruierten Teil des Graphen einen weiteren Knoten hinzu, markiert mit der Häufigkeit $k_1 + k_2$ und Kanten zu den Knoten, die für (X_1, k_1) und (X_2, k_2) eingefügt worden waren



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

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

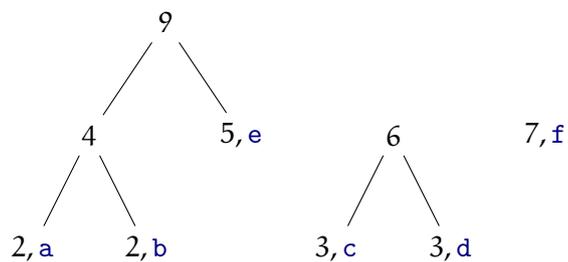
und der Graph sieht dann so aus:



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

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

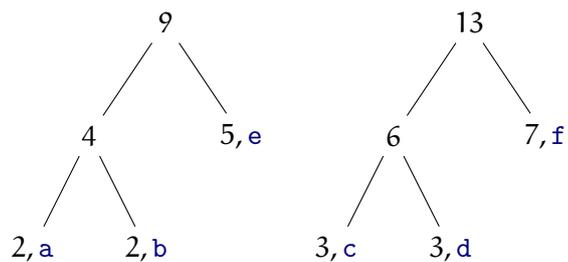
und der Graph sieht dann so aus:



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

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

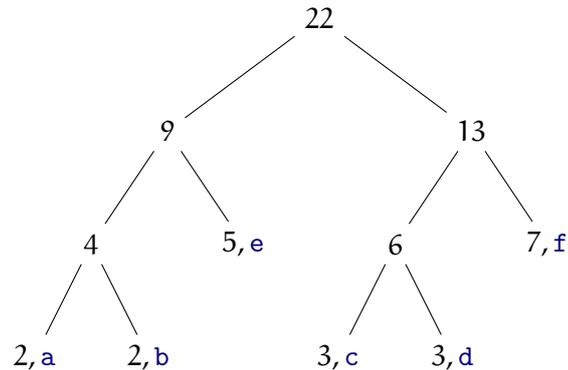
und der Graph sieht dann so aus:



Zum Schluss berechnet man noch

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

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> (24.11.08).

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 Homomorphismus die Übersetzung $h(x)$ 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

Block-Codierung

also von einer Funktion $h : A^b \rightarrow B^*$ aus, und erweitert diese zu einer Funktion $h : (A^b)^* \rightarrow B^*$.

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/> (9.10.2008) 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 mitnotiert 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:

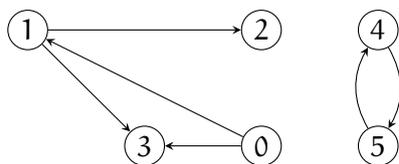


Abbildung 11.2: eine andere Zeichnung des Graphen aus Abbildung 11.1

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

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

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:

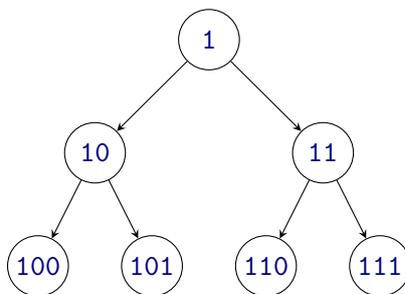


Abbildung 11.3: ein Graph, der ein sogenannter Baum ist

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

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

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

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

Schlinge

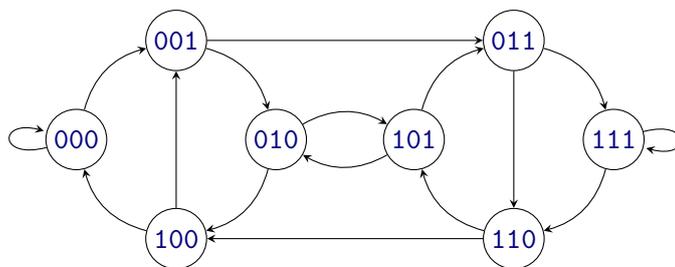


Abbildung 11.4: Der de Bruijn-Graphen mit 8 Knoten

auszugehen, dass ein Graph keine Schlingen besitzt. Ein solcher Graph heißt *schlingenfrei*.

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:

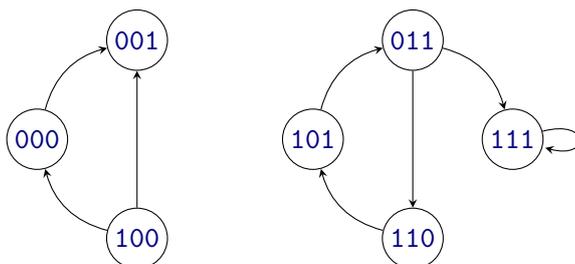


Abbildung 11.5: Ein Teilgraph des de Bruijn-Graphen aus Abbildung 11.4

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) .

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 $i \in \mathbb{C}_n$ 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 $i \in \mathbb{C}_n$.

Pfad

Länge eines Pfades

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

erreichbar

Ein Pfad (v_0, \dots, v_n) heißt *wiederholungsfrei*, wenn gilt: Die Knoten v_0, \dots, v_{n-1} sind paarweise verschieden und die Kno-

wiederholungsfreier Pfad

ten v_1, \dots, v_n sind paarweise verschieden. Die beiden einzigen Knoten, die gleich sein dürfen, sind also v_0 und v_n .

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

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.

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.

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.

Durch „Hintereinanderhängen“ dieser Pfade der Länge > 0 ergäbe sich ein Pfad von r nach r , der vom Pfad (r) 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 ihm weg führen. Der *Eingangsgrad* eines Knoten y wird mit $d^-(y)$ bezeichnet und ist definiert als

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

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 .

geschlossener Pfad
Zyklus
einfacher Zyklus

streng zusammenhängender Graph

Baum

Wurzel

Eingangsgrad

Ausgangsgrad
Grad

11.1.3 Isomorphie von Graphen

Wir haben eingangs davon gesprochen, dass 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 $f : V_1 \rightarrow V_2$ gibt mit der Eigenschaft:

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

isomorph

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 $f = \text{Id}_V$.
- 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 einen engen Zusammenhang 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 $E^2 = E \circ E^1 = E \circ E \circ \text{Id} = E \circ E$ 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 $E^0 = \text{Id}_V$, 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 $(x, y) = (x, x) \in \text{Id}_V = E^0$.

Damit haben wir uns explizit davon überzeugt, dass für alle $i \in \mathbb{C}_3$ 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 Interpretation 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 Formalisierung an

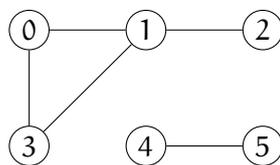


Abbildung 11.6: ein ungerichteter Graph

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 ungerichte-

ungerichteter Graph

ten Graphen *adjazent*, wenn sie durch eine Kante miteinander verbunden sind.

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

Teilgraph

Wir sagen, dass $U' = (V', E')$ ein *Teilgraph* eines ungerichteten Graphen $U = (V, E)$ 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 Teilmengen 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 $p = (v_0, \dots, v_n) \in V^{(+)}$ 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

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

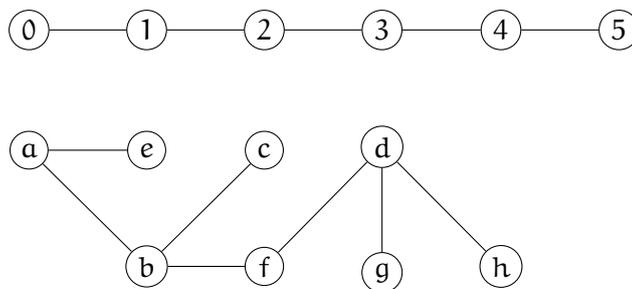


Abbildung 11.7: zwei ungerichtete Bäume

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

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

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 „automatisch“ $(y, x) \in E_g$. 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

$$(x, y) \in R \implies (y, x) \in R.$$

Und wir wollen an dieser Stelle schon einmal erwähnen, dass eine Relation, die reflexiv, transitiv und symmetrisch ist, eine sogenannte *Äquivalenzrelation* ist.

Äquivalenzrelation

Wir haben 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.

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 $m_V : V \rightarrow M_V$ 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 Probleme müssen nicht nur (vielleicht) von Verlagen für Atlanten gelöst werden, sondern sie werden auch etwa von modernen Compilern beim Übersetzen von Programmen bearbeitet.

Ein Graph mit Kantenmarkierungen oder *kantenmarkierter Graph* liegt vor, wenn zusätzlich zu $G = (V, E)$ auch noch eine Abbildung $m_E : E \rightarrow M_E$ 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 $M_E = \{0, 1\}$.

kantenmarkierter Graph

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 vielleicht auch mal Vektoren von Zahlen o. ä.;

gewichtete Graphen

jedenfalls soll die Menge der Gewichte irgendeine Art von algebraischer Struktur aufweisen, so dass man „irgendwie rechnen“ kann.

Als Motivation 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 Kanten 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 transportiert werden soll.

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](#) beginnen wir mit einem ganz naiven Algorithmus und verbessern in dann in zwei Schritten. Dabei werden Sie sehen, dass manchmal Kenntnisse über die mathematische Struktur der benutzten Objekte bei der Verbesserung von Algorithmen sehr hilfreich sein können.

Einen der klassischen Algorithmen, den von Warshall, für das Problem, werden wir in Abschnitt [12.3](#) 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.

Das ist natürlich nicht schädlich. 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 $n \geq 1$. 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 Methode, 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 {
    String name;
    Vertex[] neighbors;
}
```

Man spricht dann davon, dass für jeden Knoten die *Adjazenzliste*

Adjazenzliste

vorhanden ist. 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: die sogenannte *Adjazenzmatrix* des Graphen. (Sie haben Matrizen inzwischen in der Vorlesung „Lineare Algebra“ kennengelernt.) Die Adjazenzmatrix eines gerichteten Graphen $G = (V, E)$ mit n Knoten ist eine $n \times n$ -Matrix A mit der Eigenschaft:

Adjazenzmatrix

$$A_{ij} = \begin{cases} 1 & \text{falls } (i, j) \in E \\ 0 & \text{falls } (i, j) \notin E \end{cases}$$

Als Beispiel sind in Abbildung 12.1 ein Graph mit vier Knoten und einigen Kanten und die zugehörige Adjazenzmatrix angegeben.

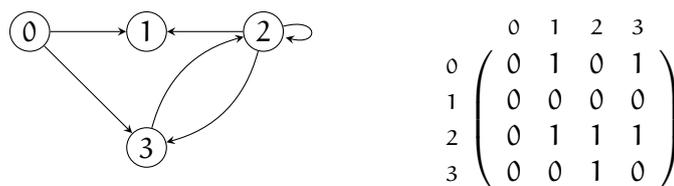


Abbildung 12.1: Ein Graph und seine Adjazenzmatrix

Im Falle eines ungerichteten Graphen $U = (V, E)$ versteht man unter seiner Adjazenzmatrix die des zugehörigen gerichteten Graphen $G = (V, E_g)$.

So, wie man die Kantenrelation E eines gerichteten Graphen als (Adjazenz-)Matrix 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 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:

- 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?
- Was kann man tun, um nicht unendlich viele Matrizen berechnen zu müssen? D. h., kann man das ∞ durch eine natürliche Zahl ersetzen?

Beginnen wir mit dem letzten 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: $p = (i_0, i_1, \dots, i_k)$ 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:

$$\forall k \geq n - 1 : E^* = \bigcup_{i=0}^k E^i$$

12.2.1 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 0 oder 1, und er ist genau dann 1, wenn $A_{ik} = A_{kj} = 1$ ist, also genau dann, wenn es Kanten von i nach k und von k nach j gibt. Das ist genau dann der Fall, 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 .

Analoge Aussagen gelten 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 . 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. Wir bezeichnen nun mit sgn die sogenannte *Signum-Funktion*

Signum-Funktion

$$\text{sgn} : \mathbb{R} \rightarrow \mathbb{R} : \text{sgn}(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases}$$

Für die Erweiterung auf Matrizen durch komponentenweise Anwendung schreiben wir auch wieder sgn :

$$\text{sgn} : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n} : (\text{sgn}(M))_{ij} = \text{sgn}(M_{ij})$$

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:

$$\text{sgn}((A^k)_{ij}) = \begin{cases} 1 & \text{falls in } G \text{ ein Pfad der Länge } k \\ & \text{von } i \text{ nach } j \text{ existiert} \\ 0 & \text{falls in } G \text{ kein Pfad der Länge } k \\ & \text{von } i \text{ nach } j \text{ existiert} \end{cases}$$

2. Für alle $k \in \mathbb{N}_0$ gilt: $\text{sgn}(A^k)$ ist die Matrix, die die Relation E^k repräsentiert. (Dabei sei für eine Matrix A per definitionem A^0 die Einheitsmatrix.)

12.2.2 Erste Möglichkeit für die Berechnung der Wegematrix

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 $\text{sgn}(\sum_{i=0}^k A^i)$ repräsentiert die Relation E^* .
- Mit anderen Worten:

$$W = \text{sgn} \left(\sum_{i=0}^k A^i \right)$$

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 Vereinigung $\bigcup_{i=0}^{n-1} E^i$ wird repräsentiert durch die Matrix $\text{sgn}(\sum_{i=0}^k \text{sgn}(A^i))$.
- In dieser Formel darf man die „inneren“ Anwendungen von sgn weglassen.

Das sieht man so:

- Zum ersten Punkt genügt es, sich zu überlegen, dass für zwei (Rest per Induktion) Relationen R und R' , die durch Matrizen M und M' dargestellt werden, die Vereinigung $R \cup R'$ durch $\text{sgn}(M + M')$ dargestellt wird.

$$\begin{aligned} \text{sgn}(M + M')_{ij} = 1 &\iff (M + M')_{ij} \geq 1 \\ &\iff M_{ij} = 1 \vee M'_{ij} = 1 \\ &\iff (i, j) \in R \vee (i, j) \in R' \\ &\iff (i, j) \in R \cup R' \end{aligned}$$

- Mit einer ähnlichen Herangehensweise wie beim ersten Punkt ergibt sich

$$\begin{aligned}
\text{sgn}(\text{sgn}(M) + \text{sgn}(M'))_{ij} = 1 &\iff (\text{sgn}(M) + \text{sgn}(M'))_{ij} \geq 1 \\
&\iff (\text{sgn}(M) + \text{sgn}(M'))_{ij} \geq 1 \\
&\iff \text{sgn}(M)_{ij} + \text{sgn}(M')_{ij} \geq 1 \\
&\iff \text{sgn}(M)_{ij} = 1 \vee \text{sgn}(M')_{ij} = 1 \\
&\iff M_{ij} \geq 1 \vee M'_{ij} \geq 1 \\
&\iff M_{ij} + M'_{ij} \geq 1 \\
&\iff (M + M')_{ij} \geq 1 \\
&\iff \text{sgn}(M + M')_{ij} = 1
\end{aligned}$$

Dabei haben wir in beiden Punkten benutzt, dass die Einträge in den interessierenden Matrizen nur 0 oder 1 sind. ■

Das sich ergebende Verfahren ist in Algorithmus 12.1 dargestellt.

Algorithmus 12.1 einfachster Algorithmus, um die Wegematrix zu berechnen

```

// Matrix A sei die Adjazenzmatrix
// Matrix W wird am Ende die Wegematrix enthalten
// Matrix M wird benutzt um Ai zu berechnen
W ← 0 // Nullmatrix
for i ← 0 to n - 1 do
  M ← Id // Einheitsmatrix
  for j ← 1 to i do
    M ← M · A // Matrixmultiplikation
  od
  W ← W + M // Matrixaddition
od
W ← sgn(W)

```

12.2.3 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 12.1 zu benutzen und ohne viel Nachdenken zu implementieren. Das bedeutet, dass wir zur Berechnung von $\text{sgn}\left(\sum_{i=0}^{n-1} A^i\right)$ folgende

Berechnungen (nicht unbedingt in dieser Reihenfolge) durchzuführen haben:

- n^2 Berechnungen der sgn Funktion;
- n Matrix-Additionen;
- $0 + 1 + \dots + n - 1 = \sum_{i=1}^{n-1} i = n(n-1)/2$ 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 Abschnitt 13.3 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} .

12.2.4 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 12.2 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 12.1.

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 nut-

Algorithmus 12.2 verbesserter Algorithmus, um die Wegematrix zu berechnen

```

// Matrix A sei die Adjazenzmatrix
// Matrix W wird am Ende die Wegematrix enthalten
// Matrix M wird benutzt um Ai zu berechnen
W ← 0 // Nullmatrix
M ← Id // Einheitsmatrix
for i ← 0 to n - 1 do
    W ← W + M // Matrixaddition
    M ← M · A // Matrixmultiplikation
od
W ← sgn(W)

```

zen 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 $F = E^0 \cup E^1 = \text{Id}_V \cup E$. 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 $2n/\lceil \log_2 n \rceil$.

12.3 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 12.3 dargestellt.

Algorithmus 12.3 Berechnung der Wegematrix nach Warshall

```

for  $i \leftarrow 0$  to  $n - 1$  do
  for  $j \leftarrow 0$  to  $n - 1$  do
     $W[i, j] \leftarrow \begin{cases} 1 & \text{falls } i = j \\ A[i, j] & \text{falls } i \neq j \end{cases}$ 
  od
od
for  $k \leftarrow 0$  to  $n - 1$  do
  for  $i \leftarrow 0$  to  $n - 1$  do
    for  $j \leftarrow 0$  to  $n - 1$  do
       $W[i, j] \leftarrow \max(W[i, j], \min(W[i, k], W[k, j]))$ 
    od
  od
od

```

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.

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 $W[i, j]$ 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 $W[i, j]$ 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:

\implies : Es sei $W_k[i, j] = 1$. Dann hat also mindestens eine der folgenden Bedingungen zugefallen:

- $W_{k-1}[i, j] = 1$: 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.
 - $W_{k-1}[i, k] = 1$ und $W_{k-1}[k, j] = 1$. 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.
- ⇐: 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 $W_{k-1}[i, k - 1] = 1$ und $W_{k-1}[k - 1, j] = 1$. Folglich wird im k -ten Durchlauf $W_k[i, j] = 1$ 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 $W_{k-1}[i, j] = 1$ und daher auch $W_k[i, j] = 1$.

■

12.4 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*. Hg. 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 Problem-Instanz 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.

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.

Laufzeit, Rechenzeit
Ressource
Speicherplatzbedarf
Komplexitätsmaß

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 5 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 [12.1](#) 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 Problem-Instanz 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.

Meistens ist es aber so, dass man nicht für jede Problem-Instanz einzeln angeben will oder kann, wie lange ein Algorithmus zu seiner Lösung benötigt. Man beschränkt sich auf eine vergrößernde Sichtweise und beschreibt z. B. die Laufzeit nur in

```
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) {
        int i ← idx;
        // Tausche a[idx] nach links bis es einsortiert ist
        while (i > 0 ∧ a[i - 1] > a[i]) {
            long tmp ← a[i - 1];
            a[i - 1] ← a[i];
            a[i] ← tmp;
            i--;
        }
    }
}
```

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?

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.

worst case

average case

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 Berechnung 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 vergrößernde 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 reellen Zahlen, also $\mathbb{R}_0^+ = \mathbb{R}^+ \cup \{0\}$. Wir betrachten Funktionen $f : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$.

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 $g : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ wächst *größenordnungsmäßig genauso schnell* wie eine Funktion $f : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$, wenn gilt:

$$\exists c, c' \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : cf(n) \leq g(n) \leq c'f(n) .$$

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 $n_0 = 0$:

$$\forall n \geq n_0 : g(n) = 10^{-2}n^2 \leq 3n^2 = c'f(n)$$

*asymptotisches
Wachstum
größenordnungsmäßiges
Wachstum*

$f \asymp g$

Die eben durchgeführte Rechnung lässt sich leicht etwas allgemeiner durchführen. Dann zeigt sich, dass man festhalten kann:
13.1 (Rechenregel) Für alle $f : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ 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.

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 Unterabschnitt 11.2.1).

- *Reflexivität:* Es ist stets $f \asymp f$, denn 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:

$$\Theta(f) = \{g \mid f \asymp g\}$$

$$= \{g \mid \exists c, c' \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : cf(n) \leq g(n) \leq c'f(n)\}$$

Aus Rechenregel 13.1 wird bei Verwendung von Θ :

13.4 (Rechenregel) Für alle $f : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ und alle Konstanten $a, b \in \mathbb{R}_0^+$ gilt: $\Theta(af(n)) = \Theta(bf(n))$.

13.2.2 Notation für obere und untere Schranken des Wachstums

In Fällen wie der unbekanntem 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:

$$O(f(n)) = \{g(n) \mid \exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \leq cf(n)\}$$

$$\Omega(f(n)) = \{g(n) \mid \exists c \in \mathbb{R}^+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : g(n) \geq cf(n)\}$$

$O(f), \Omega(f)$

Man schreibt gelegentlich auch

$g \preceq f, g \succeq f$

$$g \preceq f \text{ falls } g \in O(f)$$

$$g \succeq f \text{ falls } g \in \Omega(f)$$

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 hypothetischer 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.

$O(1)$

- Mitunter trifft man auch die Schreibweise $O(1)$ an. Was ist das? Die Definition 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 $a < b$, dass $n^a \preceq n^b$ ist, aber nicht umgekehrt.

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 $f : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ und $g : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ gilt:

$$g(n) \in O(f(n)) \iff f(n) \in \Omega(g(n)), \quad \text{also} \quad g \preceq f \iff f \succeq g$$

Man kann auch zeigen:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

$$\text{also} \quad g \asymp f \iff g \preceq f \wedge g \succeq f$$

Das war in diesem Semester erst als Übungsaufgabe gedacht. Nun ist es doch keine.

13.2.3 Die furchtbare Schreibweise

Damit Sie bei Lektüre von Büchern und Aufsätzen alles verstehen, was dort mit Hilfe von $\Theta(\cdot)$, $O(\cdot)$ 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

$$g(n) = O(f(n)) \quad \text{statt} \quad g(n) \in O(f(n)),$$

$$g(n) = \Theta(f(n)) \quad \text{statt} \quad g(n) \in \Theta(f(n)),$$

$$g(n) = \Omega(f(n)) \quad \text{statt} \quad g(n) \in \Omega(f(n)).$$

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 $f_1, f_2 : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ 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

$$M_1 + M_2 = \{g_1 + g_2 \mid g_1 \in M_1 \wedge g_2 \in M_2\}$$

$$M_1 \cdot M_2 = \{g_1 \cdot g_2 \mid g_1 \in M_1 \wedge g_2 \in M_2\}$$

Für Funktionen sei Addition und Multiplikation (natürlich?) argumentweise definiert. Dann ist z. B.

$$O(n^3) + O(n^3) = \{g_1(n) + g_2(n) \mid g_1 \in O(n^3) \wedge g_2 \in O(n^3)\}$$

z. B.

$$(2n^3 - n^2) + 7n^2 = 2n^3 + 6n^2 \in O(n^3) + O(n^3)$$

Wenn eine der Mengen M_i einelementig ist, lässt man manchmal die Mengenklammern darum weg und schreibt zum Beispiel bei Zahlenmengen

$$\text{statt } \{3\} \cdot \mathbb{N}_0 + \{1\} \quad \text{kürzer} \quad 3\mathbb{N}_0 + 1$$

oder bei Funktionenmengen

$$\text{statt } \{n^3\} + O(n^2) \quad \text{kürzer} \quad n^3 + O(n^2)$$

Solche Komplexoperationen sind übrigens nichts Neues für Sie. Die Definition des Produkts formaler Sprachen passt genau in dieses Schema (siehe Unterabschnitt ??).

13.7 Beweis. (von Lemma 13.6) Wir beweisen die beiden Inklusionen getrennt.

„ \subseteq “: Wenn $g_1 \in O(f_1)$, 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 $g_2 \in O(f_2)$, 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 $c = \max(c_1, c_2) \in \mathbb{R}_0^+$:

$$\begin{aligned} g_1(n) + g_2(n) &\leq c_1 f_1(n) + c_2 f_2(n) \\ &\leq c f_1(n) + c f_2(n) \\ &= c(f_1(n) + f_2(n)) \end{aligned}$$

„ \supseteq “: Wenn $g \in O(f_1 + f_2)$ 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 $g_1 : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ 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 $g_1 \in O(f_1)$.

Außerdem ist $g_1(n) \leq g(n)$ und folglich $g_2(n) = g(n) - g_1(n)$ stets größer gleich 0. Behauptung: $g_2 \in O(f_2)$. 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 $g_2 \in O(f_2)$. Also ist $g = g_1 + g_2 \in O(f_1) + O(f_2)$. ■

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 $g \in O(f)$, dann ist auch $O(g) \subseteq O(f)$ und $O(g + f) = O(f)$.

Es gibt noch eine Reihe weiterer Rechenregeln für $O(\cdot)$ 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:

$$\begin{array}{cc|cc} & & b_{11} & b_{12} \\ & & b_{21} & b_{22} \\ \hline a_{11} & a_{12} & a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21} & a_{22} & a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{array}$$

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:

$$\begin{array}{cc|cc} & & B_{11} & B_{12} \\ & & B_{21} & B_{22} \\ \hline A_{11} & A_{12} & A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21} & A_{22} & A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array}$$

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 $N_{mult}(n) = 8 \cdot N_{mult}(n/2)$:

$$\begin{aligned} N_{mult}(2^k) &= 8 \cdot N_{mult}(2^{k-1}) = 8 \cdot 8 \cdot N_{mult}(2^{k-2}) = \dots = 8^k \cdot N_{mult}(1) \\ &= 8^k = 8^{\log_2(n)} = 2^{3 \log_2(n)} = 2^{\log_2(n) \cdot 3} = n^3 \end{aligned}$$

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:

$$\begin{aligned} N_{add}(2^k) &= 8 \cdot N_{add}(2^{k-1}) + 4^k \\ &= 8 \cdot 8 \cdot N_{add}(2^{k-2}) + 8 \cdot 4^{k-1} + 4^k = \dots \\ &= 8 \cdot 8 \cdot N_{add}(2^{k-2}) + 2 \cdot 4^k + 4^k = \dots \\ &= 8^k N_{add}(2^0) + (2^{k-1} + \dots + 1) \cdot 4^k = \\ &= 2^k \cdot 4^k \cdot 1 + (2^k - 1) \cdot 4^k = \\ &= 2 \cdot 2^k \cdot 4^k - 4^k = 2n^3 - n^2 \end{aligned}$$

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:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_2 = (A_{21} + A_{22})B_{11}$$

$$M_3 = A_{11}(B_{12} - B_{22})$$

$$M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{12})B_{22}$$

$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

und dann

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

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:

$$\begin{aligned} N_{mult}(2^k) &= 7 \cdot N_{mult}(2^{k-1}) = 7 \cdot 7 \cdot N_{mult}(2^{k-2}) = \dots = 7^k \cdot N_{mult}(1) \\ &= 7^k = 7^{\log_2(n)} = 2^{\log_2 7 \cdot \log_2(n)} = n^{\log_2 7} \approx n^{2.807\dots} \end{aligned}$$

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 $\Theta(n^{\log_2 7}) + \Theta(n^{\log_2 7}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807\dots})$.

Es gibt sogar Algorithmen, die asymptotisch noch weniger Operationen benötigen. Das in dieser Hinsicht beste Ergebnis stammt von Coppersmith und Winograd (1990), die mit $O(n^{2.376\dots})$ elementaren arithmetischen Operationen auskommen. Auch dieses Verfahren benutzt wie das von Strasse eine Vorgehensweise, die man in vielen Algorithmen wiederfindet: Man teilt die Problem Instanz 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*).

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 $f(n) \in O(n^{\log_b a - \varepsilon})$ für ein $\varepsilon > 0$ ist, dann ist $T(n) \in \Theta(n^{\log_b a})$.

FALL 2: Wenn $f(n) \in \Theta(n^{\log_b a})$ ist, dann ist $T(n) \in \Theta(n^{\log_b a} \log n)$.

FALL 3: Wenn $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ 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 tiefeschürfenden 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 $f(n) \in O(n^{3-\varepsilon})$ (z. B. für $\varepsilon = 1/2$) und der erste Fall des Mastertheorems besagt, dass folglich $T(n) \in \Theta(n^3)$. (Und das hatten wir uns vorhin 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 $\log_b a = \log_2 7 \approx 2.807 \dots$. 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 $f(n) \in O(n^{\log_b a - \varepsilon}) = O(n^{\log_2 7 - \varepsilon})$. Folglich benötigt Strassens Algorithmus $T(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2.807\dots})$ Zeit.

13.5 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 Auswahl-tasten 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 Auswahl-tasten 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 das gewünschte Getränk ausgeworfen.

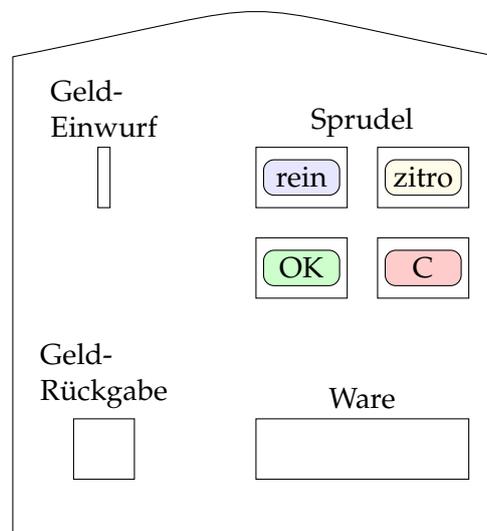


Abbildung 14.1: Ein primitiver Getränkeautomat

Dieser Getränkeautomat 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 ??) 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 $y \in \{-, R, Z\}$ die Getränkewahl repräsentiert. Wir wollen $Z = \{0, 1\} \times \{-, R, Z\}$ 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 $f : Z \times X \rightarrow Z$ 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.

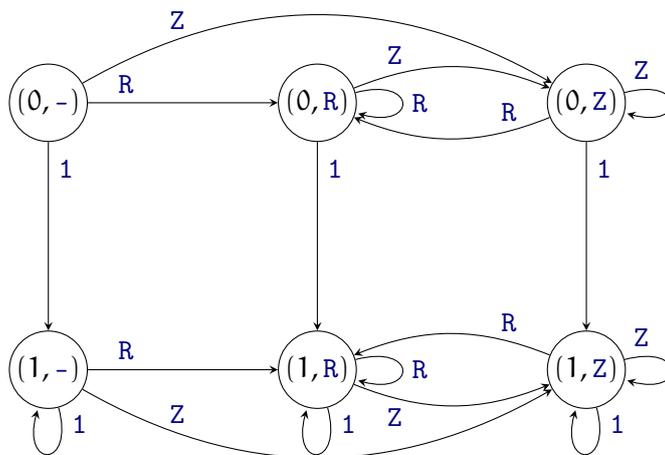


Abbildung 14.2: Graphische Darstellung der Zustandsübergänge des Getränkeautomaten für die drei Eingabesymbole **1**, **R** und **Z**.

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 **0**. Wenn bei einem Zustand

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 $g : Z \times X \rightarrow Y^*$. 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 Eingabesymbol, von diesem durch einen senkrechten Strich getrennt (manche nehmen auch ein Komma). Aus Abbildung 14.4 ergibt sich Abbildung 14.5.

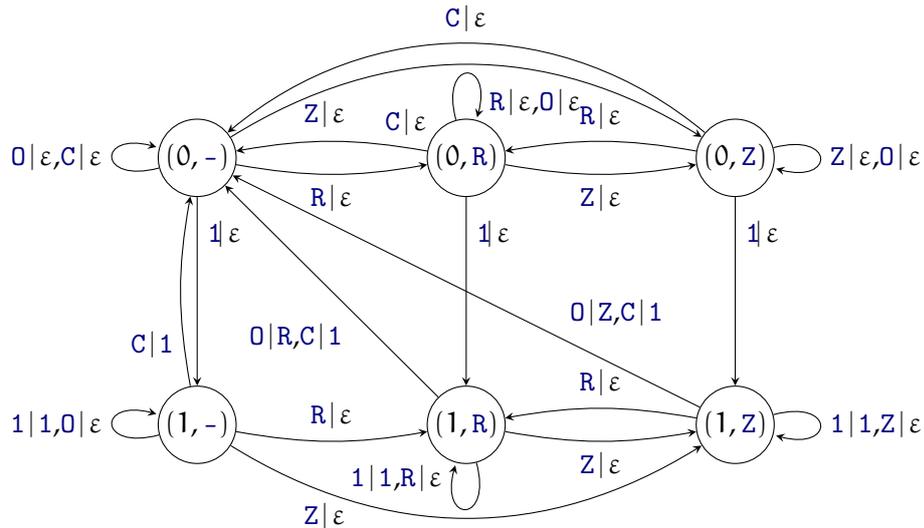


Abbildung 14.5: Graphische Darstellung der Zustandsübergänge und Ausgaben des Getränkeautomaten für alle Eingabesymbole.

14.2 MEALY-AUTOMATEN

Ein (endlicher) Mealy-Automat $A = (Z, z_0, X, f, Y, g)$ ist festgelegt durch

Mealy-Automat

- eine endliche Zustandsmenge Z ,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X ,
- eine Zustandsüberföhrungsfunktion $f : Z \times X \rightarrow Z$,
- ein Ausgabealphabet Y ,
- eine Ausgabefunktion $g : Z \times X \rightarrow Y^*$

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 (einschließlich 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 erstes Argument nicht ein Zustand sondern ein ganzes Wort von Zuständen ist; und der zweite Stern soll gegebenenfalls andeuten, dass wir uns nicht für einen einzelnen Funktionswert (von f bzw. g) interessieren, sondern wiederum für eine ganze Reihe von ihnen. Als erstes legen wir $f^* : Z \times X^* \rightarrow Z$ fest:

f^*

$$f^*(z, \varepsilon) = z$$

$$\forall w \in X^* : \forall x \in X : f^*(z, wx) = f(f^*(z, w), x)$$

Alternativ hätte man auch definieren können:

$$\bar{f}^*(z, \varepsilon) = z$$

$$\forall w \in X^* : \forall x \in X : \bar{f}^*(z, xw) = \bar{f}^*(f(z, x), w)$$

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.)

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 $f^{**} : Z \times X^* \rightarrow Z^*$ wie folgt fest:

f^{**}

$$f^{**}(z, \varepsilon) = z$$

$$\forall w \in X^* : x \in X : f^{**}(z, wx) = f^{**}(z, w) \cdot f(f^*(z, w), x)$$

Auch hier gibt es wieder eine alternative Definitionsmöglichkeit, indem man nicht das letzte, sondern das erste Symbol des Eingabewortes separat betrachtet. Nun zu den verallgemeinerten Ausgabefunktionen. Wir definieren $g^* : Z \times X^* \rightarrow Y^*$ so:

g^*

$$g^*(z, \varepsilon) = \varepsilon$$

$$g^*(z, wx) = g(f^*(z, w), x)$$

Die Funktion $g^{**} : Z \times X^* \rightarrow Y^*$ definiert man so:

g^{**}

$$g^{**}(z, \varepsilon) = \varepsilon$$

$$g^{**}(z, wx) = g^{**}(z, w) \cdot g^*(z, wx)$$

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

Moore-Automat

- eine endliche Zustandsmenge Z ,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X ,
- eine Zustandsüberföhrungsfunktion $f : Z \times X \rightarrow Z$,
- ein Ausgabealphabet Y ,
- eine Ausgabefunktion $h : Z \rightarrow Y^*$

Als einfaches Beispiel betrachten wir den Automaten in Abbildung 14.6 mit 5 Zuständen, Eingabealphabet $X = \{a, b\}$ und Ausgabealphabet $Y = \{0, 1\}$.

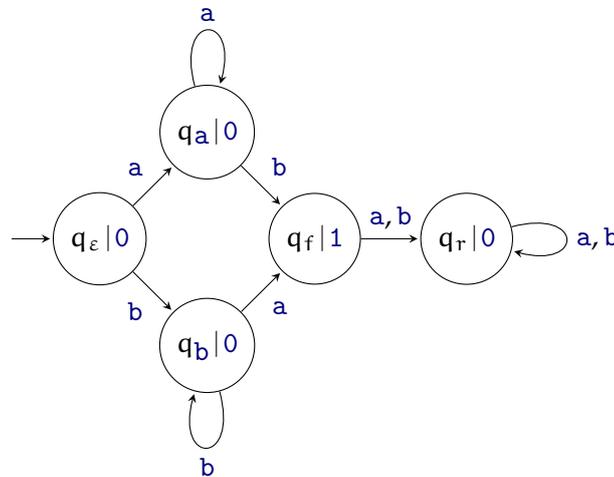


Abbildung 14.6: Ein einfacher Moore-Automat (aus der Dokumentation des \LaTeX -Pakets `tikz`; modifiziert)

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 $f^*(q_\epsilon, \mathbf{aaaba}) = q_r$, denn bei Eingabe \mathbf{aaaba} durchläuft der Automat ausgehend von q_ϵ nacheinander die Zustände

 f^*
 f^{**}

$$q_\epsilon \xrightarrow{a} q_a \xrightarrow{a} q_a \xrightarrow{a} q_a \xrightarrow{b} q_f \xrightarrow{a} q_r$$

Und folglich ist auch $f^{**}(q_\epsilon, \mathbf{aaaba}) = q_\epsilon q_a q_a q_a q_f q_r$.

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. Z. B. ist $g^* : Z \times X^* \rightarrow Y^*$ 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

$$g^*(q_\varepsilon, \mathbf{aaaba}) = h(f^*(q_\varepsilon, \mathbf{aaaba})) = h(q_r) = 0$$

das zuletzt ausgegebene Bit, wenn man vom Startzustand ausgehend \mathbf{aaaba} eingibt.

Auch $g^{**} : Z \times X^* \rightarrow Y^*$ 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. (An jener Stelle haben wir das Skript aktualisiert und notieren jetzt h^{**} statt h^* .) Die Ausgabeabbildung $h : Z \rightarrow Y^*$ induziert einen Homomorphismus $h^{**} : Z^* \rightarrow Y^*$ (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 $g^{**} = h^{**} \circ f^{**}$. g^{**}

In unserem Beispiel ist

$$\begin{aligned} g^{**}(q_\varepsilon, \mathbf{aaaba}) &= h^{**}(f^{**}(q_\varepsilon, \mathbf{aaaba})) \\ &= h^{**}(q_\varepsilon q_a q_a q_a q_f q_r) \\ &= h(q_\varepsilon)h(q_a)h(q_a)h(q_a)h(q_f)h(q_r) \\ &= 000010 \end{aligned}$$

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*.

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 Zustandsüberföhrungsfunktion $f : Z \times X \rightarrow Z$,
- eine Menge $F \subseteq Z$ akzeptierender Zustände

*akzeptierender
Zustand*
*ablehnender
Zustand*
endlicher Akzeptor

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.

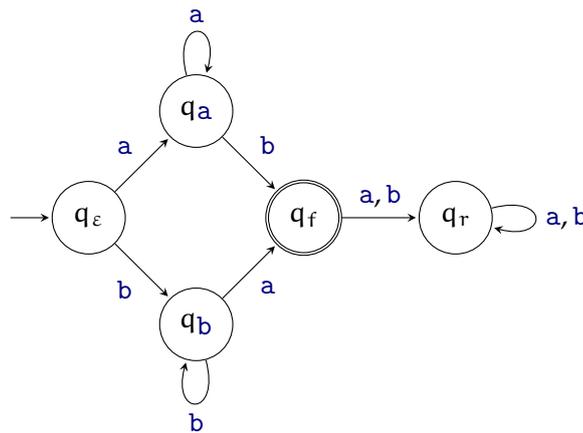


Abbildung 14.7: Ein einfacher Akzeptor (aus der Dokumentation des \LaTeX -Pakets `tikz`; modifiziert)

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 $f^*(z_0, \mathbf{aaaba}) = q_r \notin F$ 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 $a^k b$ für ein $k \in \mathbb{N}_+$. Und es werden auch alle Wörter akzeptiert, die von der Form $b^k a$ 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

$$L(A) = \{a\}^+ \{b\} \cup \{b\}^+ \{a\},$$

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.

Es gibt also (natürlich) formale Sprachen, 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

$$L = \{a^k b^k \mid k \in \mathbb{N}_0\}.$$

14.1 Lemma. Es gibt keinen endlichen Akzeptor A mit

$$L(A) = \{a^k b^k \mid k \in \mathbb{N}_0\}.$$

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 $L = \{a^k b^k \mid k \in \mathbb{N}_0\}$ 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 $w = a^m b^m$.

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: $f^*(z_0, \varepsilon) = z_0$, $f^*(z_0, a)$, $f^*(z_0, aa)$, \dots , $f^*(z_0, a^m)$. Nennen wir diese Zustände allgemein z_i , d. h. für $0 \leq i \leq m$ ist $z_i = f^*(z_0, a^i)$. Mit anderen Worten: $f^{**}(z_0, a^m) = z_0 z_1 \cdots z_m$.

Offensichtlich gilt dann: $f^*(z_m, a^m) = z_f$.

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 a als Eingabe erhält. Also ist auch $z_{m-\ell} = z_m$.

3. Nun entfernen wir einige der a in der Eingabe, so dass die Schleife einmal weniger durchlaufen wird, d. h. wir betrachten die Eingabe $w' = a^{m-\ell} b^m$. Wie verhält sich der Akzeptor bei dieser Eingabe? Nachdem er das Präfix $a^{m-\ell}$ 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 a^m ist. Und wir wissen: $f^*(z_m, b^m) = z_f \in F$. Also ist $f^*(z_0, a^{m-\ell} b^m) = f^*(f^*(z_0, a^{m-\ell}), b^m) = f^*(z_{m-\ell}, b^m) = f^*(z_m, b^m) = z_f$, d. h. A akzeptiert die Eingabe $w' = a^{m-\ell} b^m$. Aber das Wort w' gehört nicht zu L , da es verschieden viele a und 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 $f : Z \times X \rightarrow Z$ 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](#).

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, Perl) 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 6 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:

- \emptyset ist ein regulärer Ausdruck.
- Für jedes $a \in A$ ist a ein regulärer Ausdruck.
- Wenn R_1 und R_2 reguläre Ausdrücke sind, dann sind auch $(R_1 | R_2)$ und $(R_1 R_2)$ 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. $R_1 | R_2 R_3^*$ ist z. B. als $(R_1 | (R_2 (R_3^*)))$ zu verstehen. Bei mehreren gleichen binären Operatoren gilt das

regulärer Ausdruck

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

$$G = (\{R\}, \{ |, (,), *, \emptyset \} \cup A, R, P)$$

$$\text{und } P = \{R \rightarrow \emptyset, R \rightarrow (R|R), R \rightarrow (RR), R \rightarrow (R*)\}$$

$$\cup \{R \rightarrow a \mid a \in A\}$$

erzeugt werden.

Die folgenden Zeichenketten sind alle reguläre Ausdrücke über dem Alphabet $\{0, 1\}$:

- | | | |
|--------------------|---------------------------------------|---------------------|
| (a) \emptyset | (b) 0 | (c) 1 |
| (d) (01) | (e) $((01)0)$ | (f) $((01)0)0$ |
| (g) $((01)(00))$ | (h) $(\emptyset 1)$ | (i) $(0 1)$ |
| (j) $((0(0 1)) 1)$ | (k) $(0 (1 (0 0)))$ | (l) (\emptyset^*) |
| (m) (0^*) | (n) $((10)(1^*))$ | (o) $((10)1)^*$ |
| (p) $((0^*)^*)$ | (q) $(((((01)1)^*)^*) (\emptyset^*))$ | |

Wendet man die Klammereinsparungsregeln an, so ergibt sich aus den Beispielen mit Klammern:

- | | | |
|----------------|------------------------------|-------------------|
| (d) 01 | (e) 010 | (f) 0100 |
| (g) $01(00)$ | (h) $\emptyset 1$ | (i) $0 1$ |
| (j) $0(0 1) 1$ | (k) $(0 (1 (0 0)))$ | (l) \emptyset^* |
| (m) 0^* | (n) 101^* | (o) $(101)^*$ |
| (p) 0^{**} | (q) $(011)^{**} \emptyset^*$ | |

Die folgenden Zeichenketten sind dagegen auch bei Berücksichtigung der Klammereinsparungsregeln *keine* regulären Ausdrücke über $\{0, 1\}$:

- $(|1)$ falsch: vor $|$ fehlt ein regulärer Ausdruck;
- $|\emptyset|$ falsch: vor und hinter $|$ fehlt je ein regulärer Ausdruck;
- (01) falsch: zwischen $($ und $)$ fehlt ein regulärer Ausdruck;
- $((01)$ falsch: Klammern müssen „in der üblichen Weise gepaart“ auftreten;
- $*(01)$ falsch: vor $*$ fehlt ein regulärer Ausdruck;
- 2^* falsch: 2 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 $a \in A$ ist $\langle a \rangle = \{ a \}$.

- 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:

- $R = a|b$: Dann ist $\langle R \rangle = \langle a|b \rangle = \langle a \rangle \cup \langle b \rangle = \{a\} \cup \{b\} = \{a, b\}$.
- $R = (a|b)^*$: Dann ist $\langle R \rangle = \langle (a|b)^* \rangle = \langle a|b \rangle^* = \{a, b\}^*$.
- $R = (a^*b^*)^*$: Dann ist $\langle R \rangle = \langle (a^*b^*)^* \rangle = \langle a^*b^* \rangle^* = (\langle a^* \rangle \langle b^* \rangle)^* = (\{a\}^* \{b\}^*)^*$.

Mehr oder weniger kurzes Überlegen zeigt übrigens, dass für die Sprachen des zweiten und dritten Beispiels gilt: $(\{a\}^* \{b\}^*)^* = \{a, b\}^*$. 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 [Einheit 16 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.

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.

reguläre 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 A , B und C formuliert.
- Es wird behauptet:
 - $A \iff B$
 - $B \iff C$
 - $C \iff A$

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 $A \implies B$ gilt und dass $B \implies C$, dann folgt $A \implies C$ automatisch. Das sieht man anhand der folgenden Tabelle:

	A	B	C	$A \implies B$	$B \implies C$	$A \implies C$
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 $A \implies B$ als auch für $B \implies C$ ein W (für *wahr*) eingetragen ist, ist das auch für $A \implies C$ der Fall. Statt *falsch* haben wir der besseren Übersicht wegen die entsprechenden Felder freigelassen.

Wenn man $A \implies B$ und $B \implies C$ schon bewiesen hat, dann muss man also $A \implies C$ gar nicht mehr beweisen. Und beweist man nun zusätzlich noch $C \implies A$, dann

- folgt mit $A \implies B$ sofort $C \implies B$ und
- mit $B \implies C$ folgt sofort $B \implies A$,

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“.

- 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 $L' = \{a,b\}^* \setminus L$, 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 Akzeptoren erkennen kann. Denn die Grammatik $G = (\{X\}, \{a,b\}, X, \{X \rightarrow aXb \mid \varepsilon\})$ erzeugt $\{a^k b^k \mid k \in \mathbb{N}_0\}$ 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 $X \rightarrow w$ oder von der Form $X \rightarrow wY$ mit $w \in T^*$ und $X, Y \in N$. Auf der rechten Seite einer Produktion darf also höchstens ein Nichtterminalsymbol vorkommen, und wenn dann nur als letztes Symbol.

rechtslineare Grammatik

Die oben erwähnte Grammatik $G = (\{X\}, \{a,b\}, X, \{X \rightarrow aXb \mid \varepsilon\})$ ist also *nicht* rechtslinear, denn in der Produktion $X \rightarrow aXb$ 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 $\{a^k b^k \mid k \in \mathbb{N}_0\}$ erzeugt.

Zum Abschluss sei 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*.

Typ-3-Grammatiken

Typ-2-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* .

Beweise für die Behauptungen aus Satz 15.1 werden Sie vielleicht in der Vorlesung „Theoretische 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.

`a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|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*. Hg. 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).

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.

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 Decodierung 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>.

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>, 14.1.09).

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 gelesenen Symbol wie der neue Zustand der Steuereinheit.

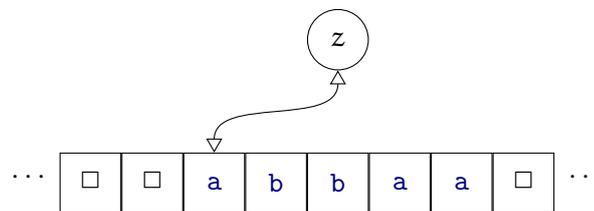


Abbildung 16.1: schematische Darstellung einer (einfachen) Turingmaschine

Formal kann man sich also eine *Turingmaschine* $T = (Z, z_0, X, f, g, m)$ festgelegt vorstellen durch

Turingmaschine

- eine Zustandsmenge Z
- einen Anfangszustand $z_0 \in Z$
- ein Bandalphabet X
- eine partielle Zustandsüberföhrungsfunktion $f : Z \times X \dashrightarrow Z$
- 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 .

Eine Turingmaschine befindet sich zu jedem Zeitpunkt in einem „Gesamtzustand“, den wir eine *Konfiguration* nennen wollen. Sie ist vollständig beschrieben durch

Konfiguration

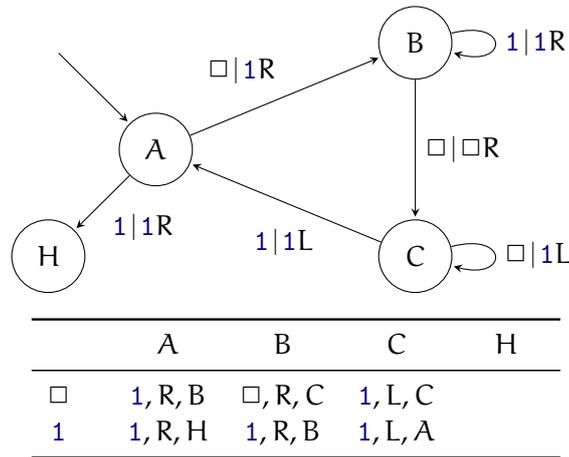


Abbildung 16.2: Zwei Spezifikationsmöglichkeiten der gleichen Turingmaschine; sie heißt BB₃.

- den aktuellen Zustand $z \in Z$ der Steuereinheit,
- die aktuelle Beschriftung des gesamten Bandes, die man als Abbildung $b : \mathbb{Z} \rightarrow X$ formalisieren kann, und
- die aktuelle Position $p \in \mathbb{Z}$ des Kopfes.

Eine Bandbeschriftung ist also ein potenziell unendliches „Gebilde“. Wie aber schon in Abschnitt 5.2 erwähnt und zu Beginn dieser Einheit noch ein 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.

Blanksymbol

16.2.1 Berechnungen

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 gelesenen 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:

Schritt einer Turingmaschine

- $z' = f(z, b(p))$
- $\forall i \in \mathbb{Z} : b'(i) = \begin{cases} b(i) & \text{falls } i \neq p \\ g(z, b(p)) & \text{falls } i = p \end{cases}$
- $p' = p + m(z, b(p))$

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.

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*.

Endkonfiguration
Halten

Die Turingmaschine aus Abbildung 16.2 wollen wir BB_3 nennen. Wenn BB_3 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:

A							
□	□	□	□	□	□	□	□
B							
□	□	□	1	□	□	□	□

Dabei haben wir den Zustand der Turingmaschine jeweils über dem gerade besuchten Bandfeld notiert. In der entstandenen Konfiguration kann BB_3 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	□	□
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.

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 Berechnung ist und ihre letzte Konfiguration eine Endkonfiguration ist.

endliche Berechnung

haltende 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*.

unendliche Berechnung

nicht haltende Berechnung

Eine nicht haltende Berechnungen 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

$$\begin{aligned} \Delta_0 &= \text{Id} \\ \Delta_{t+1} &= \Delta_1 \circ \Delta_t \end{aligned}$$

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

Entscheidungsproblem

„Berechnung von Funktionen“, bei denen der Funktionswert aus einem größeren Bereich als nur $\{0, 1\}$ kommt.

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 Blankensymbol gehört also nie zum 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

Eingabealphabet

$$b_w : \mathbb{Z} \rightarrow X$$

$$b_w(i) = \begin{cases} \square & \text{falls } i < 0 \vee i \geq |w| \\ w(i) & \text{falls } 0 \leq i \wedge i < |w| \end{cases}$$

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 $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$, 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 *akzeptierenden 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 akzeptierender 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

Wenn ein Wort w von einer Turingmaschine *nicht* akzeptiert wird, dann gibt es dafür zwei mögliche Ursachen:

Turingmaschinenakzeptor

- 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 ist. Dass das eine echt stärkere Eigenschaft ist als Aufzählbarkeit werden wir in Abschnitt 16.5 ansprechen.

entscheiden
entscheidbare
Sprache

Als Beispiel betrachten wir die Aufgabe, für jedes Eingabewort $w \in L = \{a, b\}^*$ festzustellen, ob es ein Palindrom ist oder nicht. Um das zu belegen, muss man einen Turingmaschinenakzeptor finden, der genau L entscheidet. In Abbildung 16.3 ist eine solche Turingmaschine angegeben. Ihr Anfangszustand ist l 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.

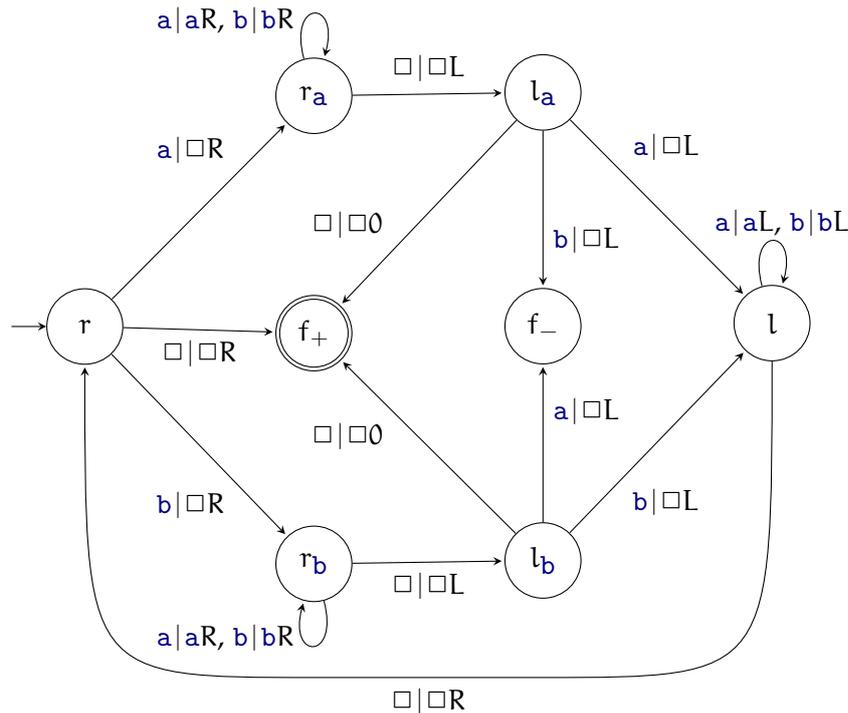


Abbildung 16.3: Eine Turingmaschine zur Palindromerkennung; f_+ sei der einzige akzeptierende Zustand

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.

16.3 BERECHNUNGSKOMPLEXITÄT

Wir beginnen mit einem wichtigen Hinweis: Der Einfachheit halber wollen wir in diesem Abschnitt und im nachfolgenden Abschnitt 16.4 davon ausgehen, dass wir ausschließlich mit Turingmaschinen zu tun haben, die für jede Eingabe halten. Warum das „in Ordnung“ ist, erfahren Sie vielleicht einmal in einer Vorlesung über Komplexitätstheorie.

In Abschnitt 16.5 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 hoffentlich schon in Abschnitt 16.5 verstehen. Viel mehr zu diesem Thema werden Sie vielleicht einmal in einer Vorlesung über Berechenbarkeit bzw. Rekursionstheorie hören.

16.3.1 Komplexitätsmaße

Bei Turingmaschinen kann man leicht zwei sogenannte *Komplexitätsmaße* definieren, die Rechenzeit und Speicherplatzbedarf charakterisieren.

Komplexitätsmaß

Für die Beurteilung des Zeitbedarfs definiert man zwei Funktionen $\text{time}_T : A^+ \rightarrow \mathbb{N}_+$ und $\text{Time}_T : \mathbb{N}_+ \rightarrow \mathbb{N}_+$ wie folgt:

$$\text{time}_T(w) = \text{dasjenige } t \text{ mit } \Delta_t(c_0(w)) = \Delta_*(c_0(w))$$

$$\text{Time}_T(n) = \max\{\text{time}(w) \mid w \in A^n\}$$

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 $\text{Time}_T(n) \in O(p(n))$.

Zeitkomplexität

*polynomielle
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
- für das Teilwort der Länge $n - 2$ ohne die Randsymbole wieder einen Palindromtest machen.

Der erste Teil erfordert $2n - 1$ Schritte. Für den Zeitbedarf $\text{Time}(n)$ gilt also jedenfalls:

$$\text{Time}(n) \leq 2n + 1 + \text{Time}(n - 2)$$

Der Zeitaufwand für Wörter der Längen 0 und 1 ist jedenfalls durch eine Konstante c beschränkt. 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 $\text{space}_T(w) : A^+ \rightarrow \mathbb{N}_+$ $\text{Space}_T(n) : \mathbb{N}_+ \rightarrow \mathbb{N}_+$ wie folgt:

$\text{space}_T(w)$ = die Anzahl der Felder, die während der Berechnung für Eingabe w benötigt werden

$$\text{Space}_T(n) = \max\{\text{space}(w) \mid w \in A^n\}$$

Üblicherweise heißt die Abbildung Space_T die *Raumkomplexität* oder *Platzkomplexität* der Turingmaschine T . Dabei gelte ein Feld

*Raumkomplexität
Platzkomplexität*

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 Beispielmachine zur Palindromerkennung hat Platzbedarf $\text{Space}(n) = n + 2 \in \Theta(n)$, weil außer den n Feldern mit den Eingabesymbolen nur noch jeweils ein weiteres Feld links bzw. rechts davon besucht werden.

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 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

Eine *Komplexitätsklasse* ist eine Menge von Problemen. Wir beschränken uns im folgenden wieder auf Entscheidungsprobleme, also formale Sprachen. Charakterisiert werden Komplexitätsklasse 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 $O(n^2)$ und die Raumkomplexität in $O(n^{3/2} \log n)$ ist, wobei hier n wieder für die Größe der Problem Instanz, also die Länge des Eingabewortes, steht.

Komplexitätsklasse

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 $O(n^2)$ Schritte, um festzustellen, ob w Palindrom ist.

Ein Beispiel eines Entscheidungsproblem 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 polynomiell 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 polynomiell Platzbedarf auch dann, wenn sie exponentielle Laufzeit hat, eine andere Turingmaschine mit nur polynomiell 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 SCHWERE PROBLEME

Es seien L_1 und L_2 zwei formale Sprachen über dem gleichen Alphabet A (oder mit anderen Worten: Entscheidungsprobleme). Man sagt, dass L_1 auf L_2 *reduzierbar* ist, wenn es eine in Polynomialzeit berechenbare Funktion $f : A^* \rightarrow A^*$ gibt mit der Eigenschaft: $w \in L_1 \iff f(w) \in L_2$. (Eine solche Funktion kann auch wieder nur polynomiell viel Platz benötigen.) Wenn man das hat und z. B. $L_2 \in \mathbf{P}$ ist, dann kann man auch L_1 in Polynomialzeit entscheiden: Für jede Eingabe w

L_1 auf L_2
reduzierbar

- berechnet man zuerst $f(w)$ und
- überprüft dann ob $f(w) \in L_2$ ist.

Beides geht in Polynomialzeit, also ist dann auch $L_1 \in \mathbf{P}$.

Und wenn $L_2 \in \mathbf{PSPACE}$ ist, dann folgt aus der gleichen Konstruktion, dass auch $L_1 \in \mathbf{PSPACE}$ ist.

Ein Problem C heißt **PSPACE-hart** oder **PSPACE-schwer** wenn es für jedes Problem $L \in \mathbf{PSPACE}$ eine Polynomialzeitreduktion auf C gibt. Ein Problem heißt **PSPACE-vollständig**, falls es **PSPACE-hart** ist und selbst in **PSPACE** liegt. **PSPACE-vollständige** Probleme gehören also sozusagen „zu den schwersten in **PSPACE**“.

PSPACE-hart
PSPACE-schwer
PSPACE-vollständig

Solche Probleme gibt es. Man kennt eine ganze Reihe von ihnen, z.B. die schon erwähnte Überprüfung, ob zwei reguläre Ausdrücke die gleiche formale Sprache beschreiben.

Eine Folge der eben gemachten Überlegungen ist: Wenn man nur für ein einziges **PSPACE**-vollständiges Problem einen Polynomialzeitalgorithmus findet, dann kann man *alle* Probleme aus **PSPACE** in Polynomialzeit entscheiden. Der etwas kuriose Stand der Wissenschaft **PSPACE**-vollständige Probleme betreffend ist: Man kennt für kein einziges einen Polynomialzeit-Algorithmus, man kann aber bislang auch nicht beweisen, dass keine existieren.

Damit Sie aus den vorangegangenen Abschnitten nicht falsche Schlüsse ziehen, wollen wir zum Abschluss noch folgendes mitteilen: Man kann beweisen, dass es noch viel schwierigere Probleme gibt. Z. B. kennt man Probleme, für die jede Turingmaschine, die eines von ihnen löst, Laufzeit 2^{2^n} hat. Und Analoges gilt für höhere „Türme“ von Exponenten.

16.5 UNENTSCHEIDBARE PROBLEME

In gewisser Weise noch schlimmer als Probleme, die exorbitanten Ressourcenaufwand 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.

16.5.1 Codierungen von Turingmaschinen

Zunächst soll eine Möglichkeit beschrieben werden, wie man jede Turingmaschine durch ein Wort über dem festen Alphabet $A = \{[,], 0, 1\}$ beschreiben kann. Natürlich kann man diese Symbole durch Wörter über $\{0, 1\}$ 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 $\text{cod}_Z(z)$ 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 $\text{cod}_X(x)$ 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 $\text{cod}_M(r)$ 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 $\text{cod}_{fgm}(z, x) = [\text{cod}_Z(z) \text{cod}_X(x) \square \square \square]$.
 - Wenn sie für ein Argumentpaar (z, x) definiert sind, wird das codiert durch das Wort $\text{cod}_{fgm}(z, x) = [\text{cod}_Z(z) \text{cod}_X(x) \text{cod}_Z(f(z, x)) \text{cod}_X(g(z, x)) \text{cod}_M(m(z, x))]$.
 - Die Codierung der gesamten Funktionen ist die Konkatination aller $\text{cod}_{fgm}(z, x)$ für alle $z \in Z$ und alle $x \in X$.
- Die gesamte Turingmaschine wird codiert als Konkatination 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.

Tatsächlich kann man sogar noch mehr: Es gibt sogenannte *universelle Turingmaschinen*. Eine universelle Turingmaschine U

*universelle
Turingmaschine*

- 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.

16.5.2 Das Halteproblem

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>, 25.1.09), 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.

Diagonalisierung

	x_0	x_1	x_2	x_3	x_4	\dots
f_0	$f_0(x_0)$	$f_0(x_1)$	$f_0(x_2)$	$f_0(x_3)$	$f_0(x_4)$	\dots
f_1	$f_1(x_0)$	$f_1(x_1)$	$f_1(x_2)$	$f_1(x_3)$	$f_1(x_4)$	\dots
f_2	$f_2(x_0)$	$f_2(x_1)$	$f_2(x_2)$	$f_2(x_3)$	$f_2(x_4)$	\dots
f_3	$f_3(x_0)$	$f_3(x_1)$	$f_3(x_2)$	$f_3(x_3)$	$f_3(x_4)$	\dots
f_4	$f_4(x_0)$	$f_4(x_1)$	$f_4(x_2)$	$f_4(x_3)$	$f_4(x_4)$	\dots
\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“) \bar{d} ,

	x_0	x_1	x_2	x_3	x_4	\dots
d	$f_0(x_0)$	$f_1(x_1)$	$f_2(x_2)$	$f_3(x_3)$	$f_4(x_4)$	\dots
\bar{d}	$\overline{f_0(x_0)}$	$\overline{f_1(x_1)}$	$\overline{f_2(x_2)}$	$\overline{f_3(x_3)}$	$\overline{f_4(x_4)}$	\dots

die sich von jeder Abbildung („Zeile“) der gegebenen Tabelle unterscheidet, denn für alle i ist

$$\bar{d}(x_i) = \overline{f_i(x_i)} \neq f_i(x_i).$$

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 Satz. *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, deren Codierung x_i ist, 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“ \bar{d} ähnlich wie oben mit den beiden folgenden sich widersprechenden Eigenschaften.

- Einerseits unterscheidet sich \bar{d} von jeder Zeile der Tabelle, also von jeder von einer Turingmaschine berechneten Funktion.
- Andererseits kann auch \bar{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 $T_{\bar{d}}$ realisieren:

- Für eine Eingabe x_i berechnet $T_{\bar{d}}$ zunächst, welches Ergebnis T_h für diese Eingabe liefern würde.
- Dann arbeitet $T_{\bar{d}}$ wie folgt weiter:
 - Wenn T_h mitteilt, dass $T_{x_i}(x_i)$ hält, dann geht $T_{\bar{d}}$ in eine Endlosschleife.
 - Wenn T_h mitteilt, dass $T_{x_i}(x_i)$ nicht hält, dann hält $T_{\bar{d}}$ (und liefert irgendein Ergebnis, etwa 0).
- Andere Möglichkeiten gibt es nicht, und in beiden Fällen ist das Verhalten von $T_{\bar{d}}$ 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 $T_{\bar{d}}$, aber jede Turingmaschine (T_{x_i}) verhält sich für mindestens eine Eingabe (x_i) anders als $T_{\bar{d}}$.

Das ist ein Widerspruch. Folglich war die Annahme, dass es die Turingmaschine T_h gibt, die H entscheidet, falsch. ■

16.5.3 Die Busy-Beaver-Funktion

In Abschnitt 16.2 hatten wir BB_3 als erstes Beispiel einer Turingmaschine gesehen. Diese Turingmaschine hat folgende Eigenschaften:

- Bandalphabet ist $X = \{\square, 1\}$.
- 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 $X = \{\square, 1\}$.
- 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 BB_3 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

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

$$bb : \mathbb{N}_+ \rightarrow \mathbb{N}_+$$

$bb(n)$ = die Anzahl von Einsen, die ein fleißiger Biber mit $n + 1$ Zuständen am Ende auf dem Band hinterlässt

Diese Funktion wird auch *Radó-Funktion* genannt nach dem ungarischen Mathematiker Tibor Radó, der sich als erster mit dieser Funktion beschäftigte. Statt $bb(n)$ wird manchmal auch $\Sigma(n)$ geschrieben.

Radó-Funktion

Da BB_3 am Ende 6 Einsen auf dem Band hinterlässt, ist also jedenfalls $bb(3) \geq 6$. Radó fand heraus, dass sogar $bb(3) = 6$ ist. Die Turingmaschine BB_3 ist also ein fleißiger Biber.

Brady hat 1983 gezeigt, dass $bb(4) = 13$ ist. Ein entsprechender fleißiger Biber ist

	A	B	C	D	H
□	1, R, B	1, L, A	1, R, H	1, R, D	
1	1, L, B	□, L, C	1, L, D	□, R, A	

H. Marxen (<http://www.drb.insel.de/~heiner/BB/>, 24.1.09) und J. Buntrock haben 1990 eine 5-Bibermaschine gefunden, die 4098 Einsen produziert und nach 47 176 870 Schritten hält. Also ist $bb(5) \geq 4098$. 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 $bb(6) \geq 4.6 \cdot 10^{1439}$. 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>, 24.1.09) teilen wir noch den folgenden Satz mit. In ihm und dem unmittelbaren Korollar wird von der Berechenbarkeit von Funktionen $\mathbb{N}_+ \rightarrow \mathbb{N}_+$ 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 Satz. Für jede berechenbare Funktion $f : \mathbb{N}_+ \rightarrow \mathbb{N}_+$ gibt es ein n_0 , so dass für alle $n \geq n_0$ gilt: $bb(n) > f(n)$.

16.4 Korollar. Die Busy-Beaver-Funktion $bb(n)$ ist nicht berechenbar.

16.6 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. So wie für **PSPACE** gibt es auch für **NP** vollständige Probleme. **NP**-vollständige Probleme spielen an noch viel mehr Stellen (auch in der Praxis) eine wichtige Rolle als **PSPACE**-vollständige. Und wieder ist es 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> (14.1.09).

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:

- $\forall x \in M : x \equiv x$,
- $\forall x \in M : \forall y \in M : x \equiv y \implies y \equiv x$
- $\forall x \in M : \forall y \in M : \forall z \in M : x \equiv y \wedge y \equiv z \implies x \equiv z$

Der Anlass für Symbole, die an das „=“ erinnern, ist natürlich der, dass Gleichheit, also die Relation $\text{Id} = \{(x, x) \mid x \in M\}$, auf jeder Menge eine Äquivalenzrelation ist, denn offensichtlich gilt:

- $\forall x \in M : x = x$,
- $\forall x \in M : \forall y \in M : x = y \implies y = x$
- $\forall x \in M : \forall y \in M : \forall z \in M : x = y \wedge y = z \implies x = z$

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

$$w_1 \equiv_L w_2 \iff (\forall w \in A^* : w_1 w \in L \iff w_2 w \in L)$$

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. (Aber ist eines in L und das andere nicht.) 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 $A = \{a, b\}$ und die formale Sprache $L = \langle a^* b^* \rangle \subset A^*$ 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 $w \in \langle a^* \rangle$ an, dann sind sowohl $w_1 w$ als auch $w_2 w$ in L .
- Hängt man an beide Wörter ein $w \in \langle a^* b b^* \rangle$ 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 $w \in \langle b^* \rangle$ 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 $w = a$ an, dann ist zwar $w_1 w = aaa \in L$, aber $w_2 w = abba \notin L$.
- Also sind die beiden Wörter *nicht* \equiv_L -äquivalent.

4. $w_1 = aba$ und $w_2 = babb$:

- Beide Wörter enthalten **ba**. Egal welches $w \in A^*$ man anhängt, bleibt das so, d. h. immer sind $w_1w \notin L$ und $w_2w \notin L$.
 - Also sind die beiden Wörter \equiv_L -äquivalent.
5. $w_1 = ab$ und $w_2 = ba$:
- 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: $w_1w \in L \iff w_1w \in L$.
- Symmetrie: Für $w_1, w_2 \in A^*$ und alle $w \in A^*$ gelte: $w_1w \in L \iff w_2w \in L$. Dann gilt offensichtlich auch immer $w_2w \in L \iff w_1w \in L$.
- Transitivität: Es seien $w_1, w_2, w_3 \in A^*$ und es möge gelten

$$\forall w \in A^* : w_1w \in L \iff w_2w \in L \quad (10)$$

$$\forall w \in A^* : w_2w \in L \iff w_3w \in L \quad (11)$$

Wir müssen zeigen: $\forall w \in A^* : w_1w \in L \iff w_3w \in L$. Sei dazu ein beliebiges $w \in A^*$ gegeben. Falls $w_1w \in L$ ist, dann ist wegen (10) auch $w_2w \in L$ und daher wegen (11) auch $w_3w \in L$. Analog folgt aus $w_1w \notin L$ der Reihe nach $w_2w \notin L$ und $w_3w \notin L$. Also gilt $w_1w \in L \iff w_3w \in L$.

■

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.

Für die Menge aller Äquivalenzklassen schreibt man $M_{/\equiv}$ und nennt das manchmal auch die *Faktormenge* oder *Faserung* von M nach \equiv , also $M_{/\equiv} = \{[x]_{\equiv} \mid x \in M\}$.

Mitunter ist es nützlich, sich anzusehen aus wievielen Äquivalenzklassen eine Faserung besteht. Nehmen wir als Beispiel wieder die durch $L = \langle a^*b^* \rangle$ 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 , b und ba äquivalent ist. Mit anderen Worten besteht $A^*_{/\equiv_L}$ aus drei Äquivalenzklassen:

- $[\varepsilon] = \langle a^* \rangle$
- $[b] = \langle a^*bb^* \rangle$
- $[ba] = \langle a^*bb^*a(a|b)^* \rangle$

Die Wahl der Repräsentanten in dieser Aufzählung ist natürlich willkürlich. Wir hätten genauso gut schreiben können:

Äquivalenzklasse

Faktormenge
Faserung

- $[aaaaa] = \langle a^* \rangle$
- $[aabbbbbb] = \langle a^*bb^* \rangle$
- $[aabbaabbba] = \langle a^*bb^*a(a|b)^* \rangle$

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

$$L = \{a^k b^k \mid k \in \mathbb{N}_0\}.$$

Für diese Sprache besteht $A_{\equiv_L}^*$ aus unendlich vielen Äquivalenzklassen. Ist nämlich $k \neq m$, dann sind $w_1 = a^k$ und $w_2 = a^m$ nicht äquivalent, wie man durch Anhängen von $w = b^k$ sieht:

- $w_1 w = a^k b^k \in L$, aber
- $w_2 w = a^m b^k \notin L$.

Also ist zumindest jedes Wort a^k , $k \in \mathbb{N}_0$ in einer anderen Äquivalenzklasse. Jede dieser Äquivalenzklassen ist aber ihrerseits unendlich groß, denn Sie können sich überlegen, dass

$$[a^k]_{\equiv_L} = \{a^x b^y \mid x, y \in \mathbb{N}_0 \wedge x - y = k\}$$

ist.

Vielleicht lässt die Tatsache, dass es für die reguläre Sprache $\langle a^* b^* \rangle$ endlich viele Äquivalenzklassen gibt, aber für die nicht reguläre Sprache $L = \{a^k b^k \mid k \in \mathbb{N}_0\}$ 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 $f : M \rightarrow M$ eine Abbildung. Man sagt, dass \equiv mit f *verträglich* ist, wenn für alle $x_1, x_2 \in M$ gilt:

$$x_1 \equiv x_2 \implies f(x_1) \equiv f(x_2).$$

verträglich

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:

$$x_1 \equiv x_2 \wedge y_1 \equiv y_2 \implies x_1 \square y_1 \equiv x_2 \square y_2.$$

verträglich

Ein typisches Beispiel sind wieder die Äquivalenzrelationen „modulo n “. Diese Relationen sind mit Addition, Subtraktion und Multiplikation verträglich. Ist etwa

$$\begin{array}{ll} x_1 \equiv x_2 \pmod{n} & \text{also } x_1 - x_2 = kn \\ \text{und } y_1 \equiv y_2 \pmod{n} & \text{also } y_1 - y_2 = mn \end{array}$$

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 $f_{w'} : A^* \rightarrow A^*$ 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.:

$$\forall w_1, w_2 \in A^* : w_1 \equiv_L w_2 \implies w_1 w' \equiv_L w_2 w'$$

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: $(w_1 w')v \in L \iff (w_2 w')v \in L$. Das geht ganz einfach. Sei $v \in A^*$ beliebig; dann gilt

$$\begin{aligned} (w_1 w')v \in L &\iff w_1(w'v) \in L \\ &\iff w_2(w'v) \in L && \text{weil } w_1 \equiv_L w_2 \\ &\iff (w_2 w')v \in L. \end{aligned}$$

17.2.2 Wohldefiniertheit von Operationen mit Äquivalenzklassen

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 $f : M \rightarrow M$.

induzierte Operation

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 $f_x : A^* \rightarrow A^* : w \mapsto wx$ mit \equiv_L verträglich ist.

Wir schreiben nun einmal hin:

$$f'_x : A^*/\equiv_L \rightarrow A^*/\equiv_L : [w] \mapsto [wx]$$

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 $L = \langle a^*b^* \rangle$ die Wörter ε, a, a^2, a^3 , usw. alle in einer Äquivalenzklasse liegen. Es ist also $[\varepsilon] = [a] = [a^2] = \dots$. 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 $[\varepsilon x] = [ax] = [a^2x] = \dots$ sein. Und das ist so, denn hier hinter steckt nichts anderes als die Forderung

$$w_1 \equiv_L w_2 \implies w_1x \equiv_L w_2x$$

also $w_1 \equiv_L w_2 \implies f_x(w_1) \equiv_L f_x(w_2)$

Und weil wir gesehen hatte, dass das gilt, sind wie man auch sagt, die Abbildungen $f'_x : A^*_{/\equiv_L} \rightarrow A^*_{/\equiv_L}$ *wohldefiniert*. Die Abbildungsvorschrift ist unabhängig von der Wahl des Repräsentanten der Äquivalenzklasse, die als Argument verwendet wird.

wohldefiniert

Allgemein gilt: Wenn \equiv mit $f : M \rightarrow M$ verträglich ist, dann ist $f' : M_{/\equiv} \rightarrow M_{/\equiv} : f'([x]) = [f(x)]$ 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

$$f : Z \times A \rightarrow Z : f([w], x) = [wx]$$

Diese Abbildung ist nach dem oben Gesagten wohldefiniert. Und sie erinnert 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 endlichen 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 Umbenennung 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:

$$xRy \wedge yRx \implies x = y$$

Antisymmetrie

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 $w_1 \sqsubseteq_p w_2 \iff \exists u \in A^* : w_1u = w_2$. 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 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 $(R \setminus \text{Id}) \setminus (R \setminus \text{Id})^2$. In unserem Beispiel ergibt sich aus Abbildung 17.1 durch Weglassen der Kanten Abbildung 17.2.

Hasse-Diagramm

17.3.2 „Extreme“ Elemente

Es sei (M, \sqsubseteq) eine halbgeordnete Menge und T eine beliebige Teilmenge von M .

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 $x \neq y$.

minimales Element

maximales Element

Ein Element $x \in T$ heißt *größtes Element* von T , wenn für alle

größtes Element

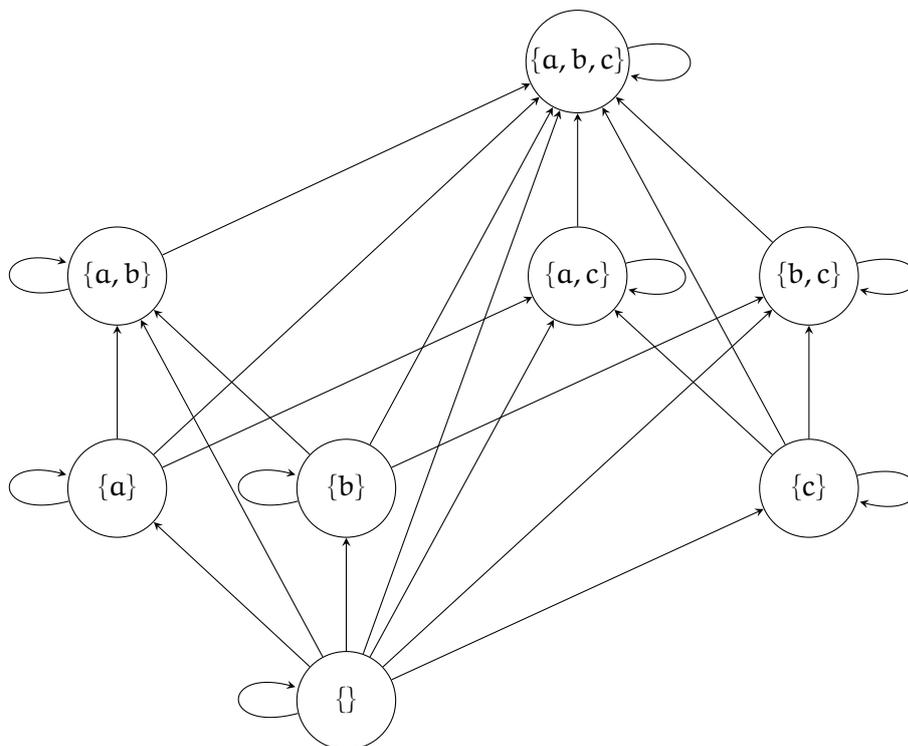


Abbildung 17.1: Die Halbordnung $(2^{\{a,b,c\}}, \subseteq)$ als Graph

$y \in T$ gilt: $y \subseteq x$. Ein Element $x \in T$ heißt *kleinstes Element* von T , wenn für alle $y \in T$ gilt: $x \subseteq y$.

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: $y \subseteq x$. Ein $x \in M$ heißt *untere Schranke* von T , wenn für alle $y \in T$ gilt: $x \subseteq y$. 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

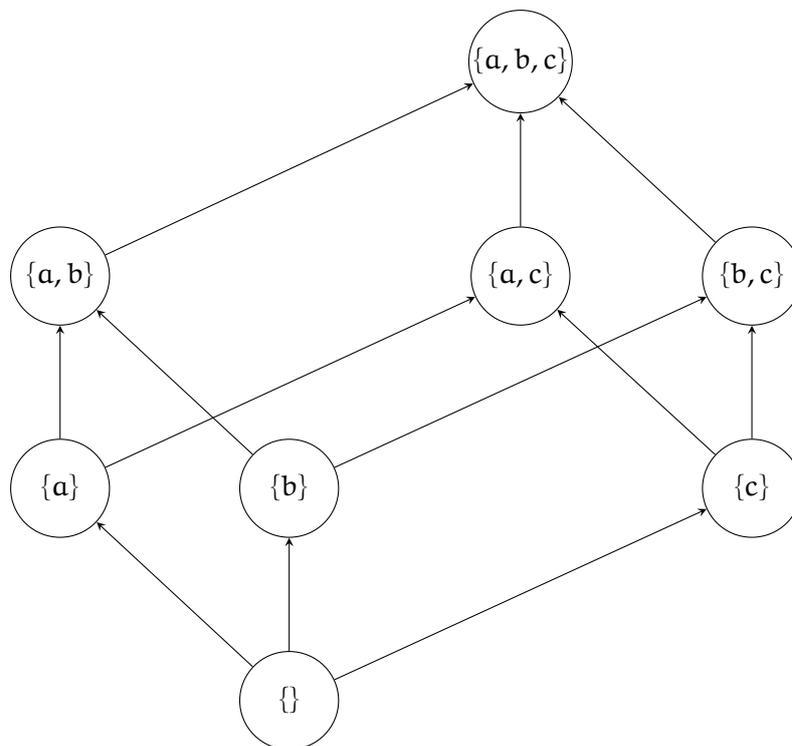


Abbildung 17.2: Hassediagramm der Halbordnung $(2^{\{a,b,c\}}, \subseteq)$

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 \perp . Für das Supremum einer aufsteigenden Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$ 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 $T_0 \subseteq T_1 \subseteq T_2 \subseteq \dots$ ist $\bigsqcup_i T_i = \bigcup T_i$.

Andererseits ist (\mathbb{N}_0, \leq) *keine* vollständige Halbordnung, denn unbeschränkt wachsende aufsteigende Ketten wie z. B. $0 \leq 1 \leq 2 \leq \dots$ 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 $\mathbb{N} = \mathbb{N}_0 \cup \{u\}$ und definiert

$$x \sqsubseteq y \iff (x, y \in \mathbb{N}_0 \wedge x \leq y) \vee (y = u)$$

Weil wir es später noch brauchen können, definieren wir auch noch $N' = \mathbb{N}_0 \cup \{u_1, u_2\}$ mit der totalen Ordnung

$$x \sqsubseteq y \iff (x, y \in \mathbb{N}_0 \wedge x \leq y) \vee (x \in \mathbb{N}_0 \cup \{u_1\} \wedge y = u_1) \vee y = u_2$$

also sozusagen

$$0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq 3 \sqsubseteq \dots \sqsubseteq u_1 \sqsubseteq u_2$$

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

Es sei \sqsubseteq eine Halbordnung auf einer Menge M . Eine Abbildung $f : M \rightarrow M$ heißt *monoton*, wenn für alle $x, y \in M$ gilt: $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$.

*monotone
Abbildung*

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 $f(3) = 3 \not\leq 0 = f(10)$.

Eine Abbildung $f : D \rightarrow D$ auf einer vollständigen Halbordnung (D, \sqsubseteq) heißt *stetig*, wenn für jede aufsteigende Kette $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$ gilt: $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$.

stetige Abbildung

Betrachten wir als erstes Beispiel noch einmal die vollständige Halbordnung $N' = \mathbb{N}_0 \cup \{u_1, u_2\}$ von oben. Die Abbildung $f : N' \rightarrow N'$ 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 $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$ 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: $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots \sqsubseteq x_i = x_{i+1} = x_{i+2} = \dots = n'$. 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 $f(x_0) \sqsubseteq f(x_1) \sqsubseteq f(x_2) \sqsubseteq \dots \sqsubseteq f(x_i) = f(x_{i+1}) = f(x_{i+2}) = \dots = f(n') = n' + 1$. 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 $g : N' \rightarrow N'$ 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 $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$ nichtnegativer ganzer Zahlen hat Supremum u_1 , so dass $g(\bigsqcup_i x_i) = u_2$ ist. Aber die Kette der Funktionswerte $g(x_0) \sqsubseteq g(x_1) \sqsubseteq g(x_2) \sqsubseteq \dots$ hat Supremum $\bigsqcup_i g(x_i) = u_1 \neq g(\bigsqcup_i x_i)$.

Der folgende Satz ist eine abgeschwächte Version des sogenannten Fixpunktsatzes von Knaster und Tarski.

17.3 Satz. *Es sei $f : D \rightarrow D$ eine monotone und stetige Abbildung auf einer vollständigen Halbordnung (D, \sqsubseteq) mit kleinstem Element \perp . Elemente $x_i \in D$ seien wie folgt definiert:*

$$\begin{aligned} x_0 &= \perp \\ \forall i \in \mathbb{N}_0 : x_{i+1} &= f(x_i) \end{aligned}$$

Dann gilt:

1. Die x_i bilden eine Kette: $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$.
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 $x_f \sqsubseteq y_f$.

17.4 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 $x_0 = \perp$ 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 x_f 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 Semantik von Programmiersprachen wieder begegnen.

Hier können wir Ihnen schon andeuten, wie er im Zusammenhang mit kontextfreien Grammatiken nützlich sein kann. Betrachten wir als Terminalzeichenalphabet $T = \{a, b\}$ und die kontextfreie Grammatik $G = (\{X\}, T, X, P)$ mit Produktionsmenge $P = \{X \rightarrow aXb \mid \varepsilon\}$. Als halbgeordnete Menge D verwenden wir die Potenzmenge $D = 2^{T^*}$ der Menge aller Wörter mit Inklusion als Halbordnungsrelation. Die Elemente der Halbordnung sind also Mengen von Wörtern, d. h. formale Sprachen. Kleinstes Element der Halbordnung ist die leere Menge \emptyset . Wie erwähnt, ist diese Halbordnung vollständig.

Es sei nun $f : D \rightarrow D$ die Abbildung mit $f(L) = \{a\}L\{b\} \cup \{\varepsilon\}$. Der Bequemlichkeit halber wollen wir nun einfach glauben, dass f stetig ist. (Wenn Ihnen das nicht passt, prüfen Sie es nach. Es ist nicht schwer.) Der Fixpunktsatz besagt, dass man den kleinsten Fixpunkt dieser Abbildung erhält als Supremum, hier also Vereinigung, aller der folgenden Mengen:

$$\begin{aligned} L_0 &= \emptyset \\ L_1 &= f(L_0) = \{a\}L_0\{b\} \cup \{\varepsilon\} \\ &= \{\varepsilon\} \\ L_2 &= f(L_1) = \{a\}L_1\{b\} \cup \{\varepsilon\} \\ &= \{ab, \varepsilon\} \\ L_3 &= f(L_2) = \{a\}L_2\{b\} \cup \{\varepsilon\} \\ &= \{aabb, ab, \varepsilon\} \end{aligned}$$

Sie sehen, wie der Hase läuft. Der kleinste Fixpunkt ist $L = \{a^k b^k \mid k \in \mathbb{N}_0\}$. Das ist auch genau die Sprache, die die Grammatik erzeugt. Und L ist Fixpunkt von f , also

$$L = \{a\}L\{b\} \cup \{\varepsilon\}$$

Es ist also sozusagen die kleinste Lösung der Gleichung $X = \{a\}X\{b\} \cup \{\varepsilon\}$. Was das mit den Produktionen der Grammatik zu tun, sehen Sie vermutlich.

17.4 ORDNUNGEN

Eine Relation $R \subseteq M \times M$ ist eine *Ordnung*, oder auch genauer *Ordnung*

totale Ordnung, wenn R Halbordnung ist und außerdem gilt:

totale Ordnung

$$\forall x, y \in M : xRy \vee yRx$$

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. $a \sqsubseteq_A b$.

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.

lexikographische
Ordnung

- Die lexikographische Ordnung \sqsubseteq_1 , nach der Wörter im Lexikon 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

$$\begin{aligned} a \sqsubseteq_1 aa \sqsubseteq_1 aaa \sqsubseteq_1 aaaa \\ \sqsubseteq_1 ab \sqsubseteq_1 aba \sqsubseteq_1 abbb \\ \sqsubseteq_1 b \sqsubseteq_1 baaaaa \sqsubseteq_1 baab \\ \sqsubseteq_1 bbbbbb \end{aligned}$$

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 $A = \{a, b\}$ mit der naheliegenden Ordnung \sqsubseteq_A so:

$$\begin{aligned} \varepsilon &\sqsubseteq_2 a \sqsubseteq_2 b \\ \sqsubseteq_2 aa &\sqsubseteq_2 ab \sqsubseteq_2 ba \sqsubseteq_2 bb \\ \sqsubseteq_2 aaa &\sqsubseteq_2 \cdots \sqsubseteq_2 bbb \\ \sqsubseteq_2 aaaa &\sqsubseteq_2 \cdots \sqsubseteq_2 bbbb \\ &\dots \end{aligned}$$

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

$$\varepsilon \sqsubseteq_1 a \sqsubseteq_1 aa \sqsubseteq_1 aaa \sqsubseteq_1 aaaa \sqsubseteq_1 \cdots$$

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

$$b \supseteq_1 ab \supseteq_1 aab \supseteq_1 aaab \supseteq_1 aaaab \supseteq_1 \cdots$$

betrachtet. Jede obere Schranke der aufsteigenden Kette muss ein b enthalten. Aber gleich, welche obere Schranke w man betrachtet, das Wort $a^{|w|}b$ 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 automa-

tischen statischen Datenflussanalyse von Programmen ausnutzen. Diese und andere Anwendungen werden Ihnen in weiteren Vorlesungen begegnen.

In dieser und zum Beispiel in den Mathematikvorlesungen wurden an vielen Stellen „logische Formeln“ verwendet, und zwar (nahezu?) ausschließlich um sich längliche umgangssprachliche Schreibweisen sparen und statt dessen eine kompakte Notation benutzen zu können. In dieser Einheit wollen wir zumindest die Grundzüge andeuten, denen man folgt, wenn man logische Formeln zum Untersuchungsgegenstand macht.

Diese Einheit ist relativ abstrakt und steht (deswegen) am Ende der Vorlesung. Trotzdem hoffen wir, auch hier wieder das ein oder andere Körnchen Neugier säen zu können, das in einer späteren Vorlesung, im vorliegenden Fall z. B. „Formale Systeme“, keimen kann.

18.1 FORMELN IN PRÄDIKATENLOGIK ERSTER STUFE

Das Vokabular für Prädikatenlogik erster Stufe besteht aus folgenden Symbolen

- einer abzählbar unendlichen Menge von *Variablensymbolen*, für die wir x_1, x_2, \dots schreiben werden, *Variablensymbole*
- einer abzählbar unendlichen Menge von *Konstantensymbolen*, für die wir c_1, c_2, \dots schreiben werden, *Konstantensymbole*
- für jedes $k \in \mathbb{N}_+$ einer abzählbar unendlichen Menge von *k-stelligen Funktionssymbolen*, für die wir f_1^k, f_2^k, \dots schreiben werden, *Funktionssymbole*
- für jedes $k \in \mathbb{N}_+$ einer abzählbar unendlichen Menge von *k-stelligen Relationssymbolen*, für die wir R_1^k, R_2^k, \dots schreiben werden, *Relationssymbole*
- den *logischen Konnektiven* \neg, \wedge, \vee und \Rightarrow , *logisches Konnektiv*
- den Klammern (und) und dem Komma , sowie
- den Quantoren \forall und \exists .

Wir haben wie in der Prädikatenlogik üblich unendlich viele Symbole vorgesehen, damit man in konkreten Anwendungen nicht beschränkt ist. In vielen konkreten Fällen sind aber (oft) endlich viele Symbole jeder Art ausreichend.

Welche Zeichenfolgen über diesem Vokabular legale prädikatenlogische Formeln erster Stufe sind, wird in drei Schritten festgelegt. Wir benutzen wieder einmal rekursive Definitionen. Aber dank des Fixpunktsatzes aus Einheit 17 wissen wir inzwischen, dass wir sozusagen den kleinsten Fixpunkt dieser Definition als das zu Definierende auffassen können.

Als erstes definieren wir, was wir unter einem *Term* verstehen wollen. *Term*

- Jedes Variablensymbol ist ein Term.
- Jedes Konstantensymbol ist ein Term.
- Wenn f ein k -stelliges Funktionssymbol ist und t_1, \dots, t_k k Terme sind, dann ist auch $f(t_1, \dots, t_k)$ ein Term.
- Ein Term, in dem keine Variablensymbole vorkommen, heißen *Grundterme*. *Grundterme*

Einfache Beispiele für Terme:

- x_1, x_3, x_{42}
- c_2, c_{42}, c_{4711}
- Terme mit Funktionssymbol:
 - $f_1^1(x_1)$
 - $f_1^2(c_3, x_2)$
 - $f_1^2(f_1^1(x_1), f_1^2(c_3, x_2))$

Als nächstes legt man fest, was eine *atomare Formel* ist:

atomare Formel

- Wenn R ein k -stelliges Relationssymbol ist und t_1, \dots, t_k k Terme sind, dann ist $R(t_1, \dots, t_k)$ eine atomare Formel.

Einfache Beispiele für atomare Formeln sind

- $R_1^1(x_1)$
- $R_1^2(c_3, x_2)$
- $R_1^2(f_1^1(x_1), f_1^2(c_3, x_2))$
- Aber z. B. x_3, c_5 oder irgendwelche anderen Terme sind *keine* atomare Formeln.

Als letztes werden *prädikatenlogische Formeln*, (oder kurz: Formeln) definiert:

prädikatenlogische Formel

- Jede atomare Formel ist Formel.
- Wenn \mathcal{F} eine Formel ist, dann ist auch $(\neg \mathcal{F})$ eine Formel.
- Wenn \mathcal{F}_1 und \mathcal{F}_2 Formeln sind, dann sind auch $(\mathcal{F}_1 \wedge \mathcal{F}_2)$, $(\mathcal{F}_1 \vee \mathcal{F}_2)$ und $(\mathcal{F}_1 \Rightarrow \mathcal{F}_2)$ Formeln.
- Wenn \mathcal{F} eine Formel ist und x ein Variablensymbol, dann sind auch $(\forall x \mathcal{F})$ und $(\exists x \mathcal{F})$ Formeln.

Wie üblich ist man in der Realität bei der Klammersetzung etwas großzügiger als obige Definition. Dann sind zum Beispiel die folgenden Zeichenketten Formeln.

- $(\neg R_5^2(x_2, x_7))$
- $R_1^1(x_6) \Rightarrow (\neg R_5^2(x_2, x_7))$
- $\exists x_2 (R_1^1(x_6) \Rightarrow (\neg R_5^2(x_2, x_7)))$

Für den Rest dieser Einheit sei vereinbart, dass wir mit „Formel“ immer eine Formel in Prädikatenlogik erster Stufe meinen.

18.2 THEORIEN UND BEWEISBARKEIT

In Mathematikvorlesungen haben Sie etwas aus der Theorie der Vektorräume gelernt. Dazu wurden als erstes definiert, „was ein Vektorraum ist“, in dem eine Reihe von Forderungen als logische Formeln aufgeschrieben wurden. So etwas nennt man Axiome. Und dann haben Sie Sätze bewiesen, d.h. Sie haben Beweise aufgeschrieben für neue Aussagen, die sich von den bisher gegebenen unterscheiden.

Aber was ist eigentlich ein Beweis? Jedenfalls geht man üblicherweise schrittweise vor und jeder einzelne Schritt ist gerechtfertigt. Eine Rechtfertigung kann sein, dass etwas aus der Ergebnissen vorgegangener Schritte „zwingend folgt“, eine andere,

dass etwas „offensichtlich“ ist wie behauptet. Das kann daran liegen, dass es sich um eines der der Theorie zu Grunde liegenden Axiome handelt, oder gar um eine Aussage, der man anhand ihrer Struktur ansieht, dass sie immer gilt. Das trüfe zum Beispiel auf jede Formel der Struktur $\mathcal{F}_1 \wedge \mathcal{F}_2 \Rightarrow \mathcal{F}_1$ zu, ganz gleich, um welche Teilformeln \mathcal{F}_1 und \mathcal{F}_2 es sich handelt.

In der formalen Logik präzisiert man das wie folgt. Ein *Beweis* ist eine endliche Folge F_1, F_2, \dots, F_k von Formeln. Dabei ist jede Formel entweder ein sogenanntes Axiom, oder sie ergibt sich aus in der Folge vorangegangenen Formeln durch eine einfache sogenannte Ableitungsregel.

Beweis

Bei den Axiomen hat man zum einen solche, die immer wahr sind, gleich welche Interpretation man zu Grunde legt. Dazu gehören zum Beispiel alle, die von einer der Formen

- $\mathcal{F}_1 \Rightarrow (\mathcal{F}_2 \Rightarrow \mathcal{F}_1)$ bzw. äquivalent $\mathcal{F}_1 \wedge \mathcal{F}_2 \Rightarrow \mathcal{F}_1$
- $(\mathcal{F}_1 \Rightarrow (\mathcal{F}_2 \Rightarrow \mathcal{F}_3)) \Rightarrow ((\mathcal{F}_1 \Rightarrow \mathcal{F}_2) \Rightarrow (\mathcal{F}_1 \Rightarrow \mathcal{F}_3))$ bzw. äquivalent $(\mathcal{F}_1 \wedge \mathcal{F}_2 \Rightarrow \mathcal{F}_3) \wedge (\mathcal{F}_1 \Rightarrow \mathcal{F}_2) \Rightarrow (\mathcal{F}_1 \Rightarrow \mathcal{F}_3)$
- $(\neg \mathcal{F}_2 \Rightarrow \neg \mathcal{F}_1) \Rightarrow ((\neg \mathcal{F}_2 \Rightarrow \mathcal{F}_1) \Rightarrow \mathcal{F}_2)$ bzw. äquivalent $(\neg \mathcal{F}_2 \Rightarrow \neg \mathcal{F}_1) \wedge (\neg \mathcal{F}_2 \Rightarrow \mathcal{F}_1) \Rightarrow \mathcal{F}_2$

sind für beliebige Formeln $\mathcal{F}_1, \mathcal{F}_2$ und \mathcal{F}_3 . Hinzu kommen weitere Formeln, in denen explizit Quantoren eine Rolle spielen. Von den beiden üblicherweise verwendeten Ableitungsregeln erwähnen wir nur eine genauer (Die zweite heißt *Generalisierung*). Sie heißt *Modus ponens* und besagt:

Generalisierung
Modus ponens

- Wenn man schon eine Formel \mathcal{F}_1 bewiesen hat und außerdem eine Formel, die von der Form $\mathcal{F}_1 \Rightarrow \mathcal{F}_2$ ist, dann darf man auch die Formel \mathcal{F}_2 als bewiesen ansehen.

Eine Theorie erster Stufe ist festgelegt durch

- die Auswahl endlich vieler Konstanten- und Funktionssymbole,
- die Auswahl mindestens eines Prädikatensymbols,
- der mit ihrer Hilfe konstruierbaren Formeln wie oben, die immer wahr sind,
- und zusätzliche theorie-spezifische Axiome (z. B. die Gruppenaxiome)
- die beiden Ableitungsregeln Modus ponens und Generalisierung.

Man könnte es als ein großes algorithmisches Problem ansehen, für eine gegebene Formel herauszufinden, ob sie in einer Theorie beweisbar ist oder nicht, und falls das der Fall ist, auch gleich noch einen Beweis zu produzieren. In der Tat ist das ein *zu* großes Problem. Es ist im allgemeinen *unentscheidbar*, ob eine Formel in einer Theorie beweisbar ist oder nicht.

Da wir hier nur einen kurzen Einblick in die Grundlagen geben wollen, beschränken wir uns auf abgeschlossene Formeln. Eine Formel heißt *abgeschlossen*, wenn jedes Vorkommen eines Variablensymbols x_i in einer Teilformel liegt, die die Form $(\forall x_i \mathcal{F})$ oder die Form $(\exists x_i \mathcal{F})$ hat.

abgeschlossene Formel

Eine *Interpretation* für eine Formel oder eine Menge von Formeln ist durch folgende Angaben festgelegt:

Interpretation

- eine Menge U , das sogenannte *Universum*,
- ein Element $\mathcal{J}(c_i) \in U$ für jedes Konstantensymbol c_i ,
- eine k -stellige Abbildung $\mathcal{J}(f_i^k) : U^k \rightarrow U$ für jedes k -stellige Funktionssymbol f_i^k und
- eine k -stellige Relation $\mathcal{J}(R_i^k) \subseteq U^k$ für jedes k -stellige Relationssymbol R_i^k .

Universum

Ohne dass wir das im Detail präzise definieren wollen, hoffen wir, dass trotzdem klar ist, dass jede geschlossene Formel in einer gegebenen Interpretation stets wahr oder falsch ist. Als erstes Beispiel betrachten wir die Formel

$$\forall x_1 R_1^2(f_1^2(x_1, c_1), x_1) \wedge \forall x_1 R_1^2(f_1^2(c_1, x_1), x_1) \quad (12)$$

Nimmt man für eine Interpretation $U = \mathbb{N}_0$, $\mathcal{J}(c_1) = 0$, $\mathcal{J}(f_1^2) = \text{Addition}$ und $\mathcal{J}(R_1^2) = \text{Id}$, dann ist die Formel wahr, denn sie besagt gerade, dass die 0 neutrales Element bezüglich Addition nichtnegativer ganzer Zahlen ist.

Nimmt man für eine Interpretation $U = \{a, b\}^*$, $\mathcal{J}(c_1) = \varepsilon$, $\mathcal{J}(f_1^2) = \text{Konkatenation}$, dann ist die Formel ebenfalls wahr, denn sie besagt gerade, dass das leere Wort neutrales Element bezüglich Konkatenation von Wörtern aus a und b ist.

Eine Interpretation heißt ein *Modell* für eine Menge abgeschlossener Formeln, wenn jede der Formeln in der Interpretation wahr ist.

Modell

Es gibt Situationen, in denen man sich mit genau einer Interpretation beschäftigt. Zum Beispiel werden Sie sich in Ihrer Vorlesung „Analysis“ oder „Höhere Mathematik“ (höchstwahrscheinlich, zumindest in den ersten Wochen) nur für das Universum $U = \mathbb{R}$ interessiert haben. Und wenn etwas bewiesen wurde, dann eben etwas über die reellen Zahlen. Jedenfalls wurde vermutlich so getan.

Alle Ableitungsregeln haben aber eines gemeinsam: Wenn die Formeln, die man als Voraussetzungen braucht, in einer Interpretation wahr sind, dann gilt das auch für die Formel, die man mit Hilfe der Ableitungsregel folgert.

Wesentliche Konsequenz dieser Tatsache ist, dass jedes Modell aller Axiome auch Modell aller daraus ableitbaren Formeln ist.

Zum Beispiel mögen Sie bitte glauben, dass man aus der Formel 12 mit Hilfe endlicher vieler Ableitungsschritte und unter

Zuhilfenahme von Axiomen, die in allen Interpretationen wahr sind, die folgende Formel beweisen kann:

$$\forall x_2 ((\forall x_1 f_1^2(x_1, x_2)=x_1 \wedge \forall x_1 f_1^2(x_2, x_1)=x_1) \Rightarrow x_2=c_1) \quad (13)$$

Was bedeutet das? Formel 12 besagt, dass in einem Modell $\mathcal{J}(c_1)$ neutrales Element bezüglich der binären Operation $\mathcal{J}(f_1^2)$ ist. Formel 13 besagt, dass ein solches neutrales Element eindeutig ist. Das gilt also in allen Modellen von Formel 12, d.h. in allen Interpretationen, die über ein neutrales Element aufweisen. Es ist also *immer* eindeutig.

18.4 BEISPIELE FÜR MODELLE FÜR GESCHLOSSENE FORMELN

Ein *normales Modell* sei eines, bei dem das Relationssymbol R_1^2 als Identität interpretiert wird. Im folgenden betrachten wir ausschließlich solche normalen Modelle. Um das deutlich zu machen schreiben wir für das Relationssymbol im folgenden auch = statt R_1^2 .

normales Modell

Welches sind die normalen Modelle der folgenden Formelmengende?

- $\forall x_1 \forall x_2 \forall x_3 f_1^2(x_1, f_1^2(x_2, x_3))=f_1^2(f_1^2(x_1, x_2), x_3)$
- $\forall x_1 (f_1^2(x_1, c_1)=x_1 \wedge f_1^2(c_1, x_1)=x_1)$
- $\forall x_1 \exists x_2 (f_1^2(x_1, x_2)=c_1 \wedge f_1^2(x_2, x_1)=c_1)$

Das erste Axiom besagt, dass in jedem normalen Modell $\mathcal{J}(f_1^2)$ eine assoziative Operation ist. Das zweite besagt, dass $\mathcal{J}(c_1)$ neutrales Element bezüglich dieser Operation ist. Und das dritte besagt, dass es zu jedem Element ein Inverses gibt. So wie man beweisen kann, dass neutrale Elemente eindeutig sind, kann man auch beweisen, dass Inverse eindeutig sind. Die normalen Modelle sind also gerade die Gruppen.

Genauso kann man zum Beispiel die Axiome für Halbordnungen hinschreiben:

- $\forall x_1 R_2^2(x_1, x_1)$
- $\forall x_1 \forall x_2 (R_2^2(x_1, x_2) \wedge R_2^2(x_2, x_1) \Rightarrow x_1=x_2)$
- $\forall x_1 \forall x_2 \forall x_3 (R_2^2(x_1, x_2) \wedge R_2^2(x_2, x_3) \Rightarrow R_2^2(x_1, x_3))$

In allen normalen Modellen wird die Interpretation $\mathcal{J}(R_2^2)$ eine Halbordnung sein.

Zum Abschluss wollen wir noch auf eine besondere Art von Interpretationen hinweisen. Die Idee dafür geht auf den Mathematiker Jacques Herbrand (12.2.1908 – 27.7.1931) zurück. Dazu sei eine Theorie gegeben, die mindestens ein Konstantensymbol enthalte.

- Als Universum U wählt man die Menge H aller Terme, die man aus allen Konstanten- und Funktionssymbolen der Theorie bilden kann (Variablensymbole sind nicht erlaubt).
- Die Interpretation $f_i^k = \mathcal{J}(f_i^k)$ eines k -stelligen Funktionssymbols f_i^k ist $f_i^k : H^k \rightarrow H$ mit

$$f_i^k(t_1, \dots, t_k) = f_i^k(t_1, \dots, t_k)$$

- Die Interpretation $R_i^k = \mathcal{J}(R_i^k)$ eines k -stelligen Relationssymbols R_i^k ist $R_i^k \subseteq H^k$ mit

$$R_i^k = \{(t_1, \dots, t_k) \mid R_i^k(t_1, \dots, t_k) \text{ ist beweisbar}\}$$

Machen Sie sich bitte unbedingt klar, dass wir eben keine Trivialitäten hingeschrieben haben. Die Abbildung f_i^k ist etwas anderes als ein einzelnes Funktionssymbol f_i^k , und die Relation R_i^k ist etwas anderes als ein einzelnes Relationssymbol R_i^k . Ein Beispiel macht dies noch deutlicher.

Betrachten wir den Fall, dass in der Theorie nur ein einziges Konstantensymbol c_1 und ein einziges Funktionssymbol f_1^1 vorkommt. Dann besteht das Universum H aus den folgenden Termen:

$$H = \{c_1, \\ f_1^1(c_1), \\ f_1^1(f_1^1(c_1)), \\ f_1^1(f_1^1(f_1^1(c_1))), \\ f_1^1(f_1^1(f_1^1(f_1^1(c_1))))), \\ \dots\}$$

Die Interpretation des einstelligen Funktionssymbols f_1^1 ist eine einstellige Funktion $s : H \rightarrow H$, für die zum Beispiel gilt:

$$s(c_1) = f_1^1(c_1) \\ s(f_1^1(c_1)) = f_1^1(f_1^1(c_1)) \\ \vdots$$

Man kann zeigen: Wenn eine Theorie überhaupt ein Modell besitzt, dann ist die Herbrand-Interpretation ein Modell.

18.5 GRENZEN VON PRÄDIKATENLOGIK ERSTER STUFE

Aus dem eben erwähnten Ergebnis zu Herbrand-Interpretationen ergibt sich sofort ein erstes Beispiel für eine Beschränkung von Prädikatenlogik erster Stufe. Das Herbrand-Universum besteht immer nur aus abzählbar unendlich vielen Elementen. Folglich muss es unmöglich sein, in Prädikatenlogik erster Stufe Axiome hinzuschreiben, deren einziges Modell die reellen Zahlen sind.

Andererseits kann man zeigen, dass jede Theorie, die überhaupt ein Modell besitzt, immer auch Modelle beliebig großer Kardinalität besitzt. Es kann also auch nicht möglich sein, in Prädikatenlogik erster Stufe Axiome hinzuschreiben, deren einziges Modell die natürlichen Zahlen sind. Die Peano-Axiome sind also nicht in Prädikatenlogik erster Stufe formulierbar.

Es zeigt sich, dass dabei sogar soviel Ausdruckskraft fehlt, dass man noch nicht einmal erzwingen kann, dass die natürlichen Zahlen das einzige Modell mit abzählbar unendlich großem Universum sind.

18.6 AUSBLICK

INDEX

- $f \simeq g$, 98
- Äquivalenzklasse, 143
- Äquivalenzrelation, 81, 141
- Äquivalenzrelation von Nerode, 142
- Übersetzung, 65
- äquivalente Aussagen, 13

- A^n , 18
- A^* , 17
- Abbildung, 11
 - bijektiv, 12
 - injektiv, 12
 - monotone, 150
 - partielle, 11
 - stetige, 150
 - surjektiv, 12
 - wohldefinierte, 146
- abgelehntes Wort, 115
- abgeschlossene Formel, 159
- ablehnender Zustand, 114
- Ableitung, 50
- Ableitungsbaum, 53
- Ableitungsfolge, 50
- Ableitungsschritt, 50
- adjazente Knoten, 74, 80
- Adjazenzliste, 85
- Adjazenzmatrix, 85
- akzeptierender Zustand
 - bei endlichen Automaten, 114
 - bei Turingmaschinen, 129
- akzeptierte formale Sprache
 - bei Turingmaschinen, 129
- akzeptierte formale Sprache
 - bei endlichen Automaten, 115
- akzeptiertes Wort
 - bei Turingmaschinen, 129
 - bei endlichen Automaten, 115
- Akzeptor, endlicher, 114
- al-Khwarizmi, Muhammad ibn Musa, 25
- Algebra, 25
- Algorithmus, 25, 124
- Algorithmus von Warshall, 92
- Allquantor, 14
- Alphabet, 8
- Anfangskonfiguration, 129
- antisymmetrische Relation, 147
- assoziative Operation, 24
- asymptotisches Wachstum, 98
- atomare Formel, 157

- aufschreiben
 - früh, 40
- aufsteigende Kette, 149
- aufzählbare Sprache, 129
- Ausgabefunktion, 111, 113
- Ausgangsgrad, 77
- Aussagen
 - äquivalente, 13
- aussagenlogische Formel, 13
- Auszeichnungssprache, 41
- Automat
 - endlicher, 111
 - Mealy-, 111
 - Moore-, 113
- average case, 97

- Bachelorarbeit, 40
- Baum
 - gerichtet, 77
 - ungerichtet, 80
- Berechnung
 - endliche, 128
 - haltend, 128
 - nicht haltend, 128
 - unendliche, 128
- Beweis
 - in der Prädikatenlogik, 158
- Biber
 - fleißiger, 139
- Bibermaschine, 139
- bijektive Abbildung, 12
- Binärdarstellung, 62
- binäre Relation, 11
- Bit, 56
- Blanksymbol, 126
- Block-Codierung, 72
- Busy-Beaver-Funktion, 139
- Byte, 56

- $c_0(w)$, 129
- Champollion, Jean-Francois, 8
- Code, 66
 - präfixfreier, 67
- Codierung, 66
- computational complexity, 96
- Coppersmith, Don, 105

- Datum, 6
- Diagonalisierung, 136
- Dokument, 40

- ε -freier Konkatenationsabschluss, 38

- einfacher Zyklus, 77
- Eingabealphabet, 129
- Eingangsgrad, 77
- Endkonfiguration, 127
- endliche Berechnung, 128
- endlicher Akzeptor, 114
- endlicher Automat, 111
- entscheidbare Sprache, 130
- Entscheidungsproblem, 128
- erreichbarer Knoten, 76
- Erscheinungsbild, 40
- erzeugte formale Sprache, 51
- Existenzquantor, 14

- f*
 - bei Moore-Automaten, 113
 - bei Mealy-Automaten, 112
- f**
 - bei Mealy-Automaten, 112
 - bei Moore-Automaten, 113
- Faktormenge, 143
- Faserung, 143
- fleißiger Biber, 139
- Form, 40
- formale Sprache, 36
 - akzeptierte , bei Turingmaschinen, 129
 - akzeptierte , bei endlichen Automaten, 115
 - erzeugte, 51
 - kontextfrei, 51
 - Potenzen, 37
- Formel
 - abgeschlossene, 159
 - aussagenlogische, 13
- früh aufschreiben, 40
- Funktion, 11
 - bijektiv, 12
 - injektiv, 12
 - partielle, 11
 - surjektiv, 12
- Funktionssymbol, 156

- g*
 - bei Mealy-Automaten, 112
 - bei Moore-Automaten, 114
- g**
 - bei Mealy-Automaten, 112
 - bei Moore-Automaten, 114
- gerichteter Graph, 74
- geschlossener Pfad, 77
- gewichteter Graph, 82
- größenordnungsmäßiges Wachstum, 98
- größtes Element, 147

Grad, 77, 81
 Graham, Ronald, 101
 Grammatik
 kontextfreie, 49
 rechtslinear, 122
 Typ-2, 122
 Typ-3, 122
 Graph
 gerichtet, 74
 gewichtet, 82
 isomorphe, 78
 kantenmarkiert, 82
 knotenmarkiert, 82
 schlingenfrei, 76, 80
 streng zusammenhängend, 77
 ungerichtet, 79
 Graphisomorphismus, 78
 Groß-O-Notation, 97
 Grundterme, 156

 Hülle
 reflexiv-transitive, 54
 h^{**} , 67
 halbgeordnete Menge, 147
 Halbordnung, 147
 vollständige, 149
 Halten
 einer Turingmaschine, 127
 haltende Berechnung, 128
 Halteproblem, 137
 harte Probleme, 134
 Hasse-Diagramm, 147
 Herbrand, Jacques, 160
 Hexadezimaldarstellung, 63
 Homomorphismus, 67
 ε -freier, 67
 Huffman-Codierung, 68

 Id_M , 53
 IETF, 56
 Induktion
 vollständige, 23
 induzierte Operation, 145
 Infimum, 148
 Infixschreibweise, 50
 Informatik, 4
 Information, 6
 Inhalt, 40
 injektive Abbildung, 12
 Inschrift, 5
 Internet Engineering Task Force, 56
 Interpretation, 159

Inzidenzlisten, 85
 isomorphe Graphen, 78
 Isomorphismus
 von Graphen, 78

 kantenmarkierter Graph, 82
 Kantenrelation, 80
 kartesisches Produkt, 11
 Kleene, Stephen Cole, 92, 118
 kleinstes Element, 148
 Knoten
 adjazente, 74, 80
 erreichbarer, 76
 knotenmarkierter Graph, 82
 Knuth, Donald, 41, 101
 kommutative Operation, 24
 Komplexitätsklasse, 133
 Komplexitätsmaß, 96, 132
 Komplexoperationen, 102
 Konfiguration, 125
 Kongruenz modulo n , 141
 Kongruenzrelation, 145
 Konkatenation
 von Wörtern, 19
 Konkatenationsabschluss, 38
 ε -freier, 38
 Konnektiv
 logisches, 156
 Konstantensymbol, 156
 kontextfreie Grammatik, 49
 kontextfreie Sprache, 51
 Kryptographie, 124

 Länge eines Pfades, 76
 Länge eines Weges, 80
 Länge eines Wortes, 16
 L^AT_EX, 41
 leeres Wort, 17
 lexikographische Ordnung, 153
 Linksableitung, 53
 linkseideutige Relation, 11
 linkstotale Relation, 11
 logisches Konnektiv, 156

 markup language, 41
 Masterarbeit, 40
 Mastertheorem, 106
 maximales Element, 147
 Mealy-Automat, 111
 Menge aller Wörter, 17
 minimales Element, 147
 Modell, 159

- modulo n , 141
- Modus ponens, 158
- monotone Abbildung, 150
- Moore-Automat, 113
- Morphogenese, 124

- Nachricht, 5
- Nerode
 - Äquivalenzrelation, 142
- nicht haltende Berechnung, 128
- nichtdeterministische Automaten, 117
- Nichtterminalsymbol, 49

- $O(f)$, 100
- $O(1)$, 100
- $\Omega(f)$, 100
- obere Schranke, 148
- Octet, 56
- Operation
 - assoziative, 24
 - induzierte, 145
 - kommutative, 24
- Ordnung, 152
 - lexikographische, 153
 - total, 153

- P**, 133
- partielle Abbildung, 11
- Patashnik, Oren, 101
- Pfad, 76
 - geschlossener, 77
 - wiederholungsfrei, 76
- Pfadlänge, 76
- Platzkomplexität, 132
- polynomiell
 - Raumkomplexität, 133
 - Zeitkomplexität, 132
- Potenzen
 - einer formalen Sprache, 37
 - einer Relation, 54
- Potenzen eines Wortes, 21
- prädikatenlogische Formel, 157
- präfixfreier Code, 67
- Produkt
 - kartesches, 11
- Produkt der Relationen, 53
- Produkt formaler Sprachen, 37
- Produktion, 50
- PSPACE**, 133
- PSPACE**-hart, 134
- PSPACE**-schwer, 134

Quantor, 14
 $\langle R \rangle$, 119
 Radó, Tibor, 139
 Rado-Funktion, 139
 Rat fürs Studium, 40
 Raumkomplexität, 132
 polynomiell, 133
 Rechenzeit, 96
 rechtseindeutige Relation, 11
 rechtslineare Grammatik, 122
 rechtstotale Relation, 12
 reduzierbar, 134
 Reduzierbarkeit, 134
 reflexiv-transitive Hülle, 54
 reflexive Relation, 54
 reguläre Sprache, 120
 regulärer Ausdruck, 118
 Relation, 11
 antisymmetrisch, 147
 binäre, 11
 linkseideutig, 11
 linkstotal, 11
 Potenz, 54
 Produkt, 53
 rechtseindeutig, 11
 rechtstotal, 12
 reflexiv-transitive Hülle, 54
 reflexive, 54
 symmetrisch, 81
 transitive, 54
 Relationssymbol, 156
 Reques for Comments, 56
 Ressource, 96
 RFC, 56
 3629, 67

 Schleifeninvariante, 33
 Schlinge, 75, 80
 schlingenfreier Graph, 76, 80
 Schranke
 obere, 148
 untere, 148
 Schritt einer Turingmaschine, 126
 Semianrarbeit, 40
 Signal, 4
 Signum-Funktion, 87
 Speicher, 56
 Speicherplatzbedarf, 96
 Speicherung, 5
 Sprache
 aufzählbare, 129

- entscheidbare, 130
- formale, 36
- Produkt, 37
- reguläre, 120
- Startsymbol, 50
- stetige Abbildung, 150
- Strassen, Volker, 105
- streng zusammenhängender Graph, 77
- Struktur, 40
- Studium
 - ein guter Rat, 40
- Supremum, 148
- surjektive Abbildung, 12
- symmetrische Relation, 81

- Tantau, Till, 40
- Teilgraph, 76, 80
- Term, 156
- Terminalsymbol, 49
- $\Theta(f)$, 100
- totale Ordnung, 153
- transitive Relation, 54
- Turing, Alan, 124
- Turingmaschine, 125
 - Akzeptor, 129
 - Halten, 127
 - Schritt, 126
 - universelle, 136
- Twain, Mark, 1
- Typ-2-Grammatiken, 53, 122
- Typ-3-Grammatiken, 122

- unendliche Berechnung, 128
- ungerichteter Baum, 80
- ungerichteter Graph, 79
- universelle Turingmaschine, 136
- Universum, 159
- untere Schranke, 148
- UTF-8, 67

- Variablensymbol, 156
- Verträglichkeit, 144
- vollständige Induktion, 23
- vollständige Halbordnung, 149
- vollständige Induktion, 22

- Wachstum
 - asymptotisches, 98
 - größenordnungsmäßig, 98
 - größenordnungsmäßig gleich, 98
- Warshall-Algorithmus, 92
- Watson, Thomas, 1

- Weg, 80
- Wegematrix, 86
- Weglänge, 80
- wiederholungsfreier Pfad, 76
- Winograd, Shmuel, 105
- wohldefinierte Abbildung, 146
- worst case, 97
- Wort, 16
 - abgelehntes, 115
 - akzeptiertes bei endlichen Automaten, 115
 - akzeptiertes, bei Turingmaschinen, 129
 - Länge, 16
 - leeres, 17
 - Potenzen, 21
- Wurzel, 77

- Zeitkomplexität, 132
 - polynomiell, 132
- Zellularautomaten, 124
- zusammenhängender ungerichteter Graph, 80
- Zustand
 - ablehnender, 114
 - akzeptierender, bei endlichen Automaten, 114
 - akzeptierender, bei Turingmaschinen, 129
- Zustandsüberföhrungsfunktion, 111, 113
- Zyklus, 77
 - einfacher, 77

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*. Hg. 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> (14.1.09).
- 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 L^AT_EX 2_ε (more precisely pdfelatex) using Hermann Zapf's *Palatino* and *Euler* type faces. The layout was inspired by Robert Bringhurst's book *The Elements of Typographic Style*. (It is similar to the `classicthesis` style available via CTAN). See <http://www.ctan.org>.