

Thomas Worsch und Simon Wacker
unter Mitarbeit von P. Berdesinski, L. Buchholz, P. Fetzer

Grundbegriffe der Informatik

Vorlesung im Wintersemester 2020/2021
(Fassung vom 10. März 2021)

KIT-Fakultät für Informatik
Karlsruher Institut für Technologie

INHALTSVERZEICHNIS (KURZ)

Inhaltsverzeichnis (kurz) [i](#)

Inhaltsverzeichnis (lang) [iii](#)

- [1 Prolog](#) [1](#)
- [2 Signale, Nachrichten, Informationen, Daten](#) [5](#)
- [3 Mengen, Alphabete, Abbildungen](#) [9](#)
- [4 Wörter](#) [23](#)
- [5 Aussagenlogik](#) [33](#)
- [6 Induktives Vorgehen](#) [49](#)
- [7 Formale Sprachen](#) [55](#)
- [8 Übersetzungen und Codierungen](#) [59](#)
- [9 Speicher](#) [79](#)
- [10 Prozessor](#) [85](#)
- [11 Dokumente](#) [99](#)
- [12 Kontextfreie Grammatiken](#) [105](#)
- [13 Prädikatenlogik](#) [117](#)
- [14 Der Begriff des Algorithmus](#) [133](#)
- [15 Graphen](#) [147](#)
- [16 Erste Algorithmen in Graphen](#) [159](#)
- [17 Quantitative Aspekte von Algorithmen](#) [177](#)
- [18 Endliche Automaten](#) [195](#)
- [19 Reguläre Ausdrücke und rechtslineare Grammatiken](#) [209](#)

20 Turingmaschinen [219](#)

21 Relationen [239](#)

22 Mima-x [253](#)

Index [265](#)

Literatur [273](#)

Colophon [275](#)

INHALTSVERZEICHNIS (LANG)

Inhaltsverzeichnis (kurz) [i](#)

Inhaltsverzeichnis (lang) [iii](#)

1 Prolog [1](#)

1.1 Aufbau der Vorlesung und Ziele [1](#)

1.2 Quellen [2](#)

Literatur [3](#)

2 Signale, Nachrichten, Informationen, Daten [5](#)

2.1 Signal [5](#)

2.2 Übertragung und Speicherung [6](#)

2.3 Nachricht [6](#)

2.4 Information [7](#)

2.5 Datum [8](#)

2.6 Zusammenfassung [8](#)

3 Mengen, Alphabete, Abbildungen [9](#)

3.1 Mengen [10](#)

3.2 Alphabete [15](#)

3.2.1 Beispiel ASCII [15](#)

3.2.2 Beispiel Unicode [16](#)

3.3 Relationen und Abbildungen [17](#)

3.3.1 Paare, Tupel und kartesische Produkte [17](#)

3.3.2 Allquantor und Existenzquantor [18](#)

3.3.3 Relationen und Abbildungen [18](#)

3.4 Mehr zu Mengen [21](#)

4 Wörter [23](#)

4.1 Wörter [23](#)

4.2 Das leere Wort [24](#)

4.3 Mehr zu Wörtern [25](#)

4.4 Konkatenation von Wörtern [25](#)

4.4.1 Konkatenation mit dem leeren Wort [27](#)

4.4.2 Eigenschaften der Konkatenation [28](#)

4.4.3 Beispiel: Aufbau von E-Mails [28](#)

4.4.4 Iterierte Konkatenation [30](#)

4.5	Formale Sprachen	30
4.6	Binäre Operationen	31
5	Aussagenlogik	33
5.1	Informelle Grundlagen	33
5.2	Syntax aussagenlogischer Formeln	34
5.3	Boolesche Funktionen	37
5.4	Semantik aussagenlogischer Formeln	38
5.5	Beweisbarkeit	43
	Literatur	47
6	Induktives Vorgehen	49
6.1	Vollständige Induktion	49
6.2	Varianten vollständiger Induktion	51
6.3	Induktive Definitionen	53
7	Formale Sprachen	55
7.1	Operationen auf formalen Sprachen	55
7.1.1	Produkt oder Konkatenation formaler Sprachen	55
7.1.2	Konkatenationsabschluss einer formalen Sprache	57
7.2	Zusammenfassung	58
8	Übersetzungen und Codierungen	59
8.1	Von Wörtern zu Zahlen und zurück	59
8.1.1	Dezimaldarstellung von Zahlen	59
8.1.2	Andere unbeschränkte Zahldarstellungen	60
8.1.3	Ganzzahlige Division mit Rest	62
8.1.4	Von Zahlen zu ihren Darstellungen	63
8.1.5	Beschränkte Zahlbereiche und Zahldarstellungen	64
8.2	Komposition von Funktionen	65
8.3	Von einem Alphabet zum anderen	66
8.3.1	Ein Beispiel: Übersetzung von Zahldarstellungen	66
8.3.2	Homomorphismen	68
8.3.3	Beispiel Unicode: UTF-8 Codierung	71
8.4	Huffman-Codierung	72
8.4.1	Algorithmus zur Berechnung von Huffman-Codes	73
8.4.2	Weiteres zu Huffman-Codes	77
8.5	Ausblick	78
9	Speicher	79

9.1	Bit und Byte	79
9.2	Binäre und dezimale Größenpräfixe	80
9.3	Speicher als Tabellen und Abbildungen	81
9.3.1	Hauptspeicher	81
9.4	Ausblick	84
10	Prozessor	85
10.1	Einfache „Hardware“-„Bausteine“	85
10.2	Grobstruktur der MIMA	88
10.3	Maschinenbefehle der MIMA	90
10.3.1	Befehle zum Datentransport	91
10.3.2	Befehle für arithmetische-logische Operationen	92
10.3.3	Sprungbefehle	93
10.4	Mikroprogrammsteuerung der MIMA	94
10.4.1	Befehlsholphase	95
10.4.2	Befehlsdecodierungsphase	95
10.4.3	Befehlsausführungsphase	96
10.5	Ein Beispielprogramm	96
11	Dokumente	99
11.1	Dokumente	99
11.2	Struktur von Dokumenten	100
11.2.1	L ^A T _E X	100
11.2.2	HTML und XHTML	102
11.2.3	Eine Grenze unserer bisherigen Vorgehensweise	103
11.3	Zusammenfassung	104
12	Kontextfreie Grammatiken	105
12.1	Rekursive Definition syntaktischer Strukturen	105
12.2	Kontextfreie Grammatiken	110
12.3	Relationen (Teil 2)	114
12.4	Ausblick	115
13	Prädikatenlogik	117
13.1	Syntax prädikatenlogischer Formeln	117
13.2	Semantik prädikatenlogischer Formeln	120
13.3	Freie und gebundene Variablenvorkommen und Substitutionen	123
13.4	Beweisbarkeit	128
14	Der Begriff des Algorithmus	133

14.1	Lösen einer Sorte quadratischer Gleichungen	134
14.2	Zum informellen Algorithmusbegriff	135
14.3	Informelle Einführung des Hoare-Kalküls	136
14.4	Ein Algorithmus zur Multiplikation nichtnegativer ganzer Zahlen	142
15	Graphen	147
15.1	Gerichtete Graphen	147
15.1.1	Graphen und Teilgraphen	147
15.1.2	Pfade und Erreichbarkeit	149
15.1.3	Isomorphie von Graphen	151
15.1.4	Ein Blick zurück auf Relationen	152
15.2	Ungerichtete Graphen	153
15.2.1	Anmerkung zu Relationen	156
15.3	Graphen mit Knoten- oder Kantenmarkierungen (nicht im Wintersemester 2020/2021)	156
15.3.1	Gewichtete Graphen	157
16	Erste Algorithmen in Graphen	159
16.1	Repräsentation von Graphen im Rechner	159
16.2	Berechnung der 2-Erreichbarkeitsrelation und Rechnen mit Matrizen	163
16.2.1	Matrizenmultiplikation	165
16.2.2	Matrizenaddition	166
16.3	Berechnung der Erreichbarkeitsrelation	167
16.3.1	Potenzen der Adjazenzmatrix	168
16.3.2	Erste Möglichkeit für die Berechnung der Wegematrix	169
16.3.3	Zählen durchzuführender arithmetischer Operationen	170
16.3.4	Weitere Möglichkeiten für die Berechnung der Wegematrix	172
16.4	Algorithmus von Warshall (nicht im Wintersemester 2020/2021)	173
16.5	Ausblick	176
	Literatur	176
17	Quantitative Aspekte von Algorithmen	177
17.1	Ressourcenverbrauch bei Berechnungen	177
17.2	Groß-O-Notation	178
17.2.1	Ignorieren konstanter Faktoren	179
17.2.2	Notation für obere und untere Schranken des Wachstums	183
17.2.3	Die furchtbare Schreibweise	184
17.2.4	Rechnen im O-Kalkül	185

17.3	Matrizenmultiplikation	187
17.3.1	Rückblick auf die Schulmethode	188
17.3.2	Algorithmus von Strassen	189
17.4	Asymptotisches Verhalten „implizit“ definierter Funktionen	190
17.5	Unterschiedliches Wachstum einiger Funktionen	191
17.6	Ausblick	192
	Literatur	193
18	Endliche Automaten	195
18.1	Erstes Beispiel: ein Getränkeautomat	195
18.2	Mealy-Automaten	198
18.3	Moore-Automaten	200
18.4	Endliche Akzeptoren	202
18.4.1	Beispiele formaler Sprachen, die von endlichen Akzeptoren akzeptiert werden können	203
18.4.2	Eine formale Sprache, die von keinem endlichen Akzeptoren akzeptiert werden kann	205
18.5	Ausblick	206
19	Reguläre Ausdrücke und rechtslineare Grammatiken	209
19.1	Reguläre Ausdrücke	209
19.2	Rechtslineare Grammatiken	213
19.3	Kantorowitsch-Bäume und strukturelle Induktion	214
19.4	Ausblick	217
	Literatur	218
20	Turingmaschinen	219
20.1	Alan Mathison Turing	219
20.2	Turingmaschinen	219
20.2.1	Berechnungen	222
20.2.2	Eingaben für Turingmaschinen	224
20.2.3	Ergebnisse von Turingmaschinen	225
20.3	Berechnungskomplexität	226
20.3.1	Komplexitätsmaße	227
20.3.2	Komplexitätsklassen	230
20.4	Unentscheidbare Probleme	231
20.4.1	Codierungen von Turingmaschinen	231
20.4.2	Das Halteproblem	232
20.4.3	Die Busy-Beaver-Funktion	235

20.5 Ausblick 237
Literatur 237

21 Relationen 239

- 21.1 Äquivalenzrelationen 239
 - 21.1.1 Definition 239
 - 21.1.2 Äquivalenzklassen und Faktormengen 240
- 21.2 Kongruenzrelationen 240
 - 21.2.1 Verträglichkeit von Relationen mit Operationen 240
 - 21.2.2 Wohldefiniertheit von Operationen mit Äquivalenzklassen 241
- 21.3 Halbordnungen 241
 - 21.3.1 Grundlegende Definitionen 241
 - 21.3.2 „Extreme“ Elemente 244
 - 21.3.3 Vollständige Halbordnungen 245
 - 21.3.4 Stetige Abbildungen auf vollständigen Halbordnungen 246
- 21.4 Ordnungen 249
- 21.5 Ausblick 251

22 Mima-x 253

- 22.1 Stapel / Stack / Keller 253
 - 22.1.1 Der Stapel 253
 - 22.1.2 Operationen auf einem Stapel 253
 - 22.1.3 Implementierung eines Stapels 255
 - 22.1.4 Die Ackermann-Funktion 255
- 22.2 MiMa-X 257
 - 22.2.1 Architektur 257
 - 22.2.2 Stapelimplementierung 257
 - 22.2.3 „Flache“ Funktionsaufrufe 258
 - 22.2.4 „Richtige“ Funktionsaufrufe mit einem Stapel 259

Index 265

Literatur 273

Colophon 275

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.

2012: In Deutschland gab es Ende des Jahres etwa 30 Millionen Smartphones.

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 am Karlsruher Institut für Technologie vorgesehen ist.

Der vorliegende Text ist ein Vorlesungsskript zu dieser Veranstaltung.

1.1 AUFBAU DER VORLESUNG UND ZIELE

Seit dem Wintersemester 2015/2016 ist die Vorlesung „Grundbegriffe der Informatik“ von vorher zwei auf drei Vorlesungsstunden pro Woche vergrößert. Der Vorlesungsinhalt ist auf eine Reihe überschaubarer Kapitel aufgeteilt.

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

die Studenten mit wichtigen Vorgehensweisen bei der Definition neuer Begriffe und beim Beweis von Aussagen vertraut gemacht werden. Induktives Vorgehen ist in diesem Zusammenhang wohl zu allererst zu nennen.

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

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

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

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

1.2 QUELLEN

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

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

Außerdem findet man im WWW diverse Versionen eines Buches zu „*Mathematics for Computer Science*“, das ursprünglich von Lehman und Leighton ausgearbeitet wurde. Die aktuelle Version stammt von Lehman, Leighton und Meyer (2013).

Durch Ihre Mitarbeit an der Vorlesung und der zugehörigen großen Übung haben im Laufe der Jahre Matthias Schulz, Matthias Janke und Simon Janke Beiträge geleistet, die auch dieses Vorlesungsskript mit eingeflossen sind. Gespräche und Diskussionen mit ihnen und anderen 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. Explizit danken möchte ich in diesem Zusammenhang den Studenten Felix Lübke, Matthias Fritsche, Thomas Schwartz und Lukas Baldner, die eine ganze Reihe von Korrekturen geliefert haben.

Im Laufe des Jahres 2020 haben die ehemaligen Tutoren Philipp Berdesinski, Luca Buchholz und Patrick Fetzer die bisher noch fehlenden Kapitel 10 und 22 zum Aufbau eines einfachen Prozessors erarbeitet. Dafür danke ich Ihnen ganz herzlich.

Thomas Worsch, im November 2020.

LITERATUR

Abeck, Sebastian (2005). *Kursbuch Informatik, Band 1*. Universitätsverlag Karlsruhe (siehe S. 2).

Goos, Gerhard (2006). *Vorlesungen über Informatik: Band 1: Grundlagen und funktionales Programmieren*. Springer-Verlag (siehe S. 2).

Lehman, Eric, Tom Leighton und Albert R. Meyer (2013). *Mathematics for Computer Science*. URL: <http://courses.csail.mit.edu/6.042/fall13/class-material.shtml> (siehe S. 2).

Spitzer, Manfred (2002). *Lernen: Gehirnforschung und Schule des Lebens*. Spektrum Akademischer Verlag (siehe S. 2).

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 Grenze 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 diesem ersten Kapitel klären wollen, sind die wichtigen Begriffe *Signal*, *Nachricht*, *Information* und *Datum*.

2.1 SIGNAL

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



Abbildung 2.1: Eine Braillezeile, Quelle: http://commons.wikimedia.org/wiki/Image:Refreshable_Braille_display.jpg (17.10.2013)

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

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

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

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

2.2 ÜBERTRAGUNG UND SPEICHERUNG

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

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

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

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

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

2.3 NACHRICHT

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

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

Nachricht

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

Mit Inschriften werden wir uns ein erstes Mal in Kapitel 3 genauer beschäftigen und mit Speicherung in Kapitel 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	Programm- ausführung	59
	Interpretation↓	Interpre↓tation		Interpre↓tation
Mensch:	zweiund- vierzig	siebzehn	Rechnen→ 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 (d. h. speichern oder übertragen).

2.6 ZUSAMMENFASSUNG

Signale übertragen und Inschriften speichern Nachrichten.

Die Bedeutung einer Nachricht ist die ihr zugeordnete *Information* — im Rahmen eines 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 MENGEN, ALPHABETE, ABBILDUNGEN

Im Kapitel über Signale, Nachrichten, ... usw. 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.

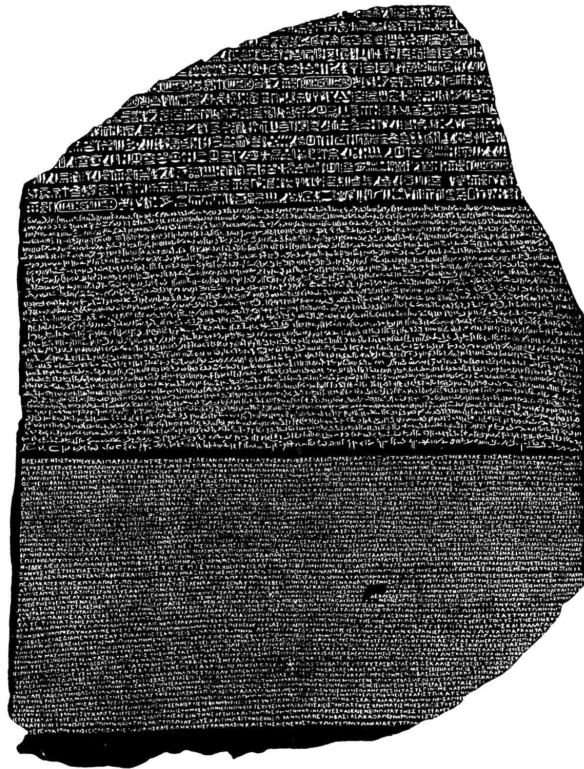
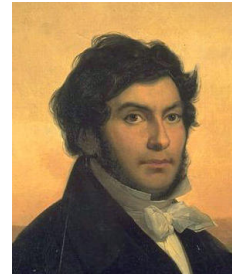


Abbildung 3.1: Der Rosetta-Stein, heute im Britischen Museum, London. Bildquelle: <http://www.gutenberg.org/ebooks/48649> (17.9.15)

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

Einen Vorrat an Zeichen, aus denen Texte zusammengesetzt werden, nennt man ein Alphabet. Das ist etwas präziser gesagt eine endliche Menge von Zeichen.

Daher werden wir in diesem Kapitel zuerst einige Anmerkungen zum Begriff der *Menge* machen. Anschließend werden wir insbesondere auf zwei wichtige Alphabete der Informatik zu sprechen kommen. Das wird dann auch gleich motivieren, warum wir uns mit *Relationen* und *Abbildungen* beschäftigen.

3.1 MENGEN

Jede Menge kann man sich als einen „Behälter“ vorstellen, der „Objekte“ enthält, zum Beispiel die Menge, die die Zahlen 1, 2 und 3 enthält und nichts sonst. Man sagt dann, dass 1 ein Element dieser Menge ist, und ebenso 2 und 3, aber keine anderen Objekte. Weder ist die 42 oder eine andere Zahl ein Element der Menge, noch andere „andersartige“ Objekte wie zum Beispiel der linke Schuh des Autors dieser Zeilen zu der Zeit, als er diese Zeilen schrieb.¹

eine ganz wichtige
allgemeine Bemerkung

An vielen Stellen in diesem Vorlesungsskript (und allgemein in der Mathematik und Informatik) steht man vor dem Problem, dass man über Dinge reden möchte, die nur (mehr oder weniger) mühsam (sprich: länglich) zu beschreiben sind. Statt dann immer wieder die längliche Beschreibung zu wiederholen, vergibt man eine *Namen* dafür. Zum Beispiel hätten wir oben sagen können:

„... zum Beispiel die Menge *A*, die die Zahlen 1, 2 und 3 ...“

Man sagt dann, dass 1 ein Element von *A* ist ...“

Damit ist *A* schlicht eine Abkürzung für etwas Längeres. Es ist ein Name für etwas ganz bestimmtes.

In einem zweiten Schritt geht man bei der Benutzung von Namen noch weiter. Gehen wir davon aus, dass *A* der Name für obige Menge ist. Dann sagt man auch so etwas wie: „Es sei *x* ein Objekt, das in *A* enthalten ist.“ Hier wird ein neuer Name eingeführt, nämlich „*x*“. Damit wird jetzt aber *kein* ganz bestimmtes Objekt bezeichnet, das in *A* enthalten ist. Vielmehr ist so etwas gemeint wie:

„Wir nehmen jetzt ein beliebiges Objekt her, das in *A* enthalten ist. Es ist gleichgültig, welches es ist. Wir nennen es *x*.“

Wenn dann im weiteren Verlauf in einer Aussage von „*x*“ die Rede ist, ist das eine Art Platzhalter, und man sollte sich klar machen, dass man ihn durch jedes konkrete Objekt ersetzen kann, das in *A* enthalten ist.

Element einer Menge

Es sei *A* eine Menge und *x* ein Objekt. Dann heißt *x* Element von *A*, wenn es in *A* enthalten ist, und sonst nicht. Wir bezeichnen mit

$$x \in A$$

¹eine blaue Sandale

die Aussage, die wahr ist, wenn x Element von A ist, und andernfalls falsch. Mit

$$x \notin A$$

bezeichnen wir das Gegenteil von $x \in A$, also die Aussage, die genau dann wahr ist, wenn x nicht Element von A ist. Je nachdem, für welche konkreten Werte die Namen x und A stehen, ist also immer eine der beiden Aussagen wahr und die andere falsch.

Anstelle von "Es sei x ein Element von A " (und "Es sei x ein Objekt so, dass $x \in A$ wahr ist") schreiben wir abkürzend "Es sei $x \in A$ ".

Jede Menge soll durch die Elemente, die sie enthält, eindeutig bestimmt sein: Für zwei Mengen A und B ist also genau dann $A = B$, wenn für jedes Objekt x gilt:

$$\text{wenn } x \in A, \text{ dann } x \in B \quad \text{und} \quad \text{wenn } x \in B, \text{ dann } x \in A .$$

Ist jedes Element einer Menge A auch Element einer Menge B , dann schreiben wir $A \subseteq B$ und sprechen davon, dass A *Teilmenge von* B sei und B *Obermenge* von A .
Folglich gilt

Teilmenge
Obermenge

3.1 Lemma. Für beliebige Mengen A und B ist

$$\text{genau dann } A = B, \text{ wenn } A \subseteq B \text{ und } B \subseteq A .$$

Mitunter will man auch die Tatsache notieren, dass A eine *echte Teilmenge* von B ist. Das notiert man oft in der Form $A \subsetneq B$ oder $A \subsetneqq B$. Manche Autoren schreiben das auch in der Form $A \subset B$, aber für andere bedeutet das das, wofür wir weiter oben $A \subseteq B$ geschrieben haben. Also seien Sie bei der Lektüre immer vorsichtig. Wir werden in diesem Skript nie $A \subset B$ schreiben.

echte Teilmenge

Kleine endliche Mengen kann man angeben, indem man alle Elemente aufzählt. Wir schließen die Aufzählung aller Elemente einer Menge in geschweifte Klammern ein:

$$\{1, 2, 3, 4, 5, 42, A, B\} .$$

Die kleinste Menge enthält gar keine Elemente. Für sie schreibt man

$$\{ \} \quad \text{oder} \quad \emptyset$$

und spricht von der *leeren Menge*.

leere Menge

Man beachte, dass man die gleiche Menge verschieden hinschreiben kann:

$$\{1, 2, 3\} = \{3, 2, 1\} = \{1, 1, 2, 3, 3, 2, 1, 3, 3, 3\} .$$

Unsere Definition der Gleichheit von Mengen verlangt nur, dass sie jeweils die gleichen Elemente enthalten müssen.

Bei großen oder gar unendlichen Mengen kann man nicht explizit alle Elemente einzeln auflisten. In solchen Fällen gehen wir anders vor. Zum einen setzen wir manche unendlichen Mengen einfach als bekannt voraus. Für die Menge der positiven ganzen Zahlen schreiben wir \mathbb{N}_+ und für die Menge der nichtnegativen ganzen Zahlen \mathbb{N}_0 . Es ist also sozusagen

$$\mathbb{N}_+ = \{1, 2, 3, 4, \dots\} \text{ und } \mathbb{N}_0 = \{0, 1, 2, 3, 4, \dots\} ,$$

wenn man einmal unterstellt, dass die Pünktchen von allen Lesern „richtig“ interpretiert werden.

set comprehension

Eine Möglichkeit auch unendliche Menge präzise zu spezifizieren bietet eine Notation, die im Englischen *set comprehension* heißt und für wir keine gute deutsche Bezeichnung kennen.² Dafür benötigt man für jedes Element x einer gewissen Grundmenge M eine Aussage $P(x)$, die wahr oder falsch ist. (Wenn $P(x)$ wahr ist, sagen wir auch, „ $P(x)$ gilt“.) Dann bezeichnet $\{x \in M \mid P(x)\}$ die Menge derjenigen Elemente $x \in M$, für die die Aussage $P(x)$ wahr ist, d. h. für jedes $y \in M$ gilt:

$$y \in \{x \in M \mid P(x)\} \text{ genau dann, wenn } P(y) \text{ wahr ist .}$$

Man liest das zum Beispiel als „die Menge aller $x \in M$, für die $P(x)$ wahr ist“. Zum Beispiel ist

$$\{x \in \mathbb{N}_+ \mid x \text{ ist ohne Rest durch 2 teilbar}\}$$

die Menge aller positiven geraden Zahlen. Wenn die Grundmenge M aus dem Kontext eindeutig hervorgeht, schreiben wir abkürzend $\{x \mid P(x)\}$.

Vereinigung

Für zwei Mengen A und B ist die *Vereinigung* $A \cup B$ eine Menge, die durch die Forderung eindeutig bestimmt ist, dass für jedes Objekt x gilt:

$$x \in A \cup B \text{ genau dann, wenn } x \in A \text{ oder } x \in B .$$

Durchschnitt

Dabei meinen wir mit „oder“ das inklusive Oder: Es ist auch erlaubt, dass x Element beider Mengen ist. Der *Durchschnitt* $A \cap B$ ist eine Menge, die durch die Forderung eindeutig bestimmt ist, dass für jedes Objekt x gilt:

$$x \in A \cap B \text{ genau dann, wenn } x \in A \text{ und } x \in B .$$

Es sei noch einmal daran erinnert, dass sozusagen jedes Element einer Menge immer nur „*einmal*“ enthalten ist. Daher ist zum Beispiel

$$\{1, 2, 3\} \cup \{3, 4, 5\} = \{1, 2, 3, 4, 5\} .$$

Zwei Mengen, deren Durchschnitt die leere Menge ist, nennt man *disjunkt*.

disjunkte Mengen

Allgemein ist zum Beispiel für jede Menge A stets $A \cup A = A$. Diese und weitere Aussagen sind in den folgenden Lemmata zusammengefasst.

3.2 Lemma. (Idempotenzgesetze) Für jede Menge A ist $A \cup A = A$ und $A \cap A = A$.

3.3 Lemma. (Kommutativgesetze) Für alle Mengen A und B ist $A \cup B = B \cup A$ und $A \cap B = B \cap A$.

3.4 Lemma. Für alle Mengen A und B ist $A \subseteq A \cup B$ und $A \cap B \subseteq A$.

3.5 Lemma. (Assoziativgesetze) Für alle Mengen A , B und C ist $(A \cup B) \cup C = A \cup (B \cup C)$ und $(A \cap B) \cap C = A \cap (B \cap C)$.

3.6 Lemma. (Distributivgesetze) Für alle Mengen A , B und C ist $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ und $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$.

Einer der wichtigsten Aspekte der Vorlesung „Grundbegriffe der Informatik“ ist, dass Sie lernen, präzise zu formulieren und zu argumentieren. Denn ein Algorithmus muss nicht nur präzise aufgeschrieben werden. Man muss auch nachweisen können, dass er das tut, „was er soll“. Dafür muss übrigens auch die Spezifikation präzise aufgeschrieben sein. Unter anderem aus diesem Grund werden wir in dieser Vorlesung viele Aussagen beweisen.

Die obigen Lemmata sind aber alle recht einfach. Deswegen beschränken wir uns hier auf einen Fall:

3.7 Beweis. (erster Teil von Lemma 3.2) Es sei A eine Menge. Um zu beweisen, dass $A = A \cup A$ ist, müssen wir nach Lemma 3.1 zeigen:

1. $A \subseteq A \cup A$
2. $A \cup A \subseteq A$

Sehen wir uns die beiden Fälle nacheinander an:

1. Wir müssen zeigen: Jedes Element $x \in A$ ist auch Element von $A \cup A$. Wenn man das ausführlich aufschreibt, ergibt sich in etwa folgendes:
 - Es sei $x \in A$.Wir beginnen mit der Voraussetzung. Und um zu zeigen, dass die Aussage für jedes Element gilt, argumentieren wir für ein „beliebiges“ Element, für das keine weiteren Annahmen gemacht werden.

²Die deutsche Bezeichnung „Intensionale Mengennotation“ ist nicht weit verbreitet.

- *Dann ist $x \in A$ oder $x \in A$.*
Wir ziehen eine einfache Schlussfolgerung, der unser umgangssprachliches Verständnis von „oder“ zugrunde liegt.
 - *Also ist $x \in A \cup A$.*
Diese Schlussfolgerung ist gerade das, was per Definition von Mengenvereinigung gelten muss, damit die Behauptung stimmt.
2. Die umgekehrte Richtung ist genauso einfach: Wir müssen zeigen: Jedes Element $x \in A \cup A$ ist auch Element von A . Die Struktur der Argumentation ist ähnlich wie eben:
- *Es sei $x \in A \cup A$.*
Wir beginnen wieder mit einem „beliebigen“ Element.
 - *Dann ist $x \in A$ oder $x \in A$.*
Das ergibt sich aufgrund der Definition von Mengenvereinigung.
 - *Also ist $x \in A$.*

■

Mengendifferenz

Neben Vereinigung und Durchschnitt ist gelegentlich auch noch die *Mengendifferenz* $A \setminus B$ nützlich:

$$A \setminus B = \{x \in A \mid x \notin B\}.$$

Es ist zum Beispiel

$$\{1, 2, 3\} \setminus \{3, 4, 5\} = \{1, 2\}.$$

In der nachfolgenden Abbildung 3.2 sind für zwei „allgemeine“ Mengen A und B die Lage einiger Teilmengen dargestellt. Es fehlt $A \cup B$; machen Sie sich klar, welcher Bereich des Bildes dazu gehört.

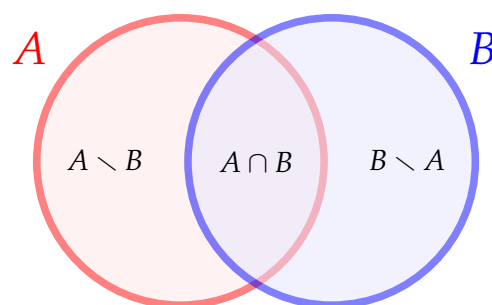


Abbildung 3.2: Teile zweier Mengen. Die linke, rote Kreisscheibe stelle eine Menge A dar und die rechte, blaue Kreisscheibe eine Menge B .

Die Anzahl der Elemente einer Menge A bezeichnet man auch als ihre *Kardinalität*, für die wir meist

Kardinalität

$$|A| \quad (\text{oder vereinzelt } \text{card } A)$$

schreiben werden. Für jede endliche Menge A ist das offensichtlich eine nichtnegative ganze Zahl. Für unendliche Mengen werden wir nicht definieren, was $|A|$ ist, und vermeiden darüber zu reden.

3.2 ALPHABETE

Unter einem *Alphabet* wollen wir eine endliche nichtleere Menge verstehen, deren Elemente wir *Zeichen* oder *Symbole* nennen. Was dabei genau „Zeichen“ sind, wollen wir nicht weiter hinterfragen. Es seien einfach die elementaren Bausteine, aus denen Inschriften zusammengesetzt sind. Stellen Sie sich zum Beispiel einige Buchstaben des deutschen Alphabetes vor. Hier sind einfache Beispiele:

Alphabet

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

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` `;`

So etwas wird Ihnen zum Beispiel in Vorlesungen über Übersetzerbau wieder begegnen.

Bereits in dieser Vorlesung werden wir im [Kapitel über aussagenlogische Formeln](#) annehmen, dass das zugrunde liegende Alphabet Symbole P_1, P_2, P_3 , usw. enthält. Oder Sie stellen sich vor, man hätte nur die Symbole P sowie die zehn Indexpfiffern $0, 1, 2, \dots, 9$, aus denen die „eigentlichen“ grundlegenden syntaktischen Bausteine zusammengesetzt sind.

3.2.1 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

ASCII

vollständige Liste findet man in Tabelle 3.1. Wie man dort sieht, fehlen in diesem Alphabet etliche Buchstaben aus nichtenglischen Alphabeten, wie zum Beispiel ä, ç, è, ã, ñ, œ, ß, û usw., von Kyrillisch, Japanisch und vielen anderen außereuropäischen Schriften ganz zu schweigen.

		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
<hr/>									
80	P	90	Z	100	d	110	n	120	x
81	Q	91	[101	e	111	o	121	y
82	R	92	\	102	f	112	p	122	z
83	S	93]	103	g	113	q	123	{
84	T	94	^	104	h	114	r	124	
85	U	95	_	105	i	115	s	125	}
86	V	96	`	106	j	116	t	126	~
87	W	97	a	107	k	117	u		
88	X	98	b	108	l	118	v		
89	Y	99	c	109	m	119	w		

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

Auf ein Zeichen in Tabelle 3.1 sei ausdrücklich hingewiesen, nämlich das mit Nummer 32. Das ist das „Leerzeichen“. Man gibt es normalerweise auf einer Rechnerntastatur 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.

3.2.2 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.

Der Unicode-Standard spezifiziert weitaus mehr als nur einen Zeichensatz.³ Für uns sind hier zunächst nur die beiden folgenden Aspekte wichtig:

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 allerdings nicht „zusammenhängend“.

Jedenfalls liegt eine Beziehung zwischen Unicode-Zeichen und nichtnegativen ganzen Zahlen vor. Man spricht von einer Relation. (Wenn Ihnen die folgenden Zeilen schon etwas sagen: schön. Wenn nicht, gedulden Sie sich bis Abschnitt 3.3 wenige Zeilen weiter.)

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

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

Entsprechendes gilt natürlich auch für den ASCII-Zeichensatz.

3.3 RELATIONEN UND ABBILDUNGEN

3.3.1 Paare, Tupel und kartesische Produkte

Es seien A und B zwei Mengen. Für Elemente $a \in A$ und $b \in B$ wird durch

$$(a, b)$$

etwas notiert, was man als *Paar*, *geordnetes Paar* oder *Tupel* bezeichnet. Der Teil a vor dem Komma heißt *erste Komponente* des Paares und der Teil b nach dem Komma *zweite Komponente*.

Paar
Tupel

Zwei Paare (a, b) und (x, y) sind genau dann gleich, wenn $a = x$ ist und $b = y$. Also sind $(1, 2)$ und $(2, 1)$ *verschiedene* Paare. Außerdem können erste und zweite

³Hinzu kommt etwa die Sortierreihenfolge von Buchstaben in verschiedenen Sprachen.

Komponente eines Paares durchaus gleich sein. Auch $(1, 1)$ ist ein Paar mit zwei Komponenten.

*kartesisches
Produkt*

Das *kartesische Produkt* der Mengen A und B , geschrieben $A \times B$, ist die Menge aller Paare (a, b) mit $a \in A$ und $b \in B$:

$$A \times B = \{(a, b) \mid a \in A \text{ und } b \in B\}.$$

Manchmal ist es bequem, mit Verallgemeinerungen von Paaren zu arbeiten, die nicht genau zwei Komponenten haben, sondern zum Beispiel drei. Man schreibt dann

$$M_1 \times M_2 \times M_3 = \{(x_1, x_2, x_3) \mid x_1 \in M_1 \text{ und } x_2 \in M_2 \text{ und } x_3 \in M_3\}.$$

Tripel

Die Elemente solcher Mengen nennt man auch *Tripel*.

n-Tupel

Allgemeiner nennt man die Elemente eines kartesischen Produkts von n Mengen auch *n-Tupel*.

Sind die Mengen M_i eines kartesischen Produktes alle gleich einer Menge M , dann spricht man auch vom *n-fachen kartesischen Produkt der Menge M mit sich selbst* und notiert dies kurz als M^n .

3.3.2 Allquantor und Existenzquantor

Gleich werden wir uns mit grundlegenden Begriffen im Zusammenhang mit Relationen und Abbildungen beschäftigen. Dabei werden oft Formulierungen gebraucht werden, die von einer beiden folgenden Strukturen sind:

Für jedes Element $x \in A$ gilt ...

oder

Es gibt ein Element $x \in A$...

*Allquantor
Existenzquantor*

Da so etwas ganz oft auftritt, haben sich Mathematiker schon im 19. Jahrhundert eine abkürzende Schreibweise überlegt. Was es mit dem sogenannten *Allquantor* \forall und dem sogenannten *Existenzquantor* \exists genauer auf sich hat, werden wir in Kapitel 13 sehen. Bis dahin erlauben wir uns aber, statt „für jedes Element $x \in A$ gilt“ kürzer „ $\forall x \in A$ “ zu schreiben und statt „es gibt ein Element $x \in A$ “ kürzer „ $\exists x \in A$ “. Dabei ist noch wichtig, dass mit der Formulierung „es gibt ein“ immer gemeint ist: „es gibt *mindestens* ein“ ! Wenn die zugrunde liegende Menge A klar ist, werden wir sie manchmal auch weglassen.

3.3.3 Relationen und Abbildungen

Die Beziehung zwischen den Unicode-Zeichen in A_U und nichtnegativen ganzen Zahlen kann man durch die Angabe aller Paare (a, n) , für die $a \in A_U$ ist und n

der zu a gehörenden Code Point, vollständig beschreiben. Für die Menge U aller dieser Paare gilt also $U \subseteq A_U \times \mathbb{N}_0$.

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 “. Das kann man auf mehr als zwei Faktoren verallgemeinern: Eine *ternäre Relation* ist zum Beispiel eine Teilmenge $R \subseteq A \times B \times C$.

Relation
binäre Relation
ternäre 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 $a \in A$ (mindestens) ein $b \in B$ existiert mit $(a, b) \in R$. Mit Quantoren können wir diese Aussage so aufschreiben: $\forall a \in A \exists b \in B : (a, b) \in R$.

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

Mit Quantoren: Es gilt *nicht*: $\exists a \in A \exists b_1 \in B \exists b_2 \in B : b_1 \neq b_2$ und $(a, b_1) \in R$ und $(a, b_2) \in R$. Unter Umständen findet man (jedenfalls im Moment noch) die folgende äquivalente Formulierung leichter verständlich: $\forall a \in A \forall b_1 \in B \forall b_2 \in B : \text{wenn } (a, b_1) \in R \text{ und } (a, b_2) \in R, \text{ dann ist } b_1 = b_2$.

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 $R : A \rightarrow B$. Es heißt dann A der *Definitionsbereich* und B der *Zielbereich* der Abbildung.

Abbildung
Funktion
Definitionsbereich
Zielbereich

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

partielle Abbildung

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 gilt:

linkseindeutig

$$\forall (a_1, b_1) \in R \forall (a_2, b_2) \in R : \text{wenn } a_1 \neq a_2, \text{ dann } b_1 \neq b_2 .$$

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

injektiv

rechtstotal

- surjektiv 7. Eine Abbildung, die rechtstotal ist, heißt *surjektiv*.
 bijektiv 8. Eine Abbildung, die sowohl injektiv als auch surjektiv ist, heißt *bijektiv*.

Im Laufe der Vorlesung wird es immer wieder notwendig sein, Abbildungen (und Relationen) zu definieren. Dazu gehört als erstes immer die Angabe von Definitionsbereich und Zielbereich (denn davon ist ja zum Beispiel unter Umständen abhängig, ob eine Abbildung injektiv ist). Wir notieren das wie oben erwähnt in der Form

$$f: A \rightarrow B.$$

Funktionswert In dieser Situation gibt es für jedes $a \in A$ genau ein $b \in B$ mit $(a, b) \in f$. Man schreibt dann üblicherweise $f(a) = b$ und nennt b den Wert oder Funktionswert von f an der Stelle a . Wenn für jedes Element a des Definitionsbereichs der Funktionswert $f(a)$ spezifiziert werden soll, schreiben wir das oft in einer der Formen

$$\begin{aligned} x &\mapsto \text{Ausdruck, in dem } x \text{ vorkommen darf,} \\ (x, y) &\mapsto \text{Ausdruck, in dem } x \text{ und } y \text{ vorkommen dürfen,} \end{aligned}$$

falls der Definitionsbereich von f eine Menge bzw. das kartesische Produkt zweier Mengen ist. (Wenn der Definitionsbereich kartesisches Produkt von mehr als zwei Mengen ist, gehen wir analog vor.) Im ersten Fall wird ein Name (hier x) für den Wert eingeführt, auf den die Abbildung angewendet wird. Im zweiten Fall wird angenommen, dass der Wert ein Paar ist, es werden sogar Namen (x und y) für seine „Bestandteile“ eingeführt und es wird darauf vertraut, dass allen klar ist, welche Bestandteile gemeint sind.

Auf der rechten Seite der Definition von Funktionswerten erlauben wir uns die Freiheit, nicht nur einfache arithmetische Ausdrücke zu notieren, sondern z. B. auch Fallunterscheidungen. Hier ist ein einfaches Beispiel:

$$f: \mathbb{N}_0 \rightarrow \mathbb{N}_0, n \mapsto \begin{cases} n/2, & \text{falls } n \text{ gerade,} \\ 3n + 1, & \text{falls } n \text{ ungerade.} \end{cases}$$

Man beachte, dass die Namen die in der Definition der Abbildung verwendet werden, völlig irrelevant sind. Genau die gleiche Abbildung wie eben wird definiert, wenn man schreibt:

$$g: \mathbb{N}_0 \rightarrow \mathbb{N}_0, x \mapsto \begin{cases} x/2, & \text{falls } x \text{ gerade,} \\ 3x + 1, & \text{falls } x \text{ ungerade.} \end{cases}$$

Mitunter ist die folgende Verallgemeinerung der Schreibweise $f(a)$ nützlich. Ist $f: A \rightarrow B$ und $M \subseteq A$, so sei

$$f(M) = \{b \in B \mid \text{es gibt ein } a \in M \text{ mit } f(a) = b\}.$$

Das ist mühsam (zu schreiben und) zu lesen, weshalb wir folgende verallgemeinerte Notation für *set comprehensions* erlauben:

$$\begin{aligned} &\text{statt } \{b \in B \mid \text{es gibt ein } a \text{ mit } f(a) = b \text{ und } P(a)\} \\ &\text{kurz } \{f(a) \mid P(a)\}. \end{aligned}$$

Damit ist $f(M) = \{f(a) \mid a \in M\}$. Für eine Abbildung $f: A \rightarrow B$ heißt $f(A)$ auch das *Bild* der Abbildung.

Bild

3.4 MEHR ZU MENGEN

Elemente einer Menge können ihrerseits wieder Mengen sein. Zum Beispiel enthält die Menge

$$M = \{1, \{2, 3\}, 4, 5, 6, \{7, 8, 9\}, \{\}\}$$

sieben (!) Elemente, von denen drei ihrerseits wieder Mengen sind, nämlich $\{2, 3\}$, $\{7, 8, 9\}$ und die leere Menge $\{\}$. Man beachte auch, dass die Menge $\{\{\}\}$ ein Element enthält (die leere Menge), also selbst offensichtlich *nicht* leer ist!

Zu jeder Menge M definiert man die *Potenzmenge* als die Menge *aller* Teilmengen von M . Wir schreiben dafür 2^M , einige andere Autoren benutzen die Notation $\mathfrak{P}(M)$. Also:

Potenzmenge

$$2^M = \{A \mid A \subseteq M\}.$$

Betrachten wir als Beispiel $M = \{1, 2, 3\}$. Dann ist

$$2^{\{1, 2, 3\}} = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Allgemein kann man also immer statt $A \subseteq M$ auch $A \in 2^M$ schreiben; und geltentlich tut man das auch.

Zu zwei beliebigen Mengen I und M betrachten wir nun eine Abbildung

$$f: I \rightarrow 2^M.$$

Wenn es Ihnen die Sache erleichtert, stellen Sie sich vor, es sei z. B. $I = \mathbb{P}_2 = \{1, 2\}$ oder $I = \mathbb{N}_0$. Es ist jedenfalls für jedes $i \in I$ der Funktionswert $f(i)$ ein Element von 2^M , also eine Teilmenge von M . Statt $f(i)$ schreiben wir im Folgenden M_i . Wenn $I = \{1, 2\}$ ist, dann sind also einfach zwei Mengen M_1 und M_2 festgelegt.

Und wenn $I = \mathbb{N}_0$ ist, dann hat man eine sogenannte *abzählbar unendliche* Folge von Mengen (genauer Teilmengen von M) M_0, M_1, M_2 , und so weiter.

Vereinigung
Durchschnitt

Als Verallgemeinerung von Durchschnitt und Vereinigung zweier Mengen definieren wir die *Vereinigung* $\bigcup_{i \in I} M_i$ und den *Durchschnitt* $\bigcap_{i \in I} M_i$ aller M_i so:

$$\bigcup_{i \in I} M_i = \{x \mid \text{es gibt ein } i \in I \text{ so, dass } x \in M_i\} ,$$

$$\bigcap_{i \in I} M_i = \{x \mid \text{für jedes } i \in I \text{ ist } x \in M_i\} .$$

Machen Sie sich bitte klar, dass im Fall $I = \{1, 2\}$ gilt:

$$\bigcup_{i \in I} M_i = M_1 \cup M_2 \text{ und}$$

$$\bigcap_{i \in I} M_i = M_1 \cap M_2 .$$

ZUSAMMENFASSUNG UND AUSBLICK

In diesem Kapitel wurde zunächst auf den Begriff der *Menge* eingegangen und es wurden einfache Operationen mit Mengen definiert.

Dann wurde der Begriff *Alphabet* eingeführt, der im weiteren Verlauf dieser Vorlesung und des Informatikstudiums noch an vielen Stellen eine Rolle spielen wird. Wer mehr über Schriften wissen möchte, findet zum Beispiel über die WWW-Seite <http://www.omniglot.com/writing/> (6.10.2015) weitere Informationen.

Als wichtige technische Hilfsmittel wurden die Begriffe *binäre Relation*, sowie *injektive*, *surjektive* und *bijektive Abbildungen* definiert.

Viele Aussagen hatten noch den Nachteil, dass sie umgangssprachlich und daher relativ weitschweifig formuliert waren. Nachdem im nächsten Kapitel die Begriffe *Wort* und *formale Sprache* eingeführt worden sein werden, werden wir uns daher anschließend ein erstes Mal mit *Aussagenlogik* (und in einem späteren Kapitel auch mit *Prädikatenlogik* erster Stufe) beschäftigen.

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 Kapiteln noch wichtig werden.

Wort über einem Alphabet A

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

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

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

Dann wollen wir jede *surjektive* Abbildung $w : \mathbb{Z}_n \rightarrow A$ als ein *Wort* auffassen. Die Zahl n heie auch die *Lnge* des Wortes, fr die man manchmal kurz $|w|$ schreibt. (Es ist in Ordnung, wenn Sie im Moment nur an Lngen $n \geq 1$ denken. Auf den Fall des sogenannten leeren Wortes ε mit Lnge $n = 0$ kommen wir im *nachfolgenden Abschnitt* gleich noch zu sprechen.)

Wort, formalistisch definiert
Lnge eines Wortes

Das Wort (im umgangssprachlichen Sinne) $w = \text{hallo}$ ist dann also formal die Abbildung $w : \mathbb{Z}_5 \rightarrow \{a, h, l, o\}$ 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 Wrter 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 ermglicht dreierlei:

- przise Argumentationen, wenn andernfalls nur vages Hndewedeln mglich wre,

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

Menge aller Wörter
 A^*

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

4.2 DAS LEERE WORT

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

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

Wenn es Ihnen hilft, können Sie sich die formalistische Definition ansehen: Es ist $\mathbb{Z}_0 = \{\}$ 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 löst sich aber, wenn man daran denkt, 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 Kapitel 2) müssen, nämlich als das leere Wort.

4.3 MEHR ZU WÖRTERN

Für die Menge aller Wörter einer festen Länge n über einem Alphabet A schreiben wir auch A^n . Wenn zum Beispiel das zu Grunde liegende Alphabet $A = \{a, b\}$ 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 das „große Vereinigungszeichen“ benutzen, das wir in Kapitel 3 kennengelernt haben:

$$A^* = \bigcup_{i \in \mathbb{N}_0} A^i$$

4.4 KONKATENATION VON WÖRTERN

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

Für Wörter definieren wir nun auch eine ganz einfache aber wichtige binäre Operation: die sogenannte *Konkatenation* von Wörtern. Das ist einfach die Hin-

Konkatenation

tereinanderschreibung zweier Wörter. Als Operationssymbol verwendet man üblicherweise wie bei der Multiplikation von Zahlen den Punkt „·“. Also zum Beispiel:

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

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**.

*Konkatenation zweier
Wörter formal*

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

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

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

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

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 $i \in \mathbb{Z}_{m+n}$ 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 $w_1 \cdot w_2 : \mathbb{Z}_{m+n} \rightarrow A_1 \cup A_2$ 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 $i_1 \in \mathbb{Z}_m$ 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 $i_2 \in \mathbb{Z}_n$ mit $w_2(i_2) = a$. Also ist $(w_1 w_2)(m + i_2) = w_2(i_2) = a$.

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

4.2 Lemma. Für jedes Alphabet A gilt:

für jedes $w_1 \in A^*$ und jedes $w_2 \in A^*$ ist $|w_1 w_2| = |w_1| + |w_2|$.

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:

für jedes $x \in \mathbb{N}_0$ ist $x + 0 = x$ und $0 + x = x$.

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

neutrales Element

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

4.3 Lemma. Für jedes Alphabet A gilt:

für jedes $w \in A^*$ ist $w \cdot \varepsilon = w$ und $\varepsilon \cdot w = w$.

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

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

4.4 Beweis. Die erste Frage, die sich stellt, ist: Wie beweist man das für alle denkbaren(?) Alphabete A ? Eine Möglichkeit ist: Man geht von einem beliebigen Alphabet A aus, an dem man für den Rest des Beweises „festhält“, und über das man *keinerlei weitere* Annahmen macht, und zeigt, dass die Aussage für dieses A gilt.

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

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

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

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

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

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

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 Konkatination

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 Konkatination von Wörtern nicht einfach die Reihenfolge vertauschen darf. (Man sagt auch, die Konkatination sei *nicht kommutativ*.)

Was ist, wenn man mehrere Wörter konkatiniert? 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 Konkatination mehrerer Wörter keine Klammern setzen muss. Man sagt auch, die Konkatination sei eine *assoziative Operation* (siehe Unterabschnitt 4.6).

4.4.3 Beispiel: Aufbau von E-Mails

RFC
Request For Comments

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

RFC 5322

Die aktuelle Fassung der Spezifikation von E-Mails findet man in RFC 5322

(<http://tools.ietf.org/html/rfc5322>, 24.10.14). Wir zitieren und kommentieren¹ einige Ausschnitte aus der Einleitung von Abschnitt 2.1 dieses RFC, wobei die Nummerierung/Strukturierung von uns stammt:

1.
 - „*This document specifies that messages are made up of characters in the US-ASCII range of 1 through 127.*“
 - Das Alphabet, aus dem die Zeichen stammen müssen, die in einer E-Mail vorkommen, ist der US-ASCII-Zeichensatz mit Ausnahme des Zeichens mit der Nummer 0.
2.
 - „*Messages are divided into lines of characters. A line is a series of characters that is delimited with the two characters carriage-return and line-feed; that is, the carriage return (CR) character (ASCII value 13) followed immediately by the line feed (LF) character (ASCII value 10). (The carriage-return/line-feed pair is usually written in this document as "CRLF".)*“
 - Eine Zeile (*line*) ist eine Folge von Zeichen, also ein Wort, das mit den beiden „nicht druckbaren“ Symbolen CR LF endet.
An anderer Stelle wird im RFC übrigens spezifiziert, dass als Zeile im Sinne des Standards nicht beliebige Wörter zulässig sind, sondern nur solche, deren Länge kleiner oder gleich 998 ist.
3.
 - *A message consists of*
 - [...] *the header section of the message [...] followed,*
 - *optionally, by a body.*“
 - Eine E-Mail (*message*) ist die Konkatenation von Kopf (*header*) der E-Mail und Rumpf (*body*) der E-Mail. Dass der Rumpf optional ist, also sozusagen fehlen darf, bedeutet nichts anderes, als dass der Rumpf auch das leere Wort sein darf. (Aus dem Rest des RFC ergibt sich, dass der Kopf nie das leere Wort sein kann.)
Aber das ist noch nicht ganz vollständig. Gleich anschließend wird der RFC genauer:
4.
 - „*The header section is a sequence of lines of characters with special syntax as defined in this specification.*“
 - *The body is simply a sequence of characters that follows the header and*
 - *is separated from the header section by an empty line (i.e., a line with nothing preceding the CRLF). [...]“*
 - Es gilt also:
 - Der Kopf einer E-Mail ist die Konkatenation (de facto mehrerer) Zeilen.

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

- Der Rumpf einer E-Mail ist (wie man an anderer Stelle im RFC nachlesen kann) ebenfalls die Konkatenation von Zeilen. Es können aber auch 0 Zeilen oder 1 Zeile sein.
- Eine Leerzeile (*empty line*) ist das Wort `CR LF`.
- Eine Nachricht ist die Konkatenation von Kopf der E-Mail, einer Leerzeile und Rumpf der E-Mail.

4.4.4 Iterierte Konkatenation

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

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

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?

Potenzen von Wörtern

Dafür benutzen wir wieder eine induktive Definition wie schon beim kartesischen Produkt mehrerer Mengen. Für *Potenzen von Wörtern* beginnen wir mit dem Fall $n = 0$:

$$w^0 = \varepsilon$$

für jedes $k \in \mathbb{N}_0 : w^{k+1} = w^k \cdot w$

Man kann nun 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.

4.5 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).

4.6 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:

kommutative Operation

für jedes $x \in M$ und für jedes $y \in M$ ist $x \diamond y = y \diamond x$.

Abgesehen davon, dass wir bald eine kompaktere Notation für „für jedes“ kennenlernen werden, könnte man etwas platzsparender auch schreiben

für alle $x \in M$ und $y \in M$ ist $x \diamond y = y \diamond x$.

Man beachte, dass dabei x und y durchaus auch für den gleichen Wert stehen dürfen. Die sich ergebende Forderung ist allerdings trivialerweise erfüllt.

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

$$\text{für alle } x \in M, y \in M \text{ und } z \in M \text{ ist } (x \diamond y) \diamond z = x \diamond (y \diamond z) .$$

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

ZUSAMMENFASSUNG UND AUSBLICK

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

Als wichtiges technisches Hilfsmittel haben wir erstmals eine *induktive Definition* gesehen. So etwas wird im weiteren Verlauf der Vorlesung immer wieder vorkommen.

Bisher haben wir an vielen Stellen Aussagen durch die Wörter „oder“ und „und“ miteinander verbunden. Da das auf Dauer etwas mühsam ist, werden wir uns kommenden Kapitel ein erstes Mal ausführlich mit Aussagenlogik beschäftigen und dabei auch Abkürzungen einführen, die kürzere, besser lesbare Formeln erlauben.

5 AUSSAGENLOGIK

5.1 INFORMELLE GRUNDLAGEN

Im früheren Kapiteln haben wir deutsche Sätze wie z. B. den folgenden geschrieben:

„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 zugrunde liegende Nachricht zusammengesetzt ist.

Der klassischen Aussagenlogik liegen zwei wesentliche Annahmen zugrunde. Die erste ist:

- Jede Aussage ist entweder falsch oder wahr.

Man spricht auch von der *Zweiwertigkeit* der Aussagenlogik. Eine Aussage kann nicht sowohl falsch als auch wahr sein, und es gibt auch keine anderen Möglichkeiten. Wenn etwas nicht entweder wahr oder falsch ist, dann ist es keine Aussage. Überlegen Sie sich, ob Ihnen Formulierungen einfallen, die keine Aussagen sind.

Zweiwertigkeit der Aussagenlogik

Außerdem bauen wir natürlich in dieser Vorlesung ganz massiv darauf, dass es keine Missverständnisse durch unterschiedliche Interpretationsmöglichkeiten einer Aussage gibt, damit auch klar ist, ob sie wahr oder falsch ist.

Häufig setzt man aus einfachen Aussagen, im folgenden kurz mit P und Q bezeichnet, kompliziertere auf eine der folgenden Arten zusammen:

- **Negation:** „Nicht P “
- **logisches Und:** „ P und Q “
- **logisches Oder:** „ P oder Q “
- **logische Folgerung:** „Wenn P , dann Q “

Die zweite Grundlage der Aussagenlogik ist dies:

- Der Wahrheitswert einer zusammengesetzten Aussage ist durch die Wahrheitswerte der Teilaussagen eindeutig festgelegt.

Ob so eine zusammengesetzte Aussage wahr oder falsch ist, soll also insbesondere *nicht* vom konkreten Inhalt der Aussagen abhängen! Das betrifft auch die aussagenlogische Folgerung. Sind zum Beispiel die Aussagen

P : „Im Jahr 2014 wurden in Japan etwa 4.7 Millionen PkW neu zugelassen.“¹ und

¹<http://de.statista.com/statistik/daten/studie/279897/umfrage/pkw-neuzulassungen-in-japan/>

Q: „Im Jahr 1999 gab es in Deutschland etwa 11.2 Millionen Internet-Nutzer“²
gegeben, dann soll „Wenn P, dann Q“

- erstens tatsächlich eine Aussage sein, die wahr oder falsch ist, obwohl natürlich *kein* kausaler Zusammenhang zwischen den genannten Themen existiert und
- zweitens soll der Wahrheitswert dieser Aussage nur von den Wahrheitswerten der Aussagen P und Q abhängen (und wie wir sehen werden im vorliegenden Beispiel *wahr* sein).

Da in der Aussagenlogik nur eine Rolle spielt, welche Wahrheitswerte die elementaren Aussagen haben, aus denen kompliziertere Aussagen zusammengesetzt sind, beschränkt und beschäftigt man sich dann in der Aussagenlogik mit sogenannten *aussagenlogischen Formeln*, die nach Regeln ähnlich den oben genannten zusammengesetzt sind und bei denen statt elementarer Aussagen einfach *Aussagevariablen* stehen, die als Werte „wahr“ und „falsch“ haben können.

aussagenlogische
Formeln
Aussagevariablen

Der Rest dieses Kapitels ist wie folgt aufgebaut: In Abschnitt 5.2 werden wir definieren, wie aussagenlogische Formeln syntaktisch aufgebaut sind. Wir werden das zunächst auf eine Art und Weise tun, für die uns schon alle Hilfsmittel zur Verfügung stehen. Eine andere, in der Informatik weit verbreitete Vorgehensweise werden wir im Kapitel über kontextfreie Grammatiken kennenlernen. In Abschnitt 5.3 führen wir sogenannte boolesche Funktionen ein. Sie werden dann in Abschnitt 5.4 benutzt, um die Semantik, d. h. die Bedeutung, aussagenlogischer Formeln zu definieren.

In diesem Kapitel orientieren wir uns stark am Skriptum zur Vorlesung „Formale Systeme“ von Schmitt (2013).

5.2 SYNTAX AUSSAGENLOGISCHER FORMELN

Grundlage für den Aufbau aussagenlogischer Formeln ist ein Alphabet

$$A_{AL} = \{ (,), \neg, \wedge, \vee, \rightarrow \} \cup Var_{AL} .$$

aussagen-
logische
Konnektive

Die vier Symbole \neg , \wedge , \vee und \rightarrow heißen auch *aussagenlogische Konnektive*. Und Var_{AL} ist ein Alphabet, dessen Elemente wir *aussagenlogische Variablen* nennen. Wir notieren sie in der Form P_i , mit $i \in \mathbb{N}_0$. Weil schon bei kleinen Beispielen die Lesbarkeit durch die Indizes unnötig beeinträchtigt wird, erlauben wir, als Abkürzungen für P_0 , P_1 , P_2 und P_3 einfach P , Q , R und S zu benutzen.

²<http://de.statista.com/statistik/daten/studie/36146/umfrage/anzahl-der-internetnutzer-in-deutschland-seit-1997/>

Die Menge der syntaktisch korrekten aussagenlogischen Formeln soll nun definiert werden als eine formale Sprache For_{AL} über einem Alphabet A_{AL} , also als Teilmenge von A_{AL}^* .

Eine erste Forderung ist, dass stets $Var_{AL} \subseteq For_{AL}$ sein soll. Damit ist also jede aussagenlogische Variable eine aussagenlogische Formel.

Als zweites wollen wir nun definieren, dass man aus „einfacheren“ aussagenlogischen Formeln „kompliziertere“ zusammensetzen kann. Für die verschiedenen Möglichkeiten definieren wir uns vier Funktionen:

$$\begin{aligned} f_{\neg} : A_{AL}^* &\rightarrow A_{AL}^* : G \mapsto (\neg G) \\ f_{\wedge} : A_{AL}^* \times A_{AL}^* &\rightarrow A_{AL}^* : (G, H) \mapsto (G \wedge H) \\ f_{\vee} : A_{AL}^* \times A_{AL}^* &\rightarrow A_{AL}^* : (G, H) \mapsto (G \vee H) \\ f_{\rightarrow} : A_{AL}^* \times A_{AL}^* &\rightarrow A_{AL}^* : (G, H) \mapsto (G \rightarrow H) \end{aligned}$$

Wie man sieht, sind die Funktionswerte jeweils definiert durch geeignete Konkationen von Argumenten und Symbolen aus dem Alphabet der Aussagenlogik. Ein sinnvolles Beispiel ist

$$f_{\wedge}((P \rightarrow Q), R) = ((P \rightarrow Q) \wedge R)$$

Ein für unser Vorhaben sinnloses Beispiel ist

$$f_{\wedge}(\neg \rightarrow), R \vee) = (\neg \rightarrow) \wedge R \vee)$$

Wie man sieht, führt jede Anwendung einer der Abbildungen dazu, dass die Anzahl der Konnektive um eins größer wird als die Summe der Anzahlen der Konnektive in den Argumenten.

Für die aussagenlogischen Formeln ist es üblich, sie wie folgt zu lesen:

$$\begin{aligned} (\neg G) &\text{ „nicht } G\text{“} \\ (G \wedge H) &\text{ „} G \text{ und } H\text{“} \\ (G \vee H) &\text{ „} G \text{ oder } H\text{“} \\ (G \rightarrow H) &\text{ „} G \text{ impliziert } H\text{“ (oder „aus } G \text{ folgt } H\text{“)} \end{aligned}$$

Wir definieren nun induktiv unendlich viele Mengen wie folgt:

$$\begin{aligned} M_0 &= Var_{AL} \\ \text{für jedes } n \in \mathbb{N}_0 : M_{n+1} &= M_n \cup f_{\neg}(M_n) \\ &\quad \cup f_{\wedge}(M_n \times M_n) \cup f_{\vee}(M_n \times M_n) \cup f_{\rightarrow}(M_n \times M_n) \\ For_{AL} &= \bigcup_{i \in \mathbb{N}_0} M_i \end{aligned}$$

Menge der
aussagenlogischen
Formeln

Die formale Sprache For_{AL} ist die Menge der aussagenlogischen Formeln für die Variablenmenge Var_{AL} . Wie man anhand der Konstruktion in der zweiten Zeile sieht, ist für jedes $n \in \mathbb{N}_0$ stets $M_n \subseteq M_{n+1}$. Die Mengen werden also „immer größer“. Außerdem enthält jedes M_n und damit auch For_{AL} nur Aussagevariablen sowie Werte der vier involvierten Abbildungen, die entstehen können, wenn die Argumente aussagenlogische Formeln sind. Wenn man also für alle aussagenlogischen Formeln etwas definieren möchte, dann genügt es auch, das nur für Aussagevariablen sowie Werte der vier involvierten Abbildungen zu tun, die entstehen können, wenn die Argumente aussagenlogische Formeln sind.

Um die Konstruktion von For_{AL} noch ein bisschen besser zu verstehen, beginnen wir der Einfachheit halber einmal mit $Var_{AL} = \{P, Q\}$. Dann ist also

$$M_0 = \{P, Q\}.$$

Daraus ergeben sich

$$\begin{aligned} f_{\neg}(M_0) &= \{(\neg P), (\neg Q)\} \\ f_{\wedge}(M_0 \times M_0) &= \{(P \wedge P), (P \wedge Q), (Q \wedge P), (Q \wedge Q)\} \\ f_{\vee}(M_0 \times M_0) &= \{(P \vee P), (P \vee Q), (Q \vee P), (Q \vee Q)\} \\ f_{\rightarrow}(M_0 \times M_0) &= \{(P \rightarrow P), (P \rightarrow Q), (Q \rightarrow P), (Q \rightarrow Q)\} \end{aligned}$$

und folglich ist

$$\begin{aligned} M_1 = \{ &P, Q, \\ &(\neg P), (\neg Q), \\ &(P \wedge P), (P \wedge Q), (Q \wedge P), (Q \wedge Q), \\ &(P \vee P), (P \vee Q), (Q \vee P), (Q \vee Q), \\ &(P \rightarrow P), (P \rightarrow Q), (Q \rightarrow P), (Q \rightarrow Q)\} \end{aligned}$$

Für M_2 ergeben sich bereits so viele Formeln, dass man sie gar nicht mehr alle hinschreiben will. Beispiele sind $(\neg(\neg P))$, $((P \rightarrow Q) \wedge P)$ oder $(Q \vee (\neg Q))$.

Weil es nützlich ist, führen wir noch eine Abkürzung ein. Sind G und H zwei aussagenlogische Formeln, so stehe $(G \leftrightarrow H)$ für $((G \rightarrow H) \wedge (H \rightarrow G))$.

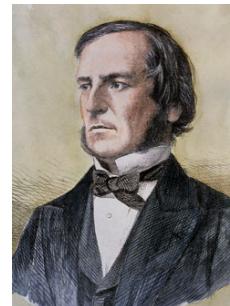
So, wie wir aussagenlogische Formeln definiert haben, ist zwar $(P \rightarrow Q)$ eine, aber $P \rightarrow Q$ nicht. (Wie könnte man das leicht beweisen?) Und bei größeren Formeln verliert man wegen der vielen Klammern leicht den Überblick. Deswegen erlauben wir folgende Abkürzungen bei der Notation aussagenlogischer Formeln. (Ihre „offizielle“ Syntax bleibt die gleiche!)

- Die äußerten umschließenden Klammern darf man immer weglassen. Zum Beispiel ist $P \rightarrow Q$ die Kurzform von $(P \rightarrow Q)$.

- Wenn ohne jede Klammern zwischen mehrere Aussagevariablen immer das gleiche Konnektiv steht, dann bedeute das „implizite Linksklammerung“. Zum Beispiel ist $P \wedge Q \wedge R$ die Kurzform von $((P \wedge Q) \wedge R)$.
- Wenn ohne jede Klammern zwischen mehrere Aussagevariablen verschiedene Konnektive stehen, dann ist von folgenden „Bindungsstärken“ der Konnektive auszugehen:
 - a) \neg bindet am stärksten
 - b) \wedge bindet am zweitstärksten
 - c) \vee bindet am drittstärksten
 - d) \rightarrow bindet am viertstärksten
 - e) \leftrightarrow bindet am schwächsten
 Zum Beispiel ist $P \vee R \rightarrow \neg Q \wedge R$ die Kurzform von $((P \vee R) \rightarrow ((\neg Q) \wedge R))$.

5.3 BOOLESCHES FUNKTIONEN

Der britische Mathematiker George Boole (1815–1864) hat mit seinem heutzutage [online](#) verfügbaren Buch „*An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*“ (Boole 1854) die Grundlagen für die sogenannte *algebraische Logik* oder auch *boolesche Algebra* gelegt. Dabei griff er die weiter vorne erwähnte Grundlage der Aussagenlogik auf, dass der Wahrheitswert einer Aussage nur von den Wahrheitswerten seiner Teilformeln abhängt. Was Boole neu hinzunahm, war die Idee, in Anlehnung an arithmetische Ausdrücke auch „boolesche Ausdrücke“ einschließlich „Variablen“ zu erlauben und solche Ausdrücke umzuformen analog zu den gewohnten Umformungen bei arithmetischen Ausdrücken.



Im weiteren Verlauf schreiben wir für die Wahrheitswerte „wahr“ und „falsch“ kurz **w** und **f** und bezeichnen die Menge mit diesen beiden Werten als $\mathbb{B} = \{\mathbf{w}, \mathbf{f}\}$.

Eine *boolesche Funktion* ist eine Abbildung der Form $f : \mathbb{B}^n \rightarrow \mathbb{B}$. In der nachfolgenden Tabelle sind einige „typische“ boolesche Funktionen aufgeführt, die wir zunächst einmal mit b_{\neg} , b_{\wedge} , b_{\vee} und b_{\rightarrow} bezeichnen:

boolesche Funktion

x_1	x_2	$b_{\neg}(x_1)$	$b_{\wedge}(x_1, x_2)$	$b_{\vee}(x_1, x_2)$	$b_{\rightarrow}(x_1, x_2)$
f	f	w	f	f	w
f	w	w	f	w	w
w	f	f	f	w	f
w	w	f	w	w	w

Üblicher ist es allerdings, die Abbildungen b_{\wedge} und b_{\vee} mit einem Infixoperator zu schreiben und b_{\neg} wird auch anders notiert. Dabei findet man (mindestens) zwei Varianten. In der nachfolgenden Tabelle sind die Möglichkeiten nebeneinander gestellt und auch gleich noch übliche Namen bzw. Sprechweisen mit angegeben:

$b_{\neg}(x)$	$\neg x$	\bar{x}	Negation bzw. Nicht
$b_{\wedge}(x, y)$	$x \wedge y$	$x \cdot y$	Und
$b_{\vee}(x, y)$	$x \vee y$	$x + y$	Oder
$b_{\rightarrow}(x, y)$			Implikation

Die an Arithmetik erinnernde Notationsmöglichkeit geht oft einher mit der Benutzung von 0 statt **f** und 1 statt **w**.

Das meiste, was durch die Definitionen dieser booleschen Abbildungen ausgedrückt wird, ist aus dem alltäglichen Leben vertraut. Nur auf wenige Punkte wollen wir explizit eingehen:

- Das „Oder“ ist „inklusiv“ (und nicht „exklusiv“): Auch wenn x und y beide wahr sind, ist $x \vee y$ wahr.
- Bei der Implikationsabbildung ist $b_{\rightarrow}(x, y)$ auf jeden Fall wahr, wenn $x = \mathbf{f}$ ist, unabhängig vom Wert von y , insbesondere auch dann, wenn $y = \mathbf{f}$ ist.
- Für jedes $x \in \mathbb{B}$ haben beiden Abbildungen b_{\wedge} und b_{\vee} die Eigenschaft, dass $b_{\wedge}(x, x) = b_{\vee}(x, x) = x$ ist, und für b_{\rightarrow} gilt $b_{\rightarrow}(x, x) = \mathbf{w}$.

Natürlich gibt es zum Beispiel insgesamt $2^{2^2} = 16$ zweistellige boolesche Funktionen. Die obigen sind aber ausreichend, um auch jede der anderen durch Hintereinanderausführung der genannten zu realisieren. Zum Beispiel ist $\mathbb{B}^2 \rightarrow \mathbb{B} : (x, y) \mapsto b_{\vee}(b_{\neg}(x), x)$ die Abbildung, die konstant **w** ist.

Für die Abbildung b_{\rightarrow} hat sich übrigens keine Operatorschreibweise durchgesetzt. Man kann sich aber klar machen, dass für jedes $x, y \in \mathbb{B}$ gilt: $b_{\rightarrow}(x, y) = b_{\vee}(b_{\neg}(x), y)$ ist. Eine Möglichkeit besteht darin, in einer Tabelle für alle möglichen Kombinationen von Werten für x und y die booleschen Ausdrücke auszuwerten:

x	y	$b_{\neg}(x)$	$b_{\vee}(b_{\neg}(x), y)$	$b_{\rightarrow}(x, y)$
f	f	w	w	w
f	w	w	w	w
w	f	f	f	f
w	w	f	w	w

5.4 SEMANTIK AUSSAGENLOGISCHER FORMELN

Unser Ziel ist es nun, jeder aussagenlogischen Formel im wesentlichen eine boolesche Funktion als Bedeutung zuzuordnen.

Eine *Interpretation* einer Menge V von Aussagevariablen ist eine Abbildung $I : V \rightarrow \mathbb{B}$. Die Menge aller Interpretationen einer Variablenmenge V ist also \mathbb{B}^V . Das kann man sich z.B. vorstellen als eine Tabelle mit je einer Spalte für jede Variable $X \in V$ und je einer Zeile für jede Interpretation I , wobei der Eintrag in „Zeile I “ und „Spalte X “ gerade $I(X)$ ist. Abbildung 5.1 zeigt beispielhaft den Fall $V = \{P_1, P_2, P_3\}$. Machen Sie sich klar, dass es für jede Variablenmenge mit $k \in \mathbb{N}_+$ Aussagevariablen gerade 2^k Interpretationen gibt.

P_1	P_2	P_3
f	f	f
f	f	w
f	w	f
f	w	w
w	f	f
w	f	w
w	w	f
w	w	w

Abbildung 5.1: Alle Interpretationen der Menge $V = \{P_1, P_2, P_3\}$ von Aussagevariablen.

Jede Interpretation I legt eine Auswertung $val_I(F)$ jeder aussagenlogischen Formel $F \in For_{AL}$ fest: Für jedes $X \in Var_{AL}$ und für jede aussagenlogische Formel G und H gelte:

$$\begin{aligned}
 val_I(X) &= I(X) \\
 val_I(\neg G) &= b_{\neg}(val_I(G)) \\
 val_I(G \wedge H) &= b_{\wedge}(val_I(G), val_I(H)) \\
 val_I(G \vee H) &= b_{\vee}(val_I(G), val_I(H)) \\
 val_I(G \rightarrow H) &= b_{\rightarrow}(val_I(G), val_I(H))
 \end{aligned}$$

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

- Man kann für komplizierte Aussagen anhand der obigen Definition „ausrechnen“, ob sie für eine Interpretation wahr oder falsch ist, weil jede aussagenlogische Formel nur durch Anwendung von genau einer erlaubten Abbildungen entstehen kann. Für die Interpretation I mit $I(P) = w$ und $I(Q) = f$ ergäbe die Anwendung der obigen Definition von val_I auf die Formel $\neg(P \wedge Q)$

z. B. schrittweise:

$$\begin{aligned}
 \text{val}_I(\neg(P \wedge Q)) &= \neg(\text{val}_I(P \wedge Q)) \\
 &= \neg(\text{val}_I(P) \wedge \text{val}_I(Q)) \\
 &= \neg(I(P) \wedge I(Q)) \\
 &= \neg(\mathbf{w} \wedge \mathbf{f}) \\
 &= \neg(\mathbf{f}) \\
 &= \mathbf{w}
 \end{aligned}$$

Oft interessiert man sich dafür, was für *jede* Interpretation der Variablen einer Formel passiert. Dann ergänzt man die Tabelle aller Interpretationen um weitere Spalten mit den Werten von $\text{val}_I(G)$. Hier ist ein Beispiel:

P	Q	$\neg P$	$\neg Q$	$\neg P \vee \neg Q$	$P \wedge Q$	$\neg(P \wedge Q)$
f	f	w	w	w	f	w
f	w	w	f	w	f	w
w	f	f	w	w	f	w
w	w	f	f	f	w	f

äquivalente
Aussagen

- Die Tabelle zeigt, dass die Aussagen $\neg(P \wedge Q)$ und $\neg P \vee \neg Q$ für *alle* Interpretationen denselben Wahrheitswert annehmen. Solche Aussagen nennt man *äquivalent*. Gleiches gilt für $\neg \neg P$ und P und viele weitere Paare von aussagenlogischen Formeln.

Wenn zwei Formeln G und H äquivalent sind, dann schreiben wir auch $G \equiv H$.

- Noch eine Anmerkung zur Implikation. So wie $b \rightarrow$ definiert wurde ist eine Aussage $P \rightarrow Q$ genau dann wahr, wenn P falsch ist oder Q wahr. Warum das sinnvoll ist, wird unter Umständen noch etwas klarer, wenn man überlegt, wie man denn die Tatsache umschreiben wollte, dass eine Implikation im naiven Sinne, also „aus P folgt Q “, *nicht* zutrifft. Das ist doch wohl dann der Fall, wenn zwar P zutrifft, aber Q nicht. Also sollte $P \rightarrow Q$ äquivalent zu $\neg(P \wedge \neg Q)$, und das ist nach obigem äquivalent zu $(\neg P) \vee (\neg \neg Q)$ und das zu $(\neg P) \vee Q$.

Man kann nun noch einen letzten Schritt tun und mit jeder aussagenlogische Formel G eine Abbildung assoziieren, die für jede (passende) Interpretation die ausgewertete Formel als Wert zuweist. Wir gehen in zwei Schritten vor. Als erstes sei V eine Menge von aussagenlogischen Variablen, die alle in G vorkommenden

Variablen enthält. Dann kann man die folgende Abbildung definieren:

$$\mathbb{B}^V \rightarrow \mathbb{B} : I \mapsto \text{val}_I(G) .$$

Das ist ganz genau genommen noch keine boolesche Abbildung, weil der Definitionsbereich nicht ein kartesisches Produkt \mathbb{B}^k ist, sondern eben die Menge aller Abbildungen von V in \mathbb{B} .

Konsequenz ist auch, dass zum Beispiel die Formeln $G = p_0 \wedge p_0$ und $H = p_2 \wedge p_2$ nicht äquivalent sind. Das sieht man an einer Interpretation I mit $I(p_0) = \mathbf{w}$ und $I(p_2) = \mathbf{f}$. Für sie ist $\text{val}_I(G) = \mathbf{w}$, aber $\text{val}_I(H) = \mathbf{f}$. Man kann sich aber durchaus auf den Standpunkt stellen, dass es einen „Kern“ gibt, der von den konkreten Namen der Aussagevariablen unabhängig und bei beiden Formeln gleich ist.

Um diesen „Kern“ herauszuarbeiten, kann man verschiedene Wege gehen. Einer, dem wir nicht folgen werden, besteht darin, zu verlangen, dass in jeder Formel G für eine geeignete nichtnegative ganze Zahl k die Nummern der in G vorkommenden Aussagevariablen gerade die Zahlen in \mathbb{Z}_k sein müssen. Falls das nicht der Fall ist, müsste man also immer Variablen umbenennen.

Wir bevorzugen eine Alternative, bei der man sozusagen auch noch die Namen der Aussagevariablen „vergisst“. Das formalisieren wir so: Da wir vorausgesetzt haben, dass alle Aussagevariablen von der Form p_i sind, können wir den Variablen in V auch eine Reihenfolge geben. Dazu seien einfach für $k = |V| \geq 1$ die Zahlen $i_1, \dots, i_k \in \mathbb{N}_0$ die Indizes der Variablen in V , und zwar seien sie so gewählt, dass für jedes $j \in \mathbb{P}_{k-1}$ gelte: $i_j < i_{j+1}$. Jedem k -Tupel $x = (x_1, \dots, x_k) \in \mathbb{B}^k$ entspricht dann in naheliegender Weise die Interpretation

$$I_x : p_{i_j} \mapsto x_j .$$

Damit ist dann die durch eine aussagenlogische Formel G beschriebene Abbildung diejenige mit $\mathbb{B}^k \rightarrow \mathbb{B} : x \mapsto \text{val}_{I_x}(G)$.

Ist I eine Interpretation für eine aussagenlogische Formel G , dann nennen wir I ein *Modell* von G , wenn $\text{val}_I(G) = \mathbf{w}$ ist. Ist I eine Interpretation für eine Menge Γ aussagenlogischer Formeln, dann nennen wir I ein *Modell* von Γ , wenn I Modell jeder Formel $G \in \Gamma$ ist.

Modell einer Formel

Modell einer Formelmenge

Ist Γ eine Menge aussagenlogischer Formeln und G ebenfalls eine, so schreibt man auch genau dann $\Gamma \models G$, wenn jedes Modell von Γ auch Modell von G ist. Enthält $\Gamma = \{H\}$ nur eine einzige Formel, schreibt man einfach $H \models G$. Ist $\Gamma = \{\}$ die leere Menge, schreibt man einfach $\models G$. Die Bedeutung soll in diesem Fall sein, dass G für *alle* Interpretationen überhaupt wahr ist, d.h. dass G eine Tautologie ist.

erfüllbare
Formel

Es zeigt sich, dass in verschiedenen Teilen der Informatik zwei Sorten aussagenlogischer Formeln von Bedeutung sind. Die eine wichtige Klasse von Formeln sind *erfüllbare Formeln*, d. h. Formeln, die für mindestens eine Interpretation wahr sind. Die Untersuchung aussagenlogischer Formeln auf Erfüllbarkeit spielt in einigen Anwendungen eine große Rolle. Für manche Formeln ist das ganz einfach, für andere anscheinend nicht. Vergleichen Sie einfach mal die Formeln $(P \wedge \neg Q) \vee (\neg P \wedge R)$ und $(P \vee \neg Q) \wedge (\neg P \vee R)$ in dieser Hinsicht. In der Vorlesung „Theoretische Grundlagen der Informatik“ werden Sie mehr zum Thema Erfüllbarkeit erfahren.

Tautologie

allgemeingültige
Formel

Das andere sind *Tautologien*; das sind Formeln, für die *jede* Interpretation ein Modell ist, die also für jede Interpretation wahr sind. Sie heißen auch *allgemeingültige Formeln*. Solche Formeln gibt es, z. B. $P \vee \neg P$ oder $P \rightarrow (Q \rightarrow P)$. Sie werden auch im nächsten Abschnitt noch eine wichtige Rolle spielen.

Zum Abschluss dieses Abschnitts geht es erst einmal darum, eine ganze Reihe von Tautologien kennenzulernen und auch Methoden, sich neue zu beschaffen.

Als erstes betrachten wir eine Formel der Form $G \leftrightarrow H$. Das wurde eingeführt als Abkürzung für $(G \rightarrow H) \wedge (H \rightarrow G)$. Für jede Interpretation I ist

$$\begin{aligned} \text{val}_I((G \rightarrow H) \wedge (H \rightarrow G)) &= \text{val}_I(G \rightarrow H) \wedge \text{val}_I(H \rightarrow G) \\ &= (\neg \text{val}_I(G) \vee \text{val}_I(H)) \wedge (\neg \text{val}_I(H) \vee \text{val}_I(G)) \end{aligned}$$

Falls nun G und H äquivalente Formeln sind, also für jede Interpretation I $\text{val}_I(G) = \text{val}_I(H)$ gilt, ergibt sich weiter

$$\text{val}_I(G \rightarrow H \wedge H \rightarrow G) = (\neg \text{val}_I(G) \vee \text{val}_I(G)) \wedge (\neg \text{val}_I(G) \vee \text{val}_I(G))$$

und gleichgültig, welchen Wert $\text{val}_I(G)$ hat ergibt die Auswertung letzten Endes immer $\text{val}_I(\dots) = \dots = \mathbf{w} \wedge \mathbf{w} = \mathbf{w}$. Also gilt:

5.1 Lemma. Wenn G und H äquivalente aussagenlogische Formeln sind, dann ist die Formel $G \leftrightarrow H$ eine Tautologie.

Zusammen mit Einsichten in vorangegangenen Abschnitten zeigt das sofort, dass für beliebige Formeln G und H z. B. folgende Formeln Tautologien sind:

- $\neg \neg G \leftrightarrow G$
- $(G \rightarrow H) \leftrightarrow (\neg G \vee H)$
- $(G \rightarrow H) \leftrightarrow (\neg H \rightarrow \neg G)$
- $(G \wedge H) \leftrightarrow \neg(\neg G \vee \neg H)$ und $(G \vee H) \leftrightarrow \neg(\neg G \wedge \neg H)$
- $\neg(G \wedge H) \leftrightarrow (\neg G \vee \neg H)$ und $\neg(G \vee H) \leftrightarrow (\neg G \wedge \neg H)$

- $G \wedge G \leftrightarrow G$ und
 $G \vee G \leftrightarrow G$
- $G \wedge H \leftrightarrow H \wedge G$ und
 $G \vee H \leftrightarrow H \vee G$

Es gilt übrigens auch die Umkehrung der obigen Argumentation:

5.2 Lemma. Wenn für zwei aussagenlogische Formeln G und H die Formel $G \leftrightarrow H$ eine Tautologie ist, dann sind G und H äquivalente Formeln.

Außerdem überzeugt man sich schnell, dass auch Formeln des folgenden Aufbaus allgemeingültig sind für beliebige aussagenlogische Formeln G , H und K :

- $G \rightarrow G$
- $\neg G \vee G$
- $G \rightarrow (H \rightarrow G)$
- $(G \rightarrow (H \rightarrow K)) \rightarrow ((G \rightarrow H) \rightarrow (G \rightarrow K))$
- $((\neg H \rightarrow \neg G) \rightarrow ((\neg H \rightarrow G) \rightarrow H))$

5.5 BEWEISBARKEIT

Der grundlegende Begriff im Zusammenhang mit Beweisbarkeit sowohl in der Aussagenlogik als auch in der Prädikatenlogik ist der des Kalküls. Im Fall der Aussagenlogik gehören zum sogenannten *Aussagenkalkül* dazu stets

Aussagenkalkül

- das Alphabet A_{AL} , aus dem die Zeichen für alle Formeln stammen,
- die Menge For_{AL} , der syntaktisch korrekten Formeln über dem Alphabet A_{AL} ,
- eine Menge sogenannter *Axiome* $Ax_{AL} \subseteq For_{AL}$,
- und sogenannte *Schlussregeln* oder *logische Folgerungsregeln*.

Axiome

Schlussregeln

logische Folgerungsregeln

Als Axiome für das Aussagenkalkül wählen wir

$$\begin{aligned} Ax_{AL} = & \{ (G \rightarrow (H \rightarrow G)) \mid G, H \in For_{AL} \} \\ & \cup \{ (G \rightarrow (H \rightarrow K)) \rightarrow ((G \rightarrow H) \rightarrow (G \rightarrow K)) \mid G, H, K \in For_{AL} \} \\ & \cup \{ (\neg H \rightarrow \neg G) \rightarrow ((\neg H \rightarrow G) \rightarrow H) \mid G, H \in For_{AL} \} \end{aligned}$$

Die Formelmengen in den drei Zeilen wollen kurz mit Ax_{AL1} , Ax_{AL2} und Ax_{AL3} bezeichnen. Für alle drei Mengen haben wir schon erwähnt, dass sie nur Tautologien enthalten. (Überlegen Sie sich bitte auch selbst.) Die Axiome sind also alle Tautologien.

Im Fall der Aussagenlogik gibt es nur eine Schlussregel, den sogenannten *Modus Ponens*. Diese Regel kann man formalisieren als Relation $MP \subseteq For_{AL}^3$ mit

Modus Ponens

$$MP = \{(G \rightarrow H, G, H) \mid G, H \in For_{AL}\}$$

Bei Kalkülen schreibt man gelegentlich Ableitungsregeln in einer speziellen Form auf. Beim Modus Ponens sieht das so aus:

$$MP : \frac{G \rightarrow H \quad G}{H}$$

Wie der Modus Ponens anzuwenden ist, ergibt sich bei der Definition dessen, was wir formal unter einer Ableitung bzw. unter einem Beweis verstehen wollen.

Hypothese
Prämisse
Ableitung

Dazu sei Γ eine Formelmenge sogenannter *Hypothesen* oder *Prämissen* und G eine Formel. Eine *Ableitung* von G aus Γ ist eine endliche Folge (G_1, \dots, G_n) von n Formeln mit der Eigenschaft, dass erstens $G_n = G$ ist und auf jede Formel G_i einer der folgenden Fälle zutrifft:

- Sie ist ein Axiom: $G_i \in Ax_{AL}$.
- Oder sie ist eine Prämisse: $G_i \in \Gamma$.
- Oder es gibt Indizes i_1 und i_2 echt kleiner i , für die gilt: $(G_{i_1}, G_{i_2}, G_i) \in MP$.

Wir schreiben dann $\Gamma \vdash G$. Ist $\Gamma = \{\}$, so heißt eine entsprechende Ableitung auch ein *Beweis* von G und G ein *Theorem* des Kalküls, in Zeichen: $\vdash G$.

Beweis
Theorem

Wir wollen als erstes einen Beweis für die Formel $(P \rightarrow P)$ angeben. Dabei ist zum besseren Verständnis in jeder Zeile in Kurzform eine Begründung dafür angegeben, weshalb die jeweilige Formel an dieser Stelle im Beweis aufgeführt werden darf.

1. $((P \rightarrow ((P \rightarrow P) \rightarrow P)) \rightarrow ((P \rightarrow (P \rightarrow P)) \rightarrow (P \rightarrow P)))$ Ax_{AL2}
2. $(P \rightarrow ((P \rightarrow P) \rightarrow P))$ Ax_{AL1}
3. $((P \rightarrow (P \rightarrow P)) \rightarrow (P \rightarrow P))$ $MP(1, 2)$
4. $(P \rightarrow (P \rightarrow P))$ Ax_{AL1}
5. $(P \rightarrow P)$ $MP(3, 4)$

Wenn man in diesem Beweis überall statt P eine andere aussagenlogische Formel G notieren würde, erhielte man offensichtlich einen Beweis für $G \rightarrow G$.

Beweisschema

Man spricht in einem solchen Fall von einem *Beweisschema*. Wir werden uns aber erlauben, im folgenden auch in solchen Fällen lax von Beweisen zu sprechen. Auch der folgende Beweis ist nicht besonders lang:

1. $(\neg P \rightarrow \neg P) \rightarrow ((\neg P \rightarrow P) \rightarrow P)$ Ax_{AL3}
2. $(\neg P \rightarrow \neg P)$ Beispiel von eben
3. $(\neg P \rightarrow P) \rightarrow P$ $MP(1, 2)$

Genaugenommen müsste man allerdings statt der Zeile 2 die fünf Zeilen von eben übernehmen und überall P durch $\neg P$ ersetzen. Aber so wie man Programme strukturiert, indem man größere Teile aus kleineren zusammensetzt, macht man das bei Beweisen auch. Insbesondere dann, wenn man kleinere Teil mehrfach verwenden kann.

Wir hatten schon darauf hingewiesen, dass alle Formeln in Ax_{AL} Tautologien sind. Sehen wir uns nun noch den Modus Ponens genauer an. Es gilt

5.3 Lemma. Modus Ponens „erhält Allgemeingültigkeit“, d.h. wenn sowohl $G \rightarrow H$ als auch G Tautologien sind, dann ist auch H eine Tautologie.

5.4 Beweis. Wenn $G \rightarrow H$ eine Tautologie ist, dann gilt für jede Interpretation I :

$$\mathbf{w} = \text{val}_I(G \rightarrow H) = b \rightarrow (\text{val}_I(G), \text{val}_I(H)) .$$

Da G aber auch Tautologie ist, ist $\text{val}_I(G) = \mathbf{w}$ und folglich

$$b \rightarrow (\text{val}_I(G), \text{val}_I(H)) = b \rightarrow (\mathbf{w}, \text{val}_I(H)) = \text{val}_I(H)$$

Also ist $\mathbf{w} = \text{val}_I(H)$ für jede Interpretation I , also ist H Tautologie. ■

Da alle Axiome Tautologien sind und Modus Ponens Allgemeingültigkeit erhält, ergibt sich

5.5 Korollar. Alle Theoreme des Aussagenkalküls sind Tautologien.

Schwieriger zu beweisen ist, dass die Umkehrung dieser Aussage auch gilt:

5.6 Lemma. Jede Tautologie ist im Aussagenkalkül beweisbar.

Zusammen mit Korollar 5.5 ergibt sich der folgende Satz.

5.7 Theorem Für jede Formel $G \in \text{For}_{AL}$ gilt $\models G$ genau dann, wenn $\vdash G$ gilt.

Wie nützlich der Kalkül ist, wollen wir noch einem Beispiel demonstrieren.

5.8 Theorem Für jedes $G \in \text{For}_{AL}$ und jedes $H \in \text{For}_{AL}$ gilt $G \vdash H$ genau dann, wenn $\vdash (G \rightarrow H)$ gilt.

5.9 Beweis. Man beweist die beiden Implikationen des Theorems getrennt.

Der einfache Fall ist der, dass $\vdash (G \rightarrow H)$ gilt. Um zu zeigen, dass dann auch $G \vdash H$ gilt, geht man von einem Beweis (G_1, \dots, G_n) für $\vdash (G \rightarrow H)$ aus. Es sei $G_n = (G \rightarrow H)$. Diese Ableitungsfolge verlängert man um zwei Formeln:

- $G_{n+1} = G$ Prämisse
- $G_{n+2} = H$ MP aus G_n und G_{n+1}

Damit hat man eine Ableitungsfolge für die Formel H aus der Prämisse G .

Gehen wir nun umgekehrt davon aus, dass $G \vdash H$ gilt. Dann gibt es eine entsprechende Ableitungsfolge (G_1, \dots, G_n) mit $G_n = H$. Wir konstruieren einen Beweis (H_1, \dots, H_m) , der $\vdash (G \rightarrow H)$ belegt, induktiv. Das Ziel ist, für jedes G_i eine Formel H_i zu haben mit $H_i = (G \rightarrow G_i)$. Es gibt drei Fälle zu betrachten:

- Wenn G_i Axiom ist, benutzt man die drei Formeln
 - $(G_i \rightarrow (G \rightarrow G_i))$ denn sie ist Axiom aus Ax_{AL1}
 - G_i denn sie ist Axiom
 - $(G \rightarrow G_i)$ gemäß MP aus den beiden vorangegangenen Formeln
- Wenn $G_i = G$ ist, benutzt man die Formeln für den Beweis der Tautologie
 - $(G_i \rightarrow G_i)$
- Wenn sich $G_i = K_2$ durch MP aus $G_{i_1} = K_1 \rightarrow K_2$ und $G_{i_2} = K_1$ mit $i_1 < i$ und $i_2 < i$ ergeben hat, dann werden in der neuen Ableitungsfolge schon die Formeln
 - $G \rightarrow G_{i_1}$
 - $G \rightarrow G_{i_2}$
 abgeleitet. man benutzt nun das Axiom
 - $(G \rightarrow (K_1 \rightarrow K_2)) \rightarrow ((G \rightarrow K_1) \rightarrow (G \rightarrow K_2))$
 aus Ax_{AL3} , das gerade die Form
 - $(G \rightarrow G_{i_1}) \rightarrow ((G \rightarrow G_{i_2}) \rightarrow (G \rightarrow G_i))$
 hat, wendet zweimal MP an, und erhält $(G \rightarrow G_i)$.

■

Deduktionstheorem

Diesen Satz nennt man auch das *Deduktionstheorem* der Aussagenlogik. Es ist die Rechtfertigung dafür, statt eines Beweises einer Implikation $(G \rightarrow H)$, „einfach“ einen Beweis von H unter der Prämisse G zu vorzulegen. Es sei an dieser Stelle bereits die Warnung ausgesprochen, dass das Deduktionstheorem in der Prädikatenlogik erster Stufe in dieser allgemeinen Form *nicht* mehr gilt.

Wir werden Lemma 5.6 in diesem Kapitel nicht beweisen. Anhand verwandter Systeme (mit anderen Schlussregeln) kann man sehen, dass es sogar möglich ist, algorithmisch (d. h. also mit Rechner) für jede Formel

- herauszufinden, ob sie Tautologie ist oder nicht, und
- gegebenenfalls dann sogar auch gleich noch einen Beweis im Kalkül.

Man beachte, dass das nur in der Aussagenlogik so schön klappt. In der Prädikatenlogik, die wir in einem späteren Kapitel auch noch ansehen werden, ist das nicht mehr der Fall.

ZUSAMMENFASSUNG UND AUSBLICK

In diesem Kapitel wurde zunächst die Syntax aussagenlogischer Formeln festgelegt. Anschließend haben wir ihre Semantik mit Hilfe boolescher Funktionen definiert. Und am Ende haben wir gesehen, wie man dem semantischen Begriff der Allgemeingültigkeit den syntaktischen Begriff der Beweisbarkeit so gegenüberstellen kann, dass sich die beiden entsprechen.

LITERATUR

- Boole, George (1854). *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. URL: <http://www.gutenberg.org/ebooks/15114> (besucht am 20. 10. 2015) (siehe S. 37).
- Schmitt, P. H. (2013). *Formale Systeme*. Vorlesungsskript. Institut für Theoretische Informatik, KIT. URL: <http://formal.iti.kit.edu/teaching/FormSysWS1415/skriptum.pdf> (besucht am 19. 10. 2015) (siehe S. 34).

6 INDUKTIVES VORGEHEN

6.1 VOLLSTÄNDIGE INDUKTION

Wir gehen in dieser Vorlesung davon aus, dass die *natürlichen Zahlen* etwas sind, worunter sich alle (jedenfalls weitgehend) das Gleiche vorstellen. Wir meinen hier immer die Zahlen einschließlich der 0 und schreiben

$$\mathbb{N}_0 = \{0, 1, 2, 3, 4, \dots\}$$

Das ist ausreichend, solange man nicht Beweise führen möchte, bei denen man auf besondere Eigenschaften der natürlichen Zahlen zurückgreifen möchte.

Genau das werden wir nun aber tun. Hätten wir eine Definition der natürlichen Zahlen angegeben, dann wäre das folgende *Bestandteil* der Definition. Mangels einer ordentlichen Definition müssen wir uns auf eine Mitteilung beschränken. Es gilt nämlich:

Wenn eine Menge M nur Zahlen aus \mathbb{N}_0 enthält,

- $0 \in M$ ist und
- für jedes $n \in \mathbb{N}_0$ aus $n \in M$ folgt, dass auch $n + 1 \in M$ ist,

dann ist $M = \mathbb{N}_0$, enthält also tatsächlich *alle* natürlichen Zahlen.

Das ist die Grundlage für das Beweisprinzip der *vollständigen Induktion*. Dabei hat man im einfachsten Fall ein „Aussageschema“ \mathcal{A} , in dem eine Variable vorkommt, für die man jede natürliche Zahl einsetzen kann. Für jeden konkreten Wert $n \in \mathbb{N}_0$ ergibt sich eine Aussage \mathcal{A}_n , die wahr oder falsch ist.

vollständige Induktion

Vollständige Induktion benutzt man, um zu beweisen, dass *jede* Aussage \mathcal{A}_n wahr ist, gleich, welchen Wert n hat. Bezeichnet

$$M = \{n \in \mathbb{N}_0 \mid \mathcal{A}_n \text{ ist wahr} \},$$

dann hat man also zu zeigen, dass $M = \mathbb{N}_0$ ist. Und das macht man gerade, indem man die oben genannte Eigenschaft der natürlichen Zahlen anwendet. Man hat also zu zeigen:

- $0 \in M$ ist und
- für jedes $n \in \mathbb{N}_0$ folgt aus $n \in M$, dass auch $n + 1 \in M$ ist.

Das bedeutet mit anderen Worten, dass man zeigen muss:

- \mathcal{A}_0 ist wahr und
- für jedes $n \in \mathbb{N}_0$ gilt: wenn \mathcal{A}_n wahr ist, dann ist auch \mathcal{A}_{n+1} wahr.

Induktionsanfang
Induktionsschritt

In einem entsprechenden Beweis nennt man den ersten Punkt *Induktionsanfang*, den zweiten den *Induktionsschritt*.

Man könnte einwenden, dass man durch diese Vorgehensweise nichts gewonnen hat, denn im Induktionsschritt muss man ja erneut beweisen, dass etwas für jedes $n \in \mathbb{N}_0$ gilt. Die gute Nachricht ist aber, dass in vielen Fällen der Nachweis, dass $\mathcal{A}_n \rightarrow \mathcal{A}_{n+1}$ für jedes $n \in \mathbb{N}_0$ gilt, deutlich einfacher ist als der Nachweis, dass \mathcal{A}_n für jedes $n \in \mathbb{N}_0$ gilt.

Für ein Beispiel, an dem man das alles sehr schön sieht, kommen wir zurück auf Konkatenation und Potenzen von Wörtern. Es sei im Folgenden A ein beliebiges Alphabet. Wir erinnern daran, dass für jedes w_1 und jedes $w_2 \in A^*$ gilt: $|w_1 w_2| = |w_1| + |w_2|$ (siehe Lemma 4.2). Außerdem hatten wir die Potenzen eines Wortes $w \in A^*$ so definiert:

$$w^0 = \varepsilon \\ \text{für jedes } k \in \mathbb{N}_0 : w^{k+1} = w^k \cdot w$$

Wir wollen nun beweisen:

6.1 Lemma. Für jedes Alphabet A und jedes Wort $w \in A^*$ gilt: Für jedes $n \in \mathbb{N}_0$ ist $|w^n| = n|w|$.

6.2 Beweis. Dass die Aussage für jedes Alphabet A und jedes Wort $w \in A^*$ gilt, beweisen wir, indem wir annehmen, dass ein beliebiges Alphabet A und ein beliebiges Wort $w \in A^*$ gegeben seien, über die keine weiteren einschränkenden Annahmen getroffen werden.

Damit müssen wir nun also zeigen: „Für jedes $n \in \mathbb{N}_0$ ist $|w^n| = n|w|$.“ Die Aussagen \mathcal{A}_n , die im obigen Schema der vollständigen Induktion benutzt wurden, sind also gerade diese:

$$\mathcal{A}_n \text{ ist } |w^n| = n|w|$$

Induktionsanfang: Im Fall $n = 0$ ist zu zeigen, dass \mathcal{A}_0 wahr ist, dass also gilt: $|w^0| = 0|w|$. Das ist einfach:

$$|w^0| = |\varepsilon| = 0 = 0 \cdot |w|.$$

Im *Induktionsschritt* muss gezeigt werden, dass für jedes $n \in \mathbb{N}_0$ gilt: $\mathcal{A}_n \rightarrow \mathcal{A}_{n+1}$. Das tun wir, indem wir „irgendein beliebiges $n \in \mathbb{N}_0$ “ hernehmen (siehe auch die Bemerkung zu Beginn von Beweis 4.4) und zeigen, dass für dieses n gilt: $\mathcal{A}_n \rightarrow \mathcal{A}_{n+1}$. Wann ist die Implikation wahr? In Abschnitt 5.3 haben wir gesehen, dass das zum einen der Fall ist, wenn \mathcal{A}_n falsch ist; dann müssen wir nichts tun. Wenn \mathcal{A}_n wahr ist, dann muss auch \mathcal{A}_{n+1} wahr sein.

Benutzen wir also die *Induktionsvoraussetzung*: $|w^n| = n|w|$. Gezeigt werden muss, dass dann auch gilt: $|w^{n+1}| = (n+1)|w|$. Dafür kann man natürlich die Definition von w^n nutzen:

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

Damit sind wir fertig. ■

6.2 VARIANTEN VOLLSTÄNDIGER INDUKTION

Manchmal ist es erzwungen oder hilfreich, Varianten des eben vorgestellten Schemas zu benutzen. Die drei folgenden Fälle werden auch in dieser Vorlesung noch vorkommen:

1. *Induktionsanfang an „anderer“ Stelle*, wenn zum Beispiel nur für jedes $n \in \mathbb{N}_+$ Aussagen \mathcal{A}_n definiert sind.
2. *im Induktionsschritt Benutzung nicht nur von \mathcal{A}_n* , sondern von „mehreren früheren“ Aussagen \mathcal{A}_i mit $i \leq n$.
3. *„mehrere Induktionsanfänge“*, wenn man z. B. explizit zeigt, dass \mathcal{A}_0 , \mathcal{A}_1 und \mathcal{A}_2 wahr sind.

Der erste Fall ist ganz einfach zu behandeln. Es sei $k \in \mathbb{N}_+$ und stellen Sie sich vor, dass man nur für jedes $n \in \mathbb{N}_0$ mit $n \geq k$ eine Aussage \mathcal{A}_n definiert hat oder dass man nur für jedes $n \geq k$ beweisen will oder kann, dass \mathcal{A}_n wahr ist. Dann kann man im Induktionsanfang beweisen, dass \mathcal{A}_k wahr ist, und im Induktionsschritt, dass für jedes $n \in \mathbb{N}_0$ mit $n \geq k$ die Implikation $\mathcal{A}_n \rightarrow \mathcal{A}_{n+1}$ gilt. Zur Begründung überlege man einfach, dass die Menge $M = \{i \in \mathbb{N}_0 \mid \mathcal{A}_{i+k} \text{ ist wahr}\}$ genau dann gleich \mathbb{N}_0 ist, wenn die Aussagen \mathcal{A}_n für jedes $n \geq k$ wahr sind.

Von der zweiten oben genannten Variante wollen wir den allgemeinsten Fall betrachten. Dazu sei für jedes $n \in \mathbb{N}_0$ eine Aussage \mathcal{B}_n definiert. Wir behaupten, dass man für den Nachweis, dass jede Aussage \mathcal{B}_n wahr ist, so vorgehen darf, dass im Induktionsschritt für den Beweis von \mathcal{B}_{n+1} als „Induktionsvoraussetzungen“ immer nicht nur \mathcal{B}_n benutzt wird, sondern *jede* der Aussagen \mathcal{B}_i mit $i \leq n$.

Um nachzuweisen, dass man dabei keinen Fehler begeht, führen wir diese Vorgehensweise auf die strikte zurück, die wir im ersten Abschnitt kennengelernt haben. Dazu wird für $n \in \mathbb{N}_0$ eine neue Aussage \mathcal{A}_n definiert als

„Für jedes $i \in \mathbb{N}_0$ mit $i \leq n$ ist \mathcal{B}_i wahr.“

Insbesondere ist also für ein $n \in \mathbb{N}_0$ eine Aussage \mathcal{B}_n wahr, sobald die Aussage \mathcal{A}_n wahr ist. Um zu beweisen, dass jede Aussage \mathcal{B}_n wahr ist, ist es also hinreichend, zu beweisen, dass jede Aussage \mathcal{A}_n wahr ist.

Was bedeutet ein Beweis der Aussagen \mathcal{A}_n durch vollständige Induktion?

- Im Induktionsanfang muss man \mathcal{A}_0 beweisen. Das ist aber äquivalent zur Aussage \mathcal{B}_0 .
- Im Induktionsschritt muss man für $n \in \mathbb{N}_0$ beweisen: $\mathcal{A}_n \rightarrow \mathcal{A}_{n+1}$.

\mathcal{A}_{n+1} ist die Aussage „Für jedes $i \in \mathbb{N}_0$ mit $i \leq n+1$ ist \mathcal{B}_i wahr.“ \mathcal{A}_{n+1} kann man aber auch so ausdrücken: „Für jedes $i \in \mathbb{N}_0$ mit $i \leq n$ gilt \mathcal{B}_i und es gilt \mathcal{B}_{n+1} .“

Also ist \mathcal{A}_{n+1} äquivalent zu der Aussage „ \mathcal{A}_n und \mathcal{B}_{n+1} “. Damit das gilt, müssen zwei Dinge bewiesen werden:

- \mathcal{A}_n : das ist aber banal, denn das ist ja gerade die Induktionsvoraussetzung
- \mathcal{B}_{n+1} : Dafür muss im Allgemeinen etwas geleistet werden, aber man darf auch hier die Induktionsvoraussetzung benutzen, die besagt: „Für jedes $i \in \mathbb{N}_0$ mit $i \leq n$ gilt \mathcal{B}_i .“ Man darf also für den Beweis von \mathcal{B}_{n+1} nicht nur auf \mathcal{B}_n , sondern auch auf alle „früheren“ Aussagen $\mathcal{B}_{n-1}, \dots, \mathcal{B}_0$ zurückgreifen.

Der dritte zu Beginn dieses Abschnittes genannte Fall, dass man unter Umständen „mehrere Induktionsanfänge“ betrachtet, tritt häufig im Zusammenhang mit entsprechenden induktiven Definitionen auf. Ein typisches Beispiel sind die sogenannten *Fibonacci-Zahlen*. Diese Folge von Zahlen f_i ist nach dem italienischen Mathematiker Leonardo Fibonacci benannt, der ungefähr von 1170 bis nach 1240 gelebt hat, und wie folgt festgelegt¹:

Fibonacci-Zahlen

$$f_0 = 0$$

$$f_1 = 1$$

$$\text{für jedes } n \in \mathbb{N}_+ \quad f_{n+1} = f_n + f_{n-1}$$

Wie man sieht, legt die dritte Zeile nur Werte f_i für $i \geq 2$ fest, und es werden zwei Werte mit kleineren Indizes benötigt. Und deswegen muss man auch die Fibonacci-

¹Machmal fängt man mit den Werten 1 und 1 an. Das ändert von der dann am Anfang fehlenden 0 abgesehen nichts an der sich ergebenden Folge der Zahlen.

Zahlen mit den *zwei* kleinsten Indizes explizit festlegen. Man kann beweisen, dass für jedes $n \in \mathbb{N}_0$ gilt:

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

Wenn man das mit vollständiger Induktion macht, dann benötigt man

- „zwei Induktionsanfänge“ für f_0 und f_1 und
- im Induktionsschritt natürlich den Rückgriff auf „zwei Induktionsvoraussetzungen“ für f_n und f_{n-1} .

6.3 INDUKTIVE DEFINITIONEN

In den ersten Kapiteln dieser Vorlesung haben wir schon an der ein oder anderen Stelle induktive Definitionen genutzt. Im Folgenden soll es um die Frage gehen, warum so etwas überhaupt sinnvoll ist.

Das ist nicht immer so klar wie z.B. zu Beginn von Abschnitt 4.4.4 bei der Definition der iterierten Konkatination.

Die sogenannte *Ackermann-Funktion* $A : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$ ist in vielerlei Hinsicht etwas ungewöhnlich. Von den leicht unterschiedlichen Versionen geben wir hier die Definition von Rósa Péter an:

Ackermann-Funktion

$$\begin{array}{ll} \text{für jedes } y \in \mathbb{N}_0: & A(0, y) = y + 1 \\ \text{für jedes } x \in \mathbb{N}_0: & A(x + 1, 0) = A(x, 1) \\ \text{für jedes } x \in \mathbb{N}_0, y \in \mathbb{N}_0: & A(x + 1, y + 1) = A(x, A(x + 1, y)) \end{array}$$

Als erstes sei eine Warnung ausgesprochen: Erwarten Sie nicht, dass die Funktionswerte (von Ausnahmen abgesehen) in einem „übersichtlichen Bereich“ liegen, und etwa von Hand oder einem Computer in überschaubarer Zeit bestimmt werden können. Schon die Berechnung von $A(3, 3)$ dauert überraschend lange und für $A(4, 4)$ reicht sozusagen weder die Zeit seit dem Urknall zur Berechnung noch das Weltall zum Aufschreiben des Funktionswertes.

Was uns hier interessiert ist die Frage, ob die obigen Formeln überhaupt eine vernünftige Definition sind. Eine Tabelle mit allen Funktionswerten könnte man z. B. so aufbauen:

	$y = 0$	$y = 1$	$y = 2$	$y = 3$	$y = 4$	
$x = 0$	$A(0,0)$	$A(0,1)$	$A(0,2)$	$A(0,3)$	$A(0,4)$	\dots
$x = 1$	$A(1,0)$	$A(1,1)$	$A(1,2)$	$A(1,3)$	$A(1,4)$	\dots
$x = 2$	$A(2,0)$	$A(2,1)$	$A(2,2)$	$A(2,3)$	$A(2,4)$	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	

Man sieht schnell, dass verschiedene Zeilen der Definition auch disjunkte Teile der Tabelle festlegen: die erste Zeile, die erste Spalte ohne den ersten Wert, und den ganzen Rest. Es bleibt also die Frage, ob überhaupt jedes $A(x,y)$ wohldefiniert ist. Überlegen Sie kurz ... Eine Möglichkeit ist, zeilenweise vorzugehen, also zu beweisen:

„Für jedes $x \in \mathbb{N}_0$ ist in Zeile x alles definiert.“

Das kann man tatsächlich per vollständiger Induktion beweisen.

- Beim Induktionsanfang für $x = 0$ muss man dann zeigen:

„Für jedes $y \in \mathbb{N}_0$ ist $A(0,y)$ definiert.“

Das ist klar.

- Für den Induktionsschritt von x nach $x + 1$ ist die Induktionsvoraussetzung

„Für jedes $y \in \mathbb{N}_0$ ist $A(x,y)$ definiert.“

und man muss beweisen

„Für jedes $y \in \mathbb{N}_0$ ist $A(x + 1, y)$ definiert.“

Dafür kann man noch einmal vollständige Induktion bemühen.

ZUSAMMENFASSUNG UND AUSBLICK

Beweise durch vollständige Induktion werden immer wieder an vielen Stellen auftauchen. Es ist wichtig, solche Beweise zu verstehen und führen zu können. Das hängt auch damit zusammen, dass induktive Definitionen in der Informatik von großer Bedeutung sind, weil sie eine Möglichkeit bieten, mittels einer endlichen Beschreibung präzise eine unendliche Menge von Objekten festzulegen.

7 FORMALE SPRACHEN

Den Begriff der formalen Sprache haben wir schon in Kapitel 4 eingeführt. Eine formale Sprache über einem Alphabet A ist eine Teilmenge $L \subseteq A^*$. In diesem kurzen Kapitel definieren wir noch einige nützliche Operationen auf formalen Sprachen.

7.1 OPERATIONEN AUF FORMALEN SPRACHEN

7.1.1 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

$$L_1 \cdot L_2 = \{w_1w_2 \mid w_1 \in L_1 \text{ und } w_2 \in L_2\}$$

das *Produkt der Sprachen* L_1 und L_2 .

Produkt von Sprachen

7.1 Lemma. Für jede formale Sprache L ist

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

7.2 Beweis. Einfaches Nachrechnen:

$$\begin{aligned} L \cdot \{\varepsilon\} &= \{w_1w_2 \mid w_1 \in L \text{ und } w_2 \in \{\varepsilon\}\} \\ &= \{w_1w_2 \mid w_1 \in L \text{ und } w_2 = \varepsilon\} \\ &= \{w_1\varepsilon \mid w_1 \in L\} \\ &= \{w_1 \mid w_1 \in L\} \\ &= L \end{aligned}$$

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

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

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

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

- Dabei sei

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

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

Potenzen L^n

Wie bei Wörtern will man *Potenzen* L^n definieren. Der „Trick“ besteht darin, für den Fall $k = 0$ etwas Sinnvolles zu finden — Lemma 7.1 gibt einen Hinweis. Die Definition geht wieder induktiv:

$$L^0 = \{\varepsilon\}$$

$$\text{für jedes } n \in \mathbb{N}_0 : L^{n+1} = L \cdot L^n$$

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

$$L^1 = L$$

$$L^2 = L \cdot L$$

$$L^3 = L \cdot L \cdot L$$

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

$$L^0 = \{\varepsilon\}$$

$$L^1 = \{\text{aa}, \text{b}\}$$

$$L^2 = \{\text{aa}, \text{b}\} \cdot \{\text{aa}, \text{b}\} = \{\text{aa} \cdot \text{aa}, \text{aa} \cdot \text{b}, \text{b} \cdot \text{aa}, \text{b} \cdot \text{b}\}$$

$$= \{\text{aaaa}, \text{aab}, \text{baa}, \text{bb}\}$$

$$L^3 = \{\text{aa} \cdot \text{aa} \cdot \text{aa}, \text{aa} \cdot \text{aa} \cdot \text{b}, \text{aa} \cdot \text{b} \cdot \text{aa}, \text{aa} \cdot \text{b} \cdot \text{b},$$

$$\text{b} \cdot \text{aa} \cdot \text{aa}, \text{b} \cdot \text{aa} \cdot \text{b}, \text{b} \cdot \text{b} \cdot \text{aa}, \text{b} \cdot \text{b} \cdot \text{b}\}$$

$$= \{\text{aaaaaa}, \text{aaaab}, \text{aabaa}, \text{aabb}, \text{baaaa}, \text{baab}, \text{bbaa}, \text{bbb}\}$$

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

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

es ist also (angedeutet)

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

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)

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

Mit anderen Worten ist

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

Man beachte, dass bei die Exponenten n_1 „vorne“ und n_2 „hinten“ verschieden sein dürfen (aber nicht müssen).

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.

7.1.2 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:

$$A^* = \bigcup_{i \in \mathbb{N}_0} A^i.$$

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

$$L^+ = \bigcup_{i \in \mathbb{N}_+} L^i \quad \text{und} \quad L^* = \bigcup_{i \in \mathbb{N}_0} L^i$$

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

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

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

Analog ist

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

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:

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

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:

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

7.2 ZUSAMMENFASSUNG

In diesem Kapitel wurden *Produkt* und der *Konkatenationsabschluss formaler Sprachen* eingeführt.

Wir haben gesehen, dass man damit jedenfalls manche formalen Sprachen kurz und verständlich beschreiben kann. Dass diese Notationsmöglichkeiten auch in der Praxis Verwendung finden, wird später noch deutlich werden. Manchmal reicht das, was wir bisher an Notationsmöglichkeiten haben, aber noch nicht. Deshalb werden wir in späteren Kapiteln auch mächtigere Hilfsmittel kennenlernen.

8 ÜBERSETZUNGEN UND CODIERUNGEN

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

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

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

8.1 VON WÖRTERN ZU ZAHLEN UND ZURÜCK

8.1.1 Dezimaldarstellung von Zahlen

Wir sind gewohnt, natürliche Zahlen im sogenannten Dezimalsystem notieren, das aus Indien kommt (siehe auch Kapitel 14 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 folgende Tabelle gegeben:

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

Man beachte, dass in der ersten Zeile der Tabelle *Zeichen* stehen, und in der zweiten Zeile *Zahlen*. Genauer gesagt stehen in der ersten Zeile Zeichen, die für sich stehen, und in der zweiten Zeile Zeichen, die Sie als Zahlen interpretieren sollen.

Also ist num_{10} eine Abbildung $\text{num}_{10}: Z_{10} \rightarrow \mathbb{Z}_{10}$.

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

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

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

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

ist, definiert man $\text{Num}_{10}: Z_{10}^* \rightarrow \mathbb{N}_0$ so:

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

8.1.2 Andere unbeschränkte 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.



Leibniz hatte 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 beschrieben

Gottfried Wilhelm Leibniz

(siehe auch http://www.hs-augsburg.de/~harsch/germanica/Chronologie/17Jh/Leibniz/lei_bina.html, 2.11.2018) und im Jahre 1703 in einer Zeitschrift veröffentlicht. Abbildung 8.1 zeigt Beispielrechnungen mit Zahlen in Binärdarstellung aus dieser Veröffentlichung.

Binärdarstellung

Bei der *Binärdarstellung* von nichtnegativen ganzen Zahlen geht man analog zur Dezimaldarstellung vor. Als Ziffernmenge benutzt man $Z_2 = \{0, 1\}$ und definiert

$$\begin{aligned}\text{num}_2(0) &= 0 \\ \text{num}_2(1) &= 1 \\ \text{Num}_2(\varepsilon) &= 0 \\ \text{sowie} \quad \forall w \in Z_2^* \quad \forall x \in Z_2 : \text{Num}_2(wx) &= 2 \cdot \text{Num}_2(w) + \text{num}_2(x)\end{aligned}$$

Damit ist dann z. B.

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

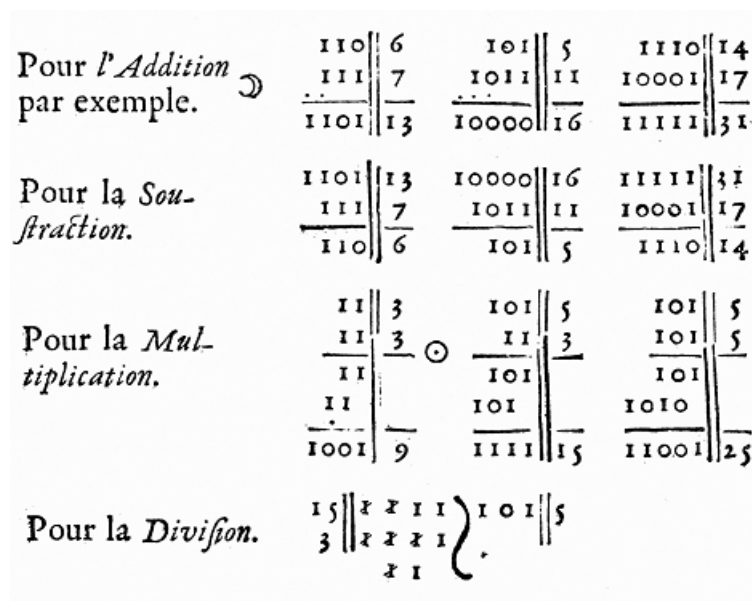


Abbildung 8.1: Ausschnitt aus dem Aufsatz „Explication de l’Arithmétique Binaire“ von Leibniz, Quelle: http://commons.wikimedia.org/wiki/Image:Leibniz_binary_system_1703.png (2.11.2018)

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

Die Zuordnung von Wörtern zu Zahlen ist im Wesentlichen gegeben durch

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

Ein Problem ergibt sich dadurch, dass die Alphabete Z_2 , Z_3 , usw. nicht disjunkt sind. Daher ist z. B. die Zeichenfolge **111** mehrdeutig: Um sagen zu können, welche Zahl hier repräsentiert wird, muss man wissen, zu welcher Basis die Zahl dargestellt ist. Zum Beispiel ist

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

8.1.3 Ganzzahlige Division mit Rest

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.

Um das einzusehen betrachten wir als Vorbereitung zwei binäre Operationen, die Ihnen im Laufe dieser und anderer Vorlesungen noch öfter über den Weg laufen werden. Sie heißen **div** und **mod** und sind wie folgt definiert: Die Operation **mod** liefert für zwei Argumente $x \in \mathbb{N}_0$ und $y \in \mathbb{N}_+$ als Funktionswert $x \bmod y$ den Rest der ganzzahligen Division von x durch y . Man beachte, dass y nicht 0 sein darf. Und die Operation **div** liefere für $x \in \mathbb{N}_0$ und $y \in \mathbb{N}_+$ als Funktionswert $x \div y$ den Wert der ganzzahligen Division von x durch y . Mit anderen Worten gilt stets:

$$x = y \cdot (x \div y) + (x \bmod y) \quad \text{und} \quad 0 \leq (x \bmod y) < y \quad (8.1)$$

Etwas anders ausgedrückt gilt das folgende: Wenn eine Zahl $x \in \mathbb{N}_0$ für ein $k \in \mathbb{N}_+$ in der Form $x = k \cdot p_x + r_x$ dargestellt ist, wobei $p_x \in \mathbb{N}_0$ und $0 \leq r_x < k$, also $r_x \in \mathbb{Z}_k$, ist, dann ist gerade $p_x = x \div k$ und $r_x = x \bmod k$.

Hieraus ergibt sich schnell folgendes Lemma.

8.1 Lemma. Für jede $x, y \in \mathbb{N}_0$ und jedes $k \in \mathbb{N}_+$ gilt:

$$(x + y) \bmod k = ((x \bmod k) + (y \bmod k)) \bmod k$$

8.2 Beweis. Es sei $x = k \cdot p_x + r_x$ und $y = k \cdot p_y + r_y$ mit $r_x, r_y \in \mathbb{Z}_k$. Dann ist

$$\begin{aligned}
 (x + y) \bmod k &= (kp_x + r_x + kp_y + r_y) \bmod k \\
 &= (r_x + r_y) \bmod k \\
 &= ((x \bmod k) + (y \bmod k)) \bmod k
 \end{aligned}$$

■

8.1.4 Von Zahlen zu ihren Darstellungen

Es sei nun $k \geq 2$.

Mit Hilfe der beiden eben eingeführten Funktionen kann man dann eine sogenannte *k-äre Darstellung* oder auch *k-äre Repräsentation* einer Zahl festlegen. Dazu benutzt man ein Alphabet Z_k mit k Ziffern, deren Bedeutungen die Zahlen in \mathbb{Z}_k sind. Für $i \in \mathbb{Z}_k$ sei $\text{repr}_k(i)$ dieses Zeichen. Die Abbildung repr_k ist also gerade die Umkehrfunktion (siehe Abschnitt 8.2) zu num_k .

k-äre Darstellung
k-äre Repräsentation

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

Wie wir gleich sehen werden, gibt es immer solche Wörter. Und es sind dann auch immer gleich unendlich viele, denn wenn $\text{Num}_k(w) = n$ ist, dann auch $\text{Num}_k(0w) = n$ (einfach nachrechnen).

Die Funktion Repr_k definieren wir so:

$$\text{Repr}_k(n) = \begin{cases} \text{repr}_k(n) & \text{falls } n < k \\ \text{Repr}_k(n \text{ div } k) \cdot \text{repr}_k(n \text{ mod } k) & \text{falls } n \geq k \end{cases} \quad (8.2)$$

Dabei bedeutet der Punkt \cdot in der zweiten Zeile Konkatenation. Hier liegt wieder einmal eine induktive Definition vor, aber es ist nicht so klar wie bisher, dass tatsächlich für jedes $n \in \mathbb{N}_0$ ein Funktionswert $\text{Repr}_k(n)$ definiert wird. Daher wollen wir das beweisen. Genauer betrachten wir für $m \in \mathbb{N}_+$ die Aussagen \mathcal{A}_m der Form:

„Für alle $n \in \mathbb{N}_0$ mit $n < k^m$ ist $\text{Repr}_k(n)$ definiert.“

oder anders aber äquivalent formuliert:

„Für alle $n \in \mathbb{N}_0$ gilt: Wenn $n < k^m$ ist, dann ist $\text{Repr}_k(n)$ definiert.“

Im *Induktionsanfang* müssen wir Aussage \mathcal{A}_1 beweisen, die besagt, dass $\text{Repr}_k(n)$ für alle $n \in \mathbb{N}_0$ mit $n < k$ definiert ist. Das ist aber klar, denn dann ist laut (8.2) gerade $\text{Repr}_k(n) = \text{repr}_k(n)$.

Für den Induktionsschritt sei nun $m \in \mathbb{N}_+$. Als *Induktionsvoraussetzung* können wir benutzen:

„Für alle $n \in \mathbb{N}_0$ mit $n < k^m$ ist $\text{Repr}_k(n)$ definiert.“

Zu zeigen ist, dass dann auch gilt:

„Für alle $n \in \mathbb{N}_0$ mit $n < k^{m+1}$ ist $\text{Repr}_k(n)$ definiert.“

Sei also $n \in \mathbb{N}_0$ und $n < k^{m+1}$. Dann sind zwei Fälle möglich:

- Es ist sogar $n < k^m$. Dann ist $\text{Repr}_k(n)$ gemäß der Induktionsvoraussetzung definiert.
- Es ist $k^m \leq n < k^{m+1}$. Dann ist $n \text{ div } k < k^m$ und somit nach Induktionsvoraussetzung $\text{Repr}_k(n \text{ div } k)$ definiert. Folglich ist aber wieder nach (8.2) auch $\text{Repr}_k(n) = \text{Repr}_k(n \text{ div } k) \cdot \text{repr}_k(n \text{ mod } k)$ definiert.

Es sei noch angemerkt, dass mit Ausnahme der 0 die Funktion Repr_k zu gegebenem $n \in \mathbb{N}_0$ das (es ist eindeutig) kürzeste Wort $w \in Z_k^*$ mit $\text{Num}_k(w) = n$ liefert. 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.

8.1.5 Beschränkte Zahlbereiche und Zahldarstellungen

In Kapitel 10 werden Sie ein erstes Modell für Prozessoren kennenlernen, die (mikroprogrammierte) *Minimalmaschine* MIMA. Sie ist nicht wirklich minimal, aber an einigen Stellen wird sie im Vergleich zu realen Prozessoren wie Sie in Desktop-Rechnern, Mobiltelefonen und anderen Geräten verbaut sind, stark vereinfacht sein. Wichtige Aspekte werden aber auch schon an der MIMA sichtbar werden.

Dazu gehört, dass man nicht beliebig große Zahlen repräsentieren kann. Einzelne Zahlen (und auch anders zu interpretierende Objekte) werden in sogenannten *Registern* gespeichert werden, und zwar in der MIMA immer als Bitwörter der Länge 24. In einem Register lassen sich also maximal 2^{24} verschiedene Zahlen speichern. Die Frage ist: welche und in welcher Repräsentation?

Eine einfache (aber nicht genutzte) Vorgehensweise wäre diese:

- Für $\ell \in \mathbb{N}_+$ sei die Funktion $\text{bin}_\ell: \mathbb{Z}_{2^\ell} \rightarrow \{0, 1\}^\ell$ definiert vermöge der Festlegung

$$\text{bin}_\ell(n) = 0^{\ell - |\text{Repr}_2(n)|} \text{Repr}_2(n)$$

Es ist also stets $|\text{bin}_\ell(n)| = \ell$ und $\text{Num}_2(\text{bin}_\ell(n)) = n$.

- Dann könnte man in der MIMA jede Zahl $n \in \mathbb{Z}_{2^{24}}$ in der Form $\text{bin}_{24}(n)$ darstellen.

Aber „natürlich“ möchte man auch negative Zahlen repräsentieren können. Und es wäre praktisch, wenn man für deren Verarbeitung keine extra Recheneinheiten benötigen würde, sondern z. B. ein sogenanntes Addierwerk immer das Gewünschte leistet, gleich, ob die Argumente für negative oder für nichtnegative Zahlen stehen.

Es zeigt sich, dass man das tatsächlich erreichen kann. Im Folgenden stellen wir eine solche Repräsentation vor, die sogenannte *Zweierkomplement-Darstellung*.

*Zweierkomplement-
Darstellung*

Auch dafür muss man als erstes eine *feste Länge* der Zahldarstellungen festlegen. Wir bezeichnen Sie mit ℓ . Es ist $\ell \in \mathbb{N}_+$ und sinnvollerweise $\ell \geq 2$. Die Menge der Zahlen, die man in Zweierkomplementdarstellung der Länge ℓ repräsentieren kann, nennen wir \mathbb{K}_ℓ (das \mathbb{K} soll an „Komplement“ erinnern), und sie ist

$$\mathbb{K}_\ell = \{x \in \mathbb{Z} \mid -2^{\ell-1} \leq x \leq 2^{\ell-1} - 1\}.$$

(Dabei schreiben wir „ $a \leq b \leq c$ “ als Abkürzung für „ $a \leq b$ und $b \leq c$ “.) Es sind also z. B.

$$\begin{aligned} \mathbb{K}_2 &= \{-2, -1, 0, 1\} \\ \text{und} \quad \mathbb{K}_8 &= \{-128, -127, \dots, -1, 0, 1, \dots, 127\} \end{aligned}$$

Die Zweierkomplementdarstellung $\text{Zkpl}_\ell: \mathbb{K}_\ell \rightarrow \{0, 1\}^\ell$ der Länge ℓ ist für alle $x \in \mathbb{K}_\ell$ wie folgt definiert:

$$\text{Zkpl}_\ell(x) = \begin{cases} 0\text{bin}_{\ell-1}(x) & \text{falls } x \geq 0 \\ 1\text{bin}_{\ell-1}(2^{\ell-1} + x) & \text{falls } x < 0 \end{cases}$$

Man kann sich überlegen, dass auch gilt:

$$\text{Zkpl}_\ell(x) = \begin{cases} \text{bin}_\ell(x) & \text{falls } x \geq 0 \\ \text{bin}_\ell(2^\ell + x) & \text{falls } x < 0 \end{cases}$$

Dass diese Zahlendarstellung Eigenschaften hat, die z. B. die technische Implementierung eines Addierwerkes erleichtern, werden Sie (hoffentlich) an anderer Stelle lernen.

Bei der Umwandlung von Zahldarstellungen zwischen verschiedenen Repräsentationsweise besteht jedenfalls eine Möglichkeit darin, den „Umweg“ über repräsentierte Zahlen zu gehen, also erst eine Num- und dann eine Repr-Abbildung hintereinander auszuführen. Ähnliches wird in späteren Kapiteln wieder auftauchen. Daher betrachten wir dieses Konzept zunächst einmal losgelöst.

8.2 KOMPOSITION VON FUNKTIONEN

Als erstes wollen wir zwei Schreibweisen vereinbaren. Sind A und B zwei Mengen, so soll B^A die Menge aller Abbildungen $f: A \rightarrow B$ bezeichnen. Der Anlass für die zunächst etwas überraschende Reihenfolge der Mengen war ursprünglich vielleicht die Tatsache, dass für endliche Mengen A und B gilt: $|B^A| = |B|^{|A|}$.

Menge aller Abbildungen

Wenn man zwei Funktionen $f: A \rightarrow B$ und $g: B \rightarrow C$ gegeben hat, dann kann man eine neue Funktion $h: A \rightarrow C$ definieren durch die Festlegung, dass für

jedes $a \in A$ gilt: $h(a) = g(f(a))$. Die Funktion h nennt man die *Komposition der Funktionen* f und g . Wie viele andere auch schreiben wir das so: $h = g \circ f$.

Dabei sind zwei Dinge zu beachten: Zum einen ist die Komposition von Funktionen nur sinnvoll zu definieren, wenn der Zielbereich der ersten Funktion Teilmenge (oder gar gleich) dem Definitionsbereich der zweiten Funktion ist. Zum anderen ist bei der Schreibweise $g \circ f$ die „rechte“ Funktion die, die *zuerst* angewendet wird. (Es gibt auch Autoren, die das genau andersherum machen.)

Um mit diesen Notationen ein bisschen zu üben, sollten Sie sich klar machen, dass für Mengen A , B und C Komposition also eine binäre Operation von der Form $\circ : C^B \times B^A \rightarrow C^A$.

Hat man vier Mengen A , B , C und D und Abbildungen $f: A \rightarrow B$, $g: B \rightarrow C$ und $h: C \rightarrow D$, dann kann man nachrechnen, dass stets $h \circ (g \circ f) = (h \circ g) \circ f$ ist. Die Komposition von Funktionen ist also assoziativ (siehe Abschnitt 4.6).

Für jede Menge M bezeichnen wir mit I_M die Abbildung

$$I_M: M \rightarrow M : x \mapsto x$$

Es für jede Abbildung $f: A \rightarrow B$ gilt $I_B \circ f = f$ und $f \circ I_A = f$. Überprüfen Sie das zur Übung für einen der beiden Fälle.

Umkehrfunktion

Es seien nun $f: A \rightarrow B$ und $g: B \rightarrow A$ zwei Abbildungen, für die also sowohl $g \circ f$ als auch $f \circ g$ definiert sind. Man sagt, dass g die *Umkehrfunktion* zu f ist, falls $g \circ f = I_A$ und $f \circ g = I_B$ ist. Man mache sich klar, dass in diesem Fall auch f die Umkehrfunktion zu g ist. Statt Umkehrfunktion sagt man manchmal auch

inverse Funktion

8.3 VON EINEM ALPHABET ZUM ANDEREN

8.3.1 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

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

Der wesentliche Punkt ist, dass die beiden Wörter **A3** und **10100011** in der jeweils „naheliegenden“ Interpretation die gleiche Bedeutung haben: die Zahl einhundert-dreiundsechzig.

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. $\text{sem}_A = \text{sem}_B \circ f$, also

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

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 (siehe Kapitel 9). Eine grundlegende Einführung in solche Fragestellungen können Sie in Vorlesungen über die Semantik von Programmiersprachen bekommen.

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

- *Lesbarkeit*: Übersetzungen können zu kürzeren und daher besser lesbaren Texten führen. [A3](#) ist leichter erfassbar als [10100011](#) (findet der Autor dieser Zeilen).
- *Verschlüsselung*: Im Gegensatz zu verbesserter Lesbarkeit übersetzt man mitunter gerade deshalb, um die Lesbarkeit möglichst unmöglich zu machen,

jedenfalls für Außenstehende. Wie man das macht, ist Gegenstand von Vorlesungen über Kryptographie.

- *Kompression*: Manchmal führen Übersetzungen zu kürzeren Texten, die also weniger Platz benötigen. Und zwar *ohne* zu einem größeren Alphabet überzugehen. Wir werden im Abschnitt 8.4 über Huffman-Codes sehen, warum und wie das manchmal möglich ist.
- *Fehlererkennung und Fehlerkorrektur*: Manchmal kann man Texte durch Übersetzung auf eine Art länger machen, dass man selbst dann, wenn ein 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 Vorlesung hören werden.

Es gibt einen öfter anzutreffenden Spezialfall, in dem man sich um die Einhaltung der Forderung $\text{sem}_A(w) = \text{sem}_B(f(w))$ keine Gedanken machen muss. Zumindest bei Verschlüsselung, aber auch bei manchen Anwendungen von Kompression ist es so, dass man vom Übersetzten $f(x)$ eindeutig zurückkommen können möchte zum ursprünglichen x . Dann ist f mit anderen Worten injektiv. In diesem Fall kann man 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.

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

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

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

8.3.2 Homomorphismen

Homomorphismus Es seien A und B zwei Alphabete und $h : A^* \rightarrow B^*$ eine Abbildung. Eine solche Abbildung h nennt man einen *Homomorphismus*, wenn für jedes $w_1 \in A^*$ und jedes $w_2 \in A^*$ gilt:

$$h(w_1 w_2) = h(w_1) h(w_2) .$$

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

Für jeden Homomorphismus h gilt $h(\varepsilon) = h(\varepsilon\varepsilon) = h(\varepsilon)h(\varepsilon)$. Also ist $|h(\varepsilon)| = |h(\varepsilon)| + |h(\varepsilon)|$, d.h. $|h(\varepsilon)| = 0$, und folglich muss für jeden Homomorphismus $h(\varepsilon) = \varepsilon$ sein.

Homomorphismen sind schon eindeutig festgelegt, wenn man das Bild jedes einzelnen Symbolen kennt:

8.3 Lemma. Es seien A und B zwei Alphabete und $h : A^* \rightarrow B^*$ und $g : A^* \rightarrow B^*$ zwei Homomorphismen. Dann gilt: Wenn nur für jedes $x \in A$ $h(x) = g(x)$ ist, dann gilt sogar schon für jedes $w \in A^*$, dass $h(w) = g(w)$ ist.

8.4 Beweis. Es seien A, B, h und g wie in den Voraussetzungen des Lemmas und für jedes $x \in A$ gelte $h(x) = g(x)$. Es ist zu zeigen, dass für jedes $w \in A^*$ gilt: $h(w) = g(w)$. Ein möglicher Beweis benutzt *Induktion über die Wortlänge*. Dafür zeigt man: Für jede Wortlänge $n \in \mathbb{N}_0$ gilt: $\forall w \in A^n : h(w) = g(w)$.

Induktion über die Wortlänge

Der *Induktionsanfang* ($n = 0$) ist leicht, da dann $w = \varepsilon$ sein muss und daher nach der Überlegung vor dem Lemma $h(\varepsilon) = \varepsilon = g(\varepsilon)$ ist.

Für den *Induktionsschritt* sei ein $n \in \mathbb{N}_0$ beliebig und für jedes $w \in A^n$ gelte die *Induktionsvoraussetzung*: $h(w) = g(w)$. Zu zeigen ist, dass auch für jedes Wort der Länge $n + 1$ beide Homomorphismen den gleichen Funktionswert liefern. Jedes Wort der Länge $n + 1$ kann man zum Beispiel schreiben in der Form wx mit $w \in A^n$ und $x \in A$. Dann gilt auch:

$$\begin{aligned} h(wx) &= h(w)h(x) && \text{da } h \text{ Homomorphismus} \\ &= g(w)h(x) && \text{Induktionsvoraussetzung} \\ &= g(w)g(x) && \text{Voraussetzung des Lemmas} \\ &= g(wx) && \text{da } g \text{ Homomorphismus} \end{aligned}$$

■

Wenn ein Homomorphismus h vorliegt, dann ist er also durch die Bilder $h(x)$ der einzelnen Symbole eindeutig festgelegt. Und tatsächlich legt auch *jede* Abbildung $f : A \rightarrow B^*$ schon einen Homomorphismus fest. Dazu definiert man eine Abbildung $f^{**} : A^* \rightarrow B^*$ vermöge

$$\begin{aligned} f^{**}(\varepsilon) &= \varepsilon \\ \text{für jedes } w \in A^* \text{ und jedes } x \in A : f^{**}(wx) &= f^{**}(w)f(x) \end{aligned}$$

Dann gilt:

8.5 Lemma. Für jede Abbildung $f : A \rightarrow B^*$ ist f^{**} ein Homomorphismus.

Den Beweis würde man wieder mittels vollständiger Induktion führen. Wir ersparen uns das an dieser Stelle.

Wann ein Homomorphismus $h : A^* \rightarrow B^*$ eine Codierung, also injektiv ist, 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 $h(x_2)$.

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

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

Wir schreiben nun zunächst einmal Folgendes hin:

$$u(w) = \begin{cases} \varepsilon, & \text{falls } w = \varepsilon \\ au(w'), & \text{falls } w = 1w' \\ bu(w'), & \text{falls } w = 01w' \\ cu(w'), & \text{falls } w = 001w' \\ \perp, & \text{sonst} \end{cases}$$

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

$$\begin{aligned} u(100101) &= au(00101) \\ &= acu(01) \\ &= acbu(\varepsilon) \\ &= acb \end{aligned}$$

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

Man spricht hier auch davon, dass die oben festgelegte Abbildung u *wohldefiniert* ist. Über Wohldefiniertheit muss man immer dann nachdenken, wenn ein

Funktionswert potenziell „auf mehreren Wegen“ festgelegt wird. Dann muss man sich entweder klar machen, dass in Wirklichkeit wie im obigen Beispiel immer nur ein Weg „gangbar“ ist, oder dass auf den verschiedenen Wegen am Ende der gleiche Funktionswert herauskommt. Für diesen zweiten Fall werden wir später noch Beispiele sehen, z. B. in dem Kapitel über Äquivalenz- und Kongruenzrelationen.

Allgemein kann man bei einem präfixfreien Code also so decodieren:

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

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

Das praktische Beispiel schlechthin für einen Homomorphismus ist die Repräsentation des ASCII-Zeichensatzes (siehe Abschnitt 3.2.1) im Rechner. Das geht einfach so: Ein Zeichen x mit der Nummer n im ASCII-Code wird codiert durch dasjenige Wort $w \in \{0, 1\}^8$ (ein sogenanntes Byte), für das $\text{Num}_2(w) = n$ ist. In Abschnitt 8.1.5 haben wir in diesem Fall auch geschrieben: $w = \text{bin}_8(n)$. Und längere Texte werden übersetzt, indem man nacheinander jedes Zeichen einzeln so abbildet.

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

8.3.3 Beispiel Unicode: UTF-8 Codierung

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

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

UTF-8

Nachfolgend ist die Definition dieses Homomorphismus in Ausschnitten wiedergegeben. Sie stammen aus *RFC 3629* (<http://tools.ietf.org/html/rfc3629>, 2.11.2018).

RFC 3629

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

Char. number range (hexadecimal)	UTF-8 octet sequence (binary)
0000 0000 - 0000 007F	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 druckbaren Zeichen aus dem ASCII-Zeichensatz dort die gleichen Nummern haben wie bei Unicode, hat dieser Homomorphismus die Eigenschaft, dass Texte, die nur ASCII-Zeichen enthalten, bei der ASCII-Codierung und bei UTF-8 die gleiche Codierung besitzen.

8.4 HUFFMAN-CODIERUNG

Huffman-Codierung

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

Dafür ist es (wie an anderen Stellen) hilfreich, eine kompakte Notation für die Zahl der Vorkommen eines Zeichens $x \in A$ in einem Wort $w \in A^*$ zu haben. Wir definieren dazu für alle Alphabete A und alle $x \in A$ Funktionen $N_x : A^* \rightarrow \mathbb{N}_0$, die wie folgt festgelegt sind:

$$N_x(\varepsilon) = 0$$

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

Dann ist zum Beispiel

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

Nun zur 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.

8.4.1 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 sogenannten Baum (mehr dazu in Kapitel 15). Die Blätter des Baumes entsprechen einzelnen Symbolen $x \in A$, 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 $w = \text{afebfecaffdeddccefbef}$ gegeben ist, dann kann sich der Baum ergeben, der in Abbildung 8.2 dargestellt ist. In diesem Fall ist dann der Homomorphismus gegeben durch

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

Es bleibt zu beschreiben, wie man den Baum konstruiert. Zu jedem Zeitpunkt hat man eine Menge M von „noch zu betrachtenden Symbolmengen mit ihren Häufigkeiten“. Diese Menge ist initial die Menge aller $\{x\}$ für $x \in A$ mit den zugehörigen Symbolhäufigkeiten, die wir so aufschreiben wollen:

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

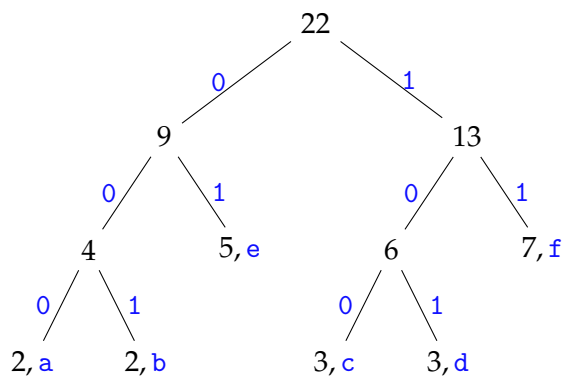
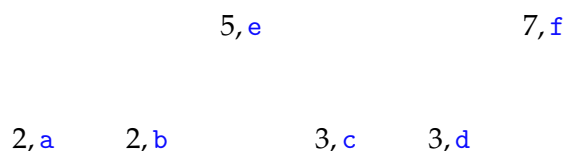


Abbildung 8.2: Ein Beispielbaum für die Berechnung eines Huffman-Codes.

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



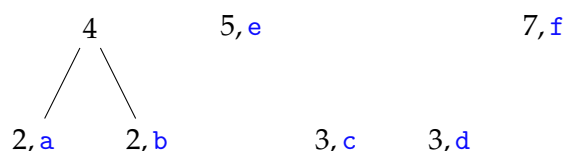
Solange eine Menge M_i noch mindestens zwei Paare enthält, tut man folgendes:

- 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

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

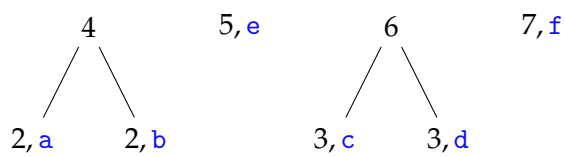
- Als zweites fügt man dem schon konstruierten Teil des Graphen einen weiteren Knoten hinzu, markiert mit der Häufigkeit $k_1 + k_2$ und Kanten zu den Knoten, die für (X_1, k_1) und (X_2, k_2) eingefügt worden waren



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

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

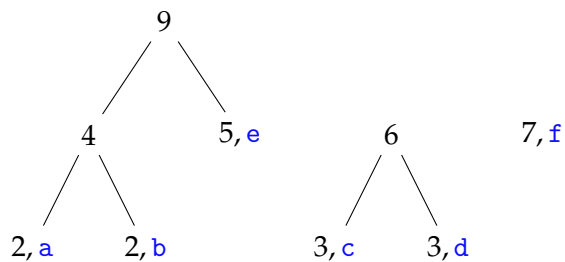
und der Graph sieht dann so aus:



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

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

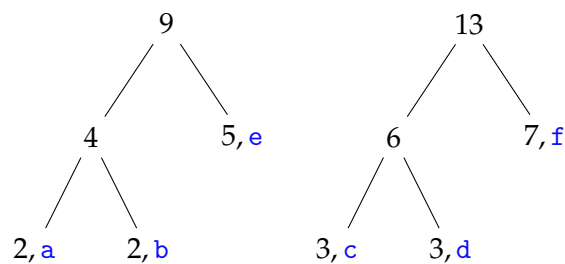
und der Graph sieht dann so aus:



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

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

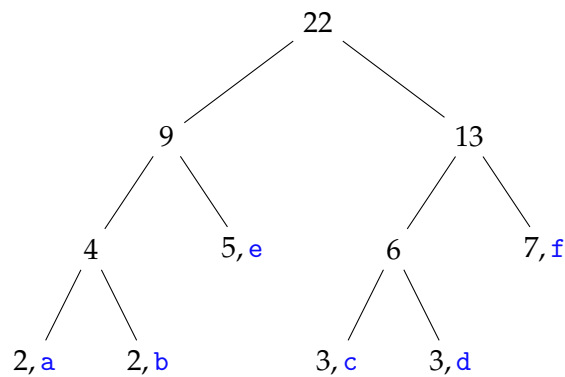
und der Graph sieht dann so aus:



Zum Schluss berechnet man noch

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

und es ergibt sich der Baum



Aus diesem ergibt sich durch die Beschriftung der Kanten die Darstellung in Abbildung 8.2.

8.4.2 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://web.archive.org/web/20090917093112/http://www.maths.abdn.ac.uk/~igc/tch/mx4002/notes/node59.html>, 6.11.2019).

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.

Block-Codierung

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

8.5 AUSBLICK

Wer Genaueres über UTF-8 und damit zusammenhängende Begriffe bei Unicode wissen will, dem sei die detaillierte Darstellung im *Unicode Technical Report UTR-17* empfohlen, den man unter <http://www.unicode.org/reports/tr17/> (6.11.2019) im WWW findet.

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

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. Eine zweite werden Sie in der Vorlesung „Theoretische Grundlagen der Informatik“ kennen lernen. 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 in früheren Kapiteln schon erwähnten *Requests for Comments* der *Internet Engineering Task Force* (siehe <http://www.ietf.org/rfc/>, 9.12.20).

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 das „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 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 fassen 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

9.3 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:

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

9.3.1 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 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 $\text{Mem} = \text{Val}^{\text{Adr}}$ die Menge aller möglichen Speicherzustände, also die Menge aller Abbildungen von Adr nach Val .

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. 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:

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

Etwas knapper aufgeschrieben:

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

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:

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

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

Allerdings 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:

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

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

Es gibt aber Speicher, bei denen diese zunächst plausibel aussehende Bedingung *nicht* erfüllt ist, z. B. sogenannte Caches (bzw. allgemeiner Assoziativspeicher). Für eine adäquate Formalisierung des Schreibens in einen solchen Speicher müsste man also die 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 Name des Empfängers angegeben. So etwas wie „Thomas Worsch“ ist aber nicht eine weitere Spezifizierung des physikalischen Ortes innerhalb von „Fasanengarten 5, 76131 Karlsruhe“, an den der Brief transportiert werden soll, sondern eine „inhaltliche“ Beschreibung. Ähnliches gibt es auch bei Speichern. Auch sie haben eine beschränkte Größe, aber im Extremfall wird für die Adressierung der gespeicherten Wörter sogar überhaupt keine Angabe über den physikalischen Ort gemacht, sondern nur eine Angabe über den Inhalt. Und dann kann Speichern eines Wertes dazu führen, dass ein anderer Wert aus dem Speicher entfernt wird.

Zum Schluss noch kurz eine Anmerkung zur Frage, wozu Gleichungen wie 9.1 und 9.2 gut sein können. Diese Gleichungen sagen ja etwas darüber aus, „wie sich Speicher verhalten soll“. Und daher kann man sie als Teile der *Spezifikation* dessen

auffassen, was Speicher überhaupt sein soll. Das wäre ein einfaches Beispiel für etwas, was Sie unter Umständen später in Ihrem Studium als algebraische Spezifikation von Datentypen kennen lernen werden.

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

9.4 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 <https://akkadia.org/drepper/cpumemory.pdf> (9.12.20) zur Verfügung.

10 PROZESSOR

In diesem Kapitel werden wir die MIMA kennenlernen. Das ist ein stark vereinfachtes Modell eines Prozessors, das aber alle fundamentalen Bestandteile und Konzepte aus realen Prozessoren enthält. In der Vorlesung „Rechnerorganisation“ wird an Hand der MIMA genauer auf den Aufbau von Prozessoren und insbesondere auf das Konzept der *Mikroprogrammierung* eingegangen werden.

MIMA

Mikroprogrammierung

In Kapitel 22 werden wir noch die Erweiterung MIMA-x kennenlernen, die das Verständnis einiger Konzepte erleichtert, die z. B. in den Vorlesungen „Programmieren“ und „Softwaretechnik“ eingeführt werden.

10.1 EINFACHE „HARDWARE“-„BAUSTEINE“

Als erstes führen wir *Drähte* ein. Das soll hier etwas noch Einfacheres sein als üblich. Ein einzelner Draht hat zwei Enden. Sinnvollerweise befindet sich am einen Ende ein „Erzeuger“ und am anderen Ende ein „Verbraucher“. Für den Zustand eines Drahtes gibt es *drei* Möglichkeiten

Draht

- Es wird eine 0 übertragen.
- Es wird eine 1 übertragen.
- Es wird *nichts* übertragen.

Der Zustand *nichts* wird hierbei häufig durch ein Z repräsentiert.

Ein *Erzeuger* hat einen *Ausgang*, auf dem er ein Bit 0 oder 1 „ausgeben“ kann, das über den Draht zum Verbraucher übertragen wird. Er kann aber auch „nichts“ ausgeben. Dann überträgt der Draht auch nichts.

Erzeuger

Ausgang

Ein *Verbraucher* hat einen *Eingang*, auf dem er ein Bit lesen kann, wenn der angeschlossene Draht eines überträgt, er muss aber nicht.

Verbraucher

Eingang

Ein Draht kann seinen Zustand nicht speichern; er hat *immer* genau das Bit als Zustand, das der Erzeuger, an den er angeschlossen ist, gerade ausgibt, bzw. Zustand Z, wenn der Erzeuger nichts ausgibt.

Wird auf einem Draht ein Bit übertragen und vom angeschlossenen *Verbraucher* gelesen, dann kann er diese Information verarbeiten. Wie er das genau tut ist dabei je nach Bedarf unterschiedlich und kann von schlichtem Weitergeben bis zum Ignorieren der Daten reichen. Insbesondere kann ein Verbraucher auch wieder ein Erzeuger (für einen anderen Draht) sein.

Verbraucher

In Abbildung 10.1 sind die drei Basisvarianten der Datenübertragung die man aus den bisher bekannten Elementen bauen kann dargestellt.

Einen einfachen Erzeuger kann man sich so vorstellen wie in Abbildung 10.2 zu sehen. Es gibt einen internen Speicher *M*, der zu *jedem* Zeitpunkt ein Bit (also

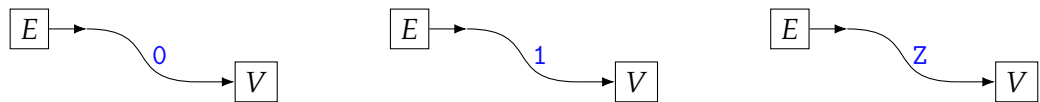


Abbildung 10.1: Verbindung eines Erzeugers mit einem Verbraucher durch einen Draht und die dabei möglichen Drahtzustände.

0 oder 1) gespeichert hat. Die technische Realisierung von M soll hier nicht weiter interessieren. Sie ist Gegenstand von Vorlesungen aus der technischen Informatik.

Die Steuerleitung S_r fungiert als Eingang. Wenn auf ihr eine 1 übertragen wird, dann wird auf der Ausgabeleitung D_o genau das Bit übertragen, das im internen Speicher M abgelegt ist.

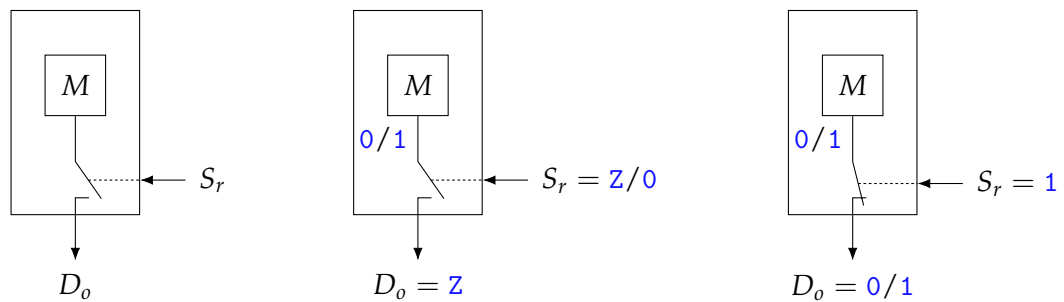


Abbildung 10.2: Struktur eines Erzeugers: Der interne Speicher M enthält ein Bit, das auf D_o ausgegeben wird, sofern S_r eine 1 transportiert.

Wenn man ein Ende eines Drahtes mit einem Ende eines anderen Drahtes verbindet, entsteht wieder ein Draht. Man darf auch je ein Ende von mehr als zwei Drähten miteinander verbinden, um kompliziertere Verbindungsstrukturen zu realisieren. Dadurch entsteht ein sogenannter *Bus*. Das sieht dann zum Beispiel so aus wie in [Abbildung 10.3](#).

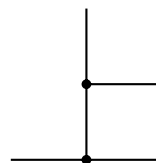


Abbildung 10.3: Ein einfacher Bus

In einem solchen Fall kann ein Erzeuger mehrere Verbraucher mit einem Bit versorgen. Man darf auch *mehrere* Erzeuger mit einem oder mehreren Verbrauchern verbinden. Dann drängt sich die Frage auf, was denn passieren soll, wenn

mehrere Erzeuger verschiedene Bits übertragen wollen. Das soll gar nicht erlaubt sein.

Genauer soll gelten:

1. Auch wenn mehrere Erzeuger miteinander verbunden sind, darf zu jedem Zeitpunkt *höchstens einer* ein Bit übertragen wollen. Das gleichzeitige Übertragen von mehreren Erzeugern auf einem Bus ist nicht erlaubt.
2. Wenn genau ein Erzeuger ein Bit auf dem Bus ausgibt, dann wird es auch alle Verbraucher übertragen und jeder von Ihnen kann es lesen (muss aber nicht).

Man kann Busse auch parallel schalten, was es uns ermöglicht mehrere Bits zu übertragen. So kann man auf einem parallelen Bus mit 8 Leitungen beispielsweise ein Byte übertragen. Ein Beispiel für einen solchen Bus ist in [Abbildung 10.4](#) gegeben.

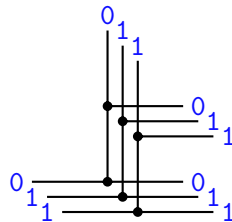


Abbildung 10.4: Ein paralleles Bussystem

Zusätzlich zum Erzeugen und Empfangen von Daten ist es oft noch nötig Daten zu speichern. Dafür verwendet man *Speicher*-Elemente. Ein Speicherelement besteht aus 3 Bestandteilen:

Speicher

1. Ein interner Speicher M , der *immer* entweder im Zustand 0 oder 1 ist.
2. Einer Datenleitung D_i (Eingang), auf der ein zu speicherndes Bit entgegengenommen wird, wenn auf einer Steuerleitung S_w (Eingang) eine 1 vorliegt.
3. Einer Datenleitung D_o (Ausgang), auf der ein gespeichertes Bit ausgegeben wird, wenn auf einer Steuerleitung S_r (Eingang) eine 1 vorliegt.

Auch hier ist die Realisierung von M ein Aspekt, der der technischen Informatik vorbehalten bleibt. Für einen Schreibvorgang muss S_w auf 1 gesetzt werden und an D_i der zu speichernde Wert anliegen. Dieser wird dann in M übertragen. Analog muss für einen Lesevorgang S_r auf 1 gesetzt werden, wodurch D_o von Z auf den aktuell gespeicherten Wert geht. Der Aufbau eines Speicherelements ist in [Abbildung 10.5](#) dargestellt. Eine wichtige Bedingung für diese Art von Speicher ist, dass *niemals* gleichzeitig S_r und S_w auf 1 gesetzt sind. Das Verhalten des Speichers ist für diesen Zustand nicht definiert.

Genau wie Busse kann man auch Speicher parallel schalten und erhält dadurch *Register*. In einem Register werden immer alle einzelnen Speicherzellen gleich-

Register

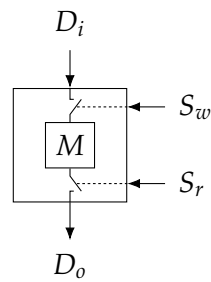


Abbildung 10.5: Ein einfaches Speicher-Element für ein Bit

zeitig angesprochen, indem immer gleichzeitig immer für alle Speicher-Elemente gleichzeitig S_r auf 1 bzw. S_w auf 1 und die D_i auf die zuspichernden Werte gesetzt wird. Die D_i werden getrennt gehalten, sodass man in verschiedenen Speicher-Elementen gleichzeitig verschiedene Bits speichern kann. Analoges gilt für die einzelnen D_o -Leitungen. In Abbildung 10.6 wird der Aufbau eines 4-Bit-Registers illustriert.

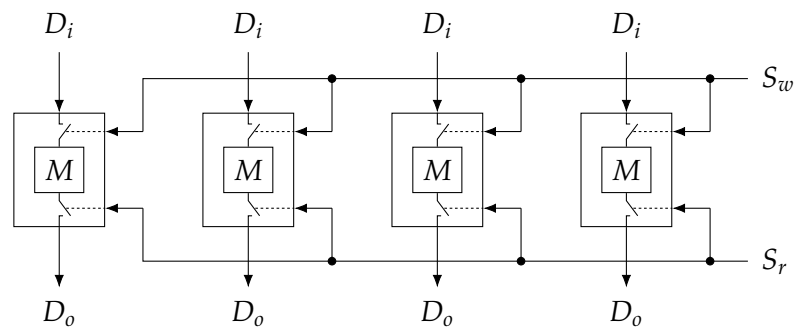


Abbildung 10.6: Ein Register mit 4 Speicherzellen

10.2 GROBSTRUKTUR DER MIMA

*Von-Neumann-
Architektur*

Die MIMA ist eine starke Vereinfachung eines realen Prozessors und folgt in ihrem Aufbau den Prinzipien der *Von-Neumann-Architektur*. Ein Rechner mit Von-Neumann-Architektur besitzt einen zentralen Bus, an den alle Komponenten angeschlossen sind. Diese Bauform ist nach dem ungarisch-amerikanischen Mathematiker John von Neumann benannt. Außerdem besitzt er, in Abgrenzung zur sogenannten Harvard-Architektur einen gemeinsamen Speicher für Daten und Programm.

Ein minimaler Prozessor benötigt drei grundlegende Baugruppen. Das sogenannte *Steuerwerk*, das den Ablauf im Prozessor steuert, das *Rechenwerk*, das die Berechnungen durchführt und das *Speicherwerk*, über das der Prozessor mit dem Hauptspeicher verbunden ist. Zusätzlich sind im Normalfall einige Register vorhanden, die zur Ablaufsteuerung oder zur Speicherung von Zwischenergebnissen im Prozessor genutzt werden. Bei der MIMA sind diese drei Teile deutlich getrennt. Eine grobe Übersicht ist in Abbildung 10.7 dargestellt.

Steuerwerk
Rechenwerk
Speicherwerk

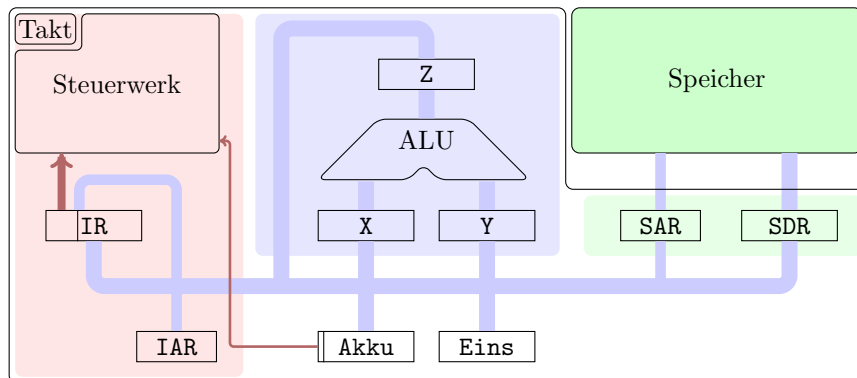


Abbildung 10.7: Der Aufbau der MIMA

Das *Steuerwerk* ist hierbei die Ablaufsteuerung des Prozessors (in Abbildung 10.7 rot hinterlegt). Es speichert in seinem Instruktionsregister (IR) den aktuellen *Maschinenbefehl* und außerdem im Instruktionsadressregister (IAR) die Adresse des nächsten auszuführenden Befehls. Im „eigentlichen“ Steuerwerk wird der Befehl aus dem IR zunächst *dekodiert* und dann *ausgeführt*. Über die Dekodierung werden wir in dieser ersten einfachen Einführung nicht weiter sprechen. Jedem Befehl ist eine ganze Folge sogenannter *Mikrobefehle* zugeordnet. Das sind elementare Anweisungen, von denen die MIMA in jedem Schritt intern jeweils eine ausführen kann. Die Ausführung eines Maschinenbefehls wird dadurch realisiert, dass die zugehörigen Mikrobefehle der Reihe nach ausgeführt werden. Im Laufe der Ausführung jedes Befehls wird das IAR um 1 erhöht, weil (von wenigen Ausnahmen abgesehen) nacheinander auszuführende Maschinenbefehle in aufeinander folgenden Adressen im Hauptspeicher abgelegt sind. Die genaue Funktionsweise eines solchen mikroprogrammierten Steuerwerks wird in den Vorlesungen der Technischen Informatik ausführlich erläutert.

Steuerwerk
Maschinenbefehl
Mikrobefehle

Das *Rechenwerk* ist der Teil der MIMA, der die „eigentlichen Berechnungen“ durchführt (in Abbildung 10.7 blau hinterlegt). Dafür erhält es in die Register X und Y die Eingabedaten und vom Steuerwerk zusätzlich noch einen sogenannten *Operations-Code*, der angibt, welche Operation auf den Daten ausgeführt werden

Rechenwerk

Akkumulator soll. Das Ergebnis wird anschließend in das Register Z geschrieben. Am Ende der meisten Befehle wird das Ergebnis noch vom Register Z in den *Akkumulator (Akku)* transportiert, bei manchen Befehlen auch in ein anderes Register. Der Akkumulator ist für viele Operationen einer der Operanden und damit gleichzeitig der wichtigste Speicher für Zwischenergebnisse. So kann man mit dem Befehl `ADD a` den Wert, der an Adresse *a* im Hauptspeicher abgelegt ist, auf den aktuellen Wert im Akkumulator aufaddieren. Das Ergebnis wird dann in den Akkumulator zurückgeschrieben. Sämtliche Befehle und ihre Operanden werden in Abschnitt 10.3 nochmals genauer erläutert.

Speicherwerk Das *Speicherwerk* ist die Anbindung der MIMA an den Hauptspeicher. Dort müssen zum einen die Maschinenbefehle des auszuführenden Programmes abgelegt sein, und zum anderen die Daten, die verarbeitet werden sollen. Die Komponenten des Speicherwerks sind in 10.7 grün dargestellt. Das Speichern und Laden von Daten funktioniert mit Hilfe von Speicherdatenregister (SDR) und Speicheradressregister (SAR). Um Daten zu speichern werden diese in das SDR geschrieben. Zusätzlich wird die Adresse, unter der gespeichert werden soll, in das SAR geschrieben. Dann gibt das Steuerwerk über eine Steuerleitung dem Speicherwerk die Anweisung, die Daten an den Hauptspeicher zu übertragen. Um Daten aus dem Speicher zu lesen, wird analog zum Speichern zunächst die Adresse, von der geladen werden soll, in das SAR geschrieben. Dann gibt das Steuerwerk über eine Steuerleitung dem Speicherwerk die Anweisung zu lesen. Die Daten werden daraufhin vom Speicher in das SDR geladen.

Hauptspeicher Der *Hauptspeicher* der MIMA ist mit 20 Bit adressiert, bietet also 2^{20} Adressen. Jede Adresse speichert einen 24 Bit langen Wert. Jeder einzelne Wert wird auch *Speicherwort* genannt. Wie schon zu Anfang des Kapitels erläutert, handelt es sich bei der MIMA um eine Von-Neumann-Architektur in der Daten und Programm physikalisch im selben Speicher liegen. Die Trennung von Datenspeicher und Programmspeicher muss bei MIMA-Programmen daher durch eigene Konventionen sichergestellt werden. Im Programmspeicher befinden sich die Maschinenbefehle für die MIMA. Die genauen Befehle werden in Abschnitt 10.3 erläutert. Der Datenspeicher wird für Eingaben, Zwischenergebnisse und Ausgaben genutzt. Zahlen werden dort in Zweierkomplementdarstellung gespeichert.

10.3 MASCHINENBEFEHLE DER MIMA

Maschinenbefehl Um Programme für die MIMA zu schreiben, benötigen wir sogenannte *Maschinenbefehle*, die aus dem Speicher geladen und ausgeführt werden können. Ein Programm liegt als Folge von Befehlen im Speicher der MIMA.

Es werden Befehle für drei verschiedenen Typen von Aufgaben unterschieden:

1. Transport von Daten
2. Verarbeitung von Daten
3. Beeinflussung der Befehlsreihenfolge

Wenn man konkret Maschinenbefehle aufschreiben will, kann man unterschiedlich vorgehen. Entweder kann man jeden Befehl als Folge von 24 Bits notieren, so wie sie auch in der MIMA intern verwendet wird. Üblicherweise nutzt man aber eine „symbolische“ und viel besser lesbare Notation. Was man sich darunter vorstellen sollte, wird anhand der nachfolgenden Abschnitte sicher klar werden. Zum Beispiel stellt die Bitfolge `0010000000000000000101010` den Befehl dar, für den wir `STV 101010` oder noch deutlicher `STV 42` schreiben. Eine vollständige Liste aller Befehle und ihrer Kodierungen wird in der Vorlesung „Rechnerorganisation“ gegeben werden. In den nachfolgenden Unterabschnitten findet man aber zumindest einen groben Überblick über die Befehle, sortiert nach den weiter vorne genannten Befehlstypen.

10.3.1 Befehle zum Datentransport

Diese Befehle dienen dazu, Daten, die an einer Stelle abgelegt sind, an eine andere zu kopieren. Das betrifft sowohl die Richtung vom Hauptspeicher in den Akkumulator als auch umgekehrt.

- `LDC const` schreibt die Konstante *const* ($0 \leq \text{const} < 2^{20}$) in den Akkumulator.
- `LDV adr` schreibt den Wert, der im Hauptspeicher an der Adresse *adr* steht, in den Akkumulator.
- `STV adr` schreibt den Wert, der im Akkumulator steht, in den Hauptspeicher an Adresse *adr*.
- `LDIV adr` schreibt den Wert, der im Hauptspeicher an der Adresse steht, die selbst im Hauptspeicher an Adresse *adr* steht, in den Akkumulator.
- `STIV adr` schreibt den Wert, der im Akkumulator steht, im Hauptspeicher an die Adresse, die selbst im Hauptspeicher an der Adresse *adr* steht.

Die Adresse *adr* muss in allen Fällen im Bereich ($0 \leq \text{adr} < 2^{20}$) liegen. Bei den Befehlen `LDIV adr` und `STIV adr` mit *indirekter Adressierung* muss auch der Wert, der im Hauptspeicher an Adresse *adr* liegt aus diesem Intervall stammen. In Tabelle 10.1 findet man ein Beispielprogramm und wie sich die gespeicherten Werte im Laufe der Abarbeitung ändern.

indirekte Adressierung

Tabelle 10.1: Ein einfaches Beispielprogramm mit direkter und indirekter Adressierung bei den Speicherzugriffen. Fehlen Einträge bedeuten undefinierte bzw. irrelevante Speicherinhalte.

	Akku	Hauptspeicher					
		...	M(12)	...	M(42)	...	M(66)
LDC 29	29						
STV 42	29				29		
LDC 66	66				29		
STV 12	66		66		29		
LDV 42	29		66		29		
STIV 12	29		66		29		29

10.3.2 Befehle für arithmetische-logische Operationen

arithmetisch-logische Befehle

Mithilfe der sogenannten *arithmetisch-logischen Befehle* kann die MIMA „mathematische Operationen“ ausführen.

- **ADD *adr*** addiert den Wert, der an der Adresse *adr* steht, auf den Wert des Akkumulators auf und schreibt das Ergebnis wieder in den Akkumulator. Die Werte werden als Zweierkomplementdarstellungen interpretiert, einen Übertrag von der höchstwertigen Stelle gibt es also nicht. Es sei an Abschnitt 8.1.5 erinnert und die Tatsache, dass man negative Zahlen daran erkennen kann, dass ihre Zweierkomplementdarstellung als höchstwertiges Bit eine 1 hat.
- Die Befehle **AND *adr***, **OR *adr***, bzw. **XOR *adr***, verknüpfen bitweise den Wert aus dem Akkumulator mit dem, der im Hauptspeicher an Adresse *adr* steht, mithilfe eines logischen Und, Oder, bzw. Exklusiven Oder und schreiben das Ergebnis in den Akkumulator. Hierbei wird eine 1 als wahr und eine 0 als falsch interpretiert.
- **NOT** invertiert alle Bits des Wertes im Akkumulator.
- **RAR** rotiert die Bits im Akkumulator um eine Stelle nach rechts. Das vormals ganz rechts stehende Bit wird an die Stelle ganz links gesetzt.

- **EQL** *adr* setzt alle Bits im Akkumulator auf 1 (das entspricht der Zahl −1), falls der Wert im Akkumulator gleich dem ist, der an Adresse *adr* im Hauptspeicher steht. Ansonsten werden alle Bits im Akkumulator auf 0 gesetzt.

In Tabelle 10.2 findet man ein Beispielprogramm, das einige der eben genannten Befehle benutzt.

Tabelle 10.2: Ein einfaches Beispielprogramm mit **LDC**, **ADD** und **EQL**

	Akku	...	M(46)	M(47)	M(48)	M(49)	...
initial	?		11	22	33	66	
LDC 0							
	0		11	22	33	66	
ADD 46							
	11		11	22	33	66	
ADD 47							
	33		11	22	33	66	
ADD 48							
	66		11	22	33	66	
EQL 49							
	-1		11	22	33	66	

10.3.3 Sprungbefehle

Normalerweise werden die Befehle eines Programms in der Reihenfolge ausgeführt, in der sie an aufeinanderfolgenden Adressen im Speicher liegen. Wie im vorherigen Kapitel beschrieben, wird dafür der Wert im Instruktionsadressregister(IAR) nach jedem Befehl um 1 erhöht. Will man diese Reihenfolge ändern, müssen dafür Sprungbefehle verwendet werden. Dadurch können Konstrukte wie bedingte Anweisungen und Schleifen, die aus vielen üblichen Programmiersprachen bekannt sind, realisiert werden.

- **JMP** *adr* ist der sogenannte *unbedingte Sprungbefehl*. Nach seiner Ausführung wird das Programm mit dem Befehl fortgesetzt, der an Adresse *adr* im Hauptspeicher steht. *unbedingter Sprungbefehl*
- **JMN** *adr* ist ein sogenannte *bedingter Sprungbefehl*. *bedingter Sprungbefehl*

Wenn der Wert im Akkumulator negativ ist, dann setzt das Programm mit dem Befehl fort, der an Adresse *adr* im Hauptspeicher steht. Andernfalls bewirkt die Ausführung von **JMN** *adr* gar nichts und das Programm wird mit

dem Befehl fortgesetzt, der im Speicher unmittelbar nach diesem Sprungbefehl steht.

- **HALT** beendet die Ausführung des Programms.

Tabelle 10.3: Ein Beispielprogramm mit **JMN** und **JMN**

000101	LDIV 100001	«lädt den Wert der Adresse, die in Adresse 33 steht»
000110	EQL 010000	«setzt Akku auf -1, falls Wert im Akku dem von Adresse 16 entspricht»
000111	JMN 111101	«springt an Adresse 61, falls der Akku negativ ist»
001000	LDC 1	
001001	ADD 100001	
001010	STV 100001	«erhöht Wert an Adresse 33 um 1»
001011	JMP 000101	«springt zum Befehl an Adresse 5»

Dieses Programm geht den Speicher der MIMA solange durch, bis ein bestimmter Wert, der an Adresse 16 steht, gefunden ist und springt dann zu Adresse 61. Der Befehl **JMN** fungiert hier als Schleife.

10.4 MIKROPROGRAMMSTEUERUNG DER MIMA

Nach den Maschinenbefehlen wenden wir uns jetzt den Mikrobefehlen zu. Programme, die aus einer Folge von Mikrobefehlen bestehen, nennt man *Mikroprogramme*. Die Ausführung jedes Maschinenbefehls wird durch die Ausführung eines Mikroprogrammes realisiert. Dabei kann man immer die *Befehlsholphase*, die *Befehlsdecodierphase* und die *Befehlsausführungsphase* unterscheiden. Jeder dieser Phasen besteht aus mehreren Mikrobefehlen.

Mikroprogramm
Befehlsholphase
Befehlsdecodierphase
Befehlsausführungsphase
Steuersignale
Meldesignale

Bei der Ausführung eines Mikrobefehles spielen *Steuersignale* und manchmal auch *Meldesignale* eine Rolle.

Steuersignale werden vom Steuerwerk über Steuerleitungen z. B. an Register (oder den externen Hauptspeicher) geleitet, um dort einen Wert zu lesen oder zu schreiben.

Meldesignale werden über Meldeleitungen zum Steuerwerk hin geleitet, damit es seine Arbeitsweise von externen Ereignissen abhängig machen kann, z. B. bei einem bedingten Sprung.

Welche Signale das in jeder Phase sind, wollen wir nachfolgend zumindest teilweise betrachten, beschränken uns dabei aber weitgehend auf Meldesignale. Zur Erinnerung zeigt Abbildung 10.8 noch einmal den groben Aufbau der MIMA:

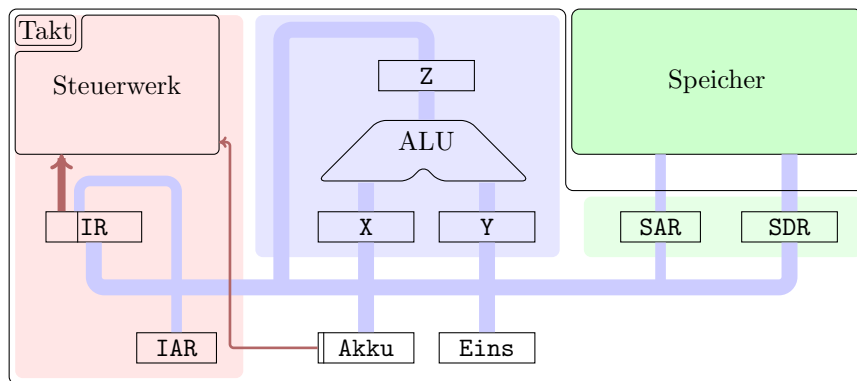


Abbildung 10.8: Grobstruktur der Mima

10.4.1 Befehlsholphase

In dieser Phase wird der nächste Maschinenbefehl aus dem Speicher ins Register IR geholt. Außerdem wird die Adresse des potenziell nächsten Befehls berechnet. Dafür muss der Inhalt des Registers IAR um 1 erhöht werden. Die tatsächlich nächste Adresse kann sich während der Ausführungsphase des aktuellen Befehls aber noch ändern, zum Beispiel wenn der Maschinenbefehl **JMP** abgearbeitet wird.

Die Holphase verläuft bei jedem Maschinenbefehl gleich, indem der Reihe nach die folgenden Mikrobefehle ausgeführt werden. Wir benutzen eine symbolische Notation, die hoffentlich auch ohne genaue Spezifikation verständlich ist:

1. $IAR \rightarrow SAR$: Adresse des Maschinenbefehls wird in's SAR geladen
2. $SAR \rightarrow \text{Speicher}; \text{Speicher} \rightarrow SDR$: Maschinenbefehl wird aus Speicher geladen und in SDR geschrieben
3. $SDR \rightarrow IR$: Maschinenbefehl wird in IR geladen
4. $IAR \rightarrow X$: Adresse des Maschinenbefehls wird in Register X der ALU geladen
5. $Eins \rightarrow Y$: Eine 1 wird in das Register Y der ALU geladen
6. $X, Y \rightarrow ALU; ALU \rightarrow Z$: Adresse des Maschinenbefehls wird um 1 inkrementiert und im Register Z gespeichert
7. $Z \rightarrow IAR$: Das Ergebnis wird in IAR geschrieben

10.4.2 Befehlsdecodierungsphase

Die 8 höchstwertigen Bits eines Speicherwortes in Register IR codieren den auszuführenden Maschinenbefehl.

In Abhängigkeit von diesen 8 Bits und dem höchstwertigen Bit des Akkumulatorinhalts verzweigt das Steuerwerk zu dem zum Maschinenbefehl zugehörigen

Mikroprogramm. Wie wir wissen gibt das höchstwertige Bit des Akkumulators an, ob der Wert im Akkumulator negativ ist. Diese Information wird für den bedingten Sprung **JMN** benötigt.

Die Details der Decodierphase sind hier nicht weiter von Bedeutung, weshalb wir das zugehörige Mikroprogramm kurz so zusammenfassen:

1. IR, Akku \rightarrow *Steuerwerk*: Steuerwerk verzweigt zu Mikroprogramm, abhängig der Werte von IR und Akku

10.4.3 Befehlsausführungsphase

Bei dieser Phase wird das zum Maschinenbefehl zugehörige Mikroprogramm ausgeführt. Während die Holphase für alle Maschinenbefehle genau gleich abläuft, gibt es in der Ausführungsphase natürlich Unterschiede. Als Beispiele wollen wir die Befehle **LDV** und **JMP** genauer betrachten. Nach der Ausführungsphase beginnt mit der Holphase das ganze für den nächsten Befehl von vorne.

Mikroprogramm für **LDV**

Das Mikroprogramm für einen Ladebefehl arbeitet ähnlich wie der Teil zum Holen eines Maschinenbefehls aus dem Hauptspeicher während der Holphase:

1. IR \rightarrow SAR: Die Adresse des Werte, der geladen werden soll, befindet sich in den 20 niedrigstwertigen Bits des Maschinenbefehls. Sie wird in Register SAR geschrieben.
2. SAR \rightarrow *Speicher*; *Speicher* \rightarrow SDR: Wert, der an dieser Adresse steht, wird aus dem Hauptspeicher nach Register SDR geladen.
3. SDR \rightarrow *Akku*: Der Wert wird von SDR in den Akkumulator transportiert.

Mikroprogramm für **JMP**

Das Mikroprogramm für den unbedingten Sprungbefehl ist ganz einfach, denn die Zieladresse befindet sich in den 20 niedrigstwertigen Bits des Maschinenbefehls.

1. IR \rightarrow IAR: Adresse des Befehls, zu dem gesprungen werden soll, wird in IAR geschrieben.

10.5 EIN BEISPIELPROGRAMM

Nachdem wir jetzt alle Grundlagen dafür haben, wollen wir uns ein erstes größeres Programm anschauen. Es soll eine Liste an Zahlen, die schon im Speicher

liegt, aufsummieren. Für eine bessere Übersichtlichkeit wurden konkrete Adressen durch *Labels* ersetzt. Außerdem wurde *list* mit der Adresse des ersten Elements der Liste und *len* mit der Listenlänge vorinitialisiert.

Genauer gehen wir davon aus, dass der Teil des Speichers, der für die Daten benutzt wird, so wie in Abbildung 10.9 im linken Teil dargestellt organisiert ist. Im rechten Teil sieht man das Programm.

<i>Label</i>	Befehl	Komm.	<i>Label</i>	Befehl	Kommentar
<i>len:</i>	5		<i>start:</i>	LDC 0	
<i>list:</i>	11	„list[0]“		STV <i>sum</i>	<i>sum</i> anfangs 0
	22	„list[1]“		STV <i>cnt</i>	<i>cnt</i> anfangs 0
	33	„list[2]“		LDC <i>list</i>	<i>ref</i> initial <i>list</i>
	44	„list[3]“		STV <i>ref</i>	
	55	„list[4]“			
<i>cnt:</i>			<i>next:</i>	LDIV <i>ref</i>	Wert des nächsten Elements,
<i>ref:</i>				ADD <i>sum</i>	aufaddiert auf Summe
<i>sum:</i>				STV <i>sum</i>	wieder speichern
				LDC 1	erhöhe Anzahl bearbeiteter
				ADD <i>cnt</i>	Listenelemente um 1
				STV <i>cnt</i>	
				LDC 1	Erhöht die Adresse des
				ADD <i>ref</i>	nächsten Elements um 1
				STV <i>ref</i>	
				LDV <i>cnt</i>	falls alle Elemente addiert
				EQL <i>len</i>	also <i>cnt=len</i>
				JMN <i>done</i>	hält das Programm
				JMP <i>next</i>	ansonsten nächstes Element
			<i>done:</i>	HALT	

Abbildung 10.9: Mima-Programm zur Summation der Elemente einer Liste von Zahlen

11 DOKUMENTE

11.1 DOKUMENTE

Die Idee zu diesem Kapitel geht auf eine Vorlesung „Informatik A“ von Till Tantau aus dem Jahre 2012 zurück.

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 abgesehen (welche fallen Ihnen ein?), 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. Seminausarbeitungen, Bachelor- und Masterarbeit, usw.) selbst zu schreiben, werden Sie merken, dass die Struktur, oder genauer gesagt das Auffinden einer geeigneten Struktur, auch zu Ihrem, also des Autors, Verständnis beitragen kann. Deswegen ist es bei solchen Arbeiten unseres Erachtens sehr ratsam, *früh damit zu beginnen*,

*ein Rat für Ihr weiteres
Studium*

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

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

11.2 STRUKTUR VON DOKUMENTEN

Auszeichnungssprache
markup language

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

11.2.1 \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/>, 27.11.2015).

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. libreoffice o. ä. verfasst wurden. Zum Beispiel liefert der Text

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

die Ausgabe

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

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

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

woraus \LaTeX die Zeile

7.2 STRUKTUR VON DOKUMENTEN

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

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

Ganz grob kann ein L^AT_EX-Dokument z. B. die folgende Struktur haben:

```
\documentclass[11pt]{report}
% so schreibt man Kommentare
% dieser Teil heißt Präambel des Dokumentes
\usepackage[T1]{fontenc}      % ohne Erklärung ....
\usepackage[ngerman]{babel}  % deutsche Trennungen etc.
\usepackage[utf8]{inputenc}  % Zeichensatz der Eingabe

\author{Thomas Worsch}
\title{Dokumente mit LaTeX}
\begin{document} % damit endet die Präambel und .....
% ..... nun der eigentliche Text .....
\maketitle        % eine Titelseite
\chapter{Überschrift}
$a^2 + b^2 = c^2$    % eine kleine Formel
\end{document}
```

Eine Liste einfacher Punkte sieht in L^AT_EX 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 L^AT_EX aufschreiben, dann könnte man z. B. zunächst geeignet die formale Sprache L_{item}

aller Texte spezifizieren, die hinter einem Aufzählungspunkt vorkommen dürfen. Dann wäre

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

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 11.2.3.

11.2.2 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 (nicht die allerneueste) Version von XHTML. Sie dürfen sich vereinfachend vorstellen, dass das ist im wesentlichen eine noch striktere Variante von HTML ist. Das nachfolgenden Fragment beschreibt teilweise, wie man syntaktisch korrekt eine Tabelle notiert.

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

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

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

Das Fragezeichen ist uns neu, hat aber eine ganz banale Bedeutung: In der uns geläufigen Notation würden wir definieren: $L^? = L^0 \cup L^1 = \{\varepsilon\} \cup L$. Mit anderen

Worten: Wenn irgendwo $L^?$ notiert ist, dann kann an dieser Stelle ein Wort aus L stehen, oder es kann fehlen. Das Auftreten eines Wortes aus L ist also mit anderen Worten optional.

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

`<!ELEMENT tbody (tr)+ >`

die folgende formale Sprache festlegt:

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

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

$$L_{\text{table}} = \{\text{<table>}\} \cdot L_{\text{caption}}^? \cdot L_{\text{thead}}^? \cdot L_{\text{tfoot}}^? \cdot (L_{\text{tbody}}^+ \cup L_{\text{tr}}^+) \cdot \{\text{</table>}\}$$

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

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

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 ist.

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.

11.2.3 Eine Grenze unserer bisherigen Vorgehensweise

Dass wir eben mit Hilfe von Produkt und Konkatenationsabschluss formaler Sprachen in einigen Fällen präzise Aussagen machen konnten, hing auch mit der Einfachheit dessen zusammen, was es zu spezifizieren galt. Es wurde, jedenfalls in

einem intuitiven Sinne, immer etwas von einer komplizierteren Art aus Bestandteilen einfacherer Art zusammengesetzt.

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

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

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

Man würde also gerne in irgendeinem Sinne L_{Klammer} mit L_{Klammer}^* und mit $\{ () \cdot L_{\text{Klammer}} \cdot \{ \} \}$ 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.

11.3 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 hinreichend einfach sind. Was man in komplizierten Fällen zum Beispiel machen kann, werden wir im nächsten Kapitel kennenlernen.

12 KONTEXTFREIE GRAMMATIKEN

12.1 REKURSIVE DEFINITION SYNTAKTISCHER STRUKTUREN

Wir hatten in [Einheit 11 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 einem 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 Erläuterungen zu „Blocks“ und „Statements“ auf <https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html>, 17.12.20). Dort stehen im Zusammenhang mit der Festlegung, was in Java eine Anweisung sei, unter anderem folgende fünf Teile:

1	Block:
	<code>{ BlockStatements_{opt} }</code>
2	BlockStatements:
	BlockStatement
	BlockStatements BlockStatement
3	BlockStatement:

	Statement
4	Statement:
	StatementWithoutTrailingSubstatement

5	StatementWithoutTrailingSubstatement:
	Block

Tabelle 12.1: Auszug aus der Spezifikation der Syntax von Java

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

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

1. Ein $\langle \text{Block} \rangle$ hat die Struktur: Ein Zeichen $\{$, eventuell gefolgt von $\langle \text{BlockStatements} \rangle$ (das tiefgestellte Suffix „opt“ besagt, dass das optional ist), gefolgt von einem Zeichen $\}$.
2. $\langle \text{BlockStatements} \rangle$ sind von einer der Strukturen
 - ein einzelnes $\langle \text{BlockStatement} \rangle$ oder
 - $\langle \text{BlockStatements} \rangle$ gefolgt von einem $\langle \text{BlockStatement} \rangle$
3. Ein $\langle \text{BlockStatement} \rangle$ kann (unter anderem ...) ein $\langle \text{Statement} \rangle$ sein.
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 Mengenklammern benutzen wollen. (Sie fragen sich, warum wir 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:

$$L = \{\varepsilon\} \cup LL \cup \{(X)L\} \quad (12.1)$$

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

Die entscheidenden Fragen sind nun natürlich:

1. Gibt es überhaupt eine formale Sprache, die Gleichung 12.1 erfüllt?

Das hätten wir gerne, und wir werden sehen, dass es tatsächlich so ist, indem wir eine Lösung der Gleichung konstruieren werden.

2. Und falls ja: Ist die formale Sprache, die Gleichung 12.1 erfüllt, nur durch die Gleichung eindeutig festgelegt?

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

Vorher wollen wir eine Lösung von Gleichung 12.1 konstruieren. Wie könnte man das tun? Wir „tasten uns an eine Lösung heran“.

Das machen wir, indem wir

- erstens eine ganze Folge L_0, L_1, \dots formaler Sprachen L_i für $i \in \mathbb{N}_0$ definieren, der Art, dass jedes L_i offensichtlich jedenfalls einige der gewünschten Wörter über dem Alphabet $\{ (,) \}$ enthält, und
- zweitens dann zeigen, dass die Vereinigung aller dieser L_i eine Lösung von Gleichung 12.1 ist.

Also:

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

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

Wir behaupten, dass $L = \bigcup_{i=0}^{\infty} L_i$ Gleichung 12.1 erfüllt.

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

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

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

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

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

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

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

Damit haben wir gezeigt, dass Gleichung 12.1 (mindestens) eine Lösung hat.

Um zu 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 12.1. Ersetzt man dann noch überall die $w \in L_i$ durch „X“, ergibt sich Abbildung 12.2.

Das ist wieder die Darstellung eines sogenannten Baumes. (Im Abschnitt 8.4 über Huffman-Codierung hatten wir etwas Ähnliches auch schon einmal gesehen.) Auf Bäume und andere Graphen als Untersuchungsgegenstand werden wir in einem späteren Kapitel zu sprechen kommen.

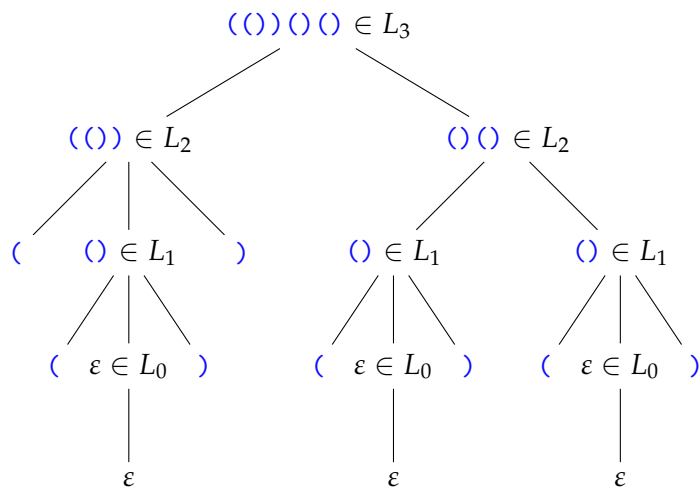


Abbildung 12.1: Eine „Begründung“, warum $()()() \in L_3$ ist.

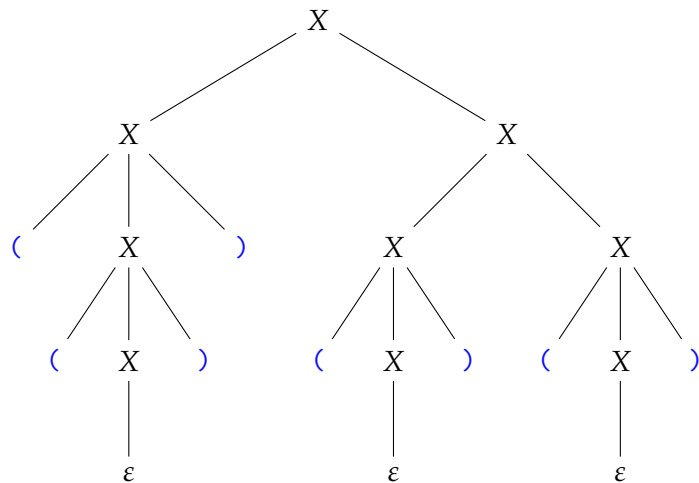


Abbildung 12.2: Eine abstraktere Darstellung der „Begründung“, warum $()()() \in L_3$ ist.

Zu kontextfreien Grammatiken ist es nun nur noch ein kleiner Schritt: Statt wie eben im Baum Verzweigungen von unten nach oben 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 .

12.2 KONTEXTFREIE GRAMMATIKEN

kontextfreie Grammatik

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

Nichtterminalsymbole

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

Terminalsymbole

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

Startsymbol

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

Produktionen

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

$X \rightarrow w$

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

Ableitungsschritt

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 $v = w_1 w w_2$.

$u \Rightarrow v$

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

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

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.

Infixschreibweise

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

Ableitung

Da die linke Seite einer Produktion immer ein Nichtterminalsymbol ist, wird bei einem Ableitungsschritt nie ein Terminalsymbol ersetzt. Wo sie stehen, ist „die Ableitung zu Ende“ (daher der Name *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 $u, v \in V^*$ und für jedes $i \in \mathbb{N}_0$ gelte

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

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*

*von einer Grammatik
erzeugte formale Sprache*

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

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

kontextfreie Sprache

Was ist bei unserer Beispielgrammatik $G = (\{X\}, \{a, b\}, X, \{X \rightarrow \varepsilon, X \rightarrow aXb\})$? 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 leichter, wenn man mit der allgemeineren Behauptung beginnt:

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

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

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

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 Produktionsmenge dann so aus:

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

Und damit können wir nun auch die richtige Interpretation für die Fragmente aus Tabelle 12.1 angeben: Es handelt sich um Produktionen einer kontextfreien Grammatik.

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

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

Und der erste Teil

1	Block:
	{ BlockStatements _{opt} }

der Tabelle repräsentiert die Produktionen

$$\langle Block \rangle \rightarrow \{ \langle BlockStatements \rangle \} \mid \{ \}$$

Wie man sieht, stehen (jedenfalls manche) Nichtterminalsymbole für strukturelle Konzepte der Programmiersprache. Im Idealfall wäre es dann so, dass man

mit der kontextfreien 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 kontextfreien 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 manchmal sehr mühsam aber nicht sonderlich erhellend. Die Grammatik für unser Klammerproblem ist ja sehr übersichtlich: $(\{X\}, \{ (,) \}, X, \{X \rightarrow XX \mid (X) \mid \varepsilon\})$. Aber

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

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 die Reihenfolge der Ableitungsschritte umsortieren und zum Beispiel immer möglichst weit links ableiten. 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 12.2. Machen Sie sich den Zusammenhang klar! (Wir verzichten bewusst darauf, Ableitungsbäumen zu formalisieren, weil es unseres Erachtens mehr verwirrt als erleuchtet.)

Ableitungsbaum

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

wohlgeformte oder korrekte Klammerausdrücke

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

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

Typ-2-Grammatiken

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

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

12.3 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 nicht schaden.

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

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

Produkt von Relationen

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

Mit I_M bezeichnen wir die Relation

$$I_M = \{(x, x) \mid x \in M\}$$

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

$$R \circ I_M = R = I_M \circ R$$

Potenzen einer Relation

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

R^i wie folgt:

$$R^0 = I_M$$
$$\text{für jedes } i \in \mathbb{N}_0 : R^{i+1} = R^i \circ R$$

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

reflexiv-transitive Hülle

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

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

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

Eine Relation heißt *reflexiv*, wenn $I_M \subseteq R$ ist.

reflexive Relation

Eine Relation heißt *transitiv*, wenn gilt:

transitive Relation

$$\text{für jedes } x \in M : \text{für jedes } y \in M : \text{für jedes } z \in M : xRy \wedge yRz \longrightarrow xRz$$

Bitte bringen Sie den logischen Implikationspfeil \longrightarrow in dieser Formel nicht mit dem Pfeil \rightarrow für Produktionen 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 $I_M = R^0 \subseteq R^*$ ist.

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

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

12.4 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.

13 PRÄDIKATENLOGIK

In Kapitel 5 haben wir Syntax und Semantik aussagenlogischer Formeln kennengelernt. Nun widmen wir uns der sogenannten *Prädikatenlogik erster Stufe*. Dabei wird auch Aussagenlogik wieder eine Rolle spielen.

13.1 SYNTAX PRÄDIKATENLOGISCHER FORMELN

Prädikatenlogische Formeln sind komplizierter aufgebaut als aussagenlogische. Man geht drei Schritten vor:

- Zunächst definiert man sogenannte *Terme*, die aus Konstanten, Variablen und Funktionssymbolen zusammengesetzt werden.
- Mit Hilfe von Relationssymbolen und Termen konstruiert man dann *atomare Formeln*.
- Aus ihnen werden mittels der schon bekannten aussagenlogischen Konnektive und zweier sogenannter Quantoren allgemeine *prädikatenlogische Formeln* gebildet.

Für die Definition von Termen sind drei Alphabete gegeben:

- ein Alphabet $Const_{PL}$ sogenannter *Konstantensymbole*, notiert als c_i (für endliche viele $i \in \mathbb{N}_0$) oder kurz als c, d . *Konstantensymbol*
- ein Alphabet Var_{PL} sogenannter *Variablensymbole*, notiert als x_i (für endliche viele $i \in \mathbb{N}_0$) oder kurz als x, y, z . *Variablensymbol*
- ein Alphabet Fun_{PL} sogenannter *Funktionsymbole*, notiert als f_i (für endliche viele $i \in \mathbb{N}_0$) oder kurz als f, g, h . Für jedes $f_i \in Fun_{PL}$ bezeichne $ar(f_i) \in \mathbb{N}_+$ die *Stelligkeit* (oder *Arität*) des Funktionssymbols. *Funktionsymbol*
Stelligkeit
Arität

Das Alphabet A_{Ter} der Symbole, aus denen Terme zusammengesetzt sind, umfasst neben den oben genannten außerdem Symbole „Klammer auf“, „Komma“ und „Klammer zu“:

$$A_{Ter} = \{ (, , ,) \} \cup Const_{PL} \cup Var_{PL} \cup Fun_{PL} .$$

Der syntaktische Aufbau von Termen wird mit Hilfe einer kontextfreien Grammatik $(N_{Ter}, A_{Ter}, T, P_{Ter})$ definiert. Dazu sei nun m die größte Stelligkeit, die ein Funktionssymbol in Fun_{PL} oder ein Relationssymbol in Rel_{PL} (siehe nächste Seite) hat. Als $m + 1$ Nichtterminalsymbole benutzen wir

$$N_{Ter} = \{ X_T \} \cup \{ X_{L,i} \mid i \in \mathbb{N}_+ \text{ und } i \leq m \} .$$

Die folgenden Produktionen erzeugen dann zusammen die Terme:

$$\begin{aligned}
 X_{L,i+1} &\rightarrow X_{L,i}, X_T && \text{für jedes } i \in \mathbb{N}_+ \text{ mit } i < m \\
 X_{L,1} &\rightarrow X_T \\
 X_T &\rightarrow c_i && \text{für jedes } c_i \in \text{Const}_{PL} \\
 X_T &\rightarrow x_i && \text{für jedes } x_i \in \text{Var}_{PL} \\
 X_T &\rightarrow f_i(X_{L,\text{ar}(f_i)}) && \text{für jedes } f_i \in \text{Fun}_{PL}
 \end{aligned}$$

Aus X_T kann man alle Terme ableiten und aus jedem $X_{L,i}$ Listen mit genau i durch Kommata getrennten Termen. Wir bezeichnen die Menge aller Terme auch mit L_{Ter} .

Grundterm Ein Term heißt *Grundterm*, wenn in ihm keine Variablensymbole vorkommen, wenn also bei seiner Ableitung niemals eine der Produktionen $X_T \rightarrow x_i$ benutzt wird.

Wenn f ein zweistelliges und g ein einstelliges Funktionssymbol sind, dann sind zum Beispiel die folgenden Ausdrücke Terme:

• c • y • $g(x)$ • $f(x, g(z))$ • $f(c, g(g(z)))$
 Syntaktisch falsch sind dagegen
 • $,,xy$ • $c(x)$ • $f)x, y($ • $g(c, c, c, x)$ • $g(f)$
 und so weiter.

atomare Formel *Atomare Formeln* werden aus Relationssymbolen und Termen zusammengesetzt. Dazu wird ein Alphabet Rel_{PL} sogenannter *Relationsymbole* festgelegt.

- Es enthält zum einen Symbole, die wir in der Form R_i notieren (für endliche viele $i \in \mathbb{N}_0$) oder kurz als R, S . Für jedes $R_i \in Rel_{PL}$ bezeichne $\text{ar}(R_i) \in \mathbb{N}_+$ die *Stelligkeit* (oder *Arität*) des Relationssymbols.
- Außerdem enthalte Rel_{PL} immer ein Symbol, für das wir \doteq schreiben, also ein „Gleichheitszeichen mit einem Punkt darüber“. Wie man an der Grammatik weiter unten sehen wird, ist \doteq ein zweistelliges Relationssymbol, das immer infix notiert wird. (Diese Notation ist aus dem Skript zur Vorlesung „Formale Systeme“ übernommen.)

Stelligkeit
Arität

Das Alphabet A_{Rel} der Symbole, aus denen atomare Formeln zusammengesetzt sind, umfasst neben den Symbolen für Terme zusätzlich nur die Relationssymbole:

$$A_{Rel} = A_{Ter} \cup Rel_{PL} .$$

Für die kontextfreie Grammatik $(N_{Rel}, A_{Rel}, X_A, P_{Rel})$, die atomare Formeln erzeugt, sei wieder m die größte Stelligkeit, die ein Funktions- oder ein Relationssymbol in Fun_{PL} bzw. Rel_{PL} haben. Dann ist

$$N_{Rel} = \{ X_R \} \cup N_{Ter} .$$

Die Produktionsmenge für atomare Formeln erweitert die für Terme wie folgt:

$$P_{Rel} = P_{Ter} \cup \{X_A \rightarrow \mathbf{R}_i(X_{L, \text{ar}(\mathbf{R}_i)}) \mid \text{für jedes } \mathbf{R}_i \in Rel_{PL}\} \\ \cup \{X_A \rightarrow X_T \doteq X_T\}.$$

Die aus X_A ableitbaren Wörter sind die atomaren Formeln. Die Menge aller atomaren Formeln bezeichnen wir mit L_{Rel} .

Wenn **f** ein zweistelliges und **g** ein einstelliges Funktionssymbol sind, **R** ein dreistelliges und **S** ein einstelliges Relationssymbol, dann sind zum Beispiel die folgenden Wörter atomare Formeln:

- $S(c)$
- $R(y, c, g(x))$
- $g(x) \doteq f(x, g(z))$

Syntaktisch falsch sind dagegen

- $x \doteq y \doteq z$
- $R(x, y)$
- $(S(S))(x)$
- $x \rightarrow R(z)$
- $R \doteq R$
- $f(S(x))$
- $R)x, y($
- $x \vee y$
- $S(x) \doteq S(x)$
- $R(S(x), x, x)$
- $gRf()$

und so weiter.

Das Alphabet A_{For} der Symbole, aus denen prädikatenlogische Formeln zusammengesetzt sind, umfasst neben den Symbolen für atomare Formeln zusätzlich die aussagenlogischen Konnektive und den sogenannten *Allquantor* \forall und den sogenannten *Existenzquantor* \exists :

Allquantor
Existenzquantor

$$A_{For} = A_{Rel} \cup \{ \neg, \wedge, \vee, \rightarrow, \forall, \exists \}.$$

Für die kontextfreie Grammatik $(N_{For}, A_{For}, X_F, P_{For})$ ist

$$N_{For} = \{ X_F \} \cup N_{Rel}$$

Die Produktionsmenge für prädikatenlogische Formeln erweitert die für atomare Formeln wie folgt:

$$\begin{aligned}
P_{For} = P_{Rel} \cup & \{ X_F \rightarrow X_A \} \\
& \cup \{ X_F \rightarrow (\neg X_F), X_F \rightarrow (X_F \wedge X_F), X_F \rightarrow (X_F \vee X_F), X_F \rightarrow (X_F \rightarrow X_F) \} \\
& \cup \{ X_F \rightarrow (\forall \mathbf{x}_i X_F) \mid \mathbf{x}_i \in Var_{PL} \} \\
& \cup \{ X_F \rightarrow (\exists \mathbf{x}_i X_F) \mid \mathbf{x}_i \in Var_{PL} \}.
\end{aligned}$$

Die aus X_F ableitbaren Wörter sind die prädikatenlogischen Formeln. Die Menge aller prädikatenlogischen Formeln bezeichnen wir mit L_{For} .

Wie bei aussagenlogischen Formeln stehe für alle Formeln G und H das Wort $(G \leftrightarrow H)$ für $((G \rightarrow H) \wedge (H \rightarrow G))$.

Und bei größeren Formeln verliert man wegen der vielen Klammern leicht den Überblick. Deswegen erlauben wir ganz analog zum Fall aussagenlogischer Formeln folgende Klammereinsparungsregeln. (Ihre „offizielle“ Syntax bleibt die gleiche!)

- Die äußerten umschließenden Klammern darf man immer weglassen. Zum Beispiel ist $P \rightarrow Q$ die Kurzform von $(P \rightarrow Q)$.
- Wenn ohne jede Klammern zwischen mehrere Aussagevariablen immer das gleiche Konnektiv steht, dann bedeute das „implizite Linksklammerung“. Zum Beispiel ist $P \wedge Q \wedge R$ die Kurzform von $((P \wedge Q) \wedge R)$.
- Wenn ohne jede Klammern zwischen mehrere Aussagevariablen verschiedene Konnektive oder/und Quantoren stehen, dann ist von folgenden „Bindungsstärken“ auszugehen:
 - a) \forall und \exists binden am stärksten
 - b) \neg bindet am zweitstärksten
 - c) \wedge bindet am drittstärksten
 - d) \vee bindet am viertstärksten
 - e) \rightarrow bindet am fünftstärksten
 - f) \leftrightarrow bindet am schwächsten

Zum Beispiel ist $\forall x R(x, y) \wedge S(x)$ die Kurzform von $(\forall x R(x, y)) \wedge S(x)$.

13.2 SEMANTIK PRÄDIKATENENLOGISCHER FORMELN

Es seien Alphabete $Const_{PL}$, Fun_{PL} und Rel_{PL} gegeben. Eine dazu passende *Interpretation* (D, I) ist durch folgende Bestandteile festgelegt:

Interpretation
Universum

- eine nichtleere Menge D , das sogenannte *Universum* der Interpretation,
- für jedes $c_i \in Const_{PL}$ ein Wert $I(c_i) \in D$,
- für jedes $f_i \in Fun_{PL}$ eine Abbildung $I(f_i) : D^{ar(f_i)} \rightarrow D$ und
- für jedes $R_i \in Rel_{PL}$ eine Relation $I(R_i) \subseteq D^{ar(R_i)}$.

Variablenbelegung

Zu jedem Alphabet Var_{PL} und einer Interpretation (D, I) ist eine *Variablenbelegung* eine Abbildung $\beta : Var_{PL} \rightarrow D$.

Sind eine Interpretation (D, I) und eine Variablenbelegung β festgelegt, so kann man

- jedem Term einen Wert aus D und
- jeder Formel einen Wahrheitswert zuordnen.

Die entsprechende Abbildung $val_{D,I,\beta} : L_{Ter} \cup L_{For} \rightarrow D \cup \mathbb{B}$ definieren wir schrittweise induktiv zunächst für Terme und anschließend für Formeln wie folgt.

Für jeden Term $t \in L_{Ter}$ und alle Terme $t_1, \dots, t_k \in L_{Ter}$ sei

$$val_{D,I,\beta}(t) = \begin{cases} \beta(\mathbf{x}_i), & \text{falls } t = \mathbf{x}_i \in Var_{PL} \\ I(\mathbf{c}_i), & \text{falls } t = \mathbf{c}_i \in Const_{PL} \\ I(\mathbf{f}_i)(val_{D,I,\beta}(t_1), \dots, val_{D,I,\beta}(t_k)), & \text{falls } t = \mathbf{f}_i(t_1, \dots, t_k) \end{cases}$$

Wie man sieht, ist für jeden Term t der Funktionswert $val_{D,I,\beta}(t)$ definiert und ein Element von D . Wir haben uns an dieser Stellen erlaubt, Pünktchen-Notation zu verwenden. Das hätte man durch zusätzlichen Aufwand vermeiden können, der das Ganze aber schlechter lesbar gemacht hätte. Deshalb machen wir diese Ausnahme hier ... und gleich noch einmal.

Jede atomare Formel ist entweder von der Form $\mathbf{R}_i(t_1, \dots, t_k)$ für Terme $(t_1, \dots, t_k) \in L_{Ter}^{ar(\mathbf{R}_i)}$ oder von der Form $t_1 \doteq t_2$ für Terme $(t_1, t_2) \in L_{Ter}^2$. Für sie wird festgelegt:

$$val_{D,I,\beta}(\mathbf{R}_i(t_1, \dots, t_k)) = \begin{cases} \mathbf{w}, & \text{falls } (val_{D,I,\beta}(t_1), \dots, val_{D,I,\beta}(t_k)) \in I(\mathbf{R}_i) \\ \mathbf{f}, & \text{falls } (val_{D,I,\beta}(t_1), \dots, val_{D,I,\beta}(t_k)) \notin I(\mathbf{R}_i) \end{cases}$$

$$val_{D,I,\beta}(t_1 \doteq t_2) = \begin{cases} \mathbf{w}, & \text{falls } val_{D,I,\beta}(t_1) = val_{D,I,\beta}(t_2) \\ \mathbf{f}, & \text{falls } val_{D,I,\beta}(t_1) \neq val_{D,I,\beta}(t_2) \end{cases}$$

Damit muss nur noch für nicht-atomare Formeln F definiert werden, was $val_{D,I,\beta}(F)$ sein soll. Falls F von einer der Formen ist, die wir schon aus der Aussagenlogik kennen, sei $val_{D,I,\beta}(F)$ dementsprechend definiert. Zum Beispiel sei für alle prädikatenlogischen Formeln H_1 und H_2 festgelegt: $val_{D,I,\beta}(H_1 \wedge H_2) = \mathbf{w} \wedge (val_{D,I,\beta}(H_1), val_{D,I,\beta}(H_2))$.

Damit bleibt nur noch, für jede Formel F zu definieren, was $val_{D,I,\beta}(\forall \mathbf{x}_i F)$ sein soll. Dafür führen wir noch folgende Hilfsdefinition ein. Für jede Variablenbelegung $\beta : Var_{PL} \rightarrow D$, jedes $\mathbf{x}_i \in Var_{PL}$ und jedes $d \in D$ sei

$$\beta_{\mathbf{x}_i}^d : Var_{PL} \rightarrow D : \mathbf{x}_j \mapsto \begin{cases} \beta(\mathbf{x}_j) & \text{falls } j \neq i \\ d & \text{falls } j = i \end{cases}$$

diejenige Variablenbelegung, die mit β für alle Variablen ungleich \mathbf{x}_i übereinstimmt, und für \mathbf{x}_i den Wert d vorschreibt.

Damit ist dann

$$val_{D,I,\beta}(\forall \mathbf{x}_i F) = \begin{cases} \mathbf{w}, & \text{falls für jedes } d \in D \text{ und } \beta' = \beta_{\mathbf{x}_i}^d \text{ gilt: } val_{D,I,\beta'}(F) = \mathbf{w} \\ \mathbf{f}, & \text{sonst} \end{cases}$$

Man kann sich überlegen, dass dann auch gilt:

$$val_{D,I,\beta}(\exists \mathbf{x}_i F) = \begin{cases} \mathbf{w}, & \text{falls für mind. ein } d \in D \text{ und } \beta' = \beta_{\mathbf{x}_i}^d \text{ gilt: } val_{D,I,\beta'}(F) = \mathbf{w} \\ \mathbf{f}, & \text{sonst} \end{cases}$$

allgemeingültige
Formel

Eine prädikatenlogische Formel F heißt *allgemeingültig*, wenn für jede passende Interpretation (D, I) und jede passende Variablenbelegung β gilt: $val_{D,I,\beta}(F) = \mathbf{w}$.

Eine einfache Methode, um zu allgemeingültigen Formeln zu gelangen, besteht darin, eine aussagenlogische Tautologie G zu nehmen und für jede in ihr vorkommende Aussagevariable P_i eine beliebige prädikatenlogische Formel G_i und dann in G alle Vorkommen von P_i syntaktisch durch G_i zu ersetzen. Wir wollen so entstehende Formeln als prädikatenlogische Tautologien bezeichnen.

Aber es gibt noch andere allgemeingültige prädikatenlogische Formeln. Ein einfaches Beispiel ist die Formel

$$(t_1 \doteq t_2) \rightarrow (t_2 \doteq t_1),$$

andere sind z. B. von der Form

$$(\forall x_i (G \rightarrow H)) \rightarrow ((\forall x_i G) \rightarrow (\forall x_i H))$$

für beliebige prädikatenlogische Formeln G und H .

Modell einer Formel

Ist (D, I) eine Interpretation für eine prädikatenlogische Formel G , dann nennen wir (D, I) ein *Modell* von G , wenn für jede Variablenbelegung β gilt, dass $val_{D,I,\beta}(G) = \mathbf{w}$ ist. Ist (D, I) eine Interpretation für eine Menge Γ prädikatenlogischer Formeln, dann nennen wir (D, I) ein *Modell* von Γ , wenn (D, I) Modell jeder Formel $G \in \Gamma$ ist.

Modell einer Formelmenge

Ist Γ eine Menge prädikatenlogischer Formeln und G ebenfalls eine, so schreibt man auch genau dann $\Gamma \models G$, wenn jedes Modell von Γ auch Modell von G ist. Enthält $\Gamma = \{H\}$ nur eine einzige Formel, schreibt man einfach $H \models G$. Ist $\Gamma = \{\}$ die leere Menge, schreibt man einfach $\models G$. Die Bedeutung soll in diesem Fall sein, dass G für *alle* Interpretationen überhaupt wahr ist, d. h. dass G allgemeingültig ist.

Als Beispiel betrachten wir die Formel G

$$\forall x \, f(x, c) \doteq x$$

und mehrere Interpretationen:

- Es sei $D = \mathbb{N}_0$, $I(c) = 0$ und $I(f)$ die Addition von Zahlen. Dann ist (D, I) ein Modell der Formel, denn anschaulich bedeutet G dann: Für jede nichtnegative ganze Zahl x ist $x + 0 = x$.

Genauer muss man sich fragen, ob für jedes β gilt: $val_{D,I,\beta}(G) = \mathbf{w}$. Dazu muss man für jedes $d \in \mathbb{N}_0$ und $\beta' = \beta_x^d$ prüfen, ob $val_{D,I,\beta'}(f(x, c) \doteq x) = \mathbf{w}$ ist. Das ist genau dann der Fall, falls $val_{D,I,\beta'}(f(x, c)) = val_{D,I,\beta'}(x)$ ist.

Die linke Seite ist $I(f)(\beta'(x), I(c)) = \beta'(x) + 0 = \beta'(x)$, was gerade gleich der rechten Seite ist.

- Ein anderes Modell von G erhält man, wenn $D = \{a, b\}^*$, $I(c) = \varepsilon$ und $I(f)$ die Konkatination von Wörtern ist.
- Ist dagegen $D = \mathbb{N}_0$, $I(c) = 0$ und $I(f)$ die Multiplikation von Zahlen, dann liegt kein Modell für G vor, denn über den nichtnegativen ganzen Zahlen ist *nicht* stets $x \cdot 0 = x$.

Betrachtet man andererseits die Formel $\forall x \forall y f(x, y) \doteq f(y, x)$, dann ist die erste der obigen Interpretationen wieder ein Modell (weil die Addition von Zahlen kommutativ ist), die zweite Interpretation ist aber kein Modell (weil die Konkatination von Wörtern nicht kommutativ ist).

13.3 FREIE UND GEBUNDENE VARIABLENVORKOMMEN UND SUBSTITUTIONEN

Dieser Abschnitt ist sehr technisch. Wofür das alles nötig ist, und dass sich der Aufwand tatsächlich lohnt, werden wir in späteren Abschnitten dieses Kapitels und auch in weiteren Kapiteln der Vorlesung sehen. Bis dahin müssen Sie sich beim Lesen etwas gedulden.

Wenn in einer prädikatenlogischen Formel G in einem Term eine Variable x steht, dann spricht man auch von einem *Vorkommen* der Variablen x in G . (Die Anwesenheit einer Variablen unmittelbar hinter einem Quantor zählt *nicht* als Vorkommen.)

Vorkommen einer Variablen

Die gleiche Variable kann natürlich an mehreren Stellen in G vorkommen. Genauer unterscheidet man sogenannte *freie* und *gebundene* Vorkommen von Variablen in G . Außerdem definiert man die Menge $fv(G)$ der frei in G vorkommenden Variablen und die Menge $bv(G)$ der gebunden in G vorkommenden Variablen. Diese Konzepte sind wie folgt definiert.

freies Vorkommen
gebundenes Vorkommen

Für jede Formel G , die atomar ist, sind alle Vorkommen von Variablen frei und es ist $bv(G) = \{\}$ und $fv(G)$ die Menge aller in G an mindestens einer Stelle vorkommenden Variablen.

Für jede Formel G der Form $\neg H$ ist $bv(G) = bv(H)$ und $fv(G) = fv(H)$ und jedes freie bzw. gebundene Vorkommen einer Variablen in H ist auch ein freies bzw. gebundenes Vorkommen in G .

Für jede Formel G , die eine der Formen $H_1 \wedge H_2$, $H_1 \vee H_2$ oder $H_1 \rightarrow H_2$ hat, ist $bv(G) = bv(H_1) \cup bv(H_2)$ und $fv(G) = fv(H_1) \cup fv(H_2)$ und jedes freie bzw. gebundene Vorkommen einer Variablen in H_1 oder H_2 ist auch ein freies bzw. gebundenes Vorkommen in G .

Für jede Formel G der Form $(\forall x_i H)$ oder $(\exists x_i H)$ ist $fv(G) = fv(H) \setminus \{x_i\}$ und $bv(G) = bv(H) \cup (\{x_i\} \cap fv(H))$, d.h. expliziter ausgedrückt

$$bv(G) = \begin{cases} bv(H) \cup \{x_i\} & \text{falls } x_i \in fv(H) \\ bv(H) & \text{sonst} \end{cases}$$

Wirkungsbereich
eines Quantors

Außerdem sind in G alle Vorkommen der Variablen x_i gebunden und man sagt auch, dass sich die in H freien Vorkommen von x_i im Wirkungsbereich des Quantors davor befinden und durch ihn gebunden werden. Alle Vorkommen anderer Variablen in G sind frei bzw. gebunden, je nachdem ob es in H freie oder gebundene Vorkommen sind.

geschlossene Formel

Eine Formel G heiße *geschlossen*, wenn $fv(G) = \{\}$ ist.

Als Beispiel betrachten wir die Formel G

$$\forall x (R(x, y) \wedge \exists y R(x, y))$$

- Das sechste Zeichen ist das erste Vorkommen von x ; es ist ein gebundenes Vorkommen, denn dieses x wird durch den Allquantor am Anfang der Formel gebunden.
- Das fünfzehnte Zeichen ist das zweite Vorkommen von x ; es ist ein gebundenes Vorkommen, denn dieses x wird ebenfalls durch den Allquantor am Anfang der Formel gebunden.
- Das achte Zeichen ist das erste Vorkommen von y ; es ist ein freies Vorkommen.
- Das siebzehnte Zeichen ist das zweite Vorkommen von y ; es ist ein gebundenes Vorkommen, dieses y wird durch den Existenzquantor nach dem \wedge gebunden.
- Es ist also $bv(G) = \{x, y\}$ und $fv(G) = \{y\}$. Wie man sieht, müssen die Menge der frei und die Menge der gebunden vorkommenden Variablen nicht disjunkt sein.
- Die Formel ist nicht geschlossen (denn y kommt frei darin vor).

Substitution

Eine *Substitution* ist eine Abbildung $\sigma : Var_{PL} \rightarrow L_{Ter}$. Sind die k Variablen x_{i_j} mit $1 \leq j \leq k$ die einzigen Variablen mit $\sigma(x_{i_j}) \neq x_{i_j}$, dann ist σ durch die Menge S der Paare $\{x_{i_j}/\sigma(x_{i_j}) \mid 1 \leq j \leq k\}$ eindeutig bestimmt. (Es ist üblich, die Paare in der Form $x_{i_j}/\sigma(x_{i_j})$ und nicht in der Form $(x_{i_j}, \sigma(x_{i_j}))$ zu notieren.) Wir notieren σ dann auch in der Form σ_S , wobei S stets rechtseindeutig ist, da σ eine Abbildung ist. Zum Beispiel bedeutet $\sigma_{\{x/c, y/f(x)\}}$ die Abbildung mit

$$\sigma(x) = c$$

$$\sigma(y) = f(x)$$

$$\sigma(z) = z \text{ für jedes } z \notin \{x, y\}$$

Man erweitert σ_S zu einer Abbildung $\sigma'_S : L_{Ter} \rightarrow L_{Ter}$, indem man die in einem Term vorkommenden Variablen „alle gleichzeitig“ gemäß S ersetzt:

$$\sigma'_S(t) = \begin{cases} \sigma_S(x), & \text{falls } t = x \text{ mit } x \in Var_{PL} \\ c, & \text{falls } t = c \text{ mit } c \in Const_{PL} \\ f(\sigma'_S(t_1), \dots, \sigma'_S(t_k)) & \text{falls } t = f(t_1, \dots, t_k) \text{ mit } f \in Fun_{PL} \text{ und } t_1, \dots, t_k \in L_{Ter} \end{cases}$$

Statt σ'_S schreibt man üblicherweise wieder einfach σ_S .

Zum Beispiel ist

$$\begin{aligned} \sigma_{\{x/c, y/f(x)\}}(x) &= c \\ \sigma_{\{x/c, y/f(x)\}}(y) &= f(x) \\ \sigma_{\{x/c, y/f(x)\}}(g(y, x)) &= g(f(x), c) \\ \sigma_{\{x/c, y/f(x)\}}(f(z, z)) &= f(z, z) \end{aligned}$$

Eine Substitution $\sigma_S : L_{Ter} \rightarrow L_{Ter}$ erweitert man schließlich in einem zweiten Schritt zu einer Abbildung $\sigma''_S : L_{For} \rightarrow L_{For}$ (für die man hinterher ebenfalls einfach wieder σ_S schreibt). Das macht man induktiv und bei Formeln ohne Quantoren auf naheliegende Art und Weise:

$$\sigma''_S(G) = \begin{cases} R(\sigma'_S(t_1), \dots, \sigma'_S(t_k)), & \text{falls } G = R(t_1, \dots, t_k) \text{ mit } R \in Rel_{PL} \text{ und } t_1, \dots, t_k \in L_{Ter} \\ \sigma'_S(t_1) \doteq \sigma'_S(t_2), & \text{falls } G = t_1 \doteq t_2 \text{ mit } t_1, t_2 \in L_{Ter} \\ \neg \sigma''_S(H), & \text{falls } G = \neg H \\ \sigma''_S(H_1) \wedge \sigma''_S(H_2), & \text{falls } G = H_1 \wedge H_2 \\ \sigma''_S(H_1) \vee \sigma''_S(H_2), & \text{falls } G = H_1 \vee H_2 \\ \sigma''_S(H_1) \rightarrow \sigma''_S(H_2), & \text{falls } G = H_1 \rightarrow H_2 \end{cases}$$

Für den verbleibenden Fall einer quantifizierten Formel definieren wir vorbereitend zu jeder Substitution σ_S und jedem $x \in Var_{PL}$ eine Substitution σ_{S-x} vermöge der Festlegung:

$$\begin{aligned} \sigma_{S-x}(x) &= x \\ \sigma_{S-x}(y) &= \sigma_S(y) \text{ für jedes } y \in Var_{PL} \text{ mit } y \neq x \end{aligned}$$

Damit können wir nun noch definieren:

$$\begin{aligned} \sigma''_S(\forall x H) &= \forall x \sigma''_{S-x}(H) \\ \sigma''_S(\exists x H) &= \exists x \sigma''_{S-x}(H) \end{aligned}$$

Das bedeutet nichts anderes als dass bei einer Substitution gebundene Vorkommen von Variablen nicht ersetzt werden. Wir betrachten dazu als Beispiel die Formel $G = S(x) \wedge \forall x R(x, y)$ und eine beliebige Substitution σ_S . Dann erhält man Schritt für Schritt zunächst einmal

$$\begin{aligned}\sigma_S(G) &= \sigma_S(S(x) \wedge \forall x R(x, y)) \\ &= \sigma_S(S(x)) \wedge \sigma_S(\forall x R(x, y)) \\ &= S(\sigma_S(x)) \wedge \forall x \sigma_{S-x}(R(x, y)) \\ &= S(\sigma_S(x)) \wedge \forall x R(\sigma_{S-x}(x), \sigma_{S-x}(y))\end{aligned}$$

Sei nun konkret die Substitution $\sigma_S = \sigma_{\{x/c, y/f(x)\}}$. Dann ist σ_{S-x} nichts anderes als $\sigma_{\{y/f(x)\}}$. Folglich ergibt sich in obigem Beispiel weiter:

$$\begin{aligned}\sigma_S(G) &= S(\sigma_S(x)) \wedge \forall x R(\sigma_{S-x}(x), \sigma_{S-x}(y)) \\ &= S(\sigma_{\{x/c, y/f(x)\}}(x)) \wedge \forall x R(\sigma_{\{y/f(x)\}}(x), \sigma_{\{y/f(x)\}}(y)) \\ &= S(c) \wedge \forall x R(x, f(x))\end{aligned}$$

In diesem Beispiel ist etwas passiert, was man bei Substitutionen häufig vermeiden will. Nach der Substitution war das zweite Argument von R durch den Allquantor gebunden, vor der Substitution aber noch nicht. Häufig möchte man nur über Substitutionen reden, bei denen das nicht passiert. Deshalb erhalten sie einen besonderen Namen.

*kollisionsfreie
Substitution*

Eine Substitution σ heie *kollisionsfrei* fr eine Formel G , wenn fr jede Variable x_i , die durch σ verndert wird (also $\sigma(x_i) \neq x_i$) und jede Stelle eines freien Vorkommens von x_i in G gilt: Diese Stelle liegt nicht im Wirkungsbereich eines Quantors $\forall x_j$ oder $\exists x_j$, wenn x_j eine Variable ist, die in $\sigma(x_i)$ vorkommt.

Zum Beispiel war in dem Beispiel weiter oben auf dieser Seite die Substitution $\sigma = \sigma_{\{y/f(x)\}}$ fr die Formel $\forall x R(x, y)$ *nicht* kollisionsfrei. Denn nach der Substitution befindet sich in der resultierenden Formel $\forall x R(x, f(x))$ das Vorkommen von x des Terms $f(x)$ im Wirkungsbereich des Quantors. Dagegen wre die Substitution $\sigma = \sigma_{\{y/f(z)\}}$ kollisionsfrei fr $\forall x R(x, y)$.

Zum Abschluss dieses Abschnitts geht es erst einmal darum, eine ganze Reihe allgemeingltiger Formeln kennenzulernen. Dazu verallgemeinern wir noch einen Begriff, der schon bei aussagenlogischen Formeln eine Rolle spielte. Zwei prdikatelogische Formeln G und H heien *logisch quivalent*, wenn fr jede passende Interpretation (D, I) und jede passende Variablenbelegung β gilt: $val_{D, I, \beta}(G) = val_{D, I, \beta}(H)$.

*logisch
quivalente
Formeln*

Man kann sich berlegen, dass zwei Formeln G und H genau dann logisch quivalent sind, wenn $\models G \leftrightarrow H$ gilt, wenn also $G \leftrightarrow H$ allgemeingltig ist.

Nachfolgend führen wir eine Reihe logisch äquivalenter Formelpaare auf, ohne auf Beweise für die logische Äquivalenz einzugehen. (Interessierte seien auf das Skript der Vorlesung „Formale Systeme“ verwiesen.) Es seien jeweils G und H beliebige prädikatenlogische Formeln.

1. $\neg \forall x_i G$ und $\exists x_i \neg G$
2. $\neg \exists x_i G$ und $\forall x_i \neg G$
3. $\forall x_i \forall x_j G$ und $\forall x_j \forall x_i G$
4. $\exists x_i \exists x_j G$ und $\exists x_j \exists x_i G$
5. $\forall x_i (G \wedge H)$ und $\forall x_i G \wedge \forall x_i H$
6. $\exists x_i (G \vee H)$ und $\exists x_i G \vee \exists x_i H$
7. wenn $x_j \notin \text{fv}(G)$ und $\sigma_{\{x_i/x_j\}}$ kollisionsfrei für G , dann sind äquivalent
 - $\forall x_i G$ und $\forall x_j \sigma_{\{x_i/x_j\}}(G)$
 - $\exists x_i G$ und $\exists x_j \sigma_{\{x_i/x_j\}}(G)$

Man spricht in diesem Fall von *gebundener Umbenennung* der Variablen.

gebundener Umbenennung

8. wenn $x_i \notin \text{fv}(G)$, dann sind äquivalent
 - $G \wedge \forall x_i H$ und $\forall x_i (G \wedge H)$
 - $G \wedge \exists x_i H$ und $\exists x_i (G \wedge H)$
 - $G \vee \forall x_i H$ und $\forall x_i (G \vee H)$
 - $G \vee \exists x_i H$ und $\exists x_i (G \vee H)$
 - $G \rightarrow \forall x_i H$ und $\forall x_i (G \rightarrow H)$
 - $G \rightarrow \exists x_i H$ und $\exists x_i (G \rightarrow H)$
9. wenn $x_i \notin \text{fv}(H)$, dann sind äquivalent
 - $\forall x_i G \rightarrow H$ und $\exists x_i (G \rightarrow H)$
 - $\exists x_i G \rightarrow H$ und $\forall x_i (G \rightarrow H)$

Weitere allgemeingültige Formeln, die aber nicht von der Form $G \leftrightarrow H$ sind, werden auch im nächsten Abschnitt benötigt. Es seien $G \in L_{\text{For}}$ und $x_i \in \text{Var}_{\text{PL}}$.

1. Wenn die Substitution $\sigma_{\{x_i/t\}}$ kollisionsfrei für G ist, dann ist
 - $(\forall x_i G) \rightarrow \sigma_{\{x_i/t\}}(G)$
 allgemeingültig.
2. Wenn $x_i \notin \text{fv}(G)$ ist, dann ist
 - $\forall x_i (G \rightarrow H) \rightarrow (G \rightarrow \forall x_i H)$
 allgemeingültig.
3. Für Variablen x_i, x_j, x_k sind die Formeln
 - $x_i \doteq x_i$
 - $x_i \doteq x_j \rightarrow x_j \doteq x_i$
 - $x_i \doteq x_j \rightarrow (x_j \doteq x_k \rightarrow x_i \doteq x_k)$
 allgemeingültig.

4. Wenn $x_1, \dots, x_k \in \text{Var}_{PL}$ und $y_1, \dots, y_k \in \text{Var}_{PL}$, f_i ein k -stelliges Funktionssymbol und R_i ein k -stelliges Relationssymbol, dann sind die Formeln
- $x_1 \doteq y_1 \rightarrow (x_2 \doteq y_2 \rightarrow (\dots (x_k \doteq y_k \rightarrow f_i(x_1, \dots, x_k) \doteq f_i(y_1, \dots, y_k)) \dots))$
 - $x_1 \doteq y_1 \rightarrow (x_2 \doteq y_2 \rightarrow (\dots (x_k \doteq y_k \rightarrow (R_i(x_1, \dots, x_k) \rightarrow R_i(y_1, \dots, y_k))) \dots))$
- allgemeingültig.

13.4 BEWEISBARKEIT

Dieser Abschnitt wurde im Wintersemester 2020/2021 ausgelassen. Die behandelten Themen sind nicht prüfungsrelevant. Interessenten können sich das Folgende natürlich trotzdem durchlesen und mit dem Vorgehen in Kapitel 5 zu Aussagenlogik, insbesondere in Abschnitt 5.5 vergleichen.

Der grundlegende Begriff im Zusammenhang mit Beweisbarkeit sowohl in der Aussagenlogik als auch in der Prädikatenlogik ist der des Kalküls. In Kapitel 5 hatten wir bereits die prinzipielle Struktur eines Kalküls anhand der Aussagenlogik kennengelernt:

- In der Menge aller syntaktisch korrekten Formeln wurde
- eine Menge von *Axiomen* ausgezeichnet, aus denen mithilfe
- von *Ableitungsregeln* in endliche vielen Schritten
- die *Theoreme* des Kalküls ableitbar sind.

Der wesentliche Punkt war dabei, dass man den Kalkül so konstruieren kann — und dann eben auch so konstruiert — dass die Menge der Theoreme mit der Menge der allgemeingültigen Formeln übereinstimmt. Man kann also im Kalkül genau die Formeln beweisen, die in jeder Interpretation wahr sind.

Dieses Programm werden wir nun für die Prädikatenlogik erster Stufe analog vorstellen. Dabei werden wir (wie in der Aussagenlogik so auch hier erst recht) darauf verzichten, den zentralen Satz zu beweisen. Das würde über den Rahmen dieser Grundlagenvorlesung hinausgehen.

Für die Prädikatenlogik gibt es, wie für die Aussagenlogik, ganz unterschiedliche Kalküle, die das gleiche leisten. Wir beschreiben nachfolgend eine Variante, die auf David Hilbert zurückgeht. Dazu sei

- A_{For} ein Zeichenvorrat (mit Variablen-, Konstanten-, Funktions- und Relationssymbolen) für prädikatenlogische Formeln
- L_{For} die Menge der syntaktisch korrekten prädikatenlogischen Formeln über dem Alphabet A_{For} ,
- $Ax_{PL} \subseteq L_{For}$ eine nachfolgend definierte Menge von Axiomen,
- und zwei Schlussregeln, nämlich neben dem schon bekannten Modus ponens noch „Generalisierung“.

Als Menge Ax_{PL} der Axiome für den Hilbert-Kalkül wählen wir die Vereinigung der folgenden Mengen von Formeln. Wir haben im vorangegangenen Abschnitt für die Formeln jeder Menge angemerkt, dass es sich jeweils um allgemeingültige prädikatenlogische Formeln handelt.

$$\begin{aligned}
Ax_{AL1} &= \{(G \rightarrow (H \rightarrow G)) \mid G, H \in L_{For}\} \\
Ax_{AL2} &= \{(G \rightarrow (H \rightarrow K)) \rightarrow ((G \rightarrow H) \rightarrow (G \rightarrow K)) \mid G, H, K \in L_{For}\} \\
Ax_{AL3} &= \{(\neg H \rightarrow \neg G) \rightarrow ((\neg H \rightarrow G) \rightarrow H) \mid G, H \in L_{For}\} \\
Ax_{PL1} &= \{(\forall x_i G) \rightarrow \sigma_{\{x_i/t\}}(G) \mid G \in L_{For}, x_i \in Var_{PL}, t \in L_{Ter} \text{ und } \sigma_{\{x_i/t\}} \text{ kollisionsfrei für } G\} \\
Ax_{PL2} &= \{(\forall x_i (G \rightarrow H)) \rightarrow (G \rightarrow \forall x_i H) \mid G, H \in L_{For}, x_i \in Var_{PL}, \text{ und } x_i \notin \text{fv}(G)\} \\
Ax_{Eq1} &= \{x_i \doteq x_i \mid x_i \in Var_{PL}\} \\
Ax_{Eq2} &= \{x_i \doteq x_j \rightarrow x_j \doteq x_i \mid x_i, x_j \in Var_{PL}\} \\
Ax_{Eq3} &= \{x_i \doteq x_j \rightarrow (x_j \doteq x_k \rightarrow x_i \doteq x_k) \mid x_i, x_j, x_k \in Var_{PL}\} \\
Ax_{Eq4} &= \{x_{i_1} \doteq x_{j_1} \rightarrow (x_{i_2} \doteq x_{j_2} \rightarrow (\dots (x_{i_n} \doteq x_{j_n} \rightarrow f_i(x_{i_1}, \dots, x_{i_n}) \doteq f_i(x_{j_1}, \dots, x_{j_n})) \dots)) \\
&\quad \mid x_{i_1}, \dots, x_{i_n}, x_{j_1}, \dots, x_{j_n} \in Var_{PL}, f_i \in Fun_{PL}\} \\
Ax_{Eq5} &= \{x_{i_1} \doteq x_{j_1} \rightarrow (x_{i_2} \doteq x_{j_2} \rightarrow (\dots (x_{i_n} \doteq x_{j_n} \rightarrow (R_i(x_{i_1}, \dots, x_{i_n}) \rightarrow R_i(x_{j_1}, \dots, x_{j_n}))) \dots)) \\
&\quad \mid x_{i_1}, \dots, x_{i_n}, x_{j_1}, \dots, x_{j_n} \in Var_{PL}, R_i \in Rel_{PL}\}
\end{aligned}$$

Im Fall der Prädikatenlogik gibt es zwei Schlussregeln. Die eine ist wieder Modus Ponens, aber dieses Mal natürlich für prädikatenlogische Formeln. Als Relation geschrieben ist $MP \subseteq L_{For}^3$ mit

$$MP = \{(G \rightarrow H, G, H) \mid G, H \in L_{For}\} \quad \text{bzw.} \quad \frac{G \rightarrow H \quad G}{H}$$

Die zweite Ableitungsregel heißt *Generalisierung*. $GEN \subseteq L_{For}^2$ ist so definiert:

Generalisierung

$$GEN = \{(G, (\forall x_i G)) \mid G \in L_{For} \text{ und } x_i \in Var_{PL}\} \quad \text{bzw.} \quad \frac{G}{(\forall x_i G)}$$

Man mache sich bitte klar, dass die Anwendung von sowohl Modus ponens als auch Generalisierung auf allgemeingültige Formeln wieder allgemeingültige Formeln liefert.

Die Formalisierung des Begriffs *Ableitung* oder auch *Beweis* erweitert die Vorgehensweise aus der Aussagenlogik um die Möglichkeit, in einem Schritt Generalisierung anzuwenden.

Ableitung

Beweis

Dazu sei Γ eine Formelmenge sogenannter *Hypothesen* oder *Prämissen* und G

Hypothese

Prämisse

Ableitung

eine Formel. Eine *Ableitung* von G aus Γ ist eine endliche Folge (G_1, \dots, G_n) von n Formeln mit der Eigenschaft, dass erstens $G_n = G$ ist und auf jede Formel G_i einer der folgenden Fälle zutrifft:

- Sie ist ein Axiom: $G_i \in Ax_{PL}$.
- Oder sie ist eine Prämisse: $G_i \in \Gamma$.
- Oder es gibt Indizes i_1 und i_2 echt kleiner i , für die gilt: $(G_{i_1}, G_{i_2}, G_i) \in MP$.
- Oder es gibt einen Index i_1 echt kleiner i , für den gilt: $(G_{i_1}, G_i) \in GEN$.

Beweis
Theorem

Wir schreiben dann $\Gamma \vdash G$. Ist $\Gamma = \{\}$, so heißt eine entsprechende Ableitung auch ein *Beweis* von G und G ein *Theorem* des Kalküls, in Zeichen: $\vdash G$.

Es ist nicht Aufgabe dieser Vorlesung in Hilberts Kalkül besonders viele oder besonders komplizierte Theoreme zu beweisen. Ein etwas ausführlicheres Beispiel soll aber klar machen, dass es der Kalkül erlaubt, Beweise im obigen Sinn zu führen, die „informell“ geführten Beweisen entsprechen. Das soll auch eine Andeutung sein der Tatsache, dass alle unsere Beweise, die wir ja immer nicht präzise in einem Kalkül führen, eine solide Grundlagen haben.

Zuvor seien noch zwei wichtige Theoreme aufgeführt, deren Beweise Sie an anderen Stellen in der Literatur oder z. B. in der Vorlesung „Formale Systeme“ finden können:

13.1 Theorem Für jede Formelmenge $\Gamma \subseteq L_{For}$ und jede Formel $G \in L_{For}$ gilt:

wenn $\Gamma \vdash G$, dann auch $\Gamma \models G$.

Korrektheit des
Hilbert-Kalküls

Man sagt auch, dass der Hilbert-Kalkül *korrekt* sei.

13.2 Theorem Für jede Formelmenge $\Gamma \subseteq L_{For}$ und jede Formel $G \in L_{For}$ gilt:

wenn $\Gamma \models G$, dann auch $\Gamma \vdash G$.

Vollständigkeit des
Hilbert-Kalküls

Man sagt auch, dass der Hilbert-Kalkül *vollständig* sei.

Außerdem sei noch eine Warnung ausgesprochen. Im Fall der Aussagenlogik hatten wir das Deduktionstheorem kennengelernt. Ohne an dieser Stelle auf Details einzugehen, sei erwähnt, dass man es *nicht* in voller Allgemeinheit auf die Prädikatenlogik übertragen kann. Wir erwähnen nur das folgende Analogon, das eine relativ starke Voraussetzung macht (es gibt schwächere hinreichende Voraussetzungen).

13.3 Theorem Für jede geschlossene Formel $G \in L_{For}$ und jedes $H \in L_{For}$ gilt $G \vdash H$ genau dann, wenn $\vdash (G \rightarrow H)$ gilt.

Zum Abschluss dieses Abschnitts betrachten wir nun beispielhaft einen Fall, in dem $Fun_{PL} = \{f\}$ und $Const_{PL} = \{c\}$. Außerdem sei $\Gamma = \{ \forall x(f(c, x) \doteq x \wedge f(x, c) \doteq x) \}$. Jedes Modell (D, I) von Γ besitzt also eine auf D definierte zweistellige Operation $I(f)$ und eine Konstante $I(c)$, die neutrales Element bezüglich der Operation ist.

Ein Modell erhält man z. B. durch die Festlegungen

- $D = \mathbb{N}_0$, $I(f) : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0 : (x, y) \mapsto x + y$ und $I(c) = 0$,

ein anderes durch

- $D = \{a, b\}^*$, $I(f) : \{a, b\}^* \times \{a, b\}^* \rightarrow \{a, b\}^* : (w_1, w_2) \mapsto w_1 \cdot w_2$ und $I(c) = \varepsilon$.

Wir wollen nun zeigen, dass in jedem solchen Modell das neutrale Element immer eindeutig ist, d. h. dass für jedes Modell von Γ die folgende Aussage G wahr ist:

$$\forall y (\forall x (f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow y \doteq c)$$

Den Nachweis wollen wir unter Ausnutzung der Korrektheit des Hilbert-Kalküls zeigen. Wir müssen also eine Ableitung von G aus den Axiomen und den Hypothesen in Γ finden.

Wir werden nicht einen genauen Beweis im Hilbert-Kalkül aufschreiben; das ist zu aufwändig. Aber wir werden Formulierungen, wie wir sie „normalerweise“ nutzen, so erweitern, dass interessierte Leser den Rest hoffentlich selbst erledigen können. Salopp gesprochen beruht die Aussage auf der Beobachtung, dass

$$y = f(c, y) = c,$$

sobald c linksneutrales Element ist und y rechtsneutrales Element. Genauer kann man in folgenden Schritten vorgehen:

Schritt 0: Man beginnt mit einem „beliebigen“ y und muss zeigen:

$$\forall x (f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow y \doteq c$$

Schritt 1: zeige: $y \doteq f(c, y)$

Schritt 1.1: wegen $\forall x (f(c, x) \doteq x \wedge f(x, c) \doteq x)$ gilt insbesondere

$$f(c, y) \doteq y \wedge f(y, c) \doteq y$$

Schritt 1.2: Also gilt $f(c, y) \doteq y$.

Schritt 1.3: Also gilt $y \doteq f(c, y)$.

Schritt 2: zeige: $\forall x (f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow y \doteq f(c, y)$

Schritt 3: zeige: $\forall x (f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow f(c, y) \doteq c$

Schritt 3.1: zeige $\forall x (f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow \forall x (f(y, x) \doteq x) \wedge \forall x (f(x, y) \doteq x)$

Schritt 3.2: zeige $\forall x (f(y, x) \doteq x) \wedge \forall x (f(x, y) \doteq x) \rightarrow \forall x (f(x, y) \doteq x)$

Schritt 3.3: also $\forall x (f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow \forall x (f(x, y) \doteq x)$

Schritt 3.4: es gilt $\forall x(f(x, y) \doteq x) \rightarrow f(c, y) \doteq c$

Schritt 3.5: also $\forall x(f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow f(c, y) \doteq c$

Schritt 4: zeige $\forall x(f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow y \doteq c$

Schritt 4.1 mit Ergebnissen von Schritt 2 und Schritt 3 zeige:

$$\forall x(f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow (y \doteq f(c, y) \wedge f(c, y) \doteq c)$$

Schritt 4.2: zeige $(y \doteq f(c, y) \wedge f(c, y) \doteq c) \rightarrow y \doteq c$

Schritt 4.3: also $\forall x(f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow y \doteq c$

Schritt 5: Aus dem Ergebnis von Schritt 4 folgt durch Generalisierung:

$$\forall y(\forall x(f(y, x) \doteq x \wedge f(x, y) \doteq x) \rightarrow y \doteq c)$$

ZUSAMMENFASSUNG UND AUSBLICK

In diesem Kapitel wurde zunächst die Syntax prädikatenlogischer Formeln festgelegt. Anschließend haben wir Interpretationen und Modelle eingeführt. Und am Ende haben wir gesehen, wie man dem semantischen Begriff der Allgemeingültigkeit den syntaktischen Begriff der Beweisbarkeit, z. B. im Hilbert-Kalkül, so gegenüberstellen kann, dass sich die beiden entsprechen.

In Zukunft werden wir das Öfteren Aussagen als prädikatenlogische Formeln notieren. Dabei werden wir es uns mitunter etwas bequem machen und folgende Abkürzungen nutzen:

- Wir werden $\forall P(x) : F$ schreiben statt $\forall x (P(x) \rightarrow F)$.
- Wir werden $\exists P(x) : F$ schreiben statt $\exists x (P(x) \wedge F)$

Die Festlegungen dieser beiden Abkürzungen sehen zwar „asymmetrisch“ aus, sind aber durchaus plausibel. Denken Sie darüber nach! Außerdem kann man dann die Tatsache, dass Formeln $\exists x : \neg F$ und $\neg \forall x F$ logisch äquivalent sind, auf den abgekürzten Fall übertragen: Auch die Formeln $\exists P(x) \neg F$ und $\neg \forall P(x) F$ sind logisch äquivalent.

Außerdem werden wir uns in späteren Kapiteln bei den Interpretationen einschränken, unter Umständen auf eine einzige, zum Beispiel $D = \mathbb{N}_0$. Welche gemeint ist, wird sich immer aus dem Kontext ergeben. Wir schreiben dann etwa direkt $+$ für das Funktionssymbol, das als Addition zu interpretieren ist, und man wird Formeln finden der Form $\forall x \geq 2 : x^2 \geq x + 2$.

14 DER BEGRIFF DES ALGORITHMUS

MUHAMMAD IBN MŪSĀ AL-KHWĀRIZMĪ lebte ungefähr von 780 bis 850. Abbildung 14.1 zeigt eine (relativ beliebte weil copyright-freie) Darstellung auf einer russischen Briefmarke anlässlich seines (jedenfalls ungefähr) 1200. Geburtstages. Im Jahr 830 (oder wenig früher) schrieb al-Khwārizmī ein wichtiges Buch mit



Abbildung 14.1: Muhammad ibn Mūsā al-Khwārizmī; Bildquelle: http://commons.wikimedia.org/wiki/Image:Abu_Abdullah_Muhammad_bin_Musa_al-Khwarizmi.jpg (24.11.2014)

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-Khwārizmī 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 14.2 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 Algorism über die indischen Zahlen“. Das „i“ am Ende von „Algorismi“ wurde später fälschlicherweise als Pluralendung des Wortes

*Herkunft des Wortes
Algorithmus*

Algorithmus angesehen.

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

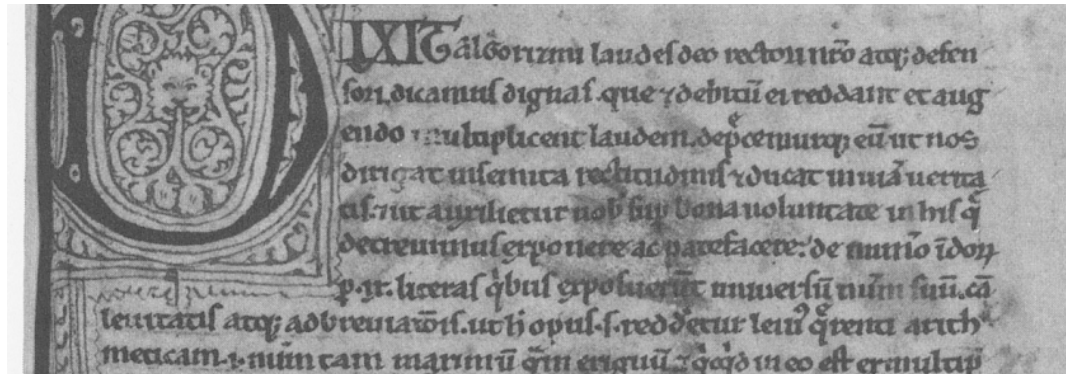


Abbildung 14.2: Ausschnitt einer Seite einer lateinischen Übersetzung der Arbeit von al-Khwārizmī über das „Rechnen mit indischen Ziffern“; Bildquelle: http://en.wikipedia.org/wiki/Image:Dixit_algorizmi.png (24.11.2014)

14.1 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:

$$h \leftarrow b/2 \quad (14.1)$$

$$q \leftarrow h^2 \quad (14.2)$$

$$s \leftarrow c + q \quad (14.3)$$

$$w \leftarrow \sqrt{s} \quad (14.4)$$

$$x \leftarrow w - h \quad (14.5)$$

(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.

14.2 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 manchmal die Fortsetzung nicht mehr eindeutig ist, sondern abhängig von Zufallsereignissen,
 - Verfahren, bei denen nicht von Anfang an alle Eingaben zur Verfügung stehen, sondern erst nach und nach (z. B. Cacheverwaltung in Prozessoren), und
 - Verfahren, die nicht terminieren (z. B. Ampelsteuerung).

14.3 INFORMELLE EINFÜHRUNG DES HOARE-KALKÜLS

Al-Khwarizmi gibt einen sehr schönen Beweis dafür an, dass die Rechnungen in (14.1)-(14.5) das gewünschte Ergebnis liefern. Er beruht auf einer recht einfachen geometrischen Überlegung (siehe z. B. <http://www-history.mcs.st-and.ac.uk/~history/Biographies/Al-Khwarizmi.html>, 24.11.2014).

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. Man nennt so etwas auch eine Zusicherung. Die Formeln sind zwischen geschweifte Klammern $\{$ und $\}$ gesetzt.

$$\{ b > 0 \wedge c > 0 \}$$

$$h \leftarrow b/2$$

$$\{ h = b/2 \}$$

$$q \leftarrow h^2$$

$$\{ q = b^2/4 \}$$

$$s \leftarrow c + q$$

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

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

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

$$x \leftarrow w - h$$

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

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

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

$$\{ x^2 + bx = c \}$$

Die oben benutzte Notation stammt von den sogenannten Hoare-Tripeln, die nach dem britischen Informatiker Charles Antony Richard Hoare benannt sind.

Bevor wir uns näher mit ihnen beschäftigen, wollen wir die Syntax einer „Mini-Programmiersprache“ skizzieren, die wir in unseren Beispielen benutzen werden. Als Schlüsselwörter werden wir **if**, **then**, **else**, **fi**, **while**, **do**, **od**, **for** und **to** be-

nutzen, und als Symbol für die Zuweisung \leftarrow . Einige gängige Funktionen und Relationen werden wir mit den üblichen Symbolen bezeichnen, die auch immer auf die gewohnte Art und Weise zu interpretieren sind, wie etwa das „+“ und das „ \leq “. Das gleiche gilt für Konstantensymbole wie etwa 0 oder 1.

Zu den Produktionen der kontextfreien Grammatik, die im wesentlichen die Syntax unserer Sprache festlegen soll, gehören unter anderem die folgenden:

$$\begin{aligned} \text{Prog} &\rightarrow \text{Stmt} \\ &\quad | \text{Stmt Prog} \\ \text{Stmt} &\rightarrow \text{Var} \leftarrow \text{Expr} \\ &\quad | \text{if Bool then Prog else Prog fi} \\ &\quad | \text{while Bool do Prog od} \\ &\quad | \text{for Var} \leftarrow \text{Expr to Expr do Prog od} \end{aligned}$$

Ein Programm soll also eine nichtleere Folge von Anweisungen (*statements*) sein. (Gelegentlich werden wir uns erlauben, zur Verdeutlichung Anweisungen durch Semikola zu trennen, obwohl die Grammatik das nicht erlaubt). Mögliche Anweisungen sind Zuweisung, **if**-Anweisung und **while**- und **for**-Schleifen. Produktionen für die Nichtterminalsymbole **Var**, **Expr** und **Bool** geben wir nicht an. Aus **Var** sollen Variablennamen der üblichen Art ableitbar sein. Aus **Expr** sollen „Ausdrücke“ (*expressions*) ableitbar sein; das sind die Gebilde, die wir in der Prädikatenlogik als Terme bezeichnet haben. Dabei sind die Programmvariablen auch prädikatenlogische Variablen. Und aus **Bool** sollen syntaktische Einheiten ableitbar sein, denen man einen Wahrheitswert zuordnen kann, also Gebilde, die wir in der Prädikatenlogik als Formeln bezeichnet haben.

Um die *Bedeutung* solcher Programme (vor allem von **while**-Schleifen) in der üblichen Weise zu definieren, fehlen uns an dieser Stelle noch zu viele Grundlagen. Wir beschränken uns daher auf einige Hinweise. In jeder Variablen ist zu jedem Zeitpunkt ein Wert aus dem zugrundeliegenden Universum D gespeichert. Welches das ist, werden wir in jedem Fall explizit sagen. Abhängig davon wird auch immer die Interpretation I aller im Programm auftretenden Konstanten-, Funktions- und Relationssymbole klar sein. Vor und nach der Ausführung jeder Zuweisung oder zusammengesetzten Anweisung liegt also ein Gesamtzustand des Speichers vor, der formal durch das beschrieben wird, was wir eine Variablenbelegung $\beta : \text{Var} \rightarrow D$ genannt haben. Der Zustand des Speichers kann sich nur durch die Ausführung einer Zuweisung $x \leftarrow E$ mit $E \in L_{Ter}$ ändern. Liegt eine Variablenbelegung β vor, dann soll sich durch Ausführung von $x \leftarrow E$ die

Variablenbelegung β' ergeben mit

$$\beta' = \beta_x^{\text{val}_{D,I,\beta}(E)}.$$

Alle Variablen ungleich x behalten also ihren Wert und der Wert von x nach der Zuweisung ist gerade der Wert, der sich bei Auswertung von E vor der Zuweisung ergibt.

Was bei der Ausführung einer bedingten Anweisung **if** B **then** S_1 **else** S_2 **fi** passiert, hängt vom Wahrheitswert der Formel B am Anfang ab. Ist $\text{val}_{D,I,\beta}(B) = \mathbf{w}$, dann „wird S_1 ausgeführt“, die Bedeutung ist also die von S_1 , und ist $\text{val}_{D,I,\beta}(B) = \mathbf{f}$, dann „wird S_2 ausgeführt“.

Bei einer Schleife **while** B **do** S **od** müssen wir am genauesten bleiben. Die Ausführung einer solchen Schleife bedeutet

- als erstes $\text{val}_{D,I,\beta}(B)$ zu bestimmen. Ergibt sich \mathbf{f} , dann ist die Ausführung der Schleife sofort beendet und „es passiert gar nichts“.
- Ist $\text{val}_{D,I,\beta}(B) = \mathbf{w}$, dann wird einmal S ausgeführt (was im allgemeinen zu einer Änderung des Speicherzustands führt) und anschließend wieder die gesamte Schleife.

Eine Schleife der Form **for** $x \leftarrow E_1$ **to** E_2 **do** S **od** wollen wir als Abkürzung auffassen für das Programmstück $x \leftarrow E_1$; **while** $x \leq E_2$ **do** S ; $x \leftarrow x + 1$ **od**.

Jede Ausführung einer Anweisung führt dazu, dass aus einer Variablenbelegung β vor der Ausführung eine Variablenbelegung β' nach der Ausführung wird. Bei der Ausführung zweier aufeinander folgender Anweisungen $S_1; S_2$ ist das sich nach Ausführung von S_1 ergebende β' die Variablenbelegung, von der für die Ausführung von S_2 ausgegangen wird.

Damit kommen wir nun zum eigentlichen Thema dieses Abschnitts. Ein *Hoare-Tripel* hat die Form $\{P\} S \{Q\}$ für eine Anweisung oder Anweisungsfolge S und zwei Aussagen P und Q , die *Zusicherungen* heißen. Man nennt P die *Vorbedingung* und Q die *Nachbedingung* des Hoare-Tripels. Zusicherungen machen Aussagen über Werte von Variablen und Ausdrücken und ihre Zusammenhänge. Im allgemeinen sind prädikatenlogische Formeln erlaubt. Damit das Ganze sinnvoll wird, gehen wir davon aus, dass man auf der rechten Seite von Zuweisungen gerade solche Ausdrücke hinschreiben darf, die auch in der Prädikatenlogik erlaubt sind.

Ein Hoare-Tripel $\{P\} S \{Q\}$ ist *gültig*, wenn für jede relevante Interpretation (siehe gleich) und jede Variablenbelegung β gilt:

Wenn vor der Ausführung $\text{val}_{D,I,\beta}(P) = \mathbf{w}$ ist und wenn die Ausführung von S für I und β endet und hinterher Variablenbelegung β' vorliegt, dann gilt am Ende $\text{val}_{D,I,\beta'}(Q) = \mathbf{w}$.

Hoare-Tripel
Zusicherungen
Vorbedingung
Nachbedingung

gültig

In den Zusicherungen werden wir auch Konstantensymbole verwenden, die *keine* offensichtliche Interpretation haben, z. B. ein c . Wir wollen vereinbaren, dass es für die Gültigkeit eines Hoare-Tripels notwendig ist, dass *für alle* Interpretationen $I(c)$ dieser Symbole die oben genannte Forderung erfüllt ist.

Die Beobachtung von Hoare war, dass man ein Kalkül (mit Axiomen und Ableitungsregeln analog zu dem, was wir bei der Aussagenlogik gesehen haben) angeben kann, in dem man gerade alle gültigen Hoare-Tripel ableiten kann. Man spricht naheliegenderweise vom *Hoare-Kalkül*.

Hoare-Kalkül

Die *Axiome* betreffen die Zuweisungen. Für jede Zuweisung $x \leftarrow E$ (mit $E \in L_{Ter}$) und jede Nachbedingung Q mit der Eigenschaft, dass die Substitution $\sigma_{\{x/E\}}$ kollisionsfrei für Q ist (siehe Abschnitt 13.3), ist $\{\sigma_{\{x/E\}}(Q)\} x \leftarrow E \{Q\}$ ein Axiom. Da man Axiome auch als Ableitungsregeln auffassen kann, die keine Voraussetzungen haben, schreibt man manchmal auch

Axiome für
Hoare-Kalkül

$$\text{HT-A: } \frac{}{\{\sigma_{\{x/E\}}(Q)\} x \leftarrow E \{Q\}}$$

Wir nennen diese „Regel“ HT-A, um mit dem A an „Axiom“ (oder *assignment*) zu erinnern. Es ist wichtig, dass $\sigma_{\{x/E\}}$ kollisionsfrei für Q ist. Andernfalls erhielte man für die Zuweisung $x \leftarrow y$ und die (erfüllbare!) Nachbedingung $\exists y : x + 1 \leq y$ als zugehörige Vorbedingung $\exists y : y + 1 \leq y$. So etwas will man offensichtlich nicht. Berücksichtigt man die Einschränkung, dann ist es eine nicht sehr schwere Übung, sich klar zu machen, dass die Hoare-Tripel der Form $\{\sigma_{\{x/E\}}(Q)\} x \leftarrow E \{Q\}$ tatsächlich gültig sind. Versuchen Sie das als Übung.

Wenn man für eine Zuweisung eine geeignete Vor- und eine geeignete Nachbedingung finden will, dann legt HT-A nahe, sich zuerst eine *Nachbedingung* zu suchen, weil sich die zugehörige Vorbedingung dann automatisch ergibt. Da diese Vorgehensweise entgegengesetzt zur Aberbeitungsreihenfolge von Programmen ist, sind Sie am Anfang vielleicht versucht, statt HT-A ein Analogon zu finden, das „von vorne nach hinten“ vorgeht. Herumprobieren wird Sie aber hoffentlich sehr schnell zu der Einsicht bringen, dass es zwar leicht ist, zu gegebener Nachbedingung Q die Vorbedingung $\sigma_{x/E}(Q)$ zu bestimmen, die umgekehrte Richtung aber in manchen Fällen völlig unklar.

Außerdem sei eine Warnung ausgesprochen: In der Literatur findet man an Stelle unserer Schreibweise $\sigma_{\{x/E\}}(Q)$ auch die Notationen $\{Q[E/x]\} x \leftarrow E \{Q\}$ oder $\{[E/x]Q\} x \leftarrow E \{Q\}$ oder $\{Q[x/E]\} x \leftarrow E \{Q\}$. Wie man sieht, ist Vorsicht geboten. Nur die letzte Notationsmöglichkeit nutzen wir gelegentlich auch.

Die beiden ersten „echten“ Ableitungsregeln des Hoare-Kalküls betreffen die Hintereinanderausführung von Programmstücken und die Möglichkeit die Vor- bzw. Nachbedingung eines Hoare-Tripels in passender Weise abzuändern.

Die Regel für die Hintereinanderausführung nennen wir HT-S (für „Sequenz“) und sie lautet:

$$\text{HT-S: } \frac{\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}}{\{P\} S_1; S_2 \{R\}}$$

Nachrechnen ergibt, dass diese Regel die Gültigkeit von Hoare-Tripeln erhält, d. h. die Eigenschaft hat, dass aus der Gültigkeit der Hoare-Tripel $\{P\} S_1 \{Q\}$ und $\{Q\} S_2 \{R\}$ auch die von $\{P\} S_1; S_2 \{R\}$ folgt.

Als einfaches Beispiel wollen wir für die Anweisungsfolge $y \leftarrow x; z \leftarrow y$ zeigen, dass

$$\{x = a\} y \leftarrow x; z \leftarrow y \{z = a\}$$

ein ableitbares, also gültiges, Hoare-Tripel ist. Da HT-A nahelegt, „von hinten nach vorne“ vorzugehen, tun wir das auch:

- Als erstes besagt HT-A, dass $\{y = a\} z \leftarrow y \{z = a\}$ ableitbar ist.
- Weiter besagt wiederum HT-A, dass $\{x = a\} y \leftarrow x \{y = a\}$ ableitbar ist.
- Aus den beiden ersten Punkten ergibt sich mit HT-S die Ableitbarkeit von $\{x = a\} y \leftarrow x; z \leftarrow y \{z = a\}$.

Wollte man z. B. den Algorithmus von Al-Khwarizmi mit Hilfe von Hoare-Tripeln verifizieren, würde man feststellen, dass noch ein Hilfsmittel fehlt, das es erlaubt, bei einem Hoare-Tripel die Vorbedingung zu „verstärken“ bzw. die Nachbedingung abzuschwächen. Wir nennen diese Regel HT-E. Sie lautet: Unter der Voraussetzung, dass $P' \rightarrow P$ und $Q \rightarrow Q'$ gelten, ist

$$\text{HT-E: } \frac{\{P\} S \{Q\}}{\{P'\} S \{Q'\}}$$

Wir verifizieren nun mit Hilfe von Hoare-Tripeln den Algorithmus von Al-Khwarizmi, schreiben das Ganze aber wieder ähnlich auf wie oben. Um etwas Platz zu sparen, schreiben wir \mathcal{A} als Abkürzung für die Aussage

$$„\sqrt{c + (b/2)^2} \text{ ist definiert und } \sqrt{c + (b/2)^2} - b/2 \text{ ist echt größer } 0.“$$

In Abbildung 14.3 sind die Zusicherungen in einer Reihenfolge durchnummeriert, in der man sie auch bestimmen kann. Ausgehend von Zusicherung 1 ergeben sich die Zusicherungen 2 bis 6 der Reihe nach jeweils aus der vorhergehenden gemäß Regel HT-A. Von Zusicherung 6 kommt man zu 7 und 8 durch Regel HT-E, indem man zu immer schwächeren Vorbedingungen übergeht. Analog kommt man mit HT-E von Zusicherung 1 letzten Endes zu 11, indem man zu immer stärkeren Nachbedingungen übergeht. Die Zwischenschritte 7, 9 und 10 sind nicht nötig, aber helfen hoffentlich beim Verständnis.

Damit hat man wegen der Regel HT-S insgesamt gezeigt, dass das Hoare-Tripel

$$\{b > 0 \wedge c > 0\} \ll \text{Algorithmus von Al-Khwarizmi} \gg \{x^2 + bx = c \wedge x > 0\}$$

- 8 $\{ b > 0 \wedge c > 0 \}$
- 7 $\{ \mathcal{A} \}$
- 6 $\{ \sqrt{c + (b/2)^2} - b/2 = \sqrt{c + b^2/4} - b/2 \wedge \mathcal{A} \}$
 $h \leftarrow b/2$
- 5 $\{ \sqrt{c + h^2} - h = \sqrt{c + b^2/4} - b/2 \wedge \mathcal{A} \}$
 $q \leftarrow h^2$
- 4 $\{ \sqrt{c + q} - h = \sqrt{c + b^2/4} - b/2 \wedge \mathcal{A} \}$
 $s \leftarrow c + q$
- 3 $\{ \sqrt{s} - h = \sqrt{c + b^2/4} - b/2 \wedge \mathcal{A} \}$
 $w \leftarrow \sqrt{s}$
- 2 $\{ w - h = \sqrt{c + b^2/4} - b/2 \wedge \mathcal{A} \}$
 $x \leftarrow w - h$
- 1 $\{ x = \sqrt{c + b^2/4} - b/2 \wedge \mathcal{A} \}$
- 9 $\{ x^2 + bx = (\sqrt{c + b^2/4} - b/2)^2 + b(\sqrt{c + b^2/4} - b/2) \wedge x > 0 \}$
- 10 $\{ x^2 + bx = c + b^2/4 - b\sqrt{c + b^2/4} + b^2/4 + b\sqrt{c + b^2/4} - b^2/2 \wedge x > 0 \}$
- 11 $\{ x^2 + bx = c \wedge x > 0 \}$

Abbildung 14.3: Verifikation des Algorithmus von von Al-Khwarizmi

gültig ist. Möglicherweise fragen Sie sich, wie man darauf kommt, ausgerechnet mit Zusicherung 1 anzufangen. Da hilft viel Übung. Im vorliegenden Fall hätten Sie aber auch noch einfach mit der untersten Zusicherung $x^2 + bx = c \wedge x > 0$ beginnen und sich dann nach oben durcharbeiten können. (Es wäre aber mehr Schreibarbeit gewesen.)

Auch für bedingte Anweisungen und Schleifen gibt es Regeln für Hoare-Tripel. Für bedingte Anweisungen geht das so:

$$\text{HT-I: } \frac{\{P \wedge B\} S_1 \{Q\} \quad \{P \wedge \neg B\} S_2 \{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{Q\}}$$

In Anlehnung an das Vorangegangene kann man diese Regel auch so darstellen wie in Abbildung 14.4. Um festzustellen, dass das äußere Hoare-Tripel gültig ist,

```

{ P }
if B
then
    { P ∧ B }
    S1
    { Q }
else
    { P ∧ ¬B }
    S2
    { Q }
fi
{ Q }

```

Abbildung 14.4: Die Regel HT-I für bedingte Anweisungen im Hoare-Kalkül

muss man sich von der Gültigkeit der beiden inneren Hoare-Tripel überzeugen.

Abbildung 14.5 zeigt ein einfaches Beispiel. Die Vorbedingung $0 = 0$ ist immer wahr. Deshalb besagt die Nachbedingung $x \geq 0$, dass sie *immer* nach Ausführung der bedingten Anweisung wahr ist. Die dazwischen eingefügten Zusicherungen ergeben sich zunächst durch HT-I und dann durch HT-A. Bei zwei aufeinander folgenden Zusicherungen muss man sich klar machen, dass HT-E zutrifft.

Der schwierigste Fall ist der von Schleifen. In der nachfolgenden Regel ist I eine „frei wählbare“ Zusicherung, eine sogenannte *Schleifeninvariante*.

Schleifeninvariante

HT-W:
$$\frac{\{I \wedge B\} S \{I\}}{\{I\} \text{ while } B \text{ do } S \text{ od } \{I \wedge \neg B\}}$$

Um zu erläutern, was es mit dieser Regel auf sich hat, betrachten wir im nächsten Abschnitt einen einfachen Algorithmus zur Multiplikation zweier Zahlen.

14.4 EIN ALGORITHMUS ZUR MULTIPLIKATION NICHTNEGATIVER GANZER ZAHLEN

Im Folgenden werden wieder die binären Operationen **div** und **mod** benutzt, die in Kapitel 8 eingeführt wurden: Die Operation **mod** liefert für zwei Argumente x und y als Funktionswert $x \bmod y$ den Rest der ganzzahligen Division von x durch


```

{ 0 = 0 }
if x < 0
then
    { 0 = 0 ∧ x < 0 }
    { -x ≥ 0 }
    x ← -x
    { x ≥ 0 }
else
    { 0 = 0 ∧ ¬(x < 0) }
    { x ≥ 0 }
    x ← x
    { x ≥ 0 }
fi
{ x ≥ 0 }

```

Abbildung 14.5: Verifikation einer einfachen bedingten Anweisung

y . Und die Operation **div** liefere für zwei Argumente x und y als Funktionswert $x \text{ div } y$ den Wert der ganzzahligen Division von x durch y . Es gilt:

$$x = y \cdot (x \text{ div } y) + (x \text{ mod } y) \quad \text{und} \quad 0 \leq (x \text{ mod } y) < y \quad (14.6)$$

Der Algorithmus ist auf der linken Seite von Abbildung 14.6 dargestellt.

Machen wir eine Beispielrechnung für den Fall $a = 6$ und $b = 9$. In der Tabelle auf der rechten Seite von Abbildung 14.6 ist in jeweils einer Zeile die Werte angegeben, die den Variablen P , X und Y jeweils zugewiesen werden, wenn Variable i einen bestimmten Wert hat. Um besser argumentieren zu können, schreiben wir z. B. P_i für den Wert, der P zugewiesen wird, wenn ein gewisser Wert von i vorliegt; es handelt sich sozusagen um den Wert „nach i Schleifendurchläufen“. (Die Variable i ist nur für diese Erläuterungen eingeführt; für das eigentliche Ergebnis des Algorithmus ist sie nicht notwendig.) Am Ende ist $X = 0$, die Schleife endet und man erhält in P den Wert 54. Das ist das Produkt der Eingabewerte 6 und 9. Im folgenden wollen wir beweisen, dass für den Grundbereich $D = \mathbb{N}_0$, und daher vereinbarungsgemäß für alle $I(a), I(b) \in \mathbb{N}_0$, der Algorithmus mit der Vorbedingung $\{0 = 0\}$ und der Nachbedingung $\{P = a \cdot b\}$ ein gültiges Hoare-Tripel ist.

Um Regel HT-W anwenden zu können, benötigen wir eine Schleifeninvariante I dergestalt, dass

$\{ \text{Eingaben: } a, b \in \mathbb{N}_0 \}$	
$i \leftarrow 0$	
$P \leftarrow 0$	
$X \leftarrow a$	$\overline{P_i X_i Y_i}$
$Y \leftarrow b$	$i = 0 \ 0 \ 6 \ 9$
while $X > 0$ do	$i = 1 \ 0 \ 318$
$i \leftarrow i + 1$	$i = 218 \ 136$
$P \leftarrow P + (X \bmod 2) \cdot Y$	$i = 354 \ 072$
$X \leftarrow X \text{ div } 2$	
$Y \leftarrow 2 \cdot Y$	
od	

Abbildung 14.6: Ein einfacher Algorithmus für die Multiplikation zweier nichtnegativer ganzer Zahlen a und b .

$\{ 0 = 0 \}$
$i \leftarrow 0$
$X \leftarrow a$
$Y \leftarrow b$
$P \leftarrow 0$
$\{ I \}$
while $X > 0$ do
$\{ I \wedge B \}$
$i \leftarrow i + 1$
$P \leftarrow P + (X \bmod 2) \cdot Y$
$X \leftarrow X \text{ div } 2$
$Y \leftarrow 2 \cdot Y$
$\{ I \}$
od
$\{ I \wedge \neg B \}$
$\{ P = a \cdot b \}$

zu einem vollständigen Beweis ergänzt werden kann. Nützliche Schleifeninvarianten zu finden, ist nicht leicht. (Nicht hilfreich ist z.B. die Aussage $2 \cdot 2 = 4$.) Mitunter hilft eine Tabelle wie in Abbildung 14.6. Hinreichend langes Hinsehen führt zu der Einsicht, dass die Formel $X \cdot Y + P = a \cdot b$ jedenfalls eine Schleifenin-

variante zu sein scheint. Das ist tatsächlich so, wie man in Abbildung 14.7 sieht.

```

{ 0 = 0 }
i ← 0
X ← a
Y ← b
P ← 0
{ X · Y + P = a · b }
while X > 0 do
  { X · Y + P = a · b ∧ X > 0 }
  { (X div 2) · (2Y) + P + (X mod 2) · Y = a · b }
  i ← i + 1
  { (X div 2) · (2Y) + P + (X mod 2) · Y = a · b }
  P ← P + (X mod 2) · Y
  { (X div 2) · (2Y) + P = a · b }
  X ← X div 2
  { X · (2Y) + P = a · b }
  Y ← 2 · Y
  { X · Y + P = a · b }
od
{ X · Y + P = a · b ∧ ¬(X > 0) }
{ P = a · b }

```

Abbildung 14.7: Die wesentlichen Zusicherungen für die Verifikation des Multiplikationsalgorithmus

Für den Übergang von der ersten zur zweiten Zusicherung im Schleifenrumpf ist die Gleichung 14.6 nützlich. Für den Übergang von der vorletzten zur letzten Zusicherung ist wichtig, dass als Grundbereich \mathbb{N}_0 angenommen wurde. Dass die ersten vier Zuweisungen immer die Gültigkeit der Invariante vor dem ersten Betreten der Schleife sicherstellen, können Sie inzwischen leicht selbst überprüfen.

15 GRAPHEN

In den bisherigen Kapiteln 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 12.2 in dem Kapitel über Grammatiken, an Huffman-Bäume wie in Abbildung 8.2 des Kapitels über Codierungen, oder an die Abbildung am Ende von Abschnitt 2.4.

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

15.1 GERICHTETE GRAPHEN

15.1.1 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 15.1: 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 15.1, oder ob man jeweils das entsprechende Element von V mit notiert wie auf der rechten Seite in Abbildung 15.1, hängt von den

adjazente Knoten

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*. Man beachte aber, diese Relation nicht symmetrisch ist, obwohl die Formulierung es suggeriert!

Außerdem ist die Anordnung der Knoten in der Darstellung irrelevant. Abbildung 15.2 zeigt den gleichen Graphen wie Abbildung 15.1:

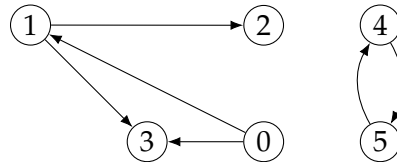


Abbildung 15.2: eine andere Zeichnung des Graphen aus Abbildung 15.1

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

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

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:

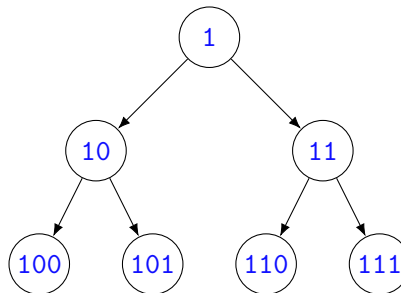


Abbildung 15.3: ein Graph, der ein sogenannter Baum ist

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 in Abbildung 15.4 darstellen. Es handelt sich um einen sogenannten De Bruijn-Graphen. Wie man sieht, können

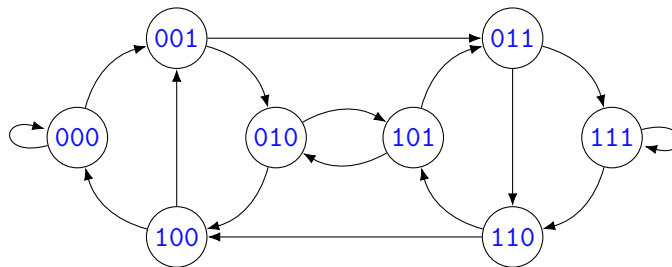


Abbildung 15.4: Der de Bruijn-Graphen mit 8 Knoten

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

Schlinge

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:

15.1.2 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 $i \in \mathbb{Z}_n$ gilt: $(v_i, v_{i+1}) \in E$. Die Anzahl

Pfad

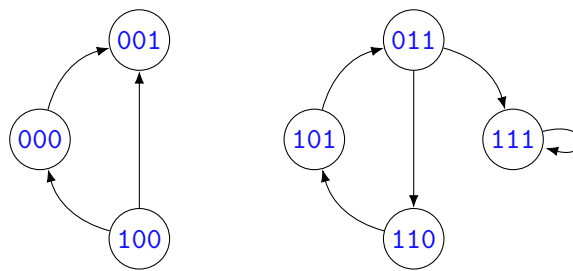


Abbildung 15.5: Ein Teilgraph des de Bruijn-Graphen aus Abbildung 15.4

Länge eines Pfades	$n = p - 1$ der Kanten (!) heißt die <i>Länge</i> 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 $i \in \mathbb{Z}_n$.
erreichbar	Wenn $p = (v_0, \dots, v_n)$ ein Pfad ist, sagt man auch, dass v_n von v_0 aus <i>erreichbar</i> ist.
Teilpfad	Streicht man von einem Pfad $p = (v_0, \dots, v_n)$ am Anfang oder/und am Ende endlich viele Knoten, aber nicht alle, so entsteht ein sogenannter <i>Teilpfad</i> von p .
wiederholungsfreier Pfad	Ein Pfad (v_0, \dots, v_n) heißt <i>wiederholungsfrei</i> , wenn gilt: Die Knoten v_0, \dots, v_{n-1} sind paarweise verschieden und die Knoten v_1, \dots, v_n sind paarweise verschieden. Die beiden einzigen Knoten, die gleich sein dürfen, sind also v_0 und v_n .
geschlossener Pfad	Ein Pfad mit $v_0 = v_n$ heißt <i>geschlossen</i> .
Zyklus	Wenn $n \geq 1$ ist, heißt ein geschlossener Pfad (v_0, \dots, v_n) auch <i>Zyklus</i> . Er heißt
einfacher Zyklus	ein <i>einfacher Zyklus</i> , wenn er außerdem wiederholungsfrei ist. Zum Beispiel ist in Abbildung 15.5 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 15.5 auch sieht, kann es unterschiedlich lange Pfade von einem Knoten zu einem anderen geben: von 100 nach 001 gibt es einen Pfad der Länge 1 und einen Pfad der Länge 2. Und es gibt in diesem Graphen auch Knotenpaare, bei denen gar kein Pfad von einem zum anderen existiert.
streng zusammenhängender Graph	Ein gerichteter Graph heißt <i>streng zusammenhängend</i> , 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 15.4 streng zusammenhängend (wie man durch Durchprobieren herausfindet), aber der Teilgraph aus Abbildung 15.5 eben nicht. Statt „streng zusammenhängend“ sagt man manchmal auch „stark zusammenhängend“.
	Sozusagen eine sehr viel einfachere Variante von Zusammenhang ist bei Gra-

phen 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 15.3 ist ein Baum dargestellt, dessen Wurzel 1 ist.

Baum

Wurzel

15.1 Lemma. Die Wurzel eines gerichteten Baumes ist eindeutig.

15.2 Beweis. Angenommen, r und r' sind Wurzeln des gleichen Baumes. Dann gibt es

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

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

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

Eingangsgrad

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

Analog nennt man

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

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

Ausgangsgrad
Grad

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

Blatt eines Baums

15.1.3 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:

isomorph

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

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

Man kann sich überlegen:

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

15.1.4 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 3.3). Für solche Relationen hatten wir in Abschnitt 12.3 Potenzen definiert. Im folgenden wollen wir uns klar machen, dass es eine enge Verbindung gibt zwischen den Relationen E^i für $i \in \mathbb{N}_0$ und Pfaden der Länge i im Graphen. Daraus wird sich dann auch eine einfache Interpretation von E^* ergeben.

Betrachten wir zunächst den Fall $i = 2$.

Ein Blick zurück in Abschnitt 12.3 zeigt, dass $E^2 = E \circ E^1 = E \circ E \circ I = E \circ E$ ist. Nach Definition des Relationenproduktes ist

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

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

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

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

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

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

Damit haben wir uns explizit davon überzeugt, dass für alle $i \in \mathbb{Z}_3$ 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.

15.3 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 $i \in \mathbb{N}_0$:

15.4 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:

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

15.2 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 es meist angebracht, üblich und übersichtlicher, in der graphischen Darstellung nicht einen Pfeil von x nach y und einen Pfeil von y nach x zu zeichnen, sondern die beiden Knoten einfach durch *einen* Strich (*ohne* Pfeilspitzen) miteinander zu verbinden. Und man spricht dann auch nur von *einer* Kante. Üblicherweise passt man dann auch die

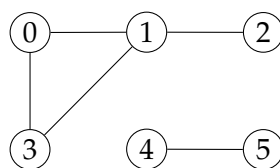


Abbildung 15.6: ein ungerichteter Graph

Formalisierung an und definiert: Ein *ungerichteter*

ungerichteter Graph ist eine Struktur $U = (V, E)$ mit einer endlichen nichtleeren Menge V von Knoten und einer Menge E von Kanten, wobei $E \subseteq \{ \{x, y\} \mid x \in V \wedge y \in V \}$.

adjazent Analog zum gerichteten Fall heißen zwei Knoten eines ungerichteten Graphen *adjazent*, wenn sie durch eine Kante miteinander verbunden sind.

Schlinge 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,

schlingenfrei heißt er auch wieder *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 Teilmenge von V resp. E sein, und die Endpunkte jeder Kante von E' müssen auch zu V' gehören.

Weg 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 $i \in \mathbb{Z}_n$ gilt: $\{v_i, v_{i+1}\} \in E$. Die Anzahl $n = |p| - 1$ der Kanten (!) heißt die *Länge* des Weges.

Kantenrelation 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:

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

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.

zu ungerichtetem Graphen
gehöriger gerichteter Graph
zusammenhängender
ungerichteter Graph

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

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

bijektive Abbildung $f : V_1 \rightarrow V_2$ gibt mit der Eigenschaft:

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

Auch für ungerichtete Graphen ist Isomorphie eine Äquivalenzrelation.

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

Damit definieren wir nun, was wir im ungerichteten Fall als Baum bezeichnen wollen: Ein ungerichteter Graph $U = (V, E)$ heißt ein *Baum*, wenn es einen gerichteten Baum $G = (V, E')$ gibt mit $E = E'_u$. Abbildung 15.7 zeigt zwei ungerichtete Bäume.

ungerichteter
Baum

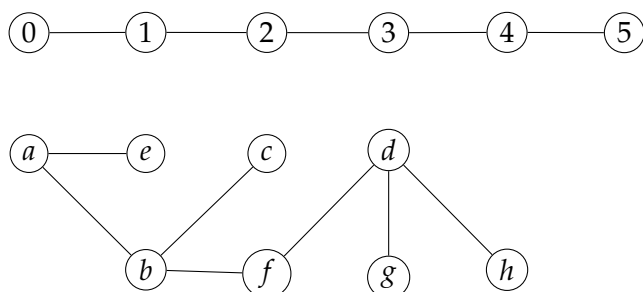


Abbildung 15.7: zwei ungerichtete Bäume

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

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

15.2.1 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:

*symmetrische
Relation*

Eine Relation $R \subseteq M \times M$ heißt *symmetrisch* wenn für alle $x \in M$ und $y \in M$ gilt:

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

Äquivalenzrelation

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

Wir haben weiter vorne in diesem Kapitel auch eine erste interessante Äquivalenzrelation kennengelernt: die Isomorphie von Graphen. Man lese noch einmal aufmerksam die drei Punkte der Aufzählung am Ende von Unterabschnitt 15.1.3.

15.3 GRAPHEN MIT KNOTEN- ODER KANTENMARKIERUNGEN (NICHT IM WINTERSEMESTER 2020/2021)

Dieser Abschnitt wurde im Wintersemester 2019/2020 ausgelassen. Der Inhalt ist nicht prüfungsrelevant. 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.

*knotenmar-
kierter Graph*

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 $M_V = \mathbb{N}_+$.

Aus Landkarten, auf denen Länder mit ihren Grenzen eingezeichnet sind, kann man auf verschiedene Weise Graphen machen. Hier ist eine Möglichkeit: Jedes Land wird durch einen Knoten des (ungerichteten) Graphen repräsentiert. Eine Kante verbindet zwei Knoten genau dann, wenn die beiden repräsentierten Länder ein Stück gemeinsame Grenzen haben. Nun ist auf Landkarten üblicherweise

das Gebiet jedes Landes in einer Farbe eingefärbt, und zwar so, dass benachbarte Länder verschiedene Farben haben (damit man sie gut unterscheiden kann). Die Zuordnung von Farben zu Knoten des Graphen ist eine Knotenmarkierung. (Man spricht auch davon, dass der Graph gefärbt sei.) Wofür man sich interessiert, sind „legale“ Färbungen, bei denen adjazente Knoten verschiedene Farben haben: $\{x, y\} \in E \implies m_V(x) \neq m_V(y)$. Ein Optimierungsproblem besteht dann z. B. darin, herauszufinden, welches die minimale Anzahl von Farben ist, die ausreicht, um den Graphen legal zu färben. Solche Färbungsprobleme müssen nicht nur (vielleicht) von Verlagen für Atlanten gelöst werden, sondern sie werden auch etwa von modernen Compilern beim Übersetzen von Programmen bearbeitet.

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

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

gewichteter Graph

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.

16 ERSTE ALGORITHMEN IN GRAPHEN

In diesem Kapitel wollen wir beginnen, Algorithmen auch unter quantitativen Gesichtspunkten zu betrachten.

Als „Aufhänger“ werden wir eine vereinfachte Problemstellung betrachten, die mit einer der am Ende des [Kapitels 15 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 [16.1](#) mit der Frage, wie man denn Graphen im Rechner repräsentiert. In [16.2](#) nähern wir uns dann langsam dem Erreichbarkeitsproblem, indem wir uns erst einmal nur für Pfade der Länge 2 interessieren. Das führt auch zu den Konzepten Matrizenaddition und Matrizenmultiplikation. Auf der Matrizenrechnung aufbauend beginnen wir dann in [16.3](#) mit einem ganz naiven Algorithmus und verbessern ihn in zwei Schritten. Einen der klassischen Algorithmen, den von Warshall, für das Problem, werden wir in Abschnitt [16.4](#) kennenlernen.

Nachdem wir uns in diesem Kapitel beispielhaft auch mit dem Thema beschäftigt haben werden, wie man — in einem gewissen Sinne — die Güte eines Algorithmus quantitativ erfassen kann, werden wir das im nachfolgenden [Kapitel 17 über quantitative Aspekte von Algorithmen](#) an weiteren Beispielen aber auch allgemein genauer beleuchten.

16.1 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 Edge {
    Vertex start;
    Vertex end;
}

class Graph {
    Vertex[] vertices;
    Edge[] edges;
}
```

Dabei hat man aber mehr hingeschrieben als man „eigentlich“ will, denn die Knoten (und auch die Kanten) sind durch die Nummerierung der Komponenten der Arrays total angeordnet worden. Das ist bei den Mengen der mathematischen Definition nicht der Fall.

Aber es schadet nicht. Da man die Nummern aber sowieso schon hat, macht man, zumindest wenn man besonders kurz und übersichtlich sein will, den Schritt und sagt, dass die Identitäten der Knoten einfach die Zahlen eines Anfangsstückes der natürlichen Zahlen sind. Solange man sich mit Problemen beschäftigt, die unter Isomorphie invariant sind, ergeben sich hierdurch keine Probleme. Deswegen ist für uns im folgenden bei allen Graphen $V = \mathbb{Z}_n$ für ein $n \geq 1$.

```
class Vertex {
    int id;
}

class Graph {
    Vertex[] vertices;
    Edge[] edges;
}

class Edge {
    Vertex start;
    Vertex end;
}
```

Gelegentlich verwendet man als Knotennummern auch Anfangsstücke der positiven ganzen Zahlen (also ohne Null). Lassen Sie sich von solchen kleinen technischen Details nicht verunsichern. Man macht, was einem gerade am besten erscheint.

Wenn man Graphen in Java wie oben skizziert implementieren würde, dann könnte man bei einer gegebenen Kante leicht auf deren Anfangs- und Endknoten zugreifen. Wie Sie bald sehen werden, will man aber mitunter umgekehrt zu einem gegebenen Knoten v z. B. auf die ihn verlassenden Kanten zugreifen. Das wäre aber nur umständlich möglich: Man müsste systematisch alle Kanten darauf hin überprüfen, ob sie bei v starten.

Es gibt (neben anderen) zwei gängige Methoden, dieses Problem zu beseitigen. Die eine besteht darin, zu jedem Knoten eine Liste der ihn verlassenden Kanten oder der über solche Kanten erreichbaren Nachbarknoten mitzuführen. Wenn man diese Liste als Array implementiert, dann wäre

```
class Vertex {
    int id;
    Vertex[] neighbors;
}

class Graph {
    Vertex[] vertices;
}
```

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

Adjazenzliste

Wenn man mit kantenmarkierten Graphen arbeiten muss, benutzt man statt dessen lieber die *Inzidenzlisten*. Das ist für einen Knoten die Liste der Kanten, die ihn als einen Endpunkt haben.

Inzidenzliste

Wir wollen im folgenden aber eine andere Methode benutzen, um die Beziehungen zwischen Knoten zu speichern. Wenn man zu einem Knoten u womöglich oft und schnell herausfinden möchte, ob ein Knoten v Nachbar von u ist, dann ist es bequem, wenn man das immer leicht herausfinden kann. Man müsste dann (unter Umständen) nur noch die Klassen für einzelne Knoten und einen Graphen implementieren, z. B. so:

```
class Vertex {
    int id;
    boolean[] is_connected_to;
}

class Graph {
    Vertex[] vertices;
}
```

Für einen Knoten, also ein Objekt u der Klasse `Vertex`, wäre `is_connected_to` also ein Feld mit genau so vielen Komponenten wie es Knoten im Graphen gibt. Und `u.is_connected_to[v.id]` sei genau dann `true`, wenn eine Kante von u nach v existiert, und ansonsten `false`.

Betrachten wir als Beispiel den Graphen aus Abbildung 16.1:

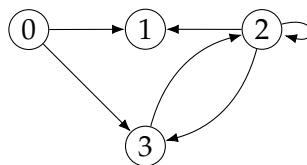


Abbildung 16.1: Ein kleiner Beispielgraph

Für das Objekt u der Klasse `Vertex`, das den Knoten 0 repräsentiert, würde z. B. gelten:

$u.id$	$u.is_connected_to$			
0	false	true	false	true
	0	1	2	3

Schreibt man das für alle vier Knoten untereinander, erhält man:

<i>u.id</i>	<i>u.is_connected_to</i>			
0	false	true	false	true
1	false	false	false	false
2	false	true	true	true
3	false	false	true	false
	0	1	2	3

Adjazenzmatrix

Wenn man in dieser zweidimensionalen Tabelle nun noch false durch 0 und true durch 1 ersetzt, erhält man die sogenannte *Adjazenzmatrix* des Graphen.

Manche haben Matrizen inzwischen in der Vorlesung „Lineare Algebra“ kennengelernt, andere haben zumindest schon ein Beispiel gesehen, in dem ein Graph als zweidimensionale Tabelle repräsentiert wurde. Im allgemeinen können Zeilenzahl m und Spaltenzahl n einer Matrix A verschieden sein. Man spricht dann von einer $m \times n$ -Matrix. Die einzelnen Einträge in Matrizen werden in dieser Vorlesung immer Zahlen sein. Für den Eintrag in Zeile i und Spalte j von A schreiben wir auch A_{ij} (oder manchmal $(A)_{ij}$ o. ä.).

Für die Menge aller $m \times n$ -Matrizen, deren Einträge alle aus einer Menge M stammen, schreiben wir gelegentlich $M^{m \times n}$.

Typischerweise notiert man eine Matrix ohne die ganzen senkrechten und waagerechten Striche, aber mit großen runden (oder manchmal auch eckigen) Klammern außen herum. Wenn es hilfreich ist, notiert man außerhalb der eigentlichen Matrix auch die Nummerierung der Zeilen bzw Spalten, wie es z. B. in Abbildung 16.2 gemacht ist.

Die Adjazenzmatrix eines gerichteten Graphen $G = (V, E)$ mit n Knoten ist eine $n \times n$ -Matrix A mit der Eigenschaft:

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

Als Beispiel ist in Abbildung 16.2 noch einmal der Graph mit vier Knoten und nun auch die zugehörige Adjazenzmatrix angegeben.

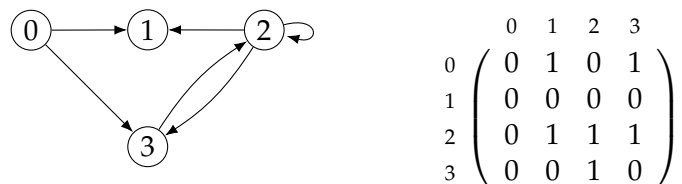


Abbildung 16.2: Ein Graph und seine Adjazenzmatrix

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

Allgemein kann man jede binäre Relation $R \subseteq M \times M$ auf einer endlichen Menge M mit n Elementen durch eine $n \times n$ -Matrix $A(R)$ repräsentieren, indem man definiert:

$$(A(R))_{ij} = \begin{cases} 1 & \text{falls } (i, j) \in R \quad \text{d. h. also } iRj \\ 0 & \text{falls } (i, j) \notin R \quad \text{d. h. also } \neg(iRj) \end{cases}$$

Dabei gehören zu verschiedenen Relationen auf der gleichen Menge M verschiedene Matrizen und umgekehrt.

So, wie man die Kantenrelation E eines gerichteten Graphen als Adjazenzmatrix darstellen kann, kann man natürlich auch jede andere Relation durch eine entsprechende Matrix darstellen, z. B. die „Erreichbarkeitsrelation“ E^* . Die zugehörige Matrix W eines Graphen wird üblicherweise *Wegematrix* genannt. Sie hat also die Eigenschaft:

Wegematrix

$$\begin{aligned} W_{ij} &= \begin{cases} 1 & \text{falls } (i, j) \in E^* \\ 0 & \text{falls } (i, j) \notin E^* \end{cases} \\ &= \begin{cases} 1 & \text{falls es in } G \text{ einen Pfad von } i \text{ nach } j \text{ gibt} \\ 0 & \text{falls es in } G \text{ keinen Pfad von } i \text{ nach } j \text{ gibt} \end{cases} \end{aligned}$$

Im folgenden wollen wir uns mit dem algorithmischen Problem beschäftigen, zu gegebener Adjazenzmatrix A die zugehörige Wegematrix W zu berechnen.

16.2 BERECHNUNG DER 2-ERREICHBARKEITSRELATION UND RECHNEN MIT MATRIZEN

Es sei A die Adjazenzmatrix eines Graphen $G = (V, E)$; Abbildung 16.3 zeigt ein Beispiel. Wir interessieren uns nun zum Beispiel für die Pfade der Länge 2 von Knoten 2 zu Knoten 4. Wie man in dem Graphen sieht, gibt es genau zwei solche Pfade: den über Knoten 1 und den über Knoten 6.

Wie findet man „systematisch“ alle solchen Pfade? Man überprüft *alle* Knoten $k \in V$ daraufhin, ob sie als „Zwischenknoten“ für einen Pfad $(2, k, 4)$ in Frage kommen. Und $(2, k, 4)$ ist genau dann ein Pfad, wenn sowohl $(2, k) \in E$, also eine Kante, ist, als auch $(k, 4) \in E$, also eine Kante, ist. Und das ist genau dann der Fall, wenn in der Adjazenzmatrix A von G sowohl $A_{2k} = 1$ als auch $A_{k4} = 1$ ist. Das kann man auch so schreiben, dass $A_{2k} \cdot A_{k4} = 1$ ist.

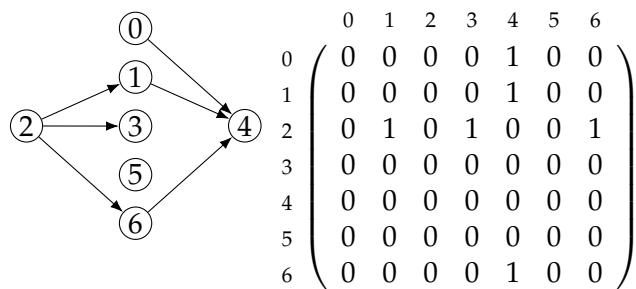


Abbildung 16.3: Beispielgraph für die Berechnung von E^2

Wenn man nacheinander alle Elemente A_{2k} inspiziert, dann durchläuft man in A nacheinander die Elemente in der *Zeile* für Knoten 2. Und wenn man nacheinander alle Elemente A_{k4} inspiziert, dann durchläuft man in A nacheinander die Elemente in der *Spalte* für Knoten 4.

In Abbildung 16.4 haben wir schräg übereinander zweimal die Matrix A hingeschrieben, wobei wir der Deutlichkeit halber einmal nur die Zeile für Knoten 2 explizit notiert haben und das andere Mal nur die Spalte für Knoten 4. Außerdem haben wir im oberen linken Viertel alle Produkte $A_{2k} \cdot A_{k4}$ angegeben. Die Frage, ob es einen Pfad der Länge zwei $(2, k, 4)$ gibt, ist gleichbedeutend mit der Frage, ob eines dieser Produkte gleich 1 ist.

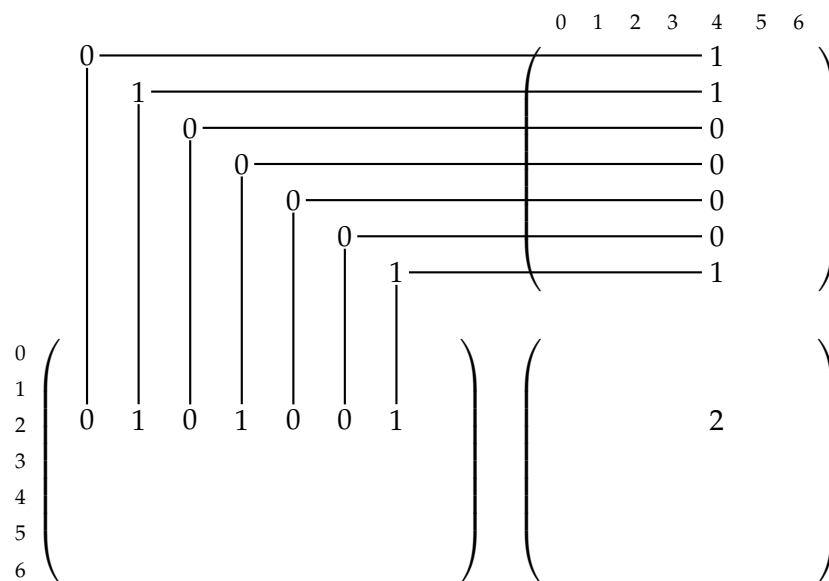


Abbildung 16.4: Der erste Schritt in Richtung Matrizenmultiplikation

Wir tun jetzt aber noch etwas anderes. Wir addieren die Werte alle zu einer Zahl

$$P_{24} = \sum_{k=0}^6 A_{2k} \cdot A_{k4}$$

und notieren Sie in einer dritten Matrix im unteren rechten Viertel der Darstellung. Dabei ist jetzt wichtig:

- Der Wert P_{24} gibt an, wieviele Pfade der Länge zwei es von Knoten 2 nach Knoten 4 gibt.
- Analog kann man für jedes andere Knotenpaar (i, j) die gleiche Berechnung durchführen:

$$P_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot A_{kj}$$

Dann ist P_{ij} die Anzahl der Pfade der Länge zwei von Knoten i zu Knoten j .

16.2.1 Matrizenmultiplikation

Was wir eben aufgeschrieben haben ist nichts anderes als ein Spezialfall von *Matrizenmultiplikation*. Ist A eine $\ell \times n$ -Matrix und B eine $n \times m$ -Matrix, so heißt die $\ell \times m$ -Matrix C , bei der für alle i und alle j

Matrizenmultiplikation

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot B_{kj}$$

gilt, das Produkt der Matrizen A und B und man schreibt auch $C = A \cdot B$. (Wichtig: Selbst wenn einmal $\ell = n = m$ ist, ist trotzdem im Allgemeinen $A \cdot B \neq B \cdot A$!)

Algorithmisch notiert sähe die naheliegende Berechnung des Produktes zweier Matrizen so aus, wie in Algorithmus 16.1 dargestellt. Wir werden im nächsten Kapitel aber sehen, dass es durchaus andere Möglichkeiten gibt!

Von besonderer Wichtigkeit sind die sogenannten *Einheitsmatrizen*. Das sind diejenigen $n \times n$ -Matrizen I , bei denen für alle i und j gilt:

Einheitsmatrix

$$I_{ij} = \begin{cases} 1 & \text{falls } i = j \\ 0 & \text{falls } i \neq j \end{cases}$$

Zu beliebiger $m \times n$ -Matrix A gilt für die jeweils passende Einheitsmatrizen:

$$I \cdot A = A = A \cdot I$$

Man beachte, dass die Einheitsmatrix auf der linken Seite Größe $m \times m$ hat und die auf der rechten Seite Größe $n \times n$.

Algorithmus 16.1: Einfachster Algorithmus, um zwei Matrizen zu multiplizieren

```

for  $i \leftarrow 0$  to  $\ell - 1$  do
  for  $j \leftarrow 0$  to  $m - 1$  do
     $C_{ij} \leftarrow 0$ 
    for  $k \leftarrow 0$  to  $n - 1$  do
       $C_{ij} \leftarrow C_{ij} + A_{ik} \cdot B_{kj}$ 
    od
  od
od

```

Ist eine Matrix quadratisch (wie z. B. die Adjazenzmatrix A eines Graphen), dann kann man A mit sich selbst multiplizieren. Dafür benutzt man wie schon an mehreren anderen Stellen in dieser Vorlesung die Potenzschreibweise:

$$A^0 = I$$

$$\forall n \in \mathbb{N}_0 : A^{n+1} = A^n \cdot A$$

16.2.2 Matrizenaddition

Matrizenaddition

Für zwei Matrizen A und B der gleichen Größe $m \times n$ ist für uns in Kürze auch noch die Summe $A + B$ von Interesse. Die *Matrizenaddition* von A und B liefert die $m \times n$ -Matrix C , bei der für alle i und alle j gilt:

$$C_{ij} = A_{ij} + B_{ij}.$$

Nullmatrix

Das neutrale Element ist die sogenannte *Nullmatrix* (genauer gesagt die Nullmatrizen; je nach Größe). Sie enthält überall nur Nullen. Wir schreiben für Nullmatrizen einfach 0.

Zwei Matrizen zu addieren, ist leicht:

```

for  $i \leftarrow 0$  to  $m - 1$  do
  for  $j \leftarrow 0$  to  $n - 1$  do
     $C_{ij} \leftarrow A_{ij} + B_{ij}$ 
  od
od

```


16.3 BERECHNUNG DER ERREICHBARKEITSRELATION

Eine naheliegende Idee für die Berechnung von E^* ist natürlich, auf die Definition

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

zurückzugreifen. Allerdings stellen sich sofort drei Probleme:

- Was kann man tun, um nicht unendlich viele Matrizen berechnen zu müssen? D. h., kann man das ∞ durch eine natürliche Zahl ersetzen?
- Woher bekommt man die Matrizen für die Relationen E^i , d. h. welcher Operation bei Matrizen entspricht das Berechnen von Potenzen bei Relationen?
- Wenn man die Matrizen hat, welcher Operation bei Matrizen entspricht die Vereinigung bei Relationen?

Beginnen wir mit dem ersten Punkt. Was ist bei Graphen spezieller als bei allgemeinen Relationen? Richtig: Es gibt nur *endlich* viele Knoten. Und das ist in der Tat eine große Hilfe: Wir interessieren uns für die Frage, ob für gegebene Knoten i und j ein Pfad in G von i nach j existiert. Sei $G = (V, E)$ mit $|V| = n$. Nehmen wir an, es existiert ein Pfad: $p = (i_0, i_1, \dots, i_k)$ 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:

16.1 Lemma. Für jeden gerichteten Graphen $G = (V, E)$ mit n Knoten gilt:

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

16.3.1 Potenzen der Adjazenzmatrix

Wenn man die Adjazenzmatrix A eines Graphen quadriert, erhält man als Eintrag in Zeile i und Spalte j

$$(A^2)_{ij} = \sum_{k=0}^{n-1} A_{ik} A_{kj}.$$

Jeder der Summanden ist genau dann 1, wenn $A_{ik} = A_{kj} = 1$ ist, also genau dann, wenn (i, k, j) ein Pfad der Länge 2 von i nach j ist. Und für verschiedene k sind das auch verschiedene Pfade. Also ist

$$(A^2)_{ij} = \sum_{k=0}^{n-1} A_{ik} A_{kj}$$

gleich der Anzahl der Pfade der Länge 2 von i nach j .

Überlegen Sie sich, dass analoge Aussagen für $(A^1)_{ij}$ und Pfade der Länge 1 von i nach j , sowie $(A^0)_{ij}$ und Pfade der Länge 0 von i nach j richtig sind. Tatsächlich gilt:

16.2 Lemma. Es sei G ein gerichteter Graph mit Adjazenzmatrix A . Für alle $k \in \mathbb{N}_0$ gilt: $(A^k)_{ij}$ ist die Anzahl der Pfade der Länge k in G von i nach j .

Der Beweis wäre eine recht einfache vollständige Induktion. Der einzige gegenüber dem Fall $k = 2$ zusätzlich zu beachtende Punkt besteht darin, dass die Verlängerung verschiedener Wege um die gleiche Kante (falls das überhaupt möglich ist), wieder zu verschiedenen Wegen führt.

Signum-Funktion

Wir bezeichnen nun mit sgn die sogenannte *Signum-Funktion*

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

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 16.2:

16.3 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.

16.3.2 Erste Möglichkeit für die Berechnung der Wegematrix

Um zu einem ersten Algorithmus zur Berechnung der Wegematrix zu kommen, müssen wir als letztes noch klären, welche Operation auf Matrizen „zur Vereinigung von Relationen passt“. Seien dazu auf der gleichen Menge M zwei binäre Relationen $R \subseteq M \times M$ und $R' \subseteq M \times M$ gegeben, repräsentiert durch Matrizen A und A' . Dann gilt:

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

Also wird die Relation $R \cup R'$ durch die Matrix $\text{sgn}(A + A')$ repräsentiert.

Aus Lemma 16.1 und dem eben aufgeführten Korollar 16.3 folgt recht schnell eine erste Formel für die Berechnung der Wegematrix:

16.4 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 .

16.5 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, die obige Überlegung zur Matrixrepräsentation von $R \cup R'$ per Induktion auf beliebig endliche viele Relationen zu übertragen.
- Mit einer ähnlichen Herangehensweise wie oben ergibt sich

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

Dabei haben wir in beiden Punkten benutzt, dass die Einträge in den interessierenden Matrizen nie negativ sind.

■

Das sich ergebende Verfahren ist in Algorithmus 16.2 dargestellt.

16.3.3 Zählen durchzuführender arithmetischer Operationen

Wir stellen nun ein erstes Mal die Frage danach wie aufwändig es ist, die Wegematrix eines Graphen auszurechnen. Unter Aufwand wollen hier zunächst einmal der Einfachheit halber die Anzahl benötigter arithmetischer Operationen verstehen. (Wir werden das im Laufe weiterer Kapitel noch genauer diskutieren.)

Wir beginnen mit der wohl naivsten Herangehensweise, wollen aber 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 16.2 zu benutzen und ohne viel Nachdenken zu implementieren. Das bedeutet, dass wir zur Berechnung von $\text{sgn}(\sum_{i=0}^{n-1} A^i)$ 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;

Algorithmus 16.2: 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  $A^i$  zu berechnen }
W ← 0                                { Nullmatrix }
for i ← 0 to n - 1 do
  M ← I                                { Einheitsmatrix }
  for j ← 1 to i do
    M ← M · A                          { Matrizenmultiplikation }
  od
  W ← W + M                            { Matrizenaddition }
od
W ← sgn(W)

```

Alle Matrizen haben Größe $n \times n$.

Für jeden der n^2 Einträge in einer Summenmatrix muss man eine Addition durchführen, also benötigt eine Matrizenaddition n^2 arithmetische Operationen.

Für jeden der n^2 Einträge in einer Produktmatrix besteht jedenfalls die nahe-
liegende Methode darin, eine Formel der Form $\sum_{k=0}^{n-1} a_{ik}b_{kj}$ auszuwerten. (Tatsäch-
lich gibt es andere Möglichkeiten, wie wir in nachfolgenden Abschnitten sehen
werden!) Das sind jeweils n Multiplikationen und $n - 1$ Additionen. Insgesamt
ergeben sich so $2n^3 - n^2$ Operationen.

Für die Berechnung der Wegematrix nach dieser Methode kommt man so auf

$$\begin{aligned}
 & n^2 + n^2 \cdot n + (2n^3 - n^2)n(n-1)/2 \\
 &= n^2 + n^3 + (2n^3 - n^2)(n^2 - n)/2 \\
 &= n^2 + n^3 + (2n^5 - 2n^4 - n^4 + n^3)/2 \\
 &= n^5 - \frac{3}{2}n^4 + \frac{3}{2}n^3 + n^2
 \end{aligned}$$

Operationen. Wenn z. B. $n = 1000$ ist, dann sind das immerhin 998 501 501 000 000
Operationen, also „fast“ $n^5 = 10^{15}$.

16.3.4 Weitere Möglichkeiten für die Berechnung der Wegematrix

Kann man die Wegematrix auch mit „deutlich“ weniger Operationen berechnen? Vielleicht haben Sie eine Möglichkeit schon gesehen: Wir haben weiter vorne so getan, als müsste man für die Berechnung von A^i immer $i - 1$ Matrizenmultiplikationen 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* Matrizenmultiplikation, um zu A^i zu gelangen. Diese Vorgehensweise ist in Algorithmus 16.3 dargestellt. Damit ergeben sich insgesamt nur n Matrizenmultiplikationen 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 16.2.

Algorithmus 16.3: 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  $A^i$  zu berechnen }
W  $\leftarrow$  0 { Nullmatrix }
M  $\leftarrow$  I { Einheitsmatrix }
for  $i \leftarrow 0$  to  $n - 1$  do
    W  $\leftarrow$  W + M { Matrizenaddition }
    M  $\leftarrow$  M  $\cdot$  A { Matrizenmultiplikation }
od
W  $\leftarrow$  sgn(W)

```

Wenn man einmal unterstellt, dass jede Operation gleich lange dauert, dann erhält man also Ergebnisse um etwa einen Faktor $n/2$ schneller.

Und es geht noch schneller: statt n Matrizenmultiplikationen wollen wir nun mit $\log_2 n$ von ihnen auskommen. Hierfür nutzen wir die Beobachtung aus, deren explizite Erwähnung in Lemma 16.1 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 hinschreiben. Dazu betrachten wir $F = E^0 \cup E^1 = I_V \cup E$. Unter Verwendung der Rechenregel (die Sie sich bitte klar machen) für Relationen

$$(A \cup B) \circ (C \cup D) = (A \circ C) \cup (A \circ D) \cup (B \circ C) \cup (B \circ D)$$

ergibt sich

$$F^2 = (E^0 \cup E^1) \circ (E^0 \cup E^1) = E^0 \cup E^1 \cup E^1 \cup E^2 = E^0 \cup E^1 \cup E^2$$

Daraus folgt

$$\begin{aligned} F^4 &= (F^2)^2 = (E^0 \cup E^1 \cup E^2) \circ (E^0 \cup E^1 \cup E^2) \\ &= \dots \\ &= E^0 \cup E^1 \cup E^2 \cup E^3 \cup E^4 \end{aligned}$$

und durch Induktion sieht man, dass für alle $m \in \mathbb{N}_0$ gilt:

$$F^{2^m} = \bigcup_{i=0}^{2^m} E^i$$

Wenn man einfach zu Beginn die Matrix für $F = E^0 + E$ berechnet und sie dann so oft quadriert, dass nach m -maligen Durchführen $2^m \geq n - 1$ ist, hat man das gewünschte Ergebnis. Offensichtlich genügt $m = \lceil \log_2 n \rceil$.

Im Matrizenrechner 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 $2n / \lceil \log_2 n \rceil$.

16.4 ALGORITHMUS VON WARSHALL (NICHT IM WINTERSEMESTER 2020/2021)

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 16.4 dargestellt.

Zum besseren Verständnis sei als erstes darauf hingewiesen, dass für zwei Bits x und y das Maximum $\max(x, y)$ dem logischen Oder entspricht, wenn man 1 als

Algorithmus 16.4: Berechnung der Wegematrix nach Warshall

```

for  $i \leftarrow 0$  to  $n - 1$  do
  for  $j \leftarrow 0$  to  $n - 1$  do
     $W_{ij} \leftarrow \begin{cases} 1 & \text{falls } i = j \\ A_{ij} & \text{falls } i \neq j \end{cases}$ 
  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
       $W_{ij} \leftarrow \max(W_{ij}, \min(W_{ik}, W_{kj}))$ 
    od
  od
od

```

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 einem späteren Kapitel eingehen.

Für den Nachweis der Korrektheit des Algorithmus von Warshall besteht die Hauptaufgabe darin, eine Schleifeninvariante für die äußerste (Laufvariable k) der drei ineinander geschachtelten Schleifen 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:

16.6 Lemma. Für alle $i, j \in \mathbb{Z}_n$ 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{Z}_k haben (also Nummern, die echt kleiner als k sind).

Hat man erst einmal nachgewiesen, dass das tatsächlich Schleifeninvariante ist, ist die Korrektheit des gesamten Algorithmus schnell bewiesen. Denn dann gilt insbesondere nach Beendigung der Schleife, also nach n Schleifendurchläufen:

- Für alle $i, j \in \mathbb{Z}_n$ gilt: Nach n Schleifendurchläufen ist W_{ij} genau dann 1, wenn es einen wiederholungsfreien Pfad von i nach j gibt, bei dem alle Zwischenknoten Nummern in \mathbb{Z}_n haben, wenn also überhaupt ein Pfad existiert (denn andere Knotennummern gibt es gar nicht).

16.7 Beweis. (von Lemma 16.6)

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 also keine Knoten außer dem ersten und dem letzten 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 $i, j \in \mathbb{Z}_n$ gilt: Nach $k - 1$ Durchläufen der äußeren Schleife des Algorithmus von Warshall ist W_{ij} genau dann 1, wenn es einen wiederholungsfreien Pfad von i nach j gibt, bei dem alle Zwischenknoten Nummern haben, die in \mathbb{Z}_{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:

—→: Es sei $W_{ij}^{[k]} = 1$. Dann hat also mindestens eine der folgenden Bedingungen zugefallen:

- $W_{ij}^{[k-1]} = 1$: In diesem Fall existiert ein Pfad, dessen Zwischenknoten alle Nummern in \mathbb{Z}_{k-1} haben, und das ist auch einer, dessen Zwischenknoten alle Nummern in \mathbb{Z}_k haben.
- $W_{ik}^{[k-1]} = 1$ und $W_{kj}^{[k-1]} = 1$. Dann existieren Pfade von i nach k und von k nach j , deren Zwischenknoten alle Nummern in \mathbb{Z}_{k-1} sind. Wenn man die Pfade zusammensetzt, erhält man einen Pfad von i nach j , dessen Zwischenknoten alle Nummern in \mathbb{Z}_k haben. Durch

Entfernen von Zyklen kann man auch einen entsprechenden Pfad konstruieren, der wiederholungsfrei ist.

\Leftarrow : Es gebe einen wiederholungsfreien Pfad p von i nach j , dessen Zwischenknoten alle Nummern in \mathbb{Z}_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{Z}_{k-1} . Das gleiche gilt für das Endstück von p , das von $k - 1$ nach j führt. Nach Induktionsvoraussetzung sind also $W_{i,k-1}^{[k-1]} = 1$ und $W_{k-1,j}^{[k-1]} = 1$. Folglich wird im k -ten Durchlauf $W_{ij}^{[k]} = 1$ gesetzt.

- Kein Zwischenknoten in p hat Nummer $k - 1$:

Dann sind die Nummern der Zwischenknoten alle in \mathbb{Z}_{k-1} . Nach Induktionsvoraussetzung ist folglich $W_{ij}^{[k-1]} = 1$ und daher auch $W_{ij}^{[k]} = 1$. ■

16.5 AUSBLICK

Wir sind in diesem Kapitel davon ausgegangen, dass die Matrizen auf die naheliegende Weise miteinander multipliziert werden. Im nächsten Kapitel 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 I“ zu sehen bekommen.

LITERATUR

Kleene, Stephen C. (1956). „Representation of Events in Nerve Nets and Finite Automata“. In: *Automata Studies*. Hrsg. von Claude E. Shannon und John McCarthy. Princeton University Press. Kap. 1, S. 3–40.

Eine Vorversion ist online verfügbar; siehe http://www.rand.org/pubs/research_memoranda/2008/RM704.pdf (10.1.2020) (siehe S. 174, 209).

Warshall, Stephen (1962). „A Theorem on Boolean Matrices“. In: *Journal of the ACM* 9, S. 11–12 (siehe S. 173).

17 QUANTITATIVE ASPEKTE VON ALGORITHMEN

17.1 RESSOURCENVERBRAUCH BEI BERECHNUNGEN

Wir haben in [Kapitel 16 mit ersten Graphalgorithmen](#) damit begonnen, festzustellen, wieviele arithmetische Operationen bei der Ausführung eines Algorithmus für eine konkrete Probleminstance 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 dieses Kapitels 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 [Kapitel 14 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 [16.2](#) zur Bestimmung der Wegematrix eines Graphen hatte auch nichts anderes als die Größe 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 [17.1](#) noch mal aufgeschrieben haben. Wie oft die **for**-Schleife in der Methode *insert* ausgeführt wird, hängt nicht einfach von der Problemgröße $n = a.length$ ab. Es hängt auch von der konkreten Probleminstance ab. Selbst bei gleicher Zahl n kann die Schleife unterschiedlich oft durchlaufen werden: Ist das Array a von Anfang an sortiert, wird die **for**-Schleife überhaupt nicht ausgeführt. Ist es

genau in entgegengesetzter Richtung sortiert, wird ihr Schleifenrumpf insgesamt $\sum_{i=1}^{n-1} i = n(n-1)/2$ mal ausgeführt.

```
public class InsertionSort {
    public static void sort(long[] a) {
        for (int i ← 1; i < a.length; i++) {
            insert(a, i);
        }
    }
    private static void insert(long[] a, int idx) {
        // Tausche a[idx] nach links bis es einsortiert ist
        for (int i ← idx; i > 0 and a[i-1] > a[i]; i--) {
            long tmp ← a[i-1];
            a[i-1] ← a[i];
            a[i] ← tmp;
        }
    }
}
```

Algorithmus 17.1: Insertionsort aus der Vorlesung „Programmieren“

Meistens ist es aber so, dass man nicht für jede Probleminstanz einzeln angeben will oder kann, wie lange ein Algorithmus zu seiner Lösung benötigt. Man beschränkt sich auf eine vergrößernde Sichtweise und beschreibt z. B. die Laufzeit nur in Abhängigkeit von der Problemgröße n . Es bleibt die Frage zu klären, was man dann angibt, wenn die Laufzeit für verschiedene Instanzen gleicher Größe variiert: Den Durchschnitt? Den schnellsten Fall? Den langsamsten Fall?

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.

17.2 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 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 einem späteren Kapitel 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 sind und trotzdem noch aussagekräftig ist.

17.2.1 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 *ohne* 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:

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

$f \asymp g$ Wir schreiben in diesem Fall auch $f \asymp g$ oder $f(n) \asymp g(n)$. Zum Beispiel gilt $3n^2 \asymp 10^{-2}n^2$. Denn einerseits gilt für $c = 10^{-3}$ und $n_0 = 0$:

$$\forall n \geq n_0 : cf(n) = 10^{-3} \cdot 3n^2 \leq 10^{-2}n^2 = g(n)$$

Andererseits gilt z. B. für $c' = 1$ und $n_0 = 0$:

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

Die eben durchgeführte Rechnung lässt sich leicht etwas allgemeiner durchführen. Dann zeigt sich, dass man festhalten kann:

17.1 (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.

Mitunter ist eine graphische Darstellung nützlich. In Abbildung 17.1 sieht man zweimal die gleiche Funktion $f(x)$ für den gleichen Argumentbereich geplottet. Auf der linken Seite sind beide Achsen linear skaliert. Auf der rechten Seite ist die y -Achse logarithmisch skaliert. In einer solchen Darstellung erhält man den Graph für $cf(x)$ aus dem für $f(x)$ durch Verschieben um $\log(c)$ nach oben. Für eine Funktion $g(x)$ mit $g(x) \asymp f(x)$ muss gelten, dass, von endlich vielen Ausnahmen abgesehen, für „fast“ alle $n \in \mathbb{N}_0$ gilt: $cf(n) \leq g(n) \leq c'f(n)$. Auf die graphische Darstellung übertragen bedeutet das, dass der Graph für g fast überall

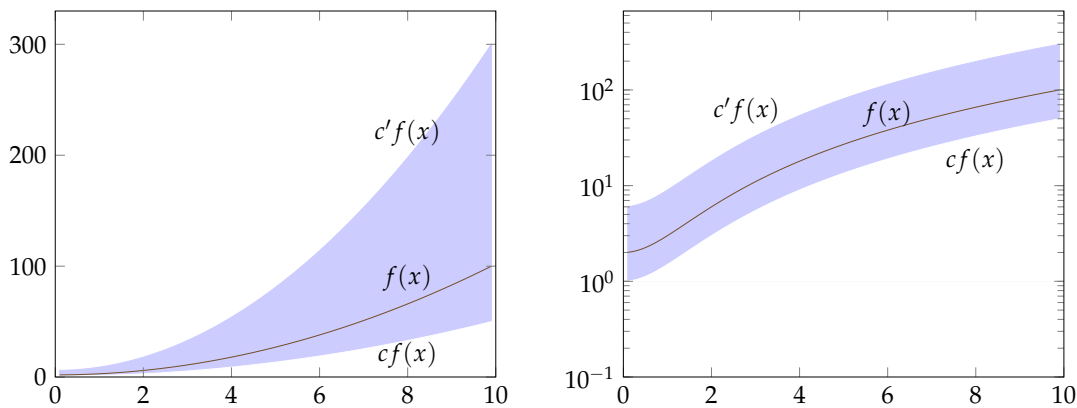


Abbildung 17.1: Zweimal die gleiche Funktion $f(x)$ und zwei Schranken $cf(x)$ und $c'f(x)$; links Darstellung mit linear skaliertem y -Achse, rechts mit logarithmisch skaliertem y -Achse

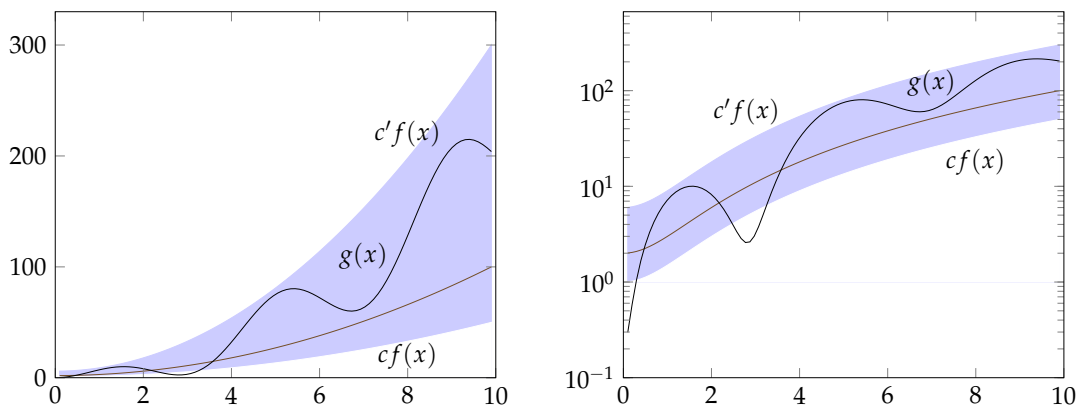


Abbildung 17.2: Zweimal die gleiche Funktion $g(x)$, die für $n \geq 4$ durch $cf(n)$ und $c'f(n)$ beschränkt ist.

in dem farblich hinterlegten „Schlauch“ um f liegen muss. In [Abbildung 17.2](#) ist ein Beispiel dargestellt.

Wer mit Grenzwerten schon vertraut ist, dem wird auch klar sein, dass z. B. nicht $n^2 \asymp 2^n$ ist: Da $\lim_{n \rightarrow \infty} 2^n / n^2 = \infty$ ist, ist offensichtlich *nicht* für alle großen n die Ungleichung $2^n \leq c'n^2$ erfüllbar. Man kann sich das aber auch „zu Fuß“ überlegen: Für die Folge von Zahlen $n_i = 2^{i+2}$ gilt:

$$\forall i \in \mathbb{N}_0 : 2^{n_i} \geq 4^i n_i^2$$

Der Induktionsanfang ist klar und es ist

$$2^{n_{i+1}} = 2^{2n_i} = 2^{n_i} \cdot 2^{n_i} \geq 4^i n_i^2 \cdot 4^{n_i/2} \geq 4^i n_i^2 \cdot 16 = 4^i \cdot 4 \cdot 4n_i^2 = 4^{i+1} \cdot n_{i+1}^2.$$

Also kann nicht für alle großen n gelten: $2^n \leq cn^2$.

Das Zeichen \asymp erinnert an das Gleichheitszeichen. Das ist auch bewusst so gemacht, denn die Relation \asymp hat wichtige Eigenschaften, die auch auf Gleichheit zutreffen:

17.2 Lemma. Die Relation \asymp ist eine Äquivalenzrelation.

17.3 Beweis. Wir überprüfen die drei definierenden Eigenschaften von Äquivalenzrelationen (siehe Abschnitt 15.2.1).

- *Reflexivität:* Es ist stets $f \asymp f$, denn wenn man $c = c' = 1$ und $n_0 = 0$ wählt, dann gilt für $n \geq n_0$ offensichtlich $cf(n) \leq f(n) \leq c'f(n)$.
- *Symmetrie:* Wenn $f \asymp g$ gilt, dann auch $g \asymp f$: Denn wenn für positive Konstanten c, c' und alle $n \geq n_0$

$$cf(n) \leq g(n) \leq c'f(n)$$

gilt, dann gilt für die gleichen $n \geq n_0$ und die ebenfalls positiven Konstanten $d = 1/c$ und $d' = 1/c'$:

$$d'g(n) \leq f(n) \leq dg(n)$$

- *Transitivität:* Wenn $f \asymp g$ ist, und $g \asymp h$, dann ist auch $f \asymp h$: Es gelte für Konstanten $c, c' \in \mathbb{R}_+$ und alle $n \geq n_0$

$$cf(n) \leq g(n) \leq c'f(n)$$

und analog für Konstanten $d, d' \in \mathbb{R}_+$ und alle $n \geq n_1$

$$dg(n) \leq h(n) \leq d'g(n) .$$

Dann gilt für alle $n \geq \max(n_0, n_1)$

$$dcf(n) \leq dg(n) \leq h(n) \leq d'g(n) \leq d'c'f(n) ,$$

wobei auch die Konstanten dc und $d'c'$ wieder positiv sind. ■

Es ist üblich, für die Menge aller Funktionen, die zu einer gegebenen Funktion $f(n)$ im Sinne von \asymp äquivalent sind, $\Theta(f)$ bzw. $\Theta(f(n))$ zu schreiben. Also:

$$\begin{aligned} \Theta(f) &= \{g \mid f \asymp g\} \\ &= \{g \mid \exists c, c' \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N}_0 : \forall n \geq n_0 : cf(n) \leq g(n) \leq c'f(n)\} \end{aligned}$$

Aus Rechenregel 17.1 wird bei Verwendung von Θ :

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

Zum Beispiel ist $\Theta(3n^2) = \Theta(8n^2)$.

17.2.2 Notation für obere und untere Schranken des Wachstums

In Fällen wie der unbekannten Anzahl von Durchläufen der **for**-Schleife in der Funktion *insert* in Algorithmus 17.1 genügt es nicht, wenn man konstante Faktoren ignorieren kann. Man kennt nur den schlimmsten Fall: $\sum_{i=1}^{n-1} i = n(n-1)/2$. Dementsprechend ist im schlimmsten Fall die Laufzeit in $\Theta(n^2)$; im allgemeinen kann sie aber auch kleiner sein. Um das bequem ausdrücken und weiterhin konstante Faktoren ignorieren zu können, definiert man:

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

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

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

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.

- Mitunter trifft man auch die Schreibweise $O(1)$ an. Was ist das? Die 1 steht hier für die Funktion, die konstant 1 ist. Die Definition sagt, dass $O(1)$ alle Funktionen g 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:

$O(1)$

$$g(n) \leq c \cdot 1 = c$$

Das sind also alle Funktionen, die man durch Konstanten beschränken kann. Dazu gehören etwa alle konstanten Funktionen, aber auch Funktionen wie $3 + \sin$. (So etwas habe ich aber noch nie eine Rolle spielen sehen.)

- Weiter vorne hatten wir benutzt, dass der Quotient n^2/n nicht für alle hinreichend großen n durch eine Konstante beschränkt werden kann. Also gilt *nicht* $n^2 \preceq n$. Andererseits gilt (machen Sie sich das bitte kurz klar) $n \preceq n^2$. Die Relation \preceq ist also *nicht* symmetrisch.

Allgemein gilt für positive reelle Konstanten $0 < a < b$, dass $n^a \preceq n^b$ ist, aber *nicht* $n^b \preceq n^a$. Ebenso ist für reelle Konstanten a und b , die beide echt größer 1 sind, $n^a \preceq b^n$ aber *nicht* $b^n \preceq n^a$.

Um vom unterschiedlichen Wachstum einiger Funktionen einen groben Eindruck zu bekommen, genügt es, sich eine Tabelle mit einigen Funktionswerten anzusehen:

$\log_2 n$	1	2	3	4	5	6
n	2	4	8	16	32	64
n^2	4	16	64	256	1024	4096
n^3	8	64	512	4096	32768	262144
2^n	2	4	8	16	32	64

Man muss nur in der Ungleichung $g(n) \leq cf(n)$ die (positive!) Konstante c auf die andere Seite bringen und schon kann man sich davon überzeugen, dass gilt:

17.5 (Rechenregel) Für alle Funktionen $f : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ und $g : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ gilt:

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

Man kann auch zeigen:

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

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

17.2.3 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

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

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

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

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 $g = O(f_1)$ und $g = O(f_2)$ zu folgern, dass $O(f_1) = O(f_2)$ ist.
- Es ist *falsch*, aus $g_1 = O(f)$ und $g_2 = O(f)$ zu folgern, dass $g_1 = g_2$ 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*. Lassen Sie sich niemals von etwas wie „ $n^2 = O(n^3)$ “ irritieren.

17.2.4 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:

17.6 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.

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

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

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

oder bei Funktionenmengen

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

Solche Komplexoperationen sind übrigens nichts Neues für Sie. Die Definition des Produkts formaler Sprachen passt genau in dieses Schema (siehe Unterabschnitt 7.1.1).

17.7 Beweis. (von Lemma 17.6) 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 $g_1 \leq g$ und folglich $g_2 = g - g_1$ stets größer gleich 0.

Behauptung: $g_2 \in O(f_2)$. Sei $n \geq n_0$. Dann ist

$$\begin{aligned}
 g_2(n) &= g(n) - g_1(n) \\
 &= \begin{cases} 0 & \text{falls } g(n) \leq cf_1(n) \\ g(n) - cf_1(n) & \text{falls } g(n) > cf_1(n) \end{cases} \\
 &\leq \begin{cases} 0 & \text{falls } g(n) \leq cf_1(n) \\ c(f_1(n) + f_2(n)) - cf_1(n) & \text{falls } g(n) > cf_1(n) \end{cases} \\
 &= \begin{cases} 0 & \text{falls } g(n) \leq cf_1(n) \\ cf_2(n) & \text{falls } g(n) > cf_1(n) \end{cases} \\
 &\leq cf_2(n),
 \end{aligned}$$

also $g_2 \in O(f_2)$. Also ist $g = g_1 + g_2 \in O(f_1) + O(f_2)$. ■

17.8 (Rechenregel) Wenn $g_1 \preceq f_1$ ist, und wenn $g_1 \asymp g_2$ und $f_1 \asymp f_2$, dann gilt auch $g_2 \preceq f_2$.

17.9 (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 17.6). Wir verzichten hier darauf, sie alle aufzuzählen.

17.3 MATRIZENMULTIPLIKATION

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} elementarer Additionen ist und
- die Anzahl N_{mult} elementarer Multiplikationen

interessieren. Deren Summe bestimmt im wesentlichen (d.h. bis auf konstante Faktoren) die Laufzeit.

17.3.1 Rückblick auf die Schulmethode

Die „Schulmethode“ für die Multiplikation von 2×2 -Matrizen geht so:

		b_{11}	b_{12}
		b_{21}	b_{22}
a_{11}	a_{12}	$a_{11}b_{11} + a_{12}b_{21}$	$a_{11}b_{12} + a_{12}b_{22}$
a_{21}	a_{22}	$a_{21}b_{11} + a_{22}b_{21}$	$a_{21}b_{12} + a_{22}b_{22}$

Wie man sieht ist dabei

- $N_{\text{mult}}(2) = 2^2 \cdot 2 = 8$ und
- $N_{\text{add}}(2) = 2^2 \cdot (2 - 1) = 4$.

Wenn n gerade ist (auf diesen Fall wollen uns im folgenden der einfacheren Argumentation wegen beschränken), dann ist die Schulmethode für $n \times n$ Matrizen äquivalent zum Fall, dass man 2×2 Blockmatrizen mit Blöcken der Größe $n/2$ vorliegen hat, die man nach dem gleichen Schema wie oben multiplizieren kann:

		B_{11}	B_{12}
		B_{21}	B_{22}
A_{11}	A_{12}	$A_{11}B_{11} + A_{12}B_{21}$	$A_{11}B_{12} + A_{12}B_{22}$
A_{21}	A_{22}	$A_{21}B_{11} + A_{22}B_{21}$	$A_{21}B_{12} + A_{22}B_{22}$

Das sind 4 Additionen von Blockmatrizen und 8 Multiplikationen von Blockmatrizen. Die Anzahl elementarer Operationen ist also

- $N_{\text{mult}}(n) = 8 \cdot N_{\text{mult}}(n/2)$ und
- $N_{\text{add}}(n) = 8 \cdot N_{\text{add}}(n/2) + 4 \cdot (n/2)^2 = 8 \cdot N_{\text{add}}(n/2) + n^2$.

Wir betrachten den Fall $n = 2^k$ (die anderen Fälle gehen im Prinzip ähnlich). Dann ergibt sich aus $N_{\text{mult}}(n) = 8 \cdot N_{\text{mult}}(n/2)$:

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

Dass man statt der Pünktchen einen Induktionsbeweis führen kann, ist inzwischen klar, oder?

Aus $N_{\text{add}}(n) = 8 \cdot N_{\text{add}}(n/2) + n^2$ ergibt sich analog:

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

17.3.2 Algorithmus von Strassen

Nun kommen wir zu der Idee von Strassen (1969). Er hat bemerkt, dass man die Blockmatrizen C_{ij} des Matrizenproduktes auch wie folgt berechnen kann:

$$\begin{aligned}M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\M_2 &= (A_{21} + A_{22})B_{11} \\M_3 &= A_{11}(B_{12} - B_{22}) \\M_4 &= A_{22}(B_{21} - B_{11}) \\M_5 &= (A_{11} + A_{12})B_{22} \\M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})\end{aligned}$$

und dann

$$\begin{aligned}C_{11} &= M_1 + M_4 - M_5 + M_7 \\C_{12} &= M_3 + M_5 \\C_{21} &= M_2 + M_4 \\C_{22} &= M_1 - M_2 + M_3 + M_6\end{aligned}$$

Das sieht erst einmal umständlicher aus, denn es sind 18 Additionen von Blockmatrizen statt nur 4 bei der Schulmethode. Aber es sind nur 7 Multiplikationen von Blockmatrizen statt 8! Und das zahlt sich aus, denn im Gegensatz zum skalaren Fall sind Multiplikationen aufwändiger als Additionen. Für die Anzahl elementarer Operationen ergibt sich:

- $N_{\text{mult}}(n) = 7 \cdot N_{\text{mult}}(n/2)$
- $N_{\text{add}}(n) = 7 \cdot N_{\text{add}}(n/2) + 18 \cdot (n/2)^2 = 7 \cdot N_{\text{add}}(n/2) + 4.5 \cdot n^2$

Für den Fall $n = 2^k$ ergibt sich:

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

Analog erhält man auch für die Anzahl der Additionen $N_{\text{add}} \in \Theta(n^{\log_2 7})$. Die Gesamtzahl elementarer arithmetischer Operationen ist also in $\Theta(n^{\log_2 7}) + \Theta(n^{\log_2 7}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807...})$.

Es gibt sogar Algorithmen, die asymptotisch noch weniger Operationen benötigen. Das in dieser Hinsicht beste Ergebnis war lange Zeit die Methode von Coppersmith und Winograd (1990), die mit $O(n^{2.376...})$ elementaren arithmetischen Operationen auskommen. Auch dieses Verfahren benutzt wie das von Strassen 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*). Die aktuelle Rekordhalterin für asymptotisch schnelle Matrizenmultiplikation ist Vassilevska Williams (2012).

17.4 ASYMPTOTISCHES VERHALTEN „IMPLIZIT“ DEFINIERTER FUNKTIONEN

Sie werden im Laufe der kommenden Semester viele Algorithmen kennenlernen, bei denen wie bei Strassens Algorithmus für Matrizenmultiplikation 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 kosten. Das beinhaltet auch den unter Umständen erforderlichen Aufwand zum Erzeugen der Teilprobleme.

Dann ergibt sich für Abschätzung (z. B.) der Laufzeit T 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 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 wächst.

Es gibt drei wichtige Fälle, in denen jeweils die Zahl $\log_b a$ eine Rolle spielt:

Fall 1: Wenn $f \in O(n^{\log_b a - \varepsilon})$ für ein $\varepsilon > 0$ ist, dann ist $T \in \Theta(n^{\log_b a})$.

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

Fall 3: Wenn $f \in \Omega(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$ ist, und wenn es eine Konstante d gibt mit $0 < d < 1$, so dass für alle hinreichend großen n gilt $af(n/b) \leq df$, dann ist $T \in \Theta(f)$.

Dass die Aussagen in diesen drei Fällen richtig sind, bezeichnet man manchmal als *Mastertheorem*, obwohl es sich sicherlich um keine sehr tiefeschürfenden Erkennt-

Mastertheorem

nisse handelt.

Betrachten wir als Beispiele noch einmal die Matrizenmultiplikation. 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; es ist also $b = 2$.

Bei der Schulmethode haben wir $a = 8$ Multiplikationen kleinerer Matrizen der Größe $n/2$ durchzuführen. In diesem Fall ist $\log_b a = \log_2 8 = 3$. Der zusätzliche Aufwand besteht in 4 kleinen Matrixadditionen, so dass $f = 4 \cdot n^2/4 = n^2$. Damit ist $f \in O(n^{3-\varepsilon})$ (z. B. für $\varepsilon = 1/2$) und der erste Fall des Mastertheorems besagt, dass folglich $T \in \Theta(n^3)$. (Und das hatten wir uns weiter vorne tatsächlich auch klar gemacht.)

Bei Strassens geschickterer Methode sind nur $a = 7$ Multiplikationen kleinerer Matrizen der Größe $n/2$ durchzuführen (es ist 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 $f = 18 \cdot n^2/4 \in \Theta(n^2)$. Auch hier gilt für ein geeignetes ε wieder $f \in O(n^{\log_b a - \varepsilon}) = O(n^{\log_2 7 - \varepsilon})$. Folglich benötigt Strassens Algorithmus $T \in \Theta(n^{\log_2 7}) = \Theta(n^{2.807\dots})$ Zeit.

17.5 UNTERSCHIEDLICHES WACHSTUM EINIGER FUNKTIONEN

Wir hatten schon darauf hingewiesen, dass gilt:

1. Für positive reelle Konstanten $0 < a < b$ ist $n^a \preceq n^b$, aber *nicht* $n^b \preceq n^a$.
2. Für reelle Konstanten a und b , die beide echt größer 1 sind, gilt $n^a \preceq b^n$ aber *nicht* $b^n \preceq n^a$.

Zur Veranschaulichung des ersten Punktes sind in Abbildung 17.3 die Funktionen $f(x) = x$, $f(x) = x^2$ und $f(x) = x^3$ geplottet. Allerdings fällt in der gewählten Darstellung $f(x) = x$ nahezu mit der x -Achse zusammen. Wie man sieht, wird in doppelt-logarithmischen Plots jede Funktion x^d durch eine Gerade repräsentiert (deren Steigung d ist, wenn die „Einheiten“ auf beiden Achsen gleich sind). Allgemeine Polynomfunktionen werden durch Linien repräsentiert, die sich für große n einer Geraden anschmiegen.

Abbildung 17.4 zeigt in doppelt-logarithmischer Darstellung zum Vergleich zwei Polynom- und zwei Exponentialfunktionen. Der Unterschied sollte klar sein.

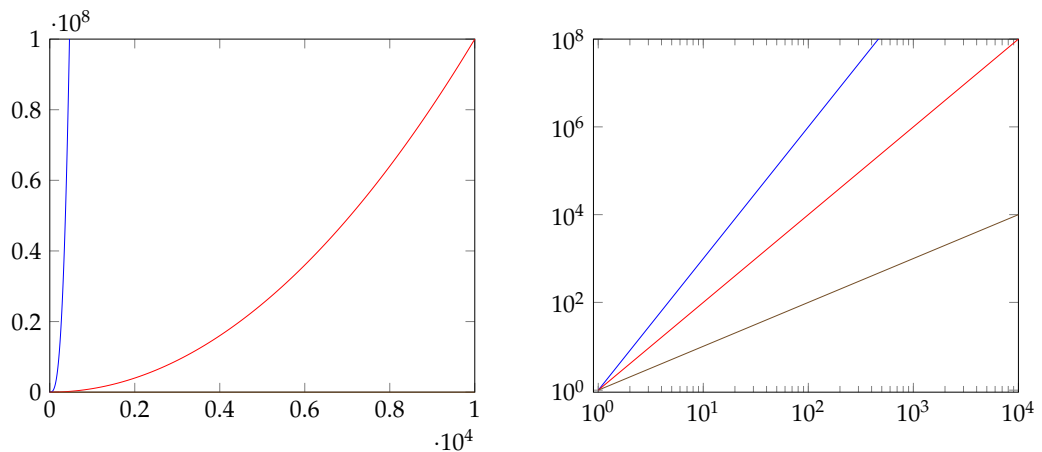


Abbildung 17.3: Die Funktionen $f(x) = x$, $f(x) = x^2$ und $f(x) = x^3$ in Plots mit linear skalierten Achsen (links) und in doppelt logarithmischer Darstellung (rechts); auf der linken Seite ist $f(x) = x$ praktisch nicht von der x -Achse zu unterscheiden.

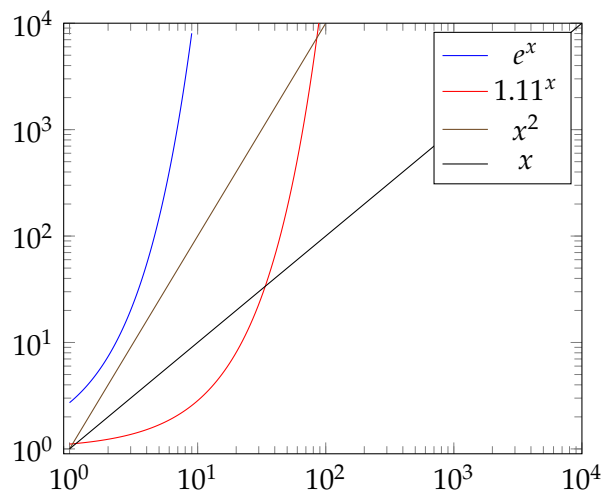


Abbildung 17.4: Zwei Polynom- und zwei Exponentialfunktionen im Vergleich; doppelt-logarithmische Darstellung.

17.6 AUSBLICK

Algorithmen, bei denen die anderen beiden Fälle des Mastertheorems zum Tragen kommen, werden Sie im kommenden Semester in der Vorlesung „Algorithmen 1“ kennenlernen.

Manchmal wird „Teile und Herrsche“ auch in etwas komplizierterer Form an-

gewendet (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 (siehe S. 189).
- Graham, Ronald L., Donald E. Knuth und Oren Patashnik (1989). *Concrete Mathematics*. Addison-Wesley (siehe S. 185).
- Strassen, Volker (1969). „Gaussian Elimination Is Not Optimal“. In: *Numerische Mathematik* 14, S. 354–356 (siehe S. 189).
- Vassilevska Williams, Virginia (2012). „Multiplying matrices faster than Coppersmith-Winograd“. In: *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*. Hrsg. von Howard J. Karloff und Toniann Pitassi. ACM, S. 887–898. ISBN: 978-1-4503-1245-5 (siehe S. 190).

18 ENDLICHE AUTOMATEN

18.1 ERSTES BEISPIEL: EIN GETRÄNKEAUTOMAT

Als erstes Beispiel betrachten wir den folgenden primitiven Getränkeautomaten (siehe Abbildung 18.1). 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 wird das gewünschte Getränk ausgeworfen.

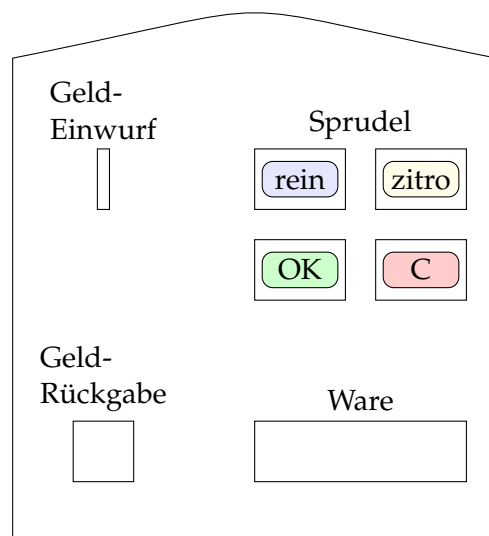


Abbildung 18.1: Ein primitiver Getränkeautomat

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 2.3) speichern. Und zwar

- zum einen, ob schon ein 1-Euro-Stück eingeworfen wurde, und
- zum anderen, ob schon ein Getränk ausgewählt wurde und wenn ja, welches.

Man kann das zum Beispiel modellieren durch Paare (x, y) , bei denen die Komponente $x \in \{0, 1\}$ den schon eingeworfenen Geldbetrag angibt und Komponente $y \in \{-, R, Z\}$ 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 18.2.

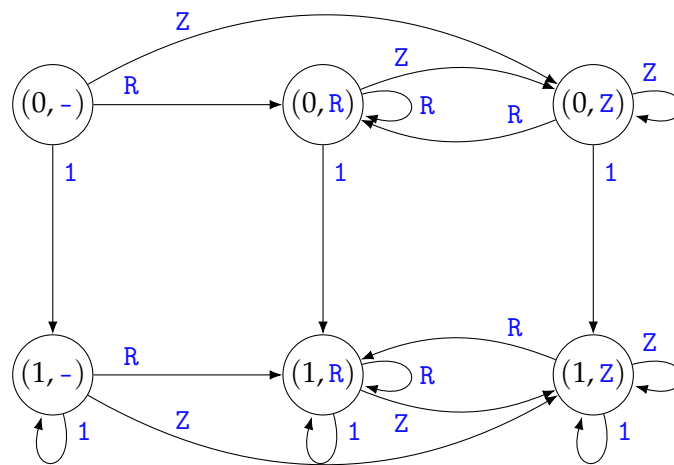


Abbildung 18.2: Graphische Darstellung der Zustandsübergänge des Getränkeautomaten für die drei Eingabesymbole $1, R$ und Z .

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 18.2 zunächst einmal nur die Zustandsübergänge für die Eingabesymbole 1 , R und Z dargestellt. Hinzu kommen noch die aus Abbildung 18.3 für die Eingaben C und 0 . Wenn bei einem Zustand für mehrere Eingabesymbole der Nachfolgezustand der gleiche ist, dann zeichnet man oft nur einen Pfeil und beschriftet ihn mit allen Eingabesymbolen, durch Kommata getrennt. In Abbildung 18.3 betrifft das den Übergang von Zustand $(1, R)$ nach Zustand $(0, -)$ für die Eingaben 0 und C (und analog von den Zuständen $(1, Z)$ und $(0, -)$).

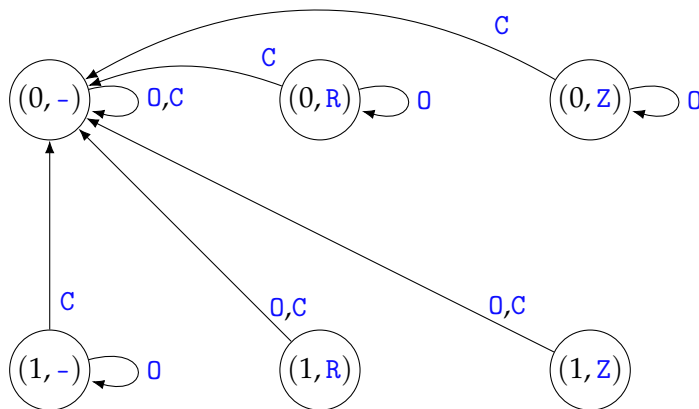


Abbildung 18.3: Graphische Darstellung der Zustandsübergänge des Getränkeautomaten für die Eingabesymbole C und 0 .

Stellt man alle Übergänge in einem Diagramm dar, ergibt sich Abbildung 18.4.

Der zweite wichtige Aspekt jedes Automaten ist, dass sich seine Arbeit, im vorliegenden Fall also die Zustandsübergänge, zumindest von Zeit zu Zeit in irgendeiner Weise auf seine Umwelt auswirken (warum sollte man ihn sonst arbeiten lassen). Beim Getränkeautomaten zeigt sich das in der Ausgabe von Geldstücken und Getränkeflaschen. Dazu sehen wir eine Menge $Y = \{1, R, Z\}$ von Ausgabesymbolen vor, deren Bedeutung klar sein sollte. Beim Getränkeautomaten ist es plausibel zu sagen, dass jedes Paar (z, x) von aktuellem Zustand z und aktueller Eingabe x eindeutig einen neuen Zustand festlegt, es ebenso eindeutig eine Ausgabe festlegt. Wir formalisieren das als eine Funktion $g : Z \times X \rightarrow Y^*$. 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 Eingabesym-

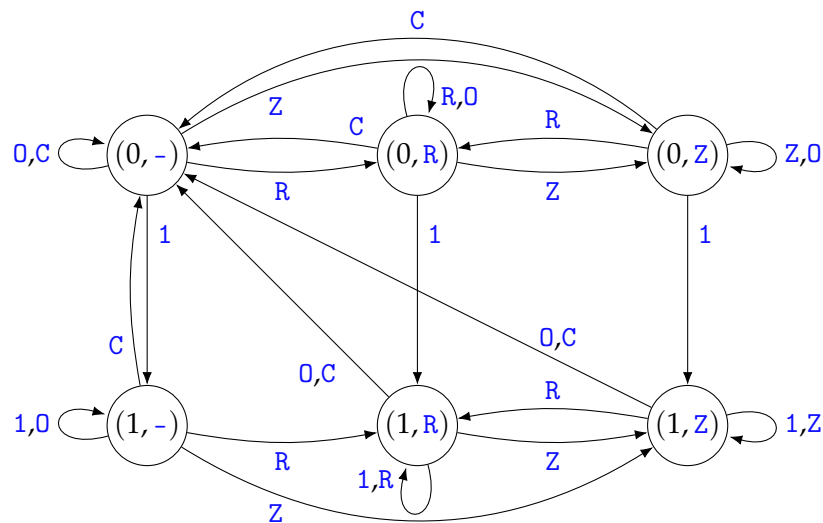


Abbildung 18.4: Graphische Darstellung der Zustandsübergänge des Getränkeautomaten für alle Eingabesymbole.

bol, von diesem durch einen senkrechten Strich getrennt (manche nehmen auch ein Komma). Aus Abbildung 18.4 ergibt sich Abbildung 18.5.

18.2 MEALY-AUTOMATEN

Mealy-Automat

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

- eine endliche Zustandsmenge Z ,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X ,
- eine 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 (ein-

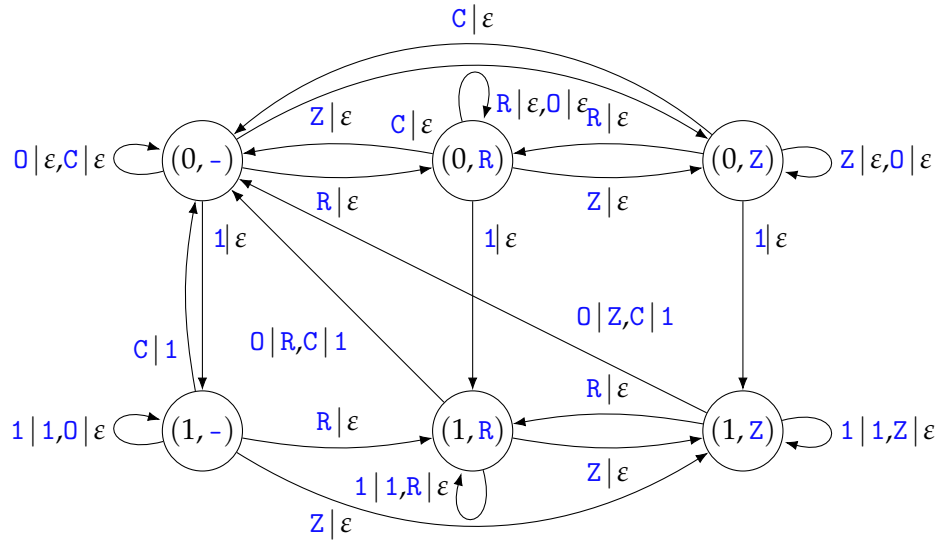


Abbildung 18.5: Graphische Darstellung der Zustandsübergänge und Ausgaben des Getränkeautomaten für alle Eingabesymbole.

schließlich des Anfangszustands) reden. Und manchmal will man auch bei den Ausgaben über allgemeinere Aspekte sprechen.

Um das bequem hinzuschreiben zu können, definieren wir Abbildungen f_* und f_{**} und analog g_* und g_{**} . Dabei soll der erste Stern andeuten, dass zweites Argument nicht ein einzelnes Eingabesymbol sondern ein ganzes Wort von Eingabesymbolen ist; und der zweite Stern soll gegebenenfalls andeuten, dass wir uns nicht für einen einzelnen Funktionswert (von f bzw. g) interessieren, sondern wiederum für ein ganzes Wort von ihnen. Als erstes legen wir $f_* : Z \times X^* \rightarrow Z$ fest:

$$f_*(z, \varepsilon) = z$$

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

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 $f_{**} : Z \times X^* \rightarrow Z^*$ für alle $z \in Z$ wie folgt fest:

$$\begin{aligned} f_{**}(z, \varepsilon) &= z \\ \forall w \in X^* : \forall x \in X : \quad f_{**}(z, wx) &= f_{**}(z, w) \cdot f_*(z, wx) \end{aligned}$$

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. Zuerst definieren wir die Funktion $g_* : Z \times X^* \rightarrow Y^*$, deren Funktionswert die zum letzten Eingabesymbol gehörende Ausgabe sein soll. Das geht für alle $z \in Z$ so:

$$\begin{aligned} g_*(z, \varepsilon) &= \varepsilon \\ \forall w \in X^* : \forall x \in X : \quad g_*(z, wx) &= g_*(f_*(z, w), x) \end{aligned}$$

Um auch über die Konkatenation der zu allen Eingabesymbolen gehörenden Ausgaben reden zu können, definieren wir die Funktion $g_{**} : Z \times X^* \rightarrow Y^*$ für alle $z \in Z$ wie folgt:

$$\begin{aligned} g_{**}(z, \varepsilon) &= \varepsilon \\ \forall w \in X^* : \forall x \in X : \quad g_{**}(z, wx) &= g_{**}(z, w) \cdot g_*(z, wx) \end{aligned}$$

18.3 MOORE-AUTOMATEN

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 $h : Z \rightarrow Y^*$

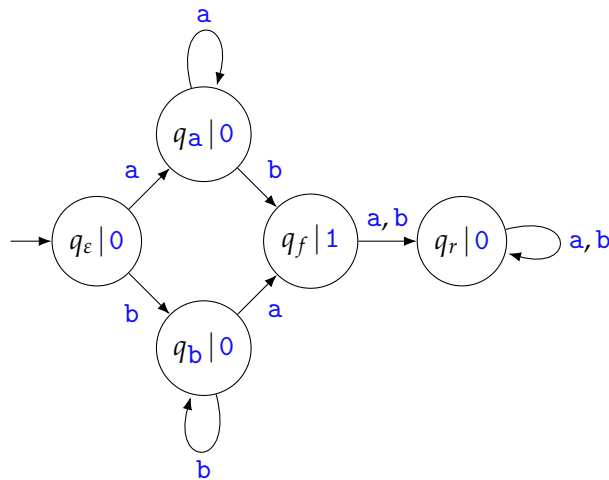


Abbildung 18.6: Ein einfacher Moore-Automat (aus der Dokumentation des \LaTeX -Pakets `tikz`; modifiziert)

Als einfaches Beispiel betrachten wir den Automaten in Abbildung 18.6 mit 5 Zuständen, Eingabealphabet $X = \{a, b\}$ und Ausgabealphabet $Y = \{0, 1\}$.

In jedem Knoten des Graphen sind jeweils ein Zustand z und, wieder durch einen senkrechten Strich getrennt, die zugehörige Ausgabe $h(z)$ notiert.

Die Definitionen für f_* und f_{**} kann man ohne Änderung von Mealy- zu Moore-Automaten übernehmen. Zum Beispiel ist im obigen Beispiel $f_*(q_\epsilon, \text{aaaba}) = q_r$, denn bei Eingabe `aaaba` durchläuft der Automat ausgehend von q_ϵ nacheinander die Zustände

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

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

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 Konkatination aller Ausgaben“ waren.

Entsprechendes kann man natürlich auch bei Moore-Automaten festlegen. Die Definitionen fallen etwas einfacher aus als bei Mealy-Automaten. Zum Beispiel ist $g_* : Z \times X^* \rightarrow Y^*$ einfach hinzuschreiben als $g_*(z, w) = h(f_*(z, w))$ (für alle $(z, w) \in Z \times X^*$). Also kurz: $g_* = h \circ f_*$.

Im obigen Beispielautomaten ist etwa

$$g_*(q_\epsilon, \text{aaaba}) = h(f_*(q_\epsilon, \text{aaaba})) = h(q_r) = 0$$

das zuletzt ausgegebene Bit, wenn man vom Startzustand ausgehend **aaaba** eingibt.

g_{**} 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 8.3.2 kennengelernt haben. 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 $g_{**} = h^{**} \circ f_{**}$.

In unserem Beispiel ist

$$\begin{aligned} g_{**}(q_\epsilon, \text{aaaba}) &= h^{**}(f_{**}(q_\epsilon, \text{aaaba})) \\ &= h^{**}(q_\epsilon q_a q_a q_a q_f q_r) \\ &= h(q_\epsilon)h(q_a)h(q_a)h(q_a)h(q_f)h(q_r) \\ &= \text{000010} \end{aligned}$$

18.4 ENDLICHE AKZEPTOREN

Ein besonders wichtiger Sonderfall endlicher Moore-Automaten sind sogenannte endliche Akzeptoren. Unser Beispiel im vorangegangenen Abschnitt war bereits einer.

Die Ausgabe ist bei einem Akzeptor immer nur ein Bit, das man interpretiert als die Mitteilung, dass die Eingabe „gut“ oder „schlecht“ war, oder mit anderen Worten „syntaktisch korrekt“ oder „syntaktisch falsch“ (für eine gerade interessierende Syntax). Formal ist bei einem endlichen Akzeptor also $Y = \{0, 1\}$ und $\forall z : h(z) \in Y$. Man macht es sich dann üblicherweise noch etwas einfacher, und schreibt statt der Funktion h einfach die Teilmenge der sogenannten *akzeptierenden Zustände* auf. Damit ist $F = \{z \mid h(z) = 1\} \subseteq Z$ gemeint. Zustände, die nicht akzeptierend sind, heißen auch *ablehnend*.

akzeptierender Zustand
ablehnender Zustand
endlicher Akzeptor

Ein *endlicher Akzeptor* $A = (Z, z_0, X, f, F)$ ist also festgelegt durch

- eine endliche Zustandsmenge Z ,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X ,
- eine Zustandsüberföhrungsfunktion $f : Z \times X \rightarrow Z$,
- eine Menge $F \subseteq Z$ akzeptierender Zustände

In graphischen Darstellungen werden die akzeptierenden Zustände üblicherweise durch doppelte Kringel statt einfacher gekennzeichnet. Abbildung 18.7 zeigt „den gleichen“ Automaten wie Abbildung 18.6, nur in der eben beschriebenen Form dargestellt. Es ist $F = \{q_f\}$, weil q_f der einzige Zustand mit Ausgabe 1 ist.

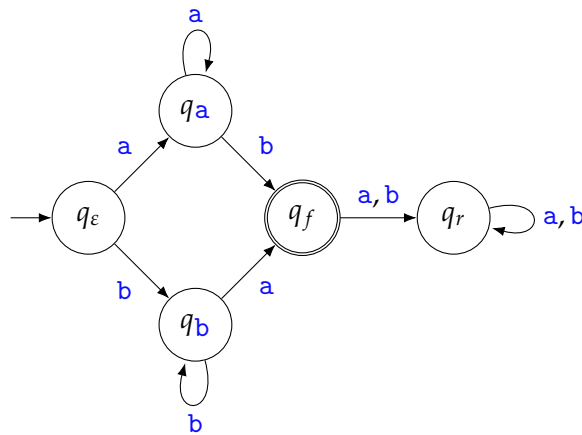


Abbildung 18.7: Ein einfacher Akzeptor (aus der Dokumentation des \LaTeX -Pakets `tikz`; modifiziert)

18.4.1 Beispiele formaler Sprachen, die von endlichen Akzeptoren akzeptiert werden können

Man sagt, ein Wort $w \in X^*$ werde *akzeptiert*, falls $f_*(z_0, w) \in F$ ist, d. h. wenn man ausgehend vom Anfangszustand bei Eingabe von w in einem akzeptierenden Zustand endet. Wird ein Wort nicht akzeptiert, dann sagt man, dass es *abgelehnt* wird. Das schon mehrfach betrachtete Wort **aaaba** wird also abgelehnt, weil $f_*(z_0, \text{aaaba}) = q_r \notin F$ 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^kb für ein $k \in \mathbb{N}_+$. Und es werden auch alle Wörter akzeptiert, die von der Form b^ka 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

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

denn außer den oben genannten Wörtern werden keine anderen akzeptiert. Das kann man sich klar machen, in dem man überlegt,

- dass Wörter ohne ein **b** oder ohne ein **a** abgelehnt werden
- dass Wörter, die sowohl mindestens zwei **a** als auch mindestens zwei **b** enthalten, abgelehnt werden, und

- dass Wörter abgelehnt werden, die z. B. nur genau ein **a** enthalten, aber sowohl davor als auch dahinter mindestens ein **b**, bzw. umgekehrt.

Eine im Alltag vorkommende Aufgabe besteht darin, aus einer Textdatei diejenigen Zeilen zu extrahieren und z. B. auszugeben, in denen ein gewisses Wort vorkommt (und alle anderen Zeilen zu ignorieren). Jede Zeile der Textdatei ist eine Zeichenkette w , die darauf hin untersucht werden muss, ob ein gewisses Textmuster m darin vorkommt. So etwas kann ein endlicher Akzeptor durchführen.

Als Beispiel betrachten wir das Textmuster $m = \text{ababb}$ über dem Eingabealphabet $X = \{a, b\}$. Ziel ist es, einen endlichen Akzeptor A zu konstruieren, der genau diejenigen Wörter akzeptiert, in denen irgendwo m als Teilwort vorkommt. Die erkannte Sprache soll also $L(A) = \{w_1 \text{ababb} w_2 \mid w_1, w_2 \in \{a, b\}^*\}$ sein.

Man kann diese Aufgabe natürlich ganz unterschiedlich angehen. Eine Möglichkeit, besteht darin, erst einmal einen Teil des Akzeptors hinzumalen, der „offensichtlich“ oder jedenfalls (hoffentlich) plausibel ist.

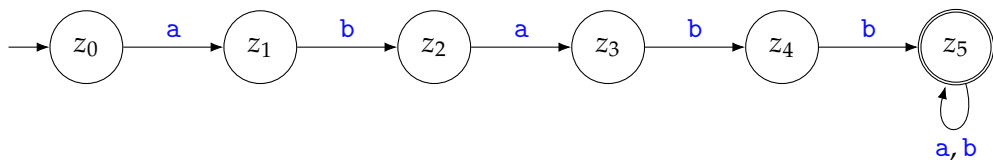


Abbildung 18.8: Teil eines Akzeptors für Wörter der Form $w_1 \text{ababb} w_2$

Damit sind wir aber noch nicht fertig. Denn erstens werden noch nicht alle gewünschten Wörter akzeptiert (z. B. **abababb**), und zweitens verlangt unsere Definition endlicher Akzeptoren, dass für *alle* Paare (z, x) der nächste Zustand $f(z, x)$ festgelegt wird.

Zum Beispiel die genauere Betrachtung des Wortes **abababb** gibt weitere Hinweise. Nach Eingabe von **abab** ist der Automat in z_4 . Wenn nun wieder ein **a** kommt, dann darf man nicht nach Zustand z_5 gehen, aber man hat zuletzt wieder **aba** gesehen. Das lässt es sinnvoll erscheinen, A wieder nach z_3 übergehen zu lassen. Durch weitere Überlegungen kann man schließlich zu dem Automaten aus Abbildung 18.9

Wir unterlassen es hier, im Detail zu beweisen, dass der Akzeptor aus Abbildung 18.9 tatsächlich die gewünschte Sprache erkennt. Man mache sich aber klar, dass für $0 \leq i \leq 4$ die folgende Aussage richtig ist:

A ist genau dann in Zustand z_i , wenn das längste Suffix der bisher gelesenen Eingabe, das Präfix von **ababb** ist, gerade Länge i hat.

Für z_5 ist die Aussage etwas anders; überlegen Sie sich eine passende Formulierung!

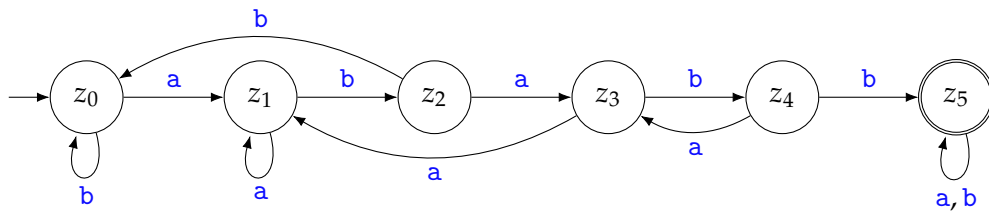


Abbildung 18.9: Der vollständige Akzeptor für alle Wörter der Form $w_1 \text{ababb} w_2$

18.4.2 Eine formale Sprache, die von keinem endlichen Akzeptoren akzeptiert werden kann

Wir haben gesehen, dass es formale Sprachen gibt, die man mit endlichen Akzeptoren erkennen kann. Es gibt aber auch formale Sprachen, die man mit endlichen Akzeptoren *nicht* erkennen kann. Ein klassisches Beispiel ist die Sprache

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

Bevor wir diese Behauptung beweisen, sollten Sie sich (falls noch nötig) klar machen, dass man natürlich ein (Java-)Programm schreiben kann, dass von einer beliebigen Zeichenkette überprüft, ob sie die eben angegebene Form hat. Man kann also mit „allgemeinen“ Algorithmen *echt mehr* Probleme lösen als mit endlichen Automaten.

18.1 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.

18.2 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 $0 \leq i \leq m$ sei $z_i = f_*(z_0, a^i)$. Mit anderen Worten: $f_{**}(z_0, a^m) = z_0 z_1 \cdots z_m$.

Offensichtlich gilt dann: $f_*(z_m, b^m) = z_f$.

Andererseits besteht die Liste $z_0 z_1 \cdots z_m$ aus $m + 1$ Werten. Aber A hat nur m verschiedene Zustände. Also kommt mindestens ein Zustand doppelt vor.

D.h. der Automat befindet sich in einer Schleife. Sei etwa $z_i = z_j$ für gewisse $i < j$. Genauer sei z_i das erste Auftreten irgendeines mehrfach auftretenden Zustandes und z_j das zweite Auftreten des gleichen Zustandes. Dann gibt es eine „Schleife“ der Länge $\ell = j - i > 0$. Und ist der Automat erst einmal in der Schleife, dann bleibt er natürlich darin, solange er weitere a als Eingabe erhält.

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^i gelesen hat, ist er in Zustand z_i . Dieser ist aber gleich dem Zustand z_j , d.h. A ist in dem Zustand in dem er auch nach der Eingabe a^j ist. Folglich ist

$$\begin{aligned} f_*(z_0, a^{m-\ell}) &= f_*(z_0, a^{i-\ell+m-i}) = f_*(f_*(z_0, a^i), a^{m-\ell-i}) \\ &= f_*(f_*(z_0, a^j), a^{m-\ell-i}) = f_*(f_*(z_0, a^{i+\ell}), a^{m-\ell-i}) \\ &= f_*(z_0, a^{i+\ell+m-\ell-i}) = f_*(z_0, a^m) = z_m \end{aligned}$$

Und wir wissen: $f_*(z_m, b^m) = z_f \in F$. Also ist $f_*(z_0, a^{m-\ell} b^m) = f_*(z_m, b^m) = z_f$, 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. ■

18.5 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

Nichtdeterminismus

Informatik eine Rolle (zum Beispiel bei Compilerbau-Werkzeugen).

Deterministischen Automaten, also die, die wir in diesem Kapitel betrachtet haben, werden Sie in Vorlesungen über Betriebssysteme und Kommunikation wiederbegegnen, z. B. im Zusammenhang mit der Beschreibung von sogenannten *Protokollen*.

19 REGULÄRE AUSDRÜCKE UND RECHTSLINEARE GRAMMATIKEN

Am Ende von [Einheit 18 über endliche Automaten](#) haben wir gesehen, dass manche formale Sprachen zwar von kontextfreien Grammatiken erzeugt, aber nicht von endlichen Akzeptoren erkannt werden können. Damit stellt sich für Sie vielleicht zum ersten Mal die Frage nach einer *Charakterisierung*, nämlich der der mit endlichen Akzeptoren erkennbaren Sprachen. Damit ist eine präzise Beschreibung dieser formalen Sprachen gemeint, die nicht (oder jedenfalls nicht offensichtlich) Automaten benutzt.

In den beiden Abschnitten dieser Einheit werden Sie zwei solche Charakterisierungen kennenlernen, die über [reguläre Ausdrücke](#) und die über [rechtslineare Grammatiken](#). In Abschnitt [19.3](#) wird es unter anderem um eine bequeme Verallgemeinerung vollständiger Induktion gehen.

19.1 REGULÄRE AUSDRÜCKE

Der Begriff *regulärer Ausdruck* geht ursprünglich auf Stephen Kleene ([1956](#)) zurück und wird heute in unterschiedlichen Bedeutungen genutzt. In dieser Einheit führen wir kurz regulären Ausdrücken nach der „klassischen“ Definition ein.

Etwas anderes (nämlich allgemeineres) sind die Varianten der *Regular Expressions*, von denen Sie möglicherweise schon im Zusammenhang mit dem ein oder anderen Programm (emacs, grep, sed, ...) oder der ein oder anderen Programmiersprache (Java, Python, ...) gelesen haben. Für Java gibt es das Paket [java.util.regex](#). Regular expressions sind eine deutliche Verallgemeinerung regulärer Ausdrücke, auf die wir in dieser Vorlesung nicht eingehen werden. Alles was wir im folgenden über reguläre Ausdrücke sagen, ist aber auch bei regular expressions anwendbar.

Wir kommen direkt zur Definition regulärer Ausdrücke. Sie wird sie hoffentlich an das ein oder andere aus der [Einheit 7 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 $x \in A$ ist x ein regulärer Ausdruck.
- Wenn R_1 und R_2 reguläre Ausdrücke sind, dann sind auch $(R_1 | R_2)$ und $(R_1 R_2)$ reguläre Ausdrücke.

regulärer Ausdruck

- 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 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

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

mit $P = \{R \rightarrow \emptyset, R \rightarrow (R | R), R \rightarrow (RR), R \rightarrow (R^*)\}$
 $\cup \{R \rightarrow x \mid x \in A\}$

erzeugt werden.

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

- \emptyset
- a
- b
- (ab)
- $((ab)a)$
- $((ab)a)a$
- $((ab)(aa))$
- $(\emptyset | b)$
- $(a | b)$
- $((a(a | b)) | b)$
- $(a | (b | (a | a)))$
- (\emptyset^*)
- (a^*)
- $((ba)(b^*))$
- $((ba)b)^*$
- $((a^*)^*)$
- $(((((ab)b)^*)^*) | (\emptyset^*))$

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

- ab
- aba
- $abaa$
- $ab(aa)$
- $\emptyset | b$
- $a | b$
- $a(a | b) | b$
- $(a | (b | (a | a)))$
- \emptyset^*
- a^*
- bab^*
- $(bab)^*$
- a^{**}
- $(abb)^{**} | \emptyset^*$

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

- $(| b)$ vor $|$ fehlt ein regulärer Ausdruck
- $| \emptyset |$ vor und hinter $|$ fehlt je ein regulärer Ausdruck
- $()ab$ zwischen $($ und $)$ fehlt ein regulärer Ausdruck
- $((ab)$ Klammern müssen „gepaart“ auftreten
- $*(ab)$ vor $*$ fehlt ein regulärer Ausdruck
- c^* c ist nicht Zeichen des Alphabetes

Reguläre Ausdrücke werden benutzt, um formale Sprachen zu spezifizieren. Auch dafür bedient man sich wieder einer induktiven Vorgehensweise; man spricht auch von einer induktiven Definition:

Die von einem regulären Ausdruck R beschriebene formale Sprache $\langle R \rangle$ ist wie folgt definiert:

durch R
beschriebene
Sprache $\langle R \rangle$

- $\langle \emptyset \rangle = \{ \}$ (d. h. die leere Menge).
- Für $x \in A$ ist $\langle x \rangle = \{ x \}$.
- Sind R_1 und R_2 reguläre Ausdrücke, so ist $\langle R_1 | R_2 \rangle = \langle R_1 \rangle \cup \langle R_2 \rangle$.
- Sind R_1 und R_2 reguläre Ausdrücke, so ist $\langle R_1 R_2 \rangle = \langle R_1 \rangle \cdot \langle R_2 \rangle$.
- Ist R ein regulärer Ausdruck, so ist $\langle R^* \rangle = \langle R \rangle^*$.

Betrachten wir drei einfache Beispiele:

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

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 im Kapitel 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 19.2 gleich noch beschreiben. Es handelt sich um einen Spezialfall kontextfreier Grammatiken.)

19.1 Satz. Für jede formale Sprache L sind die folgenden drei Aussagen äquivalent:

1. L kann von einem endlichen Akzeptor erkannt werden.
2. L kann durch einen regulären Ausdruck beschrieben werden.
3. L kann von einer rechtslinearen Grammatik erzeugt werden.

reguläre
Sprache

Eine formale Sprache, die die Eigenschaften aus Satz 19.1 hat, heißt *reguläre Sprache*. Da jede rechtslineare Grammatik eine kontextfreie Grammatik ist, ist jede reguläre Sprache eine kontextfreie Sprache.

Zwar werden wir Satz 19.1 nicht im Detail beweisen, aber wir wollen zumindest einige Dinge andeuten, insbesondere auch eine grundlegende Vorgehensweise.

Satz 19.1 hat folgende prinzipielle Struktur:

- Es werden drei Aussagen \mathcal{A} , \mathcal{B} und \mathcal{C} formuliert.
- Es wird behauptet:
 - $\mathcal{A} \longleftrightarrow \mathcal{B}$
 - $\mathcal{B} \longleftrightarrow \mathcal{C}$
 - $\mathcal{C} \longleftrightarrow \mathcal{A}$

Man kann nun natürlich einfach alle sechs Implikationen einzeln beweisen. Aber das muss man gar nicht! Dann wenn man zum Beispiel schon gezeigt hat, dass $\mathcal{A} \longrightarrow \mathcal{B}$ gilt und dass $\mathcal{B} \longrightarrow \mathcal{C}$, dann folgt $\mathcal{A} \longrightarrow \mathcal{C}$ automatisch. Das sieht man anhand der folgenden Tabelle:

	\mathcal{A}	\mathcal{B}	\mathcal{C}	$\mathcal{A} \longrightarrow \mathcal{B}$	$\mathcal{B} \longrightarrow \mathcal{C}$	$\mathcal{A} \longrightarrow \mathcal{C}$
1				W	W	W
2		W		W	W	W
3	W			W		W
4	W	W		W	W	W
5	W				W	
6	W	W			W	W
7	W	W	W			
8	W	W	W	W	W	W

In allen Zeilen 1, 2, 4 und 8, in denen sowohl für $\mathcal{A} \longrightarrow \mathcal{B}$ als auch für $\mathcal{B} \longrightarrow \mathcal{C}$ ein W (für *wahr*) eingetragen ist, ist das auch für $\mathcal{A} \longrightarrow \mathcal{C}$ der Fall. Statt *falsch* haben wir der besseren Übersicht wegen die entsprechenden Felder freigelassen.

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

- folgt mit $\mathcal{A} \longrightarrow \mathcal{B}$ sofort $\mathcal{C} \longrightarrow \mathcal{B}$ und
- mit $\mathcal{B} \longrightarrow \mathcal{C}$ folgt sofort $\mathcal{B} \longrightarrow \mathcal{A}$,

und man ist fertig.

Statt sechs Implikationen zu beweisen zu müssen, reichen also drei. Für einen Beweis von Satz 19.1 genügen daher folgende Konstruktionen:

- zu gegebenem endlichen Akzeptor A ein regulärer Ausdruck R mit $\langle R \rangle = L(A)$:

Diese Konstruktion ist „mittel schwer“. Man kann z. B. einen Algorithmus benutzen, dessen Struktur und Idee denen des Algorithmus von Warshall ähneln.

- zu gegebenem regulären Ausdruck R eine rechtslineare Grammatik G mit $L(G) = \langle R \rangle$:

Diese Konstruktion ist „relativ leicht“. Wir werden im nächsten Abschnitt noch etwas genauer darauf eingehen.

- zu gegebener rechtslinearer Grammatik G ein endlicher Akzeptor A mit $L(A) = L(G)$:

Diese Konstruktion ist die schwierigste.

Wie wertvoll Charakterisierungen wie Satz 19.1 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.

19.2 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 folgen-

*rechtslineare
Grammatik*

den 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.

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.

Typ-3-Grammatiken

Typ-2-Grammatiken

Es sei auch noch die folgende Sprechweise eingeführt: Rechtslineare Grammatiken heißen auch *Typ-3-Grammatiken* und die schon eingeführten kontextfreien Grammatiken nennt man auch *Typ-2-Grammatiken*. Hier ahnt man schon, dass es noch weiter geht. Es gibt auch noch *Typ-1-Grammatiken* und *Typ-0-Grammatiken*.

Wenn für ein $i \in \{0, 1, 2, 3\}$ eine formale Sprache L von einer Typ- i -Grammatik erzeugt wird, dann sagt man auch, L sei eine *Typ- i -Sprache* oder kurz *vom Typ i* .

Zumindest einer der Vorteile rechtslinearer Grammatiken gegenüber deterministischen endlichen Akzeptoren, wie wir sie im vorangegangenen Kapitel eingeführt haben, ist, dass sie manchmal deutlich kürzer und übersichtlicher hinzuschreiben sind. Ein genaueres Verständnis dafür, warum das so ist, werden Sie bekommen, wenn Sie im dritten Semester auch etwas über sogenannte nichtdeterministische endliche Akzeptoren gelernt haben.

19.3 KANTOROWITSCH-BÄUME UND STRUKTURELLE INDUKTION

Kantorowitsch-Baum

Reguläre Ausdrücke kann man auch als sogenannte *Kantorowitsch-Bäume* darstellen. Für den regulären Ausdruck $((b|\emptyset)a)(b^*)$ ergibt sich zum Beispiel der Graph aus Abbildung 19.1

Hier handelt es sich *nicht* um den Ableitungsbaum gemäß der Grammatik aus Abschnitt 19.1. Aber die Struktur des regulären Ausdruckes wird offensichtlich ebenso gut wiedergegeben und die Darstellung ist sogar noch kompakter.

Solche Bäume wollen wir im folgenden der Einfachheit halber *Regex-Bäume* nennen. Es sei A irgendeine Alphabet. Dann ist ein Baum ein *Regex-Baum*, wenn gilt:

- Entweder ist es ein Baum, dessen Wurzel zugleich Blatt ist, und das ist mit einem $x \in A$ oder \emptyset beschriftet,

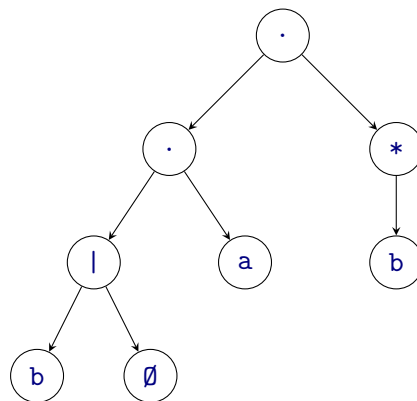


Abbildung 19.1: Der Kantorowitsch-Baum für den regulären Ausdruck $((b|\emptyset)a)(b^*)$

- oder es ist ein Baum, dessen Wurzel mit $*$ beschriftet ist und die genau einen Nachfolgeknoten hat, der Wurzel eines Regex-Baumes ist
- oder es ist ein Baum, dessen Wurzel mit \cdot oder mit $|$ beschriftet ist und die genau zwei Nachfolgeknoten hat, die Wurzeln zweier Regex-Bäume sind.

Man beachte, dass linker und rechter Unter-Regex-Baum unterhalb der Wurzel unterschiedliche Höhen haben können.

Wichtig ist für uns im folgenden, dass erstens größere Bäume „aus kleineren zusammengesetzt“ werden, und dass zweitens diese Zusammensetzung immer eindeutig ist. Außerdem kommt man bijektiv von regulären Ausdrücken zu Regex-Bäumen und umgekehrt.

Für das weitere definieren wir noch ein oft auftretendes Maß für Bäume, die sogenannte *Höhe* $h(T)$ eines Baumes T . Das geht so:

$$h(T) = \begin{cases} 0 & \text{falls die Wurzel Blatt ist} \\ 1 + \max_i h(U_i) & \text{falls die } U_i \text{ alle Unterbäume von } T \text{ sind} \end{cases}$$

*Höhe eines
Baumes*

Wenn man beweisen möchte, dass eine Aussage für alle regulären Ausdrücke gilt, dann kann man das dadurch tun, dass man die entsprechende Aussage für alle Regex-Bäume beweist. Und den Beweis für alle Regex-Bäume kann man durch vollständige Induktion über ihre Höhe führen. Bei naiver Herangehensweise tritt aber ein Problem auf: Beim Schritt zu Bäumen der Höhe $n + 1$ darf man nur auf Bäume der Höhe n zurückgreifen. Auftretende Unterbäume können aber alle Höhen $i \leq n$ haben, und man möchte gerne für sie alle die Induktionsvoraussetzung benutzen. Das darf man auch, wie wir uns in Kapitel 6 klar gemacht haben.

Das wollen wir nun anwenden, um zu einen Beweis dafür zu skizzieren, dass es für jeden regulären Ausdruck R eine rechtslineare Grammatik G gibt mit $\langle R \rangle =$

$L(G)$. Bei regulären Ausdrücken denkt man nun vorteilhafterweise an Regex-Bäume und wir machen nun zunächst eine „normale“ vollständige Induktion über die Höhe der Regex-Bäume. Im Schema von oben ist also $\mathcal{B}(n)$ die Aussage:

Für jeden Regex-Baum R der Höhe n gibt es eine rechtslineare Grammatik G mit $\langle R \rangle = L(G)$.

Wir wollen zeigen, dass $\forall n \in \mathbb{N}_0 : \mathcal{B}(n)$ gilt.

Induktionsanfang: Es ist $\mathcal{B}(0)$ zu zeigen. Man muss also rechtslineare Grammatiken angeben, die die formalen Sprachen $\{x\} = \langle x \rangle$ für $x \in A$ und die leere Menge $\{\} = \langle \emptyset \rangle$ erzeugen. Das ist eine leichte Übung.

Induktionsschluss: Es ist zu zeigen, dass für jedes $n \in \mathbb{N}_0$ aus $\mathcal{B}(n)$ auch $\mathcal{B}(n+1)$ folgt. Dazu sei für $n \in \mathbb{N}_0$ die **Induktionsvoraussetzung**, dass für jedes $i \leq n$ $\mathcal{B}(i)$ gilt, dass es also für jeden Regex-Baum R' mit einer Höhe $i \leq n$ eine rechtslineare Grammatik G mit $\langle R' \rangle = L(G)$ existiert.

Sei nun R ein beliebiger Regex-Baum der Höhe $n+1$. Dann gibt es drei mögliche Fälle:

1. Die Wurzel von R ist ein $*$ -Knoten und hat genau einen Unterbaum R' der Höhe n .
2. Die Wurzel von R ist ein $|$ -Knoten und hat genau zwei Unterbäume R_1 und R_2 . Da R Höhe $n+1$ hat, hat einer der beiden Unterbäume Höhe n , der andere hat eine Höhe $i \leq n$.
3. Die Wurzel von R ist ein „Konkatenations-Knoten“ und hat genau zwei Unterbäume R_1 und R_2 . Da R Höhe $n+1$ hat, hat einer der beiden Unterbäume Höhe n , der andere hat eine Höhe $i \leq n$.

Der entscheidende Punkt ist nun: In den Fällen 2 und 3 darf man nach Induktionsvoraussetzung annehmen, dass für *beide* Unterbäume rechtslineare Grammatiken der gewünschten Art existieren.

In allen drei Fällen kann man dann aus den Grammatiken für den Unterbaum bzw. die Unterbäume die Grammatik für den Regex-Baum, also regulären Ausdruck, R konstruieren. Das wollen wir hier nicht in allen Details durchführen und beschränken uns auf den einfachsten Fall, nämlich Fall 2: Seien also $G_1 = (N_1, A, S_1, P_1)$ und $G_2 = (N_2, A, S_2, P_2)$ Typ-3-Grammatiken, die $L(G_1) = \langle R_1 \rangle$ bzw. $L(G_2) = \langle R_2 \rangle$ erzeugen. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $N_1 \cap N_2 = \{\}$ ist. Wir wählen ein „neues“ Nichtterminalsymbol $S \notin N_1 \cup N_2$. Damit können wir eine Typ-3-Grammatik G mit $L(G) = \langle R_1 | R_2 \rangle$ ganz leicht hinschreiben:

$$G = (\{S\} \cup N_1 \cup N_2, A, S, \{S \rightarrow S_1 \mid S_2\} \cup P_1 \cup P_2)$$

Als erstes muss man sich klar machen, dass auch die Grammatik rechtslinear ist. Tun Sie das; es ist nicht schwer.

Etwas Arbeit würde es machen, zu beweisen, dass $L(G) = L(G_1) \cup L(G_2)$ ist. Das wollen wir uns an dieser Stellen sparen. Sie können das aber ruhig selbst einmal versuchen.

Und für die anderen beiden Fälle können Sie das auch einmal versuchen, geeignete (rechtslineare!) Grammatiken zu konstruieren, oder einfach glauben, dass es geht.

Um zu einer manchmal sogenannten *strukturellen Induktion* zu kommen, muss man nun nur noch das Korsett der vollständigen Induktion über die Höhe der Bäume „vergessen“. Was bleibt ist folgende prinzipielle Situation:

*strukturelle
Induktion*

1. Man beschäftigt sich mit irgendwelchen „Gebilden“ (eben waren das reguläre Ausdrücke bzw. Bäume). Dabei gibt es kleinste „atomare“ oder „elementare“ Gebilde (eben waren das die regulären Ausdrücke x für $x \in A$ und \emptyset) und eine oder mehrere Konstruktionsvorschriften, nach denen man aus kleineren Gebilden größere zusammensetzen kann (eben waren das $*$, $|$ und Konkatenation).
2. Man möchte beweisen, dass alle Gebilde eine gewisse Eigenschaft haben. Dazu macht man dann eine strukturelle Induktion:

- Im Induktionsanfang zeigt man zunächst für *alle* „atomaren“ Gebilde, dass sie eine gewünschte Eigenschaft haben und
- im Induktionsschritt zeigt man, wie sich bei einem „großen“ Gebilde die Eigenschaft daraus folgt, dass schon alle Untergebilde die Eigenschaft haben, gleich nach welcher Konstruktionsvorschrift das große Gebilde gebaut ist.

19.4 AUSBLICK

Beweise für die Behauptungen aus Satz 19.1 werden Sie vielleicht in der Vorlesung „Theoretische Grundlagen der Informatik“ oder in „Formale Systeme“ sehen. Insbesondere ist es dafür nützlich, sich mit nichtdeterministischen endlichen Automaten zu beschäftigen, auf die wir am Ende von Einheit 18 schon hingewiesen haben.

Sie werden sehen, dass reguläre Ausdrücke bei der Verarbeitung von Textdateien des Öffneren nützlich sind. Dabei kommen zu dem, was wir in Abschnitt 19.1 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. (siehe S. 218).

Kleene, Stephen C. (1956). „Representation of Events in Nerve Nets and Finite Automata“. In: *Automata Studies*. Hrsg. von Claude E. Shannon und John McCarthy. Princeton University Press. Kap. 1, S. 3–40.

Eine Vorversion ist online verfügbar; siehe http://www.rand.org/pubs/research_memoranda/2008/RM704.pdf (10.1.2020) (siehe S. 174, 209).

20 TURINGMASCHINEN

Turingmaschinen sind eine und waren im wesentlichen die erste mathematische Präzisierung des Begriffes des *Algorithmus* so wie er klassisch verstanden wird: Zu jeder endlichen Eingabe wird in endlich vielen Schritten eine endliche Ausgabe berechnet.

Algorithmus

Einen technischen Hinweis wollen wir an dieser Stelle auch noch geben: In diesem Kapitel werden an verschiedenen Stellen partielle Funktionen vorkommen. Das sind rechtseindeutige Relationen, die nicht notwendig linkstotal sind (siehe Abschnitt 3.3). 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.

20.1 ALAN MATHISON TURING

ALAN MATHISON TURING wurde am 23.6.1912 geboren.

Mitte der Dreißiger Jahre beschäftigte sich Turing mit Gödels Unvollständigkeitssätzen und Hilberts Frage, ob man für jede mathematische Aussage algorithmisch entscheiden könne, ob sie wahr sei oder nicht. Das führte zu der bahnbrechenden Arbeit „*On computable numbers, with an application to the Entscheidungsproblem*“ von 1936.

Von 1939 bis 1942 arbeitete Turing in Bletchly Park an der Dechiffrierung der verschlüsselten Texte der Deutschen. Für den Rest des zweiten Weltkriegs beschäftigte er sich in den USA mit Ver- und Entschlüsselungsfragen. Mehr zu diesem Thema (und verwandten) können Sie in Vorlesungen zum Thema *Kryptographie* erfahren.

Nach dem Krieg widmete sich Turing unter anderem dem Problem der *Morphogenese* in der Biologie. Ein kleines bisschen dazu findet sich in der Vorlesung „Algorithmen in Zellularautomaten“.

Alan Turing starb am 7.6.1954 an einer Zyankalivergiftung. Eine Art „Homepage“ findet sich unter <http://www.turing.org.uk/turing/index.html> (19.1.2011).

20.2 TURINGMASCHINEN

Als *Turingmaschine* bezeichnet man heute etwas, was Turing (1936) in seiner Arbeit eingeführt hat. Die Bezeichnung selbst geht wohl auf eine Besprechung von

Turings Arbeit durch Alonzo Church zurück (laut einer WWW-Seite von J. Miller; <http://jeff560.tripod.com/t.html>, 14.1.2010).

Eine Turingmaschine kann man als eine Verallgemeinerung endlicher Automaten auffassen, bei der die Maschine nicht mehr darauf beschränkt ist, nur feste konstante Zahl von Bits zu speichern. Im Laufe der Jahrzehnte wurden viele Varianten definiert und untersucht. Wir führen im folgenden nur die einfachste Variante ein.

Zur Veranschaulichung betrachte man Abbildung 20.1. 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 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.

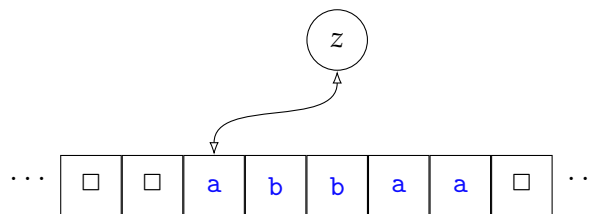


Abbildung 20.1: schematische Darstellung einer (einfachen) Turingmaschine

Turingmaschine

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

- eine Zustandsmenge Z
- einen Anfangszustand $z_0 \in Z$
- ein Bandalphabet X
- eine partielle 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 20.2 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 .

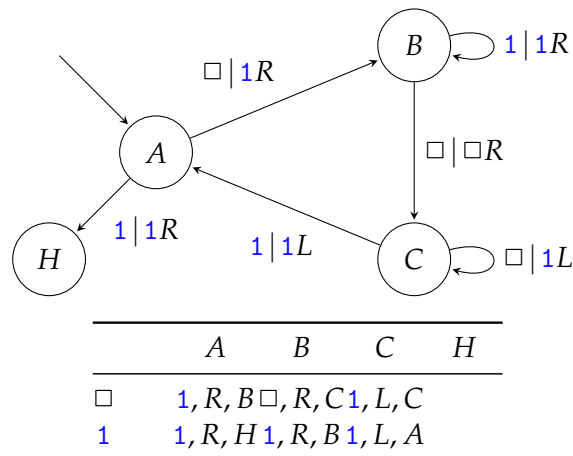


Abbildung 20.2: Zwei Spezifikationsmöglichkeiten der gleichen Turingmaschine; sie heißt BB₃.

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

- den aktuellen Zustand $z \in Z$ der Steuereinheit,
- die aktuelle Beschriftung des gesamten Bandes, die man als Abbildung $b : \mathbb{Z} \rightarrow X$ 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 14.2 erwähnt und zu Beginn dieses Kapitels noch einmal betont, interessieren in weiten Teilen der Informatik *endliche* Berechnungen, die aus *endlichen* Eingaben *endliche* Ausgaben berechnen. Um das adäquat zu formalisieren, ist es üblich, davon auszugehen, dass das Bandalphabet ein sogenanntes *Blanksymbol* enthält, für das wir $\square \in X$ schreiben. Bandfelder, die „mit \square beschriftet“ sind, wollen wir als „leer“ ansehen; und so stellen wir sie dann gelegentlich auch dar, oder lassen sie ganz weg. Jedenfalls in dieser Vorlesung (und in vielen anderen auch) sind alle tatsächlich vorkommenden Bandbeschriftungen von der Art, dass nur endlich viele Felder nicht mit \square beschriftet sind.

20.2.1 Berechnungen

Schritt einer
Turingmaschine

Wenn $c = (z, b, p)$ die aktuelle Konfiguration einer Turingmaschine T ist, dann kann es sein, dass sie einen *Schritt* durchführen kann. Das geht genau dann, wenn für das Paar $(z, b(p))$ aus aktuellem Zustand und aktuell 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:

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

Wir schreiben $c' = \Delta_1(c)$. Bezeichnet \mathcal{C}_T die Menge aller Konfigurationen einer Turingmaschine T , dann ist das also die partielle Abbildung $\Delta_1 : \mathcal{C}_T \dashrightarrow \mathcal{C}_T$, die als Funktionswert $\Delta_1(c)$ gegebenenfalls die ausgehend von c nach einem Schritt erreichte Konfiguration bezeichnet.

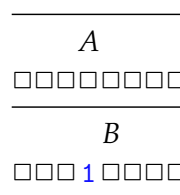
Endkonfiguration
Halten

Falls für eine Konfiguration c die Nachfolgekongfiguration $\Delta_1(c)$ nicht definiert ist, heißt c auch eine *Endkonfiguration* und man sagt, die Turingmaschine habe *gehalten*.

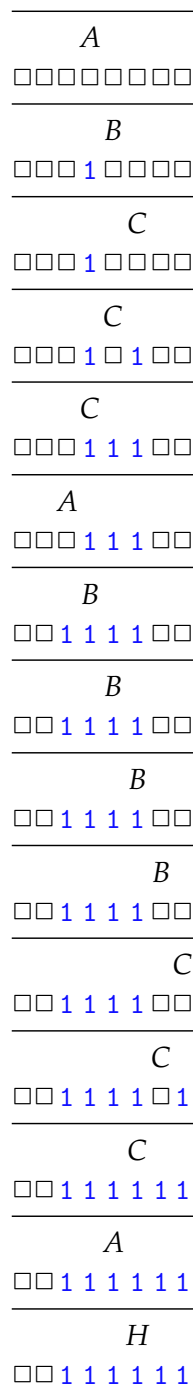
Die Turingmaschine aus Abbildung 20.2 wollen wir BB_3 nennen. Wenn BB_3 im Anfangszustand A auf einem vollständig leeren Band gestartet wird, dann macht sie wegen

- $f(A, \square) = B$,
- $g(A, \square) = 1$ und
- $m(A, \square) = R$

folgenden Schritt:



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.



In Zustand *H* ist kein Schritt mehr möglich; es ist eine Endkonfiguration erreicht und BB₃ hält.

endliche Berechnung Eine *endliche Berechnung* ist eine endliche Folge von Konfigurationen $(c_0, c_1, c_2, \dots, c_t)$ mit der Eigenschaft, dass für alle $0 < i \leq t$ gilt: $c_i = \Delta_1(c_{i-1})$. Eine Berechnung ist *haltend*, wenn es eine endliche Berechnung ist und ihre letzte Konfiguration eine Endkonfiguration ist.

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 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.

nicht haltende Berechnung Analog zu Δ_1 liefere generell für $t \in \mathbb{N}_0$ die Abbildung Δ_t als Funktionswert $\Delta_t(c)$ gegebenenfalls die ausgehend von c nach t Schritten erreichte Konfiguration. Also

$$\Delta_0 = I$$

$$\forall t \in \mathbb{N}_+ : \Delta_{t+1} = \Delta_1 \circ \Delta_t$$

Zu jeder Konfiguration c gibt es genau eine Berechnung, die mit c startet, und wenn diese Berechnung hält, dann ist der Zeitpunkt, zu dem das geschieht, natürlich auch eindeutig. Wir schreiben Δ_* für die partielle Abbildung $\mathcal{C}_T \dashrightarrow \mathcal{C}_T$ mit

$$\Delta_*(c) = \begin{cases} \Delta_t(c) & \text{falls } \Delta_t(c) \text{ definiert und} \\ & \text{Endkonfiguration ist} \\ \text{undefiniert} & \text{falls } \Delta_t(c) \text{ für alle } t \in \mathbb{N}_0 \text{ definiert ist} \end{cases}$$

20.2.2 Eingaben für Turingmaschinen

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

Eingabealphabet In beiden Fällen muss aber jedenfalls der Turingmaschine die Eingabe zur Verfügung gestellt werden. Dazu fordern wir, dass stets ein *Eingabealphabet* $A \subset X \setminus \{\square\}$ spezifiziert ist. (Das Blanksymbol gehört also nie zum Eingabealphabet.)

Und die Eingabe eines Wortes $w \in A^*$ wird bewerkstelligt, indem die Turingmaschine im Anfangszustand z_0 mit dem Kopf auf Feld 0 gestartet wird mit der Bandbeschriftung

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

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

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. Für die Eingabe mehrerer Argumente sei vereinbart, dass jedes mit `[` und `]` abgegrenzt wird und die entstehenden Wörter unmittelbar hintereinander auf das Band geschrieben werden. Für die Argumente 11 und 5 hätte der relevante Teil der initialen Bandbeschriftung also die Form `□□□[1011][101]□□`.

20.2.3 Ergebnisse von Turingmaschinen

Man betrachtet verschiedene Arten, wie eine Turingmaschine ein Ergebnis „mitteilt“, wenn sie hält. Sofern man *kein* Entscheidungsproblem vorliegen hat, sondern ein Problem, bei dem mehr als zwei verschiedene Funktionswerte $f(x)$ möglich sind, ist eine nicht unübliche Vereinbarung, dass in der Endkonfiguration das Band wieder vollständig leer ist bis auf eine Darstellung von $f(x)$ (z. B. binär, falls es um nichtnegative ganze Zahlen geht). Wir wollen so etwas eine Turingmaschine mit *Ausgabe auf dem Band* nennen.

Ausgabe auf dem Band

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
Turingmaschinenakzeptor

Einen solchen Turingmaschinenakzeptor kann man immer „umbauen“ in eine Turingmaschine mit Ausgabe auf dem Band, wobei dann Akzeptieren durch Ausgabe `1` und Ablehnen durch Ausgabe `0` repräsentiert wird. Versuchen Sie als Übung, sich zumindest ein Konzept für eine entsprechende Konstruktion zu überlegen.

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

- Die Turingmaschine hält für Eingabe w in einem nicht akzeptierenden Zustand.
- Die Turingmaschine hält für Eingabe w nicht.

Im ersten Fall bekommt man sozusagen die Mitteilung „Ich bin fertig und lehne die Eingabe ab.“ Im zweiten Fall weiß man nach jedem Schritt nur, dass die Turingmaschine noch arbeitet. Ob sie irgendwann anhält, und ob sie die Eingabe dann akzeptiert oder ablehnt, ist im allgemeinen unbekannt. Eine formale Sprache, die von einer Turingmaschine akzeptiert werden kann, heißt auch *aufzählbare Sprache*.

aufzählbare Sprache

*entscheiden
entscheidbare Sprache*

Wenn es eine Turingmaschine T gibt, die L akzeptiert und für jede Eingabe hält, dann sagt man auch, dass T die Sprache L *entscheidet* und dass L *entscheidbar* ist. Dass das eine echt stärkere Eigenschaft ist als Aufzählbarkeit werden wir in Abschnitt 20.4 ansprechen.

Als Beispiel betrachten wir die Aufgabe, für jedes Eingabewort $w \in L = \{a, b\}^*$ festzustellen, ob es ein Palindrom ist oder nicht. Es gilt also einen Turingmaschinenakzeptor zu finden, der genau L entscheidet. In Abbildung 20.3 ist eine solche Turingmaschine angegeben. Ihr Anfangszustand ist r und einziger akzeptierender Zustand ist f_+ . Der Algorithmus beruht auf der Idee, dass ein Wort genau dann Palindrom ist, wenn erstes und letztes Symbol übereinstimmen und das Teilwort dazwischen auch ein Palindrom ist.

Zur Erläuterung der Arbeitsweise der Turingmaschine ist in Abbildung 20.4 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.

20.3 BERECHNUNGSKOMPLEXITÄT

Wir beginnen mit einem wichtigen Hinweis: Der Einfachheit halber wollen wir in diesem Abschnitt davon ausgehen, dass wir ausschließlich mit Turingmaschinen zu tun haben, die für jede Eingabe halten. Die Definitionen sind dann leichter hinschreiben. Und für die Fragestellungen, die uns in diesem und im folgenden Abschnitt interessieren, ist das „in Ordnung“. Warum dem so ist, erfahren Sie vielleicht einmal in einer Vorlesung über Komplexitätstheorie.

In Abschnitt 20.4 werden wir dann aber wieder gerade von dem allgemeinen Fall ausgehen, dass eine Turingmaschine für manche Eingaben *nicht* hält. Warum das wichtig ist, werden Sie dann schnell einsehen. Viel mehr zu diesem Thema

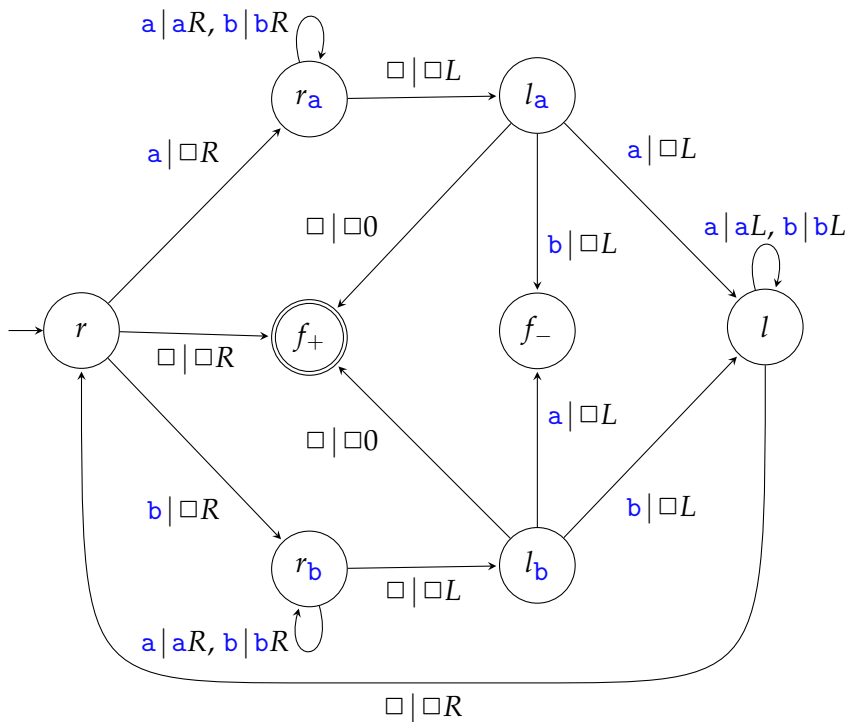


Abbildung 20.3: Eine Turingmaschine zur Palindromerkennung; f_+ sei der einzige akzeptierende Zustand

werden Sie vielleicht einmal in einer Vorlesung über Berechenbarkeit oder/und Rekursionstheorie hören.

20.3.1 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:

$$\text{time}_T(w) = \text{das } t, \text{ für das } \Delta_t(c_0(w)) \text{ Endkonfiguration ist}$$

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

Da wir im Moment davon ausgehen, dass die betrachteten Turingmaschinen immer halten, sind diese Abbildungen total. Der Einfachheit halber lassen wir das leere Wort bei diesen Betrachtungen weg.

Üblicherweise heißt die Abbildung Time_T die *Zeitkomplexität* der Turingmaschi-

Zeitkomplexität

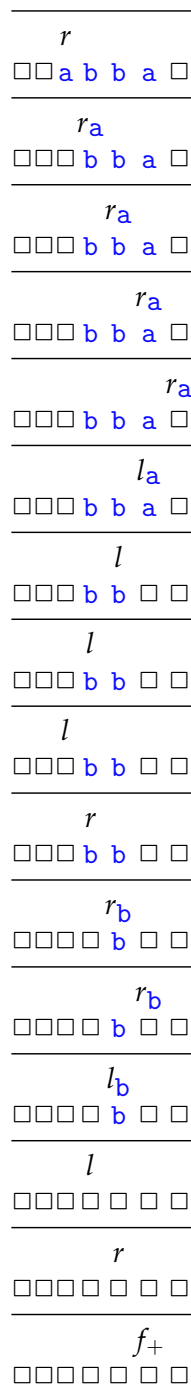


Abbildung 20.4: Akzeptierende Berechnung der Turingmaschine aus Abbildung 20.3 für Eingabe abba

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

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
- zurücklaufen an den Anfang des Restwortes der Länge $n - 2$ ohne die Rand-symbole und
- dafür wieder einen Palindromtest machen.

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

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

Der Zeitaufwand für Wörter der Länge 1 ist ebenfalls gerade $2n + 1$. Eine kurze Überlegung zeigt, dass daher die Zeitkomplexität $\text{Time}(n) \in O(n^2)$, also polynomiell ist.

Für die Beurteilung des Speicherplatzbedarfs definiert man zwei Funktionen $\text{space}_T(w) : A^+ \rightarrow \mathbb{N}_+$ $\text{Space}_T(n) : \mathbb{N}_+ \rightarrow \mathbb{N}_+$ wie folgt:

$$\begin{aligned} \text{space}_T(w) &= \text{die Anzahl der Felder, die während der} \\ &\quad \text{Berechnung für Eingabe } w \text{ benötigt werden} \\ \text{Space}_T(n) &= \max\{\text{space}_T(w) \mid w \in A^n\} \end{aligned}$$

Üblicherweise heißt die Abbildung Space_T die *Raumkomplexität* oder *Platzkomplexität* der Turingmaschine T . Dabei gelte ein Feld als „benötigt“, wenn es zu Beginn ein Eingabesymbol enthält oder irgendwann vom Kopf der Turingmaschine besucht wird. Man sagt, dass die Raumkomplexität einer Turingmaschine *polynomiell* ist, wenn ein Polynom $p(n)$ existiert mit $\text{Space}_T(n) \in O(p(n))$.

Raumkomplexität

Platzkomplexität

polynomielle

Raumkomplexität

Unsere Beispielmachine zur Palindromerkennung hat Platzbedarf $\text{Space}(n) = n + 1 \in \Theta(n)$, weil außer den n Feldern mit den Eingabesymbolen nur noch ein weiteres Feld rechts davon besucht wird.

Welche Zusammenhänge bestehen zwischen der Zeit- und der Raumkomplexität einer Turingmaschine? Wenn eine Turingmaschine für eine Eingabe w genau $\text{time}(w)$ viele Schritte macht, dann kann sie nicht mehr als $1 + \text{time}(w)$ verschiedene Felder mit ihrem Kopf besuchen. Folglich ist sicher immer

$$\text{space}(w) \leq |w| + \text{time}(w) .$$

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.

20.3.2 Komplexitätsklassen

Komplexitätsklasse

Eine *Komplexitätsklasse* ist eine Menge von Problemen. Wir beschränken uns im folgenden wieder auf Entscheidungsprobleme, also formale Sprachen. Charakterisiert werden Komplexitätsklassen oft durch Beschränkung der zur Verfügung stehenden 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 Probleminstanz, also die Länge des Eingabewortes, steht.

Es hat sich herausgestellt, dass unter anderem die beiden folgenden Komplexitätsklassen interessant sind:

- **P** ist die Menge aller Entscheidungsprobleme, die von Turingmaschinen entschieden werden können, deren Zeitkomplexität polynomiell ist.
- **PSPACE** ist die Menge aller Entscheidungsprobleme, die von Turingmaschinen entschieden werden können, deren Raumkomplexität polynomiell ist.

Zu **P** gehört zum Beispiel das Problem der Palindromerkennung. Denn wie wir uns überlegt haben, benötigt die Turingmaschine aus Abbildung 20.3 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 Kapitel 19 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 wählen, das sich von allen anderen Symbolen unterscheidet.

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 polynomielltem 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 polynomielltem Platzbedarf auch dann, wenn sie exponentielle Laufzeit hat, eine andere Turingmaschine mit nur polynomielltem 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 $\mathbf{P} \neq \mathbf{PSPACE}$.

20.4 UNENTSCHEIDBARE PROBLEME

In gewisser Weise noch schlimmer als Probleme, die exorbitanten Ressourcenbedarf zur Lösung erfordern, sind Probleme, die man algorithmisch, also z. B. mit Turingmaschinen oder Java-Programmen, überhaupt nicht lösen kann.

In diesem Abschnitt wollen wir zumindest andeutungsweise sehen, dass es solche Probleme tatsächlich gibt.

20.4.1 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 $\text{cod}_{fgm}(z, x) = [\text{cod}_Z(z) \text{cod}_X(x) [] [] []]$.
 - 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 Konkatenation aller $\text{cod}_{fgm}(z, x)$ für alle $z \in Z$ und alle $x \in X$.
- Die gesamte Turingmaschine wird codiert als Konkatenation der Codierung des Zustands mit der größten Nummer, des Bandsymbols mit der größten Nummer und der Codierung der gesamten Funktionen f , g und m .
- Wir schreiben auch T_w für die Turingmaschine mit Codierung w .

Auch ohne dass wir das hier im Detail ausführen, können Sie hoffentlich zumindest glauben, dass man eine Turingmaschine konstruieren kann, die für jedes beliebige 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 und w_2 Codierung einer Eingabe für T_{w_1} ist, und
- falls ja, simuliert sie 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 (oder vielmehr die Codierung dessen).

20.4.2 Das Halteproblem

Der Kern des Nachweises der Unentscheidbarkeit des Halteproblems ist *Diagonalisierung*. Die Idee geht zurück auf Georg Ferdinand Ludwig Philipp Cantor (1845–1918, siehe z. B. <http://www-history.mcs.st-and.ac.uk/Biographies/Cantor.html>, 17.1.2019), 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 w_j ($j \in \mathbb{N}_0$). Eintrag in Zeile i und Spalte j der Tabelle sei gerade der Funktionswert $f_i(w_j)$. Die f_i mögen als Funktionswerte zumindest 0 und 1 annehmen können.

Diagonalisierung

	w_0	w_1	w_2	w_3	w_4	\dots
f_0	$f_0(w_0)$	$f_0(w_1)$	$f_0(w_2)$	$f_0(w_3)$	$f_0(w_4)$	\dots
f_1	$f_1(w_0)$	$f_1(w_1)$	$f_1(w_2)$	$f_1(w_3)$	$f_1(w_4)$	\dots
f_2	$f_2(w_0)$	$f_2(w_1)$	$f_2(w_2)$	$f_2(w_3)$	$f_2(w_4)$	\dots
f_3	$f_3(w_0)$	$f_3(w_1)$	$f_3(w_2)$	$f_3(w_3)$	$f_3(w_4)$	\dots
f_4	$f_4(w_0)$	$f_4(w_1)$	$f_4(w_2)$	$f_4(w_3)$	$f_4(w_4)$	\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 $d(w_i) = f_i(w_i)$ und dann *alle* Einträge *ändert*, also

$$\bar{d}(w_i) = \overline{f_i(w_i)} = \begin{cases} 1 & \text{falls } f_i(w_i) = 0 \\ 0 & \text{sonst} \end{cases}$$

dann erhält man eine Abbildung („Zeile“) \bar{d} ,

	w_0	w_1	w_2	w_3	w_4	\dots
d	$f_0(w_0)$	$f_1(w_1)$	$f_2(w_2)$	$f_3(w_3)$	$f_4(w_4)$	\dots
\bar{d}	$\overline{f_0(w_0)}$	$\overline{f_1(w_1)}$	$\overline{f_2(w_2)}$	$\overline{f_3(w_3)}$	$\overline{f_4(w_4)}$	\dots

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

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

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.

20.1 Theorem Das Halteproblem ist unentscheidbar, d. h. es gibt keine Turingmaschine, die H entscheidet.

20.2 Beweis. Wir benutzen (eine Variante der) Diagonalisierung. In der obigen großen Tabelle seien nun die w_i alle Codierungen von Turingmaschinen in irgendeiner Reihenfolge. Und f_i sei die Funktion, die von der Turingmaschine mit Codierung w_i , also T_{w_i} , berechnet wird.

Da wir es mit Turingmaschinen zu tun haben, werden manche der $f_i(w_j)$ nicht definiert sein, da T_{w_i} für Eingabe w_j nicht hält. Da die w_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 w_i hält und als Ergebnis mitteilt, ob T_{w_i} für Eingabe w_i hält. Wir gehen davon aus, dass es sich dabei um eine Turingmaschine mit Ausgabe auf dem Band handelt.

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 w_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 $T_{w_i}(w_i)$ hält, dann geht $T_{\bar{d}}$ in eine Endlosschleife.
 - Wenn T_h mitteilt, dass $T_{w_i}(w_i)$ 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 w_i anders als das von T_{w_i} für die gleiche Eingabe.

Also: Wenn Turingmaschine T_h existiert, dann existiert auch Turingmaschine $T_{\bar{d}}$, aber jede Turingmaschine T_{w_i} verhält sich für mindestens eine Eingabe, nämlich w_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. ■

20.4.3 Die Busy-Beaver-Funktion

In Abschnitt 20.2 hatten wir BB₃ 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 BB₃ haben wir gesehen, dass sie 6 Einsen auf dem Band erzeugt.

Bibermaschine

Die *Busy-Beaver-Funktion* ist wie folgt definiert:

Busy-Beaver-Funktion

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

$\text{bb}(n)$ = die maximale Anzahl von Einsen, die eine n -Bibermaschine 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 $\text{bb}(n)$ wird manchmal auch $\Sigma(n)$ geschrieben.

Radó-Funktion

Eine n -Bibermaschine heißt *fleißiger Biber*, wenn sie am Ende $\text{bb}(n)$ Einsen auf dem Band hinterlässt.

fleißiger Biber



Abbildung 20.5: Europäischer Biber (*castor fiber*), Bildquelle: http://upload.wikimedia.org/wikipedia/commons/d/d4/Beaver_2.jpg (10.1.2020)

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 fließiger Biber.

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

	A	B	C	D	H
\square	1, R, B	1, L, A	1, R, H	1, R, D	
1	1, L, B	\square , L, C	1, L, D	\square , R, A	

H. Marxen 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. Die frühere WWW-Seite <http://www.drb.insel.de/~heiner/BB/> mit weiteren Informationen existiert nicht mehr. Die „Wayback Machine“ auf <https://archive.org/index.php> fand zuletzt am 20.6.2017 etwas Archivierbares (<https://web.archive.org/web/20170620023155/http://www.drb.insel.de/~heiner/BB/>).

Im Jahre 2010 hat Pavel Kropitz eine 6-Bibermaschine gefunden, die mehr als $3.514 \cdot 10^{18267}$ Einsen schreibt und erst nach mehr als $7.412 \cdot 10^{36534}$ Schritten hält. 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>, 17.1.19) 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.

20.3 Theorem Für jede berechenbare Funktion $f : \mathbb{N}_+ \rightarrow \mathbb{N}_+$ gibt es ein n_0 , so dass für alle $n \geq n_0$ gilt: $bb(n) > f(n)$.

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

20.5 AUSBLICK

Sie werden an anderer Stelle lernen, dass es eine weitere wichtige Komplexitätsklasse gibt, die **NP** heißt. Man weiß, dass $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$ gilt, aber für keine der beiden Inklusionen weiß man, ob sie echt ist. Gewisse Probleme aus **NP** und **PSPACE** (sogenannte *vollständige* Probleme) spielen an vielen Stellen (auch in der Praxis) eine ganz wichtige Rolle. Leider ist es in allen Fällen so, dass alle bekannten Algorithmen exponentielle Laufzeit haben, aber man nicht beweisen kann, dass das so sein muss.

LITERATUR

Turing, A. M. (1936). „On computable numbers, with an application to the Entscheidungsproblem“. In: *Proceedings of the London Mathematical Society*. 2. Ser. 42, S. 230–265.

Die Pdf-Version einer HTML-Version ist online verfügbar; siehe <http://web.comlab.ox.ac.uk/oucl/research/areas/ieg/e-library/sources/tp2-ie.pdf> (10.1.2020) (siehe S. 219).

21 RELATIONEN

21.1 ÄQUIVALENZRELATIONEN

21.1.1 Definition

In Abschnitt 15.2.1 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:

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

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

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

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 .

21.1.2 Äquivalenzklassen und Faktormengen

Äquivalenzklasse 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.

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

Faserung

Ist konkret \equiv die Äquivalenzrelation „modulo n “ auf den ganzen Zahlen, dann schreibt man für die Faktormenge auch \mathbb{Z}_n .

21.2 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.

21.2.1 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.

verträglich 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:

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

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:

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

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

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

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

21.2.2 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

21.3 HALBORDNUNGEN

Eine Ihnen wohlvertraute Halbordnung ist die Mengeninklusion \subseteq . Entsprechende Beispiele tauchen daher im folgenden immer wieder auf.

21.3.1 Grundlegende Definitionen

Eine Relation $R \subseteq M \times M$ heißt *antisymmetrisch*, wenn für alle $x, y \in M$ gilt:

Antisymmetrie

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

Eine Relation $R \subseteq M \times M$ heißt *Halbordnung*, wenn sie

Halbordnung

- reflexiv,
- antisymmetrisch und
- transitiv

ist. Wenn R eine Halbordnung auf einer Menge M ist, sagt man auch, die Menge sei *halbgeordnet*.

halbgeordnete Menge

Auf der Menge aller Wörter über einem Alphabet A ist die Relation \sqsubseteq_p ein einfaches Beispiel, die definiert sei vermöge der Festlegung $w_1 \sqsubseteq_p w_2 \leftrightarrow \exists u \in A^* : w_1 u = w_2$. Machen Sie sich zur Übung klar, dass sie tatsächlich die drei definierenden Eigenschaften einer Halbordnung hat.

Es sei M' eine Menge und $M = 2^{M'}$ die Potenzmenge von M' . Dann ist die Mengeninklusion \subseteq eine Halbordnung auf M . Auch hier sollten Sie noch einmal aufschreiben, was die Aussagen der drei definierenden Eigenschaften einer Halbordnung sind. Sie werden merken, dass die Antisymmetrie eine Möglichkeit an die Hand gibt, die Gleichheit zweier Mengen zu beweisen (wir haben das auch schon ausgenutzt).

Wenn R Halbordnung auf einer *endlichen* Menge M ist, dann stellt man sie manchmal graphisch dar. Wir hatten schon in Unterabschnitt 15.1.4 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 21.1. Wie man sieht wird das ganze recht schnell

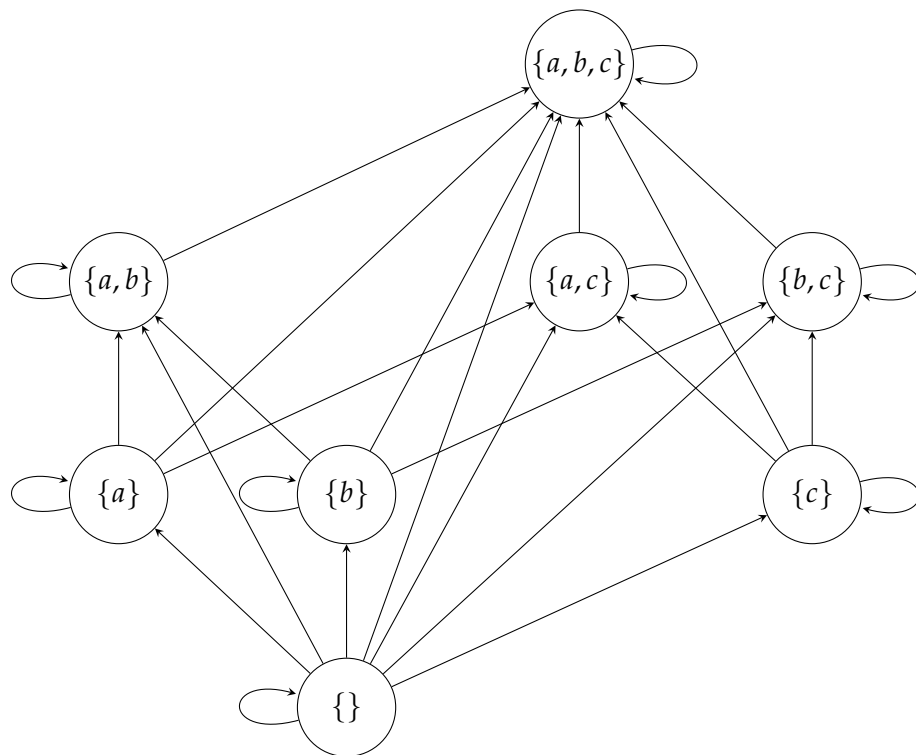


Abbildung 21.1: Die Halbordnung $(2^{\{a,b,c\}}, \subseteq)$ als Graph

relativ unübersichtlich. Dabei ist ein Teil der Kanten nicht ganz so wichtig, weil deren Existenz ohnehin klar ist (wegen der Reflexivität) oder aus anderen Kanten gefolgert werden kann (wegen der Transitivität). Deswegen wählt man meist die Darstellung als sogenanntes *Hasse-Diagramm* dar. Das ist eine Art „Skelett“ der Halbordnung, bei dem die eben angesprochenen Kanten fehlen. Genauer gesagt ist es der Graph der Relation $H_R = (R \setminus I) \setminus (R \setminus I)^2$. In unserem Beispiel ergibt sich aus Abbildung 21.1 durch Weglassen der Kanten Abbildung 21.2.

Vom Hassediagramm kommt man „ganz leicht“ wieder zur ursprünglichen Halbordnung: Man muss nur die reflexiv-transitive Hülle bilden.

Hasse-Diagramm

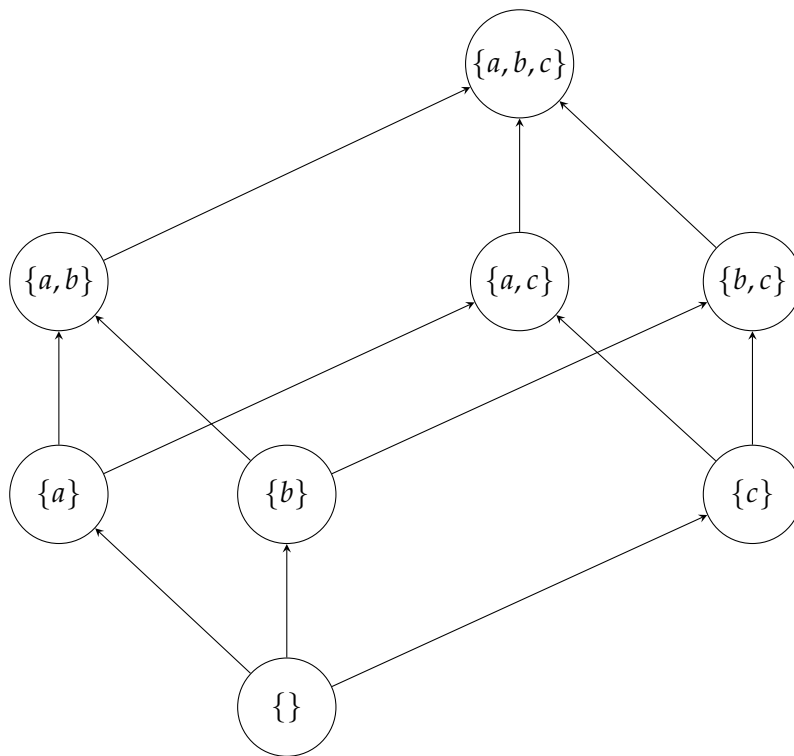


Abbildung 21.2: Hassediagramm der Halbordnung $(2^{\{a,b,c\}}, \subseteq)$

21.1 Lemma. Wenn R eine Halbordnung auf einer endlichen Menge M ist und H_R das zugehörige Hassediagramm, dann ist $H_R^* = R$.

21.2 Beweis. R und H_R sind beides Relationen über der gleichen Grundmenge M (also ist $R^0 = H_R^0$) und offensichtlich ist $H_R \subseteq R$. Eine ganz leichte Induktion (machen Sie sie) zeigt, dass für alle $i \in \mathbb{N}_0$ gilt: $H_R^i \subseteq R^i$ und folglich $H_R^* \subseteq R^*$. Da R reflexiv und transitiv ist, ist $R^* = R$, also $H_R^* \subseteq R$.

Nun wollen wir zeigen, dass umgekehrt auch gilt: $R \subseteq H_R^*$. Sei dazu $(x, y) \in R$. Falls $x = y$ ist, ist auch $(x, y) \in I \subseteq H_R^*$.

Sei daher im folgenden $x \neq y$, also $(x, y) \in R \setminus I$, und sein (x_0, x_1, \dots, x_m) eine Folge von Elementen mit folgenden Eigenschaften:

- $x_0 = x$ und $x_m = y$
- für alle $0 \leq i < m$ ist $x_i \sqsubseteq x_{i+1}$
- für alle $0 \leq i < m$ ist $x_i \neq x_{i+1}$

In einer solchen Folge kann kein Element $z \in M$ zweimal auftauchen. Wäre nämlich $x_i = z$ und $x_k = z$ mit $k > i$, dann wäre jedenfalls $k \geq i + 2$ und $x_{i+1} \neq z$.

Folglich wäre einerseits $z = x_i \sqsubseteq x_{i+1}$ und andererseits $x_{i+1} \sqsubseteq \dots \sqsubseteq x_k$, also wegen Transitivität $x_{i+1} \sqsubseteq x_k = z$. Aus der Antisymmetrie von \sqsubseteq würde $x_{i+1} = z$ folgen im Widerspruch zum eben Festgehaltenen.

Da in einer Folge (x_0, x_1, \dots, x_m) der beschriebenen Art kein Element zweimal vorkommen kann und M endlich ist, gibt es auch maximal lange solche Folgen. Sei im folgenden (x_0, x_1, \dots, x_m) maximal lang. Dann gilt also für alle $0 \leq i < m$, dass man zwischen zwei Elemente x_i und x_{i+1} kein weiteres Element einfügen kann, dass also gilt: $\neg \exists z \in M : x_i \sqsubseteq z \sqsubseteq x_{i+1} \wedge x_i \neq z \wedge x_{i+1} \neq z$.

Dafür kann man auch schreiben: $\neg \exists z \in M : (x_i, z) \in R \setminus I \wedge (z, x_{i+1}) \in R \setminus I$, d. h. $(x_i, x_{i+1}) \notin (R \setminus I)^2$.

Also gilt für alle $0 \leq i < m$: $(x_i, x_{i+1}) \in (R \setminus I) \setminus (R \setminus I)^2 = H_R$. Daher ist $(x, y) = (x_0, x_m) \in H_R^m \subseteq H_R^*$. ■

gerichtete azyklische
Graphen
Dag

Graphen, die das Hassediagramm einer endlichen Halbordnung sind, heißen auch *gerichtete azyklische Graphen* (im Englischen *directed acyclic graph* oder kurz *Dag*), weil sie keine Zyklen mit mindestens einer Kante enthalten. Denn andernfalls hätte man eine Schlinge oder (fast die gleiche Argumentation wie eben im Beweis 21.2) des Lemmas verschiedene Elemente x und y mit $x \sqsubseteq y$ und $y \sqsubseteq x$.

Gerichtete azyklische Graphen tauchen an vielen Stellen in der Informatik auf, nicht nur natürlich bei Problemstellungen im Zusammenhang mit Graphen, sondern z. B. auch bei der Darstellung von Datenflüssen, im Compilerbau, bei sogenannten *binary decision diagrams* zur Darstellung logischer Funktionen usw.

21.3.2 „Extreme“ Elemente

minimales Element
maximales Element

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 $x \neq y$.

größtes Element
kleinstes Element


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

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 $\{a, b, c\}$.

obere Schranke
untere Schranke

Ein Element $x \in M$ heißt *obere Schranke* von T , wenn für alle $y \in T$ gilt: $y \sqsubseteq x$. Ein $x \in M$ heißt *untere Schranke* von T , wenn für alle $y \in T$ gilt: $x \sqsubseteq y$. In der Halbordnung $(2^{\{a,b,c\}}, \subseteq)$ besitzt zum Beispiel $T = \{\{\}, \{a\}, \{b\}\}$ zwei obere

Schranken: $\{a, b\}$ und $\{a, b, c\}$. Die Teilmenge $T = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$ besitzt die gleichen oberen Schranken.

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 $\sqcup T$ (oder $\sup(T)$).

Supremum
 $\sqcup T, \sup(T)$

Besitzt die Menge der unteren Schranken einer Teilmenge T ein größtes Element, so heißt dies das *Infimum* von T . Das werden wir in dieser Vorlesung aber nicht benötigen.

Infimum

Wenn eine Teilmenge kein Supremum besitzt, dann kann das daran liegen, dass sie gar keine oberen Schranken besitzt, oder daran, dass die Menge der oberen Schranken kein kleinstes Element hat. Liegt eine Halbordnung der Form $(2^M, \subseteq)$, dann besitzt aber jede Teilmenge $T \subseteq 2^M$ ein Supremum. $\sqcup T$ ist dann nämlich die Vereinigung aller Elemente von T (die Teilmengen von M sind).

21.3.3 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 $\sqcup_i x_i$.

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 $\sqcup_i T_i = \bigcup T_i$.

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 $N = \mathbb{N}_0 \cup \{u\}$ und definiert

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

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

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

also sozusagen

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

Ein anderes Beispiel einer (sogar totalen) Ordnung, die *nicht* vollständig ist, werden wir am Ende von Abschnitt 21.4 sehen.

21.3.4 Stetige Abbildungen auf vollständigen Halbordnungen

monotone Abbildung 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: $x \sqsubseteq y \rightarrow f(x) \sqsubseteq f(y)$.

Die Abbildung $f(x) = x + 1$ etwa ist auf der Halbordnung (\mathbb{N}_0, \leq) monoton. Die Abbildung $f(x) = x \bmod 5$ ist auf der gleichen Halbordnung dagegen nicht monoton, denn es ist zwar $3 \leq 10$, aber $f(3) = 3 \not\leq 0 = f(10)$.

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

Betrachten wir als erstes Beispiel noch einmal die vollständige Halbordnung $N' = \mathbb{N}_0 \cup \{u_1, u_2\}$ von 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 $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ gibt es nur zwei Möglichkeiten:

- Die Kette wird konstant. Es gibt also ein $n' \in N'$ und ein $i \in \mathbb{N}_0$ so dass gilt: $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_i = x_{i+1} = x_{i+2} = \cdots = n'$. Dann ist jedenfalls $\bigsqcup_i x_i = n'$. Es gibt nun drei Unterfälle zu betrachten:
 - Wenn $n' = u_2$ ist, dann ist wegen $f(u_2) = u_2$ ist auch $\bigsqcup_i f(x_i) = u_2$, also ist $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$.
 - Wenn $n' = u_1$, gilt eine analoge Überlegung.
 - Wenn $n' \in \mathbb{N}_0$ ist, dann ist $f(\bigsqcup_i x_i) = f(n') = n' + 1$. Andererseits ist die Kette der Funktionswerte $f(x_0) \sqsubseteq f(x_1) \sqsubseteq f(x_2) \sqsubseteq \cdots \sqsubseteq f(x_i) = f(x_{i+1}) = f(x_{i+2}) = \cdots = f(n') = n' + 1$. Also ist $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$.
- Der einzige andere Fall ist: die Kette wird nicht konstant. Dann müssen alle $x_i \in \mathbb{N}_0$ sein, und die Kette wächst unbeschränkt. Das gleiche gilt dann auch für die Kette der Funktionswerte. Also haben beide als Supremum u_1 und wegen $f(u_1) = u_1$ ist $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$.

Der letzte Fall zeigt einem auch gleich schon, dass dagegen die folgende Funktion $g : N' \rightarrow N'$ 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.

21.3 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 $x_f \sqsubseteq y_f$.

21.4 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 gilt für die kleinste obere Schranke: $x_f = \bigsqcup_i x_i \sqsubseteq y_f$. ■

Dieser Fixpunktsatz (und ähnliche) finden in der Informatik an mehreren Stellen Anwendung. Zum Beispiel kann er Ihnen in Vorlesungen über 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 hat, sehen Sie vermutlich.

21.4 ORDNUNGEN

Eine Relation $R \subseteq M \times M$ ist eine *Ordnung*, oder auch genauer *totale Ordnung*, wenn R Halbordnung ist und außerdem gilt:

Ordnung
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 Alphabets 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 in Lexika usw. sortiert sind, kann man wie folgt definieren. Seien $w_1, w_2 \in A^*$. Dann gibt es das eindeutig bestimmte maximal lange gemeinsame Präfix von w_1 und w_2 , also das maximal lange Wort $v \in A^*$, so dass es $u_1, u_2 \in A^*$ gibt mit $w_1 = v u_1$ und $w_2 = v u_2$. Drei Fälle sind möglich:
 1. Falls $v = w_1$ ist, gilt $w_1 \sqsubseteq_1 w_2$.
 2. Falls $v = w_2$ ist, gilt $w_2 \sqsubseteq_1 w_1$.
 3. Falls $w_1 \neq v \neq w_2$, gibt es $x, y \in A$ und $u'_1, u'_2 \in A^*$ mit
 - $x \neq y$ und
 - $w_1 = v x u'_1$ und $w_2 = v y u'_2$.

Dann gilt $w_1 \sqsubseteq_1 w_2 \leftrightarrow x \sqsubseteq_A y$.

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, indem man festlegt, dass $w_1 \sqsubseteq_2 w_2$ genau dann gilt, 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:

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

Wir wollen noch darauf hinweisen, dass die lexikographische Ordnung \sqsubseteq_1 als Relation auf der Menge aller Wörter einige Eigenschaften hat, an die man als Anfänger vermutlich nicht gewöhnt ist. Zunächst einmal merkt man, dass die Ordnung nicht vollständig ist. Die aufsteigende Kette

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

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 kleinere 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.

21.5 AUSBLICK

Vollständige Halbordnungen spielen zum Beispiel eine wichtige Rolle, wenn man sich mit sogenannter denotationaler Semantik von Programmiersprachen beschäftigt und die Bedeutung von while-Schleifen und Programmen mit rekursiven Funktionsaufrufen präzisieren will. Den erwähnten Fixpunktsatz (oder verwandte Ergebnisse) kann man auch zum Beispiel bei der automatischen statischen Datenflussanalyse von Programmen ausnutzen. Diese und andere Anwendungen werden ihnen in weiteren Vorlesungen begegnen.

22 MIMA - X

Das folgende Kapitel erweitert die MIMA um Register für die bequeme Implementierung eines Stapels/Kellers und ineinander geschachtelter Funktionsaufrufe. Die Hilfsmittel sind zwar primitiv, dennoch ausreichend um das Konzept eines „*Callstacks*“ zu illustrieren.

Callstacks

22.1 STAPEL / STACK / KELLER

Ein Stapel ist auch unter vielen Namen bekannt, sei es nun Stack, Keller oder LIFO. Vorstellen kann man sich einen Stapel als einen wortwörtlichen Stapel (z. B. von Papierblättern). Man kann weitere Blätter auf den Stapel legen, ein Blatt von oben wegnehmen oder sich das oberste Blatt ansehen. Um diese Zugriffe zu vereinheitlichen, definieren wir im Folgenden zunächst einmal formal das Konzept eines Stapels.

22.1.1 Der Stapel

Ein *Stapel* ist festgelegt durch eine Liste/Tupel von Elementen aus einer Grundmenge V . Die Menge aller Stapel über V sei S .

Stapel

$$S = \bigcup_{i \in \mathbb{N}_0} V^i$$

Das erste (linke) Element der Liste ist das „unterste“ Element des Stapels und analog ist das letzte Element der Liste das „oberste“ Element des Stapels. Auch die leere Liste soll ein legaler Stapel sein, der sogenannte *leere Stapel*.

leerer Stapel

22.1.2 Operationen auf einem Stapel

Um dem Stapel etwas hinzuzufügen benutzen wir *push*.

push

$$\begin{aligned} \text{push: } S \times V &\rightarrow S \\ ((), v) &\mapsto (v) \\ ((x_1, \dots, x_n), v) &\mapsto (x_1, \dots, x_n, v) \end{aligned}$$

Die Operation *push* legt das Element $v \in V$ „oben“ auf den Stapel ohne den Rest des Stapels zu verändern.

Das oberste Element eines Stapels lässt sich mit *pop* entnehmen.

pop

$$\begin{aligned}\text{pop}: S &\dashrightarrow S \\ (x_1, \dots, x_n, v) &\mapsto (x_1, \dots, x_n)\end{aligned}$$

Achtung! pop ist für den leeren Stapel nicht definiert.

Nur mit push und pop können wir uns jeden beliebigen Stapel bauen. Um die Informationen in unserem Stapel auch nutzen zu können definieren wir *top*.

$$\begin{aligned}\text{top}: S &\dashrightarrow V \\ (x_1, \dots, x_n) &\mapsto x_n\end{aligned}$$

Achtung! top ist für den leeren Stapel ebenfalls nicht definiert.

Die Operation top liefert also immer das oberste Element eines Stapels zurück, ohne diesen zu verändern. Manche Programmiersprachen bzw. Programmbibliotheken bieten direkt Implementierungen für entsprechende Datenstrukturen an. Dort wird top manchmal zusammen mit pop in einer Funktion integriert, was Code vereinfachen kann. Formal ist es uns aber lieber, diese beiden Operationen getrennt zu halten, da so immer klar wird ob Informationen ausgelesen werden oder der Stapel geändert wird.

peek Der Bequemlichkeit halber kann man zusätzlich auch noch *peek* definieren. In manchen Fällen spart man sich so Programmieraufwand, es ist aber so auch einfach, den Sinn eines Stapels zu missachten, indem man beliebige Element betrachtet.

$$\begin{aligned}\text{peek}: S \times \mathbb{N}_0 &\dashrightarrow V \\ ((), k) &\mapsto \text{undefiniert} \\ ((x_1, \dots, x_n), k) &\mapsto \begin{cases} x_{n-k}, & \text{falls } 0 \leq k < n \\ \text{undefiniert}, & \text{sonst} \end{cases}\end{aligned}$$

height Um in Algorithmen Aufrufe von pop und peek zu vermeiden, wenn der Stapel leer ist, brauchen wir noch eine Möglichkeit, etwas über die „Größe“ des Stapels zu erfahren. Wir nennen die Anzahl der Elemente eines Stapel seine *Höhe*.

$$\begin{aligned}\text{height}: S &\rightarrow \mathbb{N}_0 \\ (x_1, \dots, x_n) &\mapsto n \\ () &\mapsto 0\end{aligned}$$

Wenn ihnen jetzt auffällt, dass height gerade die Anzahl der Element in einem Stapel zurückgibt, dann liegen sie richtig.

In Abbildung 22.1 sind ein paar Beispiele zu sehen.

$$\begin{aligned}
&\text{push}((3, 5, 7), 2) = (3, 5, 7, 2) \\
&\text{push}((3, 5, 7, 2), 42) = (3, 5, 7, 2, 42) \\
&\text{peek}((3, 5, 7, 2, 42), 2) = 7 \\
&\text{pop}((3, 5, 7, 2)) = (3, 5, 7) \\
&\text{pop}((3, 5, 7)) = (3, 5) \\
&\text{pop}((3, 5)) = (3) \\
&\text{pop}((3)) = () \\
&\text{peek}((0, 5, 11, 9), 0) = \text{top}((0, 5, 11, 9)) = 9
\end{aligned}$$

Abbildung 22.1: Beispiel-Operationen

22.1.3 Implementierung eines Stapels

Zur Implementierung eines Stapels benutzt man meist eine Liste aufeinanderfolgender Speicherplätze als Grundlage. Zusätzlich zu dem eigentlichen Speicherbereich muss man sich die Adresse des *nächsten freien* oder /und des *letzten belegten* Eintrags speichern. Wir werden immer die Adresse des nächsten freien Eintrags speichern.

Weiterhin muss man sich entscheiden in welche Richtung der Stapel wachsen soll. Hier werden wir Stapel verwenden, welche von niedrigen zu großen Adressen hin wachsen („nach oben“). Das bedeutet dass das oberste Element des Stapels die größte Adresse hat.

Richtung

In vielen modernen Systemen benutzt man Stapel, die von großen zu kleinen Adressen wachsen.

genauer: Wenn Offsets in die falsche Richtung gehen kann das sehr schnell zu einem korrupten Programm führen. Es ist deswegen sehr wichtig, beim Entwerfen von Programmen die Definition des verwendeten Stapels sehr genau zu beachten. Da dies aber durch push, pop, top und peek sehr gut abstrahiert wird, ist der Debrauch von Stacks in modernen Programmiersprachen erheblich erleichtert.

22.1.4 Die Ackermann-Funktion

Die *Ackermann-Funktion* ist ein gutes Beispiel für Rekursion da man sie nicht ohne Rekursion berechnen kann. Manche Funktionen, die rekursiv definiert sind, können auch anders berechnet werden, wie zum Beispiel die Fakultät.

Ackermann-Funktion

Die Definition der Ackermann-Funktion ist in Abbildung 22.2 zu sehen.

Das Berechnen der Ackermann-Funktion wird extrem schnell extrem schwierig, da die Rekursionsschritte mit einer ungeheueren Geschwindigkeit zunehmen. Dies lässt sich in Abbildung 22.10 erkennen. Bei $A(2,2)$ benötigt man schon 27

$$\begin{aligned}
A: \mathbb{N}_0 \times \mathbb{N}_0 &\rightarrow \mathbb{N}_0 \\
\forall y \in \mathbb{N}_0 : & \quad A(0, y) = y + 1 \\
\forall x \in \mathbb{N}_0 : & \quad A(x + 1, 0) = A(x, 1) \\
\forall x, y \in \mathbb{N}_0 : & \quad A(x + 1, y + 1) = A(x, A(x + 1, y))
\end{aligned}$$

Abbildung 22.2: Die Ackermann-Funktion

Schritte. Dennoch fällt bei der Berechnung auf, dass immer nur die *zwei letzten* Argumente betrachtet werden müssen.

Hier bietet sich ein auf einem Stapel basierender Algorithmus an. S ist hier zu Beginn ein Stack über \mathbb{N}_0 : (x,y). Der Algorithmus in Abbildung 22.3 berechnet $A(x, y)$. Bei genauerer Betrachtung des Codes der Abbildung 22.3 sollte der

```

while S.height() != 1
    y ← S.peek(0)
    S.pop()
    x ← S.peek(0)
    S.pop()
    if x = 0
        S ← S.push(y + 1)
    else if y = 0
        S ← S.push(x - 1)
        S ← S.push(1)
    else
        S ← S.push(x - 1)
        S ← S.push(x)
        S ← S.push(y - 1)
    fi
do

```

Abbildung 22.3: Die Ackermann-Funktion als Algorithmus

Leser bemerken, dass der Algorithmus eine simple Implementierung der in Abbildung 22.2 beschriebenen Definition ist. Leider muss hier von der „schönen“ Definition abgewichen werden, da die Argumente nur direkt vorliegen. Der Algorithmus holt die Argumente X und Y in *umgekehrter* Reihenfolge vom Stapel und legt das Ergebnis wieder auf dem Stapel ab. Der Algorithmus terminiert sobald

Abbildung 22.4: Die MIMA-x

die letzten zwei Argumente verarbeitet wurden und so die Höhe des Stapels auf eins gesunken ist. Dieses letzte Element ist dann das Ergebnis von $A(x,y)$.

22.2 MIMA-X

Die MIMA ist aufgrund des simplen Befehlsatzes ein gutes Beispiel um Computer kennenzulernen. Sie allein unterstützt jedoch keine Funktionsaufrufe oder einen leicht zu verwendenden Stapel. Im folgenden Kapitel definieren wir deswegen die MIMA-X.

22.2.1 Architektur

Die MIMA-x führt 3 neue Register ein. Um eventuelle Konflikt mit dem Befehlsteil des Wortes zu vermeiden benutzt die MIMA-x-Architektur 28 bit breite Wörter und 20 bit breite Adressen. *Return Adress* (RA) speichert bei Funktionsaufrufen die *Rücksprungadresse* ab. RA ist 20 bit breit und kann somit also genau eine Adresse abspeichern.

return adress

Der *Stack Pointer* (SP) zeigt auf den *nächsten freien* Eintrag des Stacks. Damit ist gemeint, dass die Adresse in SP immer die Adresse des obersten Elements + 1 ist. Als Adressregister hat SP natürlich eine Breite von 20 bit.

stack pointer

Der *Frame Pointer* (FP) zeigt auf das oberste Element des Stapels vor einem Funktionsaufruf. Die genaue Funktionsweise dieses Registers wird in den folgenden Unterkapiteln nicht weiter erläutert, da dies den Umfang dieses Kapitels sprengen würde. FP ist ebenfalls 20 bit breit.

frame pointer

Zum Laden von Werten aus RA, SP und FP stellen wir *LDRA*, *LDSP* und *LDFP* zur Verfügung. Diese 3 Befehle kopieren jeweils den Wert von RA, SP oder FP in den Akkumulator und füllen die obersten 8 bit mit 0 auf.

LD-

Analog dazu gibt es *STRA*, *STSP* und *STFP*, welche den Wert im Akkumulator in jeweils RA, SP oder FP speichern. Die obersten 8 bit werden dabei ignoriert. Alle 6 Befehle nehmen keine Argumente.

ST-

22.2.2 Stapelimplementierung

Die Implementierung eines Stapels in der MIMA-x basiert auf der Verwendung des Registers SP, welches die Adresse des nächsten Freien Elements beinhaltet. Der Stapel wächst dabei nach oben, ein push erhöht also SP um 1. Um mit dem

<i>push:STVR 0(RA)</i>	<i>pop:LDSP</i>	<i>top:LDVR -1</i>
LDSP	ADC -1	
ADC 1	STSP	
STSP		

Abbildung 22.5: push, pop und top für die MIMA-x

Stapel arbeiten zu können benutzen wir eine indirekte Adressierung.

LDVR *LDVR ± offset* (SP) lädt den Wert an Speicherstelle <SP> ± offset in den Akkumulator. ± offset ist hier eine 24 bit breite Variable im Zweierkomplement. Damit kann ± offset jeden ganzzahligen Wert zwischen -2^{23} und $2^{23} - 1$ annehmen.

STVR Analog benutzen wir *STVR ± offset* (SP) zum Speichern des Akkumulators an Adresse <SP> ± offset. Indirekte Adressierung ist hier verwendet, da man bei Operationen auf einem Stapel meist Elemente in Abhängigkeit der Höhe adressieren möchte. top lässt sich z.B. durch *LDVR -1*(SP) erzielen.

Es ist hier wichtig zu beachten, dass (SP) hier kein variables Argument von LDVR bzw. STVR ist. Es hilft beim Lesen und Entwerfen von MIMA-x-Code sich zu erinnern welches Register als Basis für den offset genommen wird.

Um den vergleichsweise langen Code etwas zu kürzen führen wir hier zusätzlich noch *ADC ± offset* ein. Dieser Befehl addiert den offset auf den Akkumulator. Ohne *ADC ± offset* wäre der in Abbildung 22.6 gezeigte Code nötig.

In Abbildung 22.5 sieht man Implementierungen von push, pop und top.

Da nicht definiert ist wo der Stapel anfängt muss man aufpassen, dass der Stapel nicht „unterläuft“.

22.2.3 „Flache“ Funktionsaufrufe

Den meisten Lesern sollte zu diesem Zeitpunkt das Konzept eines Funktionsaufrufs bekannt sein. Zur Umsetzung solcher Funktionalität benötigt man Antworten auf folgende Fragen:

1. Wie spezifiziere ich die Funktion die ich aufrufen möchte?
2. Wie speichere ich, was nach der Funktion ausgeführt werden soll?
3. Wie übergebe ich Argumente?
4. Wie übergebe ich eventuelle Rückgabewerte der Funktion?

CALL Zur Beantwortung von Frage 1 und 2 führen wir zwei neue Befehle ein. *CALL* adresse ruft eine Funktion im Speicher an der Adresse adresse auf und speichert

die Adresse *plus eins* der CALL-Instruktion im Register RA ab.

RET springt wie ein JMP-Befehl zu der in RA gespeicherten Adresse. Nur durch diese zwei Befehle lassen sich rudimentäre Funktionen erstellen. RET

```
inc:    STV tempvar
        LDC 1
        ADD tempvar
        RET
tempvar:LDC 0
```

Abbildung 22.6: Eine „flache“ Funktion die den Akkumulator inkrementiert

In Abbildung 22.6 ist eine „flache“ Funktion beschrieben, welche den Akkumulator um eins erhöht. Man würde sie mit „CALL inc“ aufrufen.

„Flach“ sind diese Funktionen deshalb, da sie selbst keine Funktionen aufrufen können. Würde innerhalb der Funktion *inc* eine weitere Funktion (auch sie selbst) aufgerufen, so überschreibt der zweite CALL -Befehl den Rücksprungwert in RA. RET würde dann nicht mehr zum Aufrufer zurückkehren und die MIMA-x in einem ungewollten Zustand hinterlassen. „flache funktionen“

Fragen 3 und 4 wurden in Abbildung 22.6 bereits teilweise beantwortet. Für eine Eingabe und eine Ausgabe von 28 bit kann jeweils der Akkumulator genutzt werden. Bei komplexeren Eingaben muss man auf die Techniken des nächsten Unterkapitels zurückgreifen.

22.2.4 „Richtige“ Funktionsaufrufe mit einem Stapel

Das Problem mit den „flachen“ Funktionen ist das Überschreiben der Rücksprungadresse und der lokalen Parameter. Da bei hintereinandergestaffelten Funktionsaufrufen immer nur die „tiefste“ Funktion ausgeführt wird, bietet sich hier ein Stapel an. Der so genannte *Callstack* hält die derzeit pausierten Funktionen und ihre Parameter. *Callstack*

Die Einträge welche zu einem Funktionsaufruf gehören, nennt man *Frame*. Eine Möglichkeit diese „Frames“ zu organisieren wird in Abbildung 22.7 illustriert. *Frame*

Das Erläutern des Frame Pointers geht über den Umfang dieses Skripts hinaus. Wir verwenden deswegen die in Abbildung 22.7 gezeigte Struktur ohne einen Frame Pointer und dessen Einträge.

Ein Funktionsaufruf ist also wie folgt aufgebaut:

1. Der Aufrufer legt die Argumente auf den Stapel
2. Der CALL Befehl legt die Rücksprungadresse in RA ab und springt in die Funktion

3. Die Funktion legt die in RA gespeicherte Adresse auf den Stapel
4. Die Funktion rechnet und darf auch selbst Funktionen aufrufen, da die Rücksprungadresse auf dem Stapel gespeichert ist.
5. Die Funktion lädt die Rücksprungadresse vom Stapel in RA.
6. Die Funktion setzt den Stack Pointer auf das erste Argument. Dies „löscht“ den Stapeleintrag.
7. Die Funktion lädt den Rückgabewert in den Akkumulator oder schreibt diesen auf den Stapel.
8. Die Funktion gibt mit **RET** die Ausführung wieder an den Aufrufer zurück.

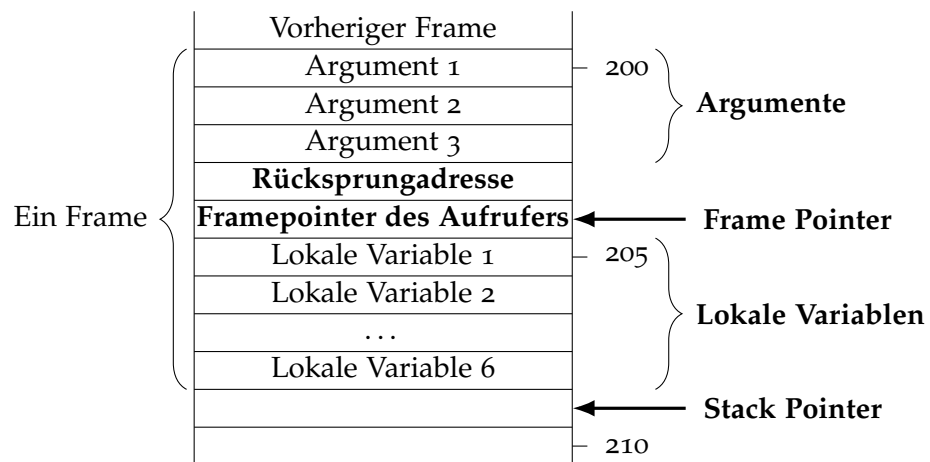


Abbildung 22.7: Ein Frame im Callstack

In der Abbildung 22.8 ist eine Umsetzung des Ganzen zu sehen. In Abbildung 22.9 sieht man den dazugehörigen C-Code.

Die genaue Funktionsweise von Funktionsaufrufen (auch auf komplexeren Maschinenmodellen) wird in Vorlesungen höherer Fachsemester behandelt.

```

factorial: LDRA      //Die Rücksprungadresse
          STVR 0      //wird auf dem Stack gespeichert
          LDSP        //Stack Pointer wird um 1 erhöht
          ADC 1
          STSP
          LDVR -2      //Das Argument x wird geladen
          ADC -1        //x - 1
          JMN fa_base  // if (x - 1) < 0 goto fa_base
          STVR 0        //Wir pushen x - 1 auf den Stack
          LDSP
          ADC 1
          STSP
          CALL factorial //factorial wird mit x - 1 aufgerufen
          LDVR -1      //Das Ergebnis [(x - 1)!] wird in fa_temp gespeichert
          STV fa_temp
          LDVR -3      //Das ursprüngliche Argument x wird geladen
          MUL fa_temp  // x * (x - 1)! wird in fa_temp gespeichert
          STV fa_temp
          LDSP        //Der Rückgabewert wird vom Stapel entfernt
          ADC -1
          STSP
          JMP fa_return //Sprung zum Funktionsende
fa_base   LDC 1        //Im Basisfall ist x = 0 (0! = 1)
          STV fa_temp
fa_return LDV fa_temp  //Der Rückgabewert wird geladen
          STVR -2      //Der Rückgabewert ersetzt das Argument
          LDSP        //Der Stapel wird verkleinert bis der Rückgabewert oben liegt
          ADC -1
          STSP
          LDVR 0        //Die Rücksprungadresse liegt über dem Stapel
          STRA        //Sie ist nicht mehr Teil des Stapels, existiert aber noch
          RET          //Die Funktion endet
fa_temp:   //Dieser Speicherplatz wird als temporäre Variable benötigt

```

Abbildung 22.8: Eine Funktion zur Berechnung der Fakultät (x!)

```
int factorial(int i){
    int result;
    if (i - 1 < 0){
        result = 1; //fa_base
    } else {
        result = i * factorial(i - 1); //Rekursionsfall
    }
    return result;
}
```

Abbildung 22.9: Die Fakultät in C

$$\begin{aligned}
1 \quad & A(2,2) \\
&= A(1, A(2,1)) \\
&= A(1, A(1, A(2,0))) \\
&= A(1, A(1, A(1,1))) \\
&= A(1, A(1, A(0, A(1,0)))) \\
&= A(1, A(1, A(0, A(0,1)))) \\
&= A(1, A(1, A(0,2))) \\
&= A(1, A(1,3)) \\
&= A(1, A(0, A(1,2))) \\
&= A(1, A(0, A(0, A(1,1)))) \\
&= A(1, A(0, A(0, A(0, A(1,0))))) \\
&= A(1, A(0, A(0, A(0, A(0,1))))) \\
&= A(1, A(0, A(0, A(0,2)))) \\
&= A(1, A(0, A(0,3))) \\
&= A(1, A(0,4)) \\
&= A(1,5) \\
&= A(0, A(1,4)) \\
&= A(0, A(0, A(1,3))) \\
&= A(0, A(0, A(0, A(1,2)))) \\
&= A(0, A(0, A(0, A(0, A(1,1))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(1,0))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0,1))))) \\
&= A(0, A(0, A(0, A(0, A(0,2)))) \\
&= A(0, A(0, A(0, A(0,3)))) \\
&= A(0, A(0, A(0,4))) \\
&= A(0, A(0,5)) \\
&= A(0,6) \\
&= 7
\end{aligned}$$

Abbildung 22.10: Die Berechnung von $A(2,2)$

INDEX

- $f \asymp g$, 180
- A Theorem on Boolean Matrices*, 173, 176, 266
- A^n , 25
- A^* , 24
- Abbildung, 19
- bijektiv, 20
 - identische, 66
 - injektiv, 19
 - Menge aller, 65
 - monotone, 246
 - partielle, 19
 - stetige, 246
 - surjektiv, 20
- Abeck, Sebastian, 2, 3, 265
- abgelehntes Wort, 203
- ablehnender Zustand, 202
- Ableitung, 110
- im Hilbert-Kalkül, 129
 - in einem Kalkül, 44
- Ableitungsbaum, 113
- Ableitungsfolge, 111
- Ableitungsschritt, 110
- Ackermann-Funktion, 53
- Addition
- von Matrizen, 166
- adjazente Knoten, 148, 154
- Adjazenzliste, 161
- Adjazenzmatrix, 162
- akzeptierender Zustand
- bei Turingmaschinen, 225
 - bei endlichen Automaten, 202
- akzeptierte formale Sprache
- bei endlichen Automaten, 203
 - bei Turingmaschinen, 225
- akzeptiertes Wort
- bei Turingmaschinen, 225
 - bei endlichen Automaten, 203
- Akzeptor, endlicher, 202
- al-Khwarizmi, Muhammad ibn Musa, 133
- Algebra, 133
- Algorithmus, 134, 219
- informell, 135
- Algorithmus von Warshall, 173
- allgemeingültige Formel, 42
- Alphabet, 15
- An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*, 37, 47, 265
- Anfangskonfiguration, 225
- antisymmetrische Relation, 241
- ASCII, 15
- Tabelle, 16
- assoziative Operation, 32
- asymptotisches Wachstum, 180
- aufschreiben
- früh, 99
- aufsteigende Kette, 245
- aufzählbare Sprache, 226
- Ausgabe auf dem Band
- bei Turingmaschinen, 225
- Ausgabefunktion, 198, 200
- Ausgang, 85
- Ausgangsgrad, 151
- Aussagen
- äquivalente, 40
- Aussagenlogik
- Zweiwertigkeit, 33
- aussagenlogische Formel, 34
- aussagenlogische Formel, 36
- Aussagevariablen, 34

Auszeichnungssprache, 100
 Automat
 endlicher, 198
 Mealy-, 198
 Moore-, 200
 average case, 178
 Axiome
 für Hoare-Kalkül, 139

 Bachelorarbeit, 99
 Baum
 gerichtet, 151
 Höhe, 215
 Kantorowitsch-Baum, 214
 ungerichtet, 155
 bedingter Sprungbefehl, 93
 Befehlsausführungsphase, 94
 Befehlsdecodierphase, 94
 Befehlsholphase, 94
 Berechnung
 endliche, 224
 haltend, 224
 nicht haltend, 224
 unendliche, 224
 Beweis, 44, 129, 130
 Biber
 fleißiger, 235
 Bibermaschine, 235
 bijektive Abbildung, 20
 Binärdarstellung, 60
 binäre Relation, 19
 Bit, 79
 Blanksymbol, 221
 Block-Codierung, 78
 Boole, George, 37, 47, 265
 boolesche Funktion, 37
 Busy-Beaver-Funktion, 235
 Byte, 79

$c_0(w)$, 225
 Champollion, Jean-Francois, 9
 Code, 68
 präfixfreier, 70
 Codewort, 68
 Codierung, 68
 computational complexity, 177
Concrete Mathematics, 185, 193, 265
 Coppersmith, Don, 189
 Coppersmith, Don, 189, 193, 265

 Dag, 244
 Darstellung von Zahlen, 63
 Datum, 8
 Decodierphase, 94
 Deduktionstheorem
 der Aussagenlogik, 46
 Definitionsbereich, 19
 Diagonalisierung, 232
 div, 62
 Dokument, 99

 ε -freier Konkatenationsabschluss, 57
 E-Mail, 28
 einfacher Zyklus, 150
 Eingabealphabet, 224
 Eingang, 85
 Eingangsgrad, 151
 Endkonfiguration, 222
 endliche Berechnung, 224
 endlicher Akzeptor, 202
 endlicher Automat, 198
 entscheidbare Sprache, 226
 Entscheidungsproblem, 224
 erreichbarer Knoten, 150
 Erscheinungsbild, 99
 Erzeuger, 85
 erzeugte formale Sprache, 111

 f_*

- bei Moore-Automaten, 201
 - bei Mealy-Automaten, 199
- f_{**}
 - bei Moore-Automaten, 201
 - bei Mealy-Automaten, 200
- Faktormenge, 240
- Faserung, 240
- Fibonacci=Zahlen, 52
- fleißiger Biber, 235
- Form, 99
- formale Sprache
 - erzeugte, 111
- formale Sprache, 31
 - akzeptierte , bei endlichen Auto-
maten, 203
 - akzeptierte , bei Turingmaschinen,
225
 - kontextfrei, 111
 - Potenzen, 56
- Formale Systeme*, 34, 47, 265
- Formel
 - allgemeingültige, 42
 - aussagenlogische, 34, 36
- Friedl, Jeffrey, 218, 265
- früh aufschreiben, 99
- Funktion, 19
 - skomposition, 66
 - bijektiv, 20
 - boolesche, 37
 - identische, 66
 - injektiv, 19
 - partielle, 19
 - surjektiv, 20
- g^*
 - bei Moore-Automaten, 201
 - bei Mealy-Automaten, 200
- g_{**}
 - bei Mealy-Automaten, 200
- bei Moore-Automaten, 202
- Gaussian Elimination Is Not Optimal*, 189,
193, 265
- Generalisierung, 129
- geordnetes Paar, 17
- gerichteter azyklischer Graph, 244
- gerichteter Graph, 147
- geschlossener Pfad, 150
- gewichteter Graph, 157
- Goos, Gerhard, 2, 3, 265
- Grad, 151, 155
- Graham, Ronald, 185
- Graham, Ronald L., 185, 193, 265
- Grammatik
 - kontextfreie, 110
 - rechtslinear, 213
 - Typ-2, 214
 - Typ-3, 214
- Graph
 - gerichtet, 147
 - gerichteter azyklischer, 244
 - gewichtet, 157
 - isomorphe, 151, 154
 - kantenmarkiert, 157
 - knotenmarkiert, 156
 - schlingenfrei, 149, 154
 - streng zusammenhängend, 150
 - ungerichtet, 154
- Graphisomorphismus, 152
- Groß-O-Notation, 178
- größenordnungsmäßiges Wachstum, 180
- größtes Element, 244
- gültiges Hoare-Tripel, 138
- h^{**} , 68, 69
- halbgeordnete Menge, 241
- Halbordnung, 241
 - vollständige, 245
- Halten

- einer Turingmaschine, 222
- haltende Berechnung, 224
- Halteproblem, 233
- Hasse-Diagramm, 242
- Hexadezimal-
 - darstellung, 61
- Hilbert-Kalkül
 - Ableitung, 129
 - Theorem, 130
- Hoare-Kalkül, 139
- Hoare-Tripel, 138
 - gültig, 138
- Hoare=Kalkül
 - Axiome, 139
- Holphase
 - der Befehlsausführung, 94
- Homomorphismus, 68
 - ε -freier, 68
- Huffman-Codierung, 72
- Hypothese, 44
- Höhe eines Baumes, 215
- Hülle
 - reflexiv-transitive, 115
- I_M , 66, 114
- identische
 - Abbildung, 66
 - Funktion, 66
- IETF, 79
- Induktion
 - strukturelle, 217
 - vollständige, 49
 - über die Wortlänge, 69
- Induktionsanfang, 50
- Induktionsschritt, 50
- induzierte Operation, 241
- Infimum, 245
- Infixschreibweise, 110
- Informatik, 5

- Information, 7
- Inhalt, 99
- injektive Abbildung, 19
- Inschrift, 6
- Internet Engineering Task Force, 79
- Interpretation, 39
- inverse
 - Abbildung, 66
 - Funktion, 66
- Inzidenzlisten, 161
- isomorphe Graphen, 151, 154
- Isomorphismus
 - von Graphen, 152
- Kalkül
 - Ableitung, 44, 129
 - Beweis, 129
 - Theorem, 44, 130
- kantenmarkierter Graph, 157
- Kantenrelation, 154
- Kantorowitsch-Baum, 214
- kartesisches Produkt, 18
- Klammerausdruck
 - korrekter, 113
 - wohlgeformter, 113
- Kleene, Stephen C., 174, 176, 209, 218, 265
- Kleene, Stephen Cole, 174, 209
- kleinstes Element, 244
- Knoten
 - adjazente, 148, 154
 - erreichbarer, 150
- knotenmarkierter Graph, 156
- Knuth, Donald, 185
- Knuth, Donald, 100
- Knuth, Donald E., 185, 193, 265
- kollisionsfreie Substitution, 126
- kommutative Operation, 31
- Komplexitätsklasse, 229

Komplexitätsmaß, 177, 228
 Komplexoperationen, 185
 Komposition
 von Funktionen, 66
 Konfiguration, 221
 Kongruenz modulo n , 239
 Kongruenzrelation, 241
 Konkatenation
 von Wörtern, 25
 Konkatenationsabschluss, 57
 ε -freier, 57
 Konnektive
 aussagenlogische, 34
 kontextfreie Grammatik, 110
 kontextfreie Sprache, 111
 korrekter Klammerausdruck, 113
 Kryptographie, 219
Kursbuch Informatik, Band 1, 2, 3, 265

 LaTeX, 100
 leeres Wort, 24
 Lehman, Eric, 2, 3, 265
 Leighton, Tom, 2, 3, 265
Lernen: Gehirnforschung und Schule des Lebens, 2, 3, 265
 lexikographische Ordnung, 249
 Linksableitung, 113
 linkseindeutige Relation, 19
 linkstotale Relation, 19
 Länge eines Pfades, 150
 Länge eines Weges, 154
 Länge eines Wortes, 23

 markup language, 100
 Masterarbeit, 99
Mastering Regular Expressions, 218, 265
 Mastertheorem, 190
Mathematics for Computer Science, 2, 3, 265

Matrix Multiplication via Arithmetic Progressions, 189, 193, 265
 Matrizenaddition, 166
 Matrizenmultiplikation, 165
 maximales Element, 244
 Mealy-Automat, 198
 Menge
 aller Abbildungen, 65
 Menge aller Wörter, 24
 Meyer, Albert R., 2, 3, 265
 Mikroprogramm, 94
 Mikroprogrammierung, 85
 Mima, 85
 minimales Element, 244
 mod, 62
 Modell
 für aussagenlogische Formeln, 41
 für prädikatenlogische Formeln, 122
 modulo n , 239
 Modus ponens, 43, 129
 monotone Abbildung, 246
 Moore-Automat, 200
 Morphogenese, 219
Multiplying matrices faster than Coppersmith-Winograd, 190, 193, 265

 Nachbedingung
 eines Hoare-Tripels, 138
 Nachricht, 7
 nicht haltende Berechnung, 224
 nichtdeterministische Automaten, 206
 Nichtterminalsymbol, 110
 Nullmatrix, 166

 $O(f)$, 183
 $O(1)$, 183
 $\Omega(f)$, 183
 obere Schranke, 244
 Octet, 79

On computable numbers, with an application to the Entscheidungsproblem, 219, 237, 265

Operation

assoziative, 32
induzierte, 241
kommutative, 31

Ordnung, 249

lexikographische, 249
total, 249

P, 230

Paar, 17

partielle Abbildung, 19

Patashnik, Oren, 185, 193, 265

Pfad, 149

geschlossener, 150
wiederholungsfrei, 150

Pfadlänge, 150

Platzkomplexität, 229

polynomiell

Raumkomplexität, 229
Zeitkomplexität, 228

Potenzen

einer formalen Sprache, 56
einer Relation, 114

Potenzen eines Wortes, 30

Potenzmenge, 21

Produkt

formaler Sprachen, 55
kartesisches, 18

Produkt der Relationen, 114

Produktion, 110

präfixfreier Code, 70

Prämisse, 44

PSPACE, 230

Quantor

Wirkungsbereich, 124

$\langle R \rangle$, 211

Radó, Tibor, 235

Rado-Funktion, 235

Rat fürs Studium, 99

Raumkomplexität, 229
polynomiell, 229

Rechenzeit, 177

rechtseindeutige Relation, 19

rechtslineare Grammatik, 213

rechtstotale Relation, 19

reflexiv-transitive Hülle, 115

reflexive Relation, 115

reguläre Sprache, 212

regulärer Ausdruck, 209

Relation, 19

antisymmetrisch, 241

binäre, 19

linkseindeutig, 19

linkstotal, 19

Potenz, 114

Produkt, 114

rechtseindeutig, 19

rechtstotal, 19

reflexiv-transitive Hülle, 115

reflexive, 115

symmetrisch, 156

ternäre, 19

transitive, 115

*Representation of Events in Nerve Nets
and Finite Automata*, 174, 176,
209, 218, 265

Repräsentation von Zahlen, 63

Reques for Comments, 79

Ressource, 177

RFC, 79

5322 (E-Mail), 28

3629, 71

E-Mail (5322), 28

Schleifeninvariante, 142
 Schlinge, 149, 154
 schlingenfreier Graph, 149, 154
 Schlussregel
 Generalisierung, 129
 Modus ponens, 43, 129
 Schmitt, P. H., 34, 47, 265
 Schranke
 obere, 244
 untere, 244
 Schritt einer Turingmaschine, 222
 Seminararbeit, 99
 Signal, 6
 Signum-Funktion, 168
 Speicher, 79
 Speicherplatzbedarf, 177
 Speicherung, 6
 Spitzer, Manfred, 2, 3, 265
 Sprache
 aufzählbare, 226
 entscheidbare, 226
 formale, 31
 Produkt, 55
 reguläre, 212
 Sprungbefehl
 bedingter, 93
 unbedingter, 93
 Startsymbol, 110
 stetige Abbildung, 246
 Strassen, Volker, 189, 193, 265
 streng zusammenhängender Graph, 150
 Struktur, 99
 strukturelle Induktion, 217
 Studium
 ein guter Rat, 99
 Substitution, 124
 kollisionsfrei, 126
 Supremum, 245
 surjektive Abbildung, 20
 symmetrische Relation, 156
 Tantau, Till, 99
 Tautologie, 42
 Teilgraph, 149, 154
 Teilpfad, 150
 Terminalsymbol, 110
 ternäre Relation, 19
 Theorem, 44, 130
 $\Theta(f)$, 182
 totale Ordnung, 249
 transitive Relation, 115
 Tupel, 17, 18
 Turing, A. M., 219, 237, 265
 Turing, Alan, 219
 Turingmaschine, 220
 Akzeptor, 225
 Ausgabe auf dem Band, 225
 Halten, 222
 Schritt, 222
 universelle, 231
 Twain, Mark, 1
 Typ-2-Grammatiken, 114, 214
 Typ-3-Grammatiken, 214
 Umkehrabbildung, 66
 Umkehrfunktion, 66
 unbedingter Sprungbefehl, 93
 unendliche Berechnung, 224
 ungerichteter Baum, 155
 ungerichteter Graph, 154
 universelle Turingmaschine, 231
 untere Schranke, 244
 UTF-8, 71
 Variable
 Vorkommen in einer Formel, 123
 Vassilevska Williams, Virginia, 190, 193, 265

Verbraucher, 85
 Verträglichkeit, 240
 vollständige Halbordnung, 245
 vollständige Induktion, 49
 Vorbedingung
 eines Hoare-Tripels, 138
 Vorkommen einer Variablen, 123
Vorlesungen über Informatik: Band 1: Grundlagen und funktionales Programmieren, 2, 3, 265
 Wachstum
 asymptotisches, 180
 größenordnungsmäßig, 180
 größenordnungsmäßig gleich, 180
 Warshall, Stephen, 173, 176, 266
 Warshall-Algorithmus, 173
 Watson, Thomas, 1
 Weg, 154
 Wegematrix, 163
 Weglänge, 154
 wiederholungsfreier Pfad, 150
 Winograd, Shmuel, 189, 193, 265
 Wirkungsbereich eines Quantors, 124
 Wohldefiniertheit, 70
 wohlgeformter Klammerausdruck, 113
 worst case, 178
 Wort, 23
 abgelehntes, 203
 akzeptiertes bei endlichen Automaten, 203
 akzeptiertes, bei Turingmaschinen, 225
 leeres, 24
 Länge, 23
 Potenzen, 30
 Wurzel, 151
 Zeitkomplexität, 228
 polynomiell, 228
 Zellularautomaten, 219
 Zielbereich, 19
 zusammenhängender ungerichteter Graph, 154
 Zusicherung, 138
 Zustand
 ablehnender, 202
 akzeptierender, bei endlichen Automaten, 202
 akzeptierender, bei Turingmaschinen, 225
 Zustandsüberföhrungsfunktion, 198, 200
 Zweierkomplement, 64
 Zweiwertigkeit, 33
 Zyklus, 150
 einfacher, 150
 Äquivalenzklasse, 240
 Äquivalenzrelation, 156, 239
 Übersetzung, 67
 äquivalente Aussagen, 40

LITERATUR

- Abeck, Sebastian (2005). *Kursbuch Informatik, Band 1*. Universitätsverlag Karlsruhe (siehe S. 2).
- Boole, George (1854). *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. URL: <http://www.gutenberg.org/ebooks/15114> (besucht am 20. 10. 2015) (siehe S. 37).
- Coppersmith, Don und Shmuel Winograd (1990). „Matrix Multiplication via Arithmetic Progressions“. In: *Journal of Symbolic Computation* 9, S. 251–280 (siehe S. 189).
- Friedl, Jeffrey (2006). *Mastering Regular Expressions*. 3rd edition. O’Reilly Media, Inc. (siehe S. 218).
- Goos, Gerhard (2006). *Vorlesungen über Informatik: Band 1: Grundlagen und funktionales Programmieren*. Springer-Verlag (siehe S. 2).
- Graham, Ronald L., Donald E. Knuth und Oren Patashnik (1989). *Concrete Mathematics*. Addison-Wesley (siehe S. 185).
- Kleene, Stephen C. (1956). „Representation of Events in Nerve Nets and Finite Automata“. In: *Automata Studies*. Hrsg. von Claude E. Shannon und John McCarthy. Princeton University Press. Kap. 1, S. 3–40.
Eine Vorversion ist online verfügbar; siehe http://www.rand.org/pubs/research_memoranda/2008/RM704.pdf (10.1.2020) (siehe S. 174, 209).
- Lehman, Eric, Tom Leighton und Albert R. Meyer (2013). *Mathematics for Computer Science*. URL: <http://courses.csail.mit.edu/6.042/fall13/class-material.shtml> (siehe S. 2).
- Schmitt, P. H. (2013). *Formale Systeme*. Vorlesungsskript. Institut für Theoretische Informatik, KIT. URL: <http://formal.itl.kit.edu/teaching/FormSysWS1415/skriptum.pdf> (besucht am 19. 10. 2015) (siehe S. 34).
- Spitzer, Manfred (2002). *Lernen: Gehirnforschung und Schule des Lebens*. Spektrum Akademischer Verlag (siehe S. 2).
- Strassen, Volker (1969). „Gaussian Elimination Is Not Optimal“. In: *Numerische Mathematik* 14, S. 354–356 (siehe S. 189).
- 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> (10.1.2020) (siehe S. 219).
- Vassilevska Williams, Virginia (2012). „Multiplying matrices faster than Coppersmith-Winograd“. In: *Proceedings of the 44th Symposium on Theory of Computing Confe-*

rence, *STOC 2012, New York, NY, USA, May 19 - 22, 2012*. Hrsg. von Howard J. Karloff und Toniann Pitassi. ACM, S. 887–898. ISBN: 978-1-4503-1245-5 (siehe S. 190).

Warshall, Stephen (1962). „A Theorem on Boolean Matrices“. In: *Journal of the ACM* 9, S. 11–12 (siehe S. 173).

COLOPHON

These lecture notes were prepared using *GNU Emacs* (<http://www.gnu.org/software/emacs/>) and in particular the *AucTeX* package (<http://www.gnu.org/software/auctex/>).

The notes are typeset with \LaTeX 2 ϵ (more precisely `pdfelatex`) with Peter Wilson's `memoir` document class using Hermann Zapf's *Palatino* and *Euler* type faces. The layout was inspired by Robert Bringhurst's book *The Elements of Typographic Style*.