

Theoretische Informatik II

Vorlesungsskript

VON

E. Best

Stand: Februar 2006

Vorwort

Das vorliegende Skript führt in die klassischen Schwerpunkte der Theoretischen Informatik ein: Formale Sprachen und Automaten, Berechenbarkeit und Komplexitätstheorie. Es beruhte ursprünglich auf einem Skriptum von V. Claus und E.R. Olderog, wurde inzwischen in mehreren Iterationen jedoch erheblich verändert. Neben der Neuorganisation des Stoffes (beginnend jetzt mit endlichen Automaten statt mit Turingmaschinen) und einer Überarbeitung der Stoffauswahl (u.a. Weglassen allgemeiner Rekursion und Aufnahme von Abschnitten über Codierungen, Programme und polynomielle Reduktionen) haben viele neue Beispiele Eingang gefunden. Auch die meisten Beweise (z.B. der Beweis der Äquivalenz von Turing-Akzeptierbarkeit und grammatischer Erzeugbarkeit) wurden neu gefasst.

Oldenburg, Februar 2006

E. Best

Das Skript wird ständig gepflegt. Wenn Ihnen beim Lesen Fehler auffallen, geben Sie diese bitte schriftlich im Sekretariat der Theoretischen Informatik bekannt oder schicken Sie eine Mail an

`eike.best@informatik.uni-oldenburg.de`.

Stand der Änderungen (Version): 11. Februar 2006.

ACHTUNG: Das Skript wird in mehreren Teilen verteilt. Beim ersten Teil sind ein vorläufiges Inhaltsverzeichnis und ein vorläufiger Index dabei. Diese beiden Teile werden zum Schluss der Vorlesung hin neu berechnet und aktualisiert zur Verfügung gestellt.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Modellbildungen in der Theoretischen Informatik	1
1.2	Literaturverzeichnis	3
2	Grundlagen	5
2.1	Mengentheoretische Grundbegriffe	6
2.1.1	Logik und Mengen	6
2.1.2	Relationen	7
2.1.3	Funktionen und Operationen	11
2.1.4	Zahlentheoretische Grundfunktionen	14
2.1.5	Mächtigkeit und Abzählbarkeit	15
2.2	Graphen und Bäume	20
2.3	Alphabete, Wörter und Sprachen	22
2.4	Elemente einer Programmiersprache	26
2.4.1	Deklarationen und Datentypen	26
2.4.2	Aufbau eines Programms	27
2.4.3	Primitive Kommandos	28
2.4.4	Kommandoverknüpfungen	28
2.4.5	WHILE- und LOOP-Registerprogramme	31
2.4.6	WHILE- und LOOP-berechenbare zahlentheoretische Funktionen	31
2.5	Codierungen	32
2.6	Übungsaufgaben	35
3	Ersetzungssysteme und Grammatiken	37
3.1	Ersetzungssysteme	37
3.2	Grammatiken	44
3.3	Übungsaufgaben	51

4	Endliche Automaten und reguläre Sprachen	53
4.1	Endliche Automaten: Definition und Beispiele	53
4.2	Endlich akzeptierbare Sprachen und Chomsky-3-Sprachen	56
4.3	Abschlusseigenschaften	62
4.4	Reguläre Ausdrücke	65
4.5	Struktureigenschaften regulärer Sprachen	68
4.5.1	Das Pumping-Lemma für reguläre Sprachen	69
4.5.2	Sprachkongruenzen	70
4.5.3	Deterministische Minimalautomaten	73
4.6	Entscheidbarkeitsfragen	77
4.7	Übungsaufgaben	79
5	Kellerautomaten und kontextfreie Sprachen	81
5.1	Kontextfreie Grammatiken und kontextfreie Sprachen	81
5.1.1	Notation und Beispiele	81
5.1.2	Links- und Rechtsableitungen	82
5.1.3	Parsebäume	83
5.1.4	Mehrdeutigkeit	86
5.1.5	Normalformen kontextfreier Grammatiken	87
5.2	Das Pumping-Lemma für kontextfreie Sprachen	89
5.3	Kellerautomaten	93
5.3.1	Definition und Beispiel	93
5.3.2	Äquivalenz der Akzeptanzbedingungen	98
5.3.3	Äquivalenz von Kellerautomaten und kontextfreien Grammatiken	100
5.4	Abschlusseigenschaften	108
5.5	Deterministisch kontextfreie Sprachen	110
5.6	Entscheidbarkeitsfragen	114
5.7	Übungsaufgaben	118
6	Turingmaschinen	121
6.1	Aufbau einer Turingmaschine	122
6.2	Beispiele	124
6.3	Das Verhalten einer Turingmaschine	129
6.4	Umwandlung einer NTM in eine DTM	131
6.5	Turingmaschinen als Sprachakzeptoren	133
6.6	Turingmaschinen als Berechner von partiellen Funktionen	136
6.7	Turingmaschinen und Grammatiken	140
6.7.1	Chomsky-0-Sprachen	140
6.7.2	Chomsky-1-Sprachen	143
6.8	Übungsaufgaben	145

7	Berechenbarkeit und Entscheidbarkeit	147
7.1	Aufzählbarkeit und Entscheidbarkeit	147
7.1.1	Definitionen	147
7.1.2	Codierung von Entscheidungsproblemen als Sprachen	149
7.1.3	Beziehungen zwischen Aufzählbarkeit und Entscheidbarkeit	153
7.2	Unentscheidbarkeit und Unberechenbarkeit	155
7.2.1	Abzählbarkeitsargumente und DTM-Codierungen	155
7.2.2	Das spezielle Halteproblem	156
7.2.3	Algorithmische Reduktion	157
7.2.4	Weitere Halteprobleme und universelle Turingmaschinen	159
7.2.5	Der Satz von Rice	162
7.3	Unentscheidbarkeitsresultate bei Grammatiken	165
7.3.1	Unentscheidbarkeitsresultate bei allgemeinen Grammatiken	165
7.3.2	Das Postsche Korrespondenzproblem	166
7.3.3	Unentscheidbarkeitsresultate bei kontextfreien Grammatiken	171
7.4	Zusammenfassende Bemerkungen zur Chomsky-Hierarchie	173
7.5	Übungsaufgaben	176
8	Komplexität	179
8.1	Modifizierte Turingmaschinenmodelle	180
8.1.1	Mehrspurmaschinen	180
8.1.2	Mehrbandmaschinen	180
8.1.3	Offline-Maschinen	181
8.2	Zeit- und Platzkomplexitätsbegriffe	182
8.3	O -Notationen	186
8.4	Die Klassen P , NP und $PSPACE$	186
8.5	Beispiele für Probleme aus der Klasse NP	188
8.6	Die Klasse NPC und polynomielle Reduzierbarkeit	189
8.7	Das Erfüllbarkeitsproblem für Boolesche Ausdrücke	192
8.8	Einige weitere NP -vollständige Probleme	198

Kapitel 1

Einleitung

1.1 Modellbildungen in der Theoretischen Informatik

Anhand des Begriffs der Berechenbarkeit von Funktionen kann man verfolgen, wie die Theoretische Informatik Modelle zur Beantwortung allgemeiner Fragen bildet. Was ist zum Beispiel der Unterschied zwischen den beiden Funktionen:

$$f: \begin{cases} \mathbb{N} \rightarrow \mathbb{N} \\ x \mapsto x+1 \end{cases} \quad \text{und} \quad g: \begin{cases} \mathbb{N} \rightarrow \mathbb{N} \\ x \mapsto x \cdot (x+1) \end{cases} \quad ?$$

Vom mathematischen Standpunkt aus könnte man antworten: f und g sind nicht die gleichen Funktionen; für alle Werte außer für $x=1$ liefern sie verschiedene Ergebnisse. Vom algorithmischen Standpunkt aus fällt in erster Linie auf, dass g etwas „schwerer berechenbar“ zu sein scheint als f , denn außer der Addition $+1$ muss auch noch eine Multiplikation geleistet werden. In dieser Antwort steckt schon implizit eine Berechnungsvorschrift (d.h.: ein *Algorithmus*) zur tatsächlichen Berechnung von f bzw. g , d.h., zur automatischen Erzeugung der Ausgabewerte $f(x)$ bzw. $g(x)$ nach Eingabe von x .

Ganz unbedarft könnte man (etwa wegen der „Einfachheit“ der Menge \mathbb{N}) annehmen, dass alle Funktionen von \mathbb{N} nach \mathbb{N} im intuitiven Sinne berechenbar sein müssen. Das ist jedoch mit an Sicherheit grenzender Wahrscheinlichkeit nicht der Fall. Diese Erkenntnis hat sich erst langsam durchgesetzt, was keineswegs verwunderlich ist. Denn nehmen wir an, wir möchten beweisen, dass es eine nicht-berechenbare Funktion von \mathbb{N} nach \mathbb{N} gibt. Um dies streng zu zeigen, muss eine große Schwierigkeit überwunden werden, denn wir benötigen einen *formalen Berechenbarkeitsbegriff*. Erst wenn solch ein Begriff vorliegt, kann unzweideutig das Vorhandensein (oder auch die Nichtexistenz) einer nicht-berechenbaren Funktion nachgewiesen werden. Die *Turingmaschine*, die wir in dieser Vorlesung definieren werden (nach Alan Turing, 1936), ist gerade zu diesem Zweck erfunden worden. Sie ist ein (Denk-)Modell, durch das Turing formal diejenigen Objekte zu erfassen versuchte, die „im Prinzip“ mit einer Maschine (einem Computer) berechnet werden können.

Seit den dreißiger Jahren sind neben der Turingmaschine noch viele weitere Modelle und damit Formalisierungen des Begriffs „Berechenbarkeit“ entstanden (es entstehen auch heute immer noch weitere). Unter den bekanntesten sind die *μ -rekursiven Funktionen* (1936; hier wird eine Funktion berechenbar genannt, wenn sie sich durch eine rekursive Definition bestimmter Art darstellen lässt), der *λ -Kalkül* (1936; hier

wird eine Funktion berechenbar genannt, wenn sie sich durch einen gewissen Ausdruck eines Kalküls definieren lässt) und die *formalen Grammatiken* (nach Noam Chomsky und anderen, ab 1959; hier wird eine Funktion berechenbar genannt, wenn sie sich – als Relation und damit als Sprache betrachtet – durch eine gewisse Grammatik erzeugen lässt).

Ist man an der Existenz von nicht-berechenbaren Funktionen interessiert, müsste man eigentlich die Existenz solcher Funktionen im Prinzip für jedes einzelne dieser Berechenbarkeitsmodelle separat nachweisen. Andererseits (und für diesen Zweck: glücklicherweise) hat sich herausgestellt, dass diese Modelle immer auf die gleiche Klasse von Funktionen hinauslaufen. Das bedeutet: ist eine Funktion Turing-berechenbar, dann ist sie auch μ -rekursiv, und umgekehrt; ist eine Funktion μ -rekursiv, dann ist sie auch λ -definierbar, und umgekehrt; usw. Dies bewirkt eine erhebliche Vereinfachung der einzelnen Existenzbeweise, denn ist eine Funktion nicht Turing-berechenbar, dann ist diese gleiche Funktion (wegen der besagten Äquivalenzen) auch weder μ -rekursiv, noch λ -definierbar, noch durch eine Grammatik erzeugbar.

Die *These von Church-Turing* besagt, dass der intuitive Berechenbarkeitsbegriff gerade durch eine dieser vielen äquivalenten formalen Berechenbarkeitsbegriffe erfasst wird. Diese These ist seit langem anerkannt und wird durch jeden neu hinzukommenden Äquivalenzbeweis erhärtet, kann aber (weil der intuitive Berechenbarkeitsbegriff *per se* nicht formal gefasst ist) nicht streng bewiesen werden.

Die Funktionen, die auf Grund der genannten (äquivalenten) Definitionen als „berechenbar“ bezeichnet werden, bilden nur eine kleine Klasse der Menge aller Funktionen, die aber immer noch sehr groß und unüberschaubar ist. In der Praxis ist man hauptsächlich an bestimmten Klassen berechenbarer Funktionen interessiert. Es werden in der Theoretischen Informatik – und auch in dieser Vorlesung – vor allem zwei Techniken zur Aussonderung von Teilklassen berechenbarer Funktionen betrachtet: *strukturelle Einschränkungen* und *algorithmische Einschränkungen*.

Zur Definition struktureller Einschränkungen taugen vor allem das formale Maschinenmodell (beruhend auf der Turingmaschine) und das Grammatikmodell. Beiden Modellen ist eigen, dass man an gewissen Parametern so „drehen“ (d.h., sie geeignet verändern bzw. einschränken) kann, dass jeweils höchst interessante Teilklassen entstehen. Die Analogie geht sogar so weit, dass man durch analoges „Drehen“ an einander entsprechenden Parametern bei beiden Modellen jeweils äquivalente Teilmodelle erhält (im Hinblick auf die definierte Teilklass der berechenbaren Funktionen). In dieser Vorlesung werden wir dies anhand der beiden Maschinen-Teilmodelle *Endliche Automaten* und *Kellerautomaten* und der ihnen entsprechenden Grammatik-Teilmodelle *rechts- bzw. linkslineare Grammatiken* und *kontextfreie Grammatiken* nachvollziehen.

Zur Definition algorithmischer Einschränkungen bietet sich das Maschinenmodell am unmittelbarsten an, denn hierin ist es am einfachsten möglich, Begriffe zu definieren, die dem „Zeitbedarf“ und dem „Speicherplatzbedarf“ von Computern entsprechen. Diese beiden Parameter, Zeit und Speicher, sind die wichtigsten quantitativen Leistungsmerkmale für die Berechnung durch einen Algorithmus. Leisten zwei Algorithmen das gleiche, verbraucht einer aber weniger Zeit und weniger Speicherplatz, dann ist dieser natürlich vorzuziehen. In der Praxis kommt sehr häufig ein Mischfall vor: versucht man, einen Algorithmus in Bezug auf seinen Zeitbedarf zu optimieren, muss man mehr Speicherplatz investieren, und umgekehrt. Trotzdem (oder gerade deswegen) ist es sinnvoll, wenn möglich, Schranken für den Zeit- und Speicherbedarf zur Berechnung einer gegebenen Funktion zu finden, und allgemeiner, die Klasse der berechenbaren Funktionen in „leicht berechenbare“ und „schwer berechenbare“ einzuteilen. Es stellt sich heraus, dass eine noch feinere Unterteilung möglich ist, und dies werden wir in der vorliegenden Vorlesung ebenfalls andeuten.

Das Skriptum ist folgendermaßen aufgebaut. Im Rest dieses Kapitels geben wir Literatur an. Kapitel 2 bietet dem Leser (der Leserin) eine Einführung in die mathematischen Strukturen, die im Verlauf der

Vorlesung benötigt werden: *Mengen* (darunter auch *Funktionen*), *Sprachen* und *Algorithmen*.

In Kapitel 3 werden formale Grammatiken mitsamt ihren wichtigen strukturellen Einschränkungen definiert. Wir konzentrieren diese Definitionen in einem Kapitel, weil sie sich homogen angeben lassen.

In Kapitel 4 betrachten wir die am stärksten einschränkenden strukturellen Eigenschaften: *endliche Automaten* und *links- bzw. rechtslineare Grammatiken*, sowie die dadurch beschriebenen *regulären Sprachen*. Unser Ziel ist die Charakterisierung dieser Sprachen.

Das Kapitel 5 ist einer etwas weniger eingeschränkten Sprachklasse, der Klasse der *kontextfreien Sprachen*, gewidmet. Wir zeigen, dass für die Erzeugung solcher Sprachen äquivalent entweder *kontextfreie Grammatiken* oder *Kellerautomaten* verwendet werden können.

In Kapitel 6 wenden wir uns den beiden grundlegenden Berechnungsmodellen zu: *Turingmaschinen* und *Grammatiken* zur Berechnung von Funktionen bzw. Erzeugung von Sprachen. Wir beweisen unter anderem, dass in Bezug auf die akzeptierten bzw. erzeugten Sprachen Turingmaschinen und formale Grammatiken gleich mächtig sind.

Das Kapitel 7 beantwortet die zu Beginn aufgeworfene – und nunmehr, nach den vorbereitenden Überlegungen des Kapitels 6, streng mathematisch beantwortbare – Frage: ob es nicht-berechenbare Funktionen (bzw. nicht algorithmisch erzeugbare Sprachen) gibt? Es wird eine erste Funktion dieser Art angegeben, und danach ein Prinzip, nach dem weitere gefunden werden können.

Das Kapitel 8 ist schließlich algorithmischen Untersuchungen vorbehalten. Wir analysieren den Zeitbedarf und den Speicherplatzbedarf von Turingmaschinen. Geleistet wird hier eine Klasseneinteilung der berechenbaren Funktionen von \mathbb{N} nach \mathbb{N} in „relativ leicht berechenbare“ (diese bilden die Klasse P, für *polynomiell* berechenbar), „vermutlich schwer berechenbare“ (diese bilden die Klasse NP, für *nichtdeterministisch polynomiell* berechenbar) und „den Rest“, d.h. „ziemlich sicher sehr schwer berechenbare“. Eigentlich sollte es genauer heißen, dass diese Einteilung vermutlich eine solche ist. Insbesondere ist das Problem, $P = NP$ zu beweisen oder zu widerlegen, ungelöst. Dieses Problem wird oft als das berühmteste offene Problem der Informatik bezeichnet.

1.2 Literaturverzeichnis

Die Themen dieser Vorlesung sind Standardstoff in den Informatik-Curricula auf der ganzen Welt. Es existiert eine große Menge an Literatur, von der wir nur eine kleine Auswahl nennen können. Als begleitende Literatur für diese Vorlesung empfehlen wir primär:

- U. Schöning: *Theoretische Informatik - kurz gefasst*. Verlag Spektrum 4. Auflage, 2001, 198 S., ISBN 3827410991.

Dieses Buch konzentriert sich auf das Wesentliche und vermittelt den Stoff in ansprechender und lesbarer Form ohne Verzicht auf Präzision.

- J.E. Hopcroft, J.D. Ullmann: *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading (Mass.), 1979.

Dieses Buch ist trotz seines relativen Alters ein Klassiker, unbedingt zu empfehlen.

Es gibt eine „abgespeckte“ und sehr lesbare Überarbeitung:

J.E. Hopcroft, R. Motwani, J.D. Ullmann: *Introduction to Automata Theory, Languages, and Computation, 2/E*. Addison Wesley Higher Education, 2001, 521 Seiten, ISBN: 0-201-44124-1.

sowie eine deutsche Übersetzung:

J.E. Hopcroft, R. Motwani, J.D. Ullmann: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 2. Auflage, Pearson Studium, 2002, 528 Seiten, ISBN: 3-8273-7020-5.

- A. Asteroth, Ch. Baier: *Theoretische Informatik*. Pearson Studium, 2002, ISBN 3-8273-7033-7.
- K. Erk, L. Priese: *Theoretische Informatik - eine umfassende Einführung*. Springer-Verlag, 2., erw. Aufl., 2002, 467 Seiten, ISBN 3-540-42624-8.

Als Sekundärliteratur nennen wir die folgenden Werke:

- N. Blum: *Theoretische Informatik - eine anwendungsorientierte Einführung*. Oldenbourg, 2001 (2. Auflage, 339 Seiten).
- E. Börger: *Berechenbarkeit, Komplexität, Logik*. Vieweg, Braunschweig 1986.
- W. Brauer: *Automatentheorie*. Teubner Verlag, Stuttgart 1984.
- E. Engeler, P. Läuchli: *Berechnungstheorie für Informatiker*. Teubner, Stuttgart 1988.
- G. Goos: *Vorlesungen über Informatik, Band 3: Berechenbarkeit, formale Sprachen, Spezifikationen*. Springer-Verlag, Berlin, 1997.
- P.R. Halmos: *Naive Set Theory*. Springer-Verlag, Undergraduate Texts in Mathematics (1960 und 1974). Gut lesbar.
- D. Harel: *Algorithmics - The Spirit of Computing*. Addison-Wesley, Reading (Mass.), 1987.
- H. Hermes: *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit*. Springer-Verlag, Berlin (1961, 1971, 1978).
- A.R. Lewis, C.H. Papadimitriou: *Elements of the Theory of Computation*. Prentice Hall, Englewood Cliffs 1981.
- A. Merceron: *Languages and Logic*. Pearson Education, 2001 (158 Seiten).
- K.R. Reischuk: *Einführung in die Komplexitätstheorie*. Teubner Verlag, Stuttgart 1990.
- A. Salomaa: *Computation and Automata*. Cambridge University Press, Cambridge 1985.
- A. Salomaa: *Formal Languages*. Academic Press, New York 1973.
- K. Wagner: *Einführung in die Theoretische Informatik - Grundlagen und Modelle*. Springer-Lehrbuch (1994).

Kapitel 2

Grundlagen

Die Abschnitte 2.1 (Mengen, Relationen, Funktionen) und 2.2 (Graphen, Bäume) enthalten Definitionen, die eigentlich schon in anderen Vorlesungen erschöpfend behandelt worden sein sollten (vor allem im Modul *Diskrete Strukturen*, aber auch in anderen Mathematikveranstaltungen und in den Modulen zu *Algorithmen und Datenstrukturen*).

Warum diese Wiederholungen?

Nun, einerseits kommen Begriffe wie *injektiv*, *transitiv* usw. an vielen Stellen im weiteren Verlauf unserer Untersuchungen vor, andererseits bestehen manchmal auch subtile Unterschiede in der Behandlung von Grundbegriffen. Fängt \mathbb{N} mit der Null an oder mit der Eins? Wird die relationale Komposition von links nach rechts geschrieben oder von rechts nach links? Diese beiden Fragen beantworten wir folgendermaßen:

- **Achtung:** Die Menge \mathbb{N} der natürlichen Zahlen fängt mit der 0 an: $\mathbb{N} = \{0, 1, 2, 3, \dots\}$.
- **Achtung:** Die relationale Komposition wird von links nach rechts geschrieben:

$$(x, y) \in (\rho \circ \sigma) \quad \text{bedeutet} \quad \exists z: (x, z) \in \rho \wedge (z, y) \in \sigma,$$

und demzufolge heißt bei Funktionen f und g

$$(f \circ g)(x) = g(f(x)) \quad (\text{und nicht etwa } = f(g(x))).$$

Beides wird gleich noch genauer erklärt.

Abschnitt 2.3 (Wörter und Sprachen) geht speziell auf die Grundlagen dieser Vorlesung ein. Ein Wort wird dort zunächst ganz unscheinbar als die Aneinanderreihung von endlich vielen Buchstaben eingeführt. Dahinter steckt jedoch ein sehr mächtiges Konzept, denn wir werden sehen, dass sich eigentlich alle Objekte, die hier überhaupt von Interesse sind (Zahlen, Daten, Grammatiken, Automaten, Programme, Computer) in Wörter übersetzen lassen. Von großer (und bekannter) Bedeutung ist dabei das zweistellige Alphabet, das Binäralphabet, das nur die Buchstaben 0 und 1 beinhaltet; wir lassen aber auch andere Alphabete zu.

In Abschnitt 2.4 definieren wir eine kleine Programmiersprache, die uns vor allem dazu dienen wird, Wörter und andere endliche Objekte ineinander umzurechnen. Diese Sprache ist eine Art Mini-Java bzw. Mini-C. Schließlich gehen wir in Abschnitt 2.5 auf solche Umrechnungen ein. Ein bekanntes Beispiel ist die Umrechnung der Dezimaldarstellung in die Dualdarstellung einer Zahl, aber wir werden noch andere und kompliziertere Umrechnungen benötigen.

Dieser Abschnitt enthält viele kleine Beispiele und eignet sich deshalb vor allem zum Selbststudium. In der Vorlesung gehen wir nur auf die spezifisch wichtigen Begriffe ein. Es gibt im Text eine Reihe von „Anmerkungen“, die beim ersten Lesen ohne Schaden ausgelassen werden können.

2.1 Mengentheoretische Grundbegriffe

2.1.1 Logik und Mengen

Aussagelogische Konstanten sind **false** („falsch“) und **true** („wahr“). An aussagelogischen Konnektiven benutzen wir \wedge (logisches „und“, Konjunktion), \vee (logisches „oder“, Disjunktion), \neg (logisches „nicht“, Negation), \Rightarrow (Implikation) und \Leftrightarrow (Äquivalenz). Mit \exists und \forall werden die prädikatenlogischen Konnektive „es existiert ein“ und „für alle“ bezeichnet.

Beispiel:

Der prädikatenlogische Ausdruck

$$\forall x \exists y: x < y \wedge \text{prim}(y)$$

besagt, dass es für jedes x ein y gibt, das größer als x und prim ist. Diese Aussage ist für die natürlichen Zahlen und mit den üblichen Interpretationen von „größer“ und „prim“ wahr.

Ende des Beispiels

Die *leere Menge* wird mit \emptyset bezeichnet. $x \in X$ ($x \notin X$) bedeutet, dass x ein (bzw. kein) *Element* von X ist. $X \cap Y = \{w \mid w \in X \wedge w \in Y\}$ und $X \cup Y = \{w \mid w \in X \vee w \in Y\}$ sind der *Durchschnitt* bzw. die *Vereinigung* von X und Y ; $X \setminus Y = \{x \in X \mid x \notin Y\}$ bezeichnet die *Differenzmenge* von X und Y ; $X \subseteq Y$ bedeutet, dass X *Teilmenge* von Y ist (man sagt auch: Y *umfasst* X); X heißt *echte* Teilmenge von Y (geschrieben $X \subset Y$ oder $X \subsetneq Y$), wenn zusätzlich $X \neq Y$ gilt. Ist Z eine (Grund-)Menge und gilt $X \subseteq Z$, dann ist das *Komplement von X bezüglich Z* definiert als $\overline{X} = Z \setminus X$.

Eine Menge, die ein einziges Element hat, heißt *Einermenge*. $2^X = \{A \mid A \subseteq X\}$ ist die *Potenzmenge* von X , d.h. die Menge aller Teilmengen von X (einschließlich \emptyset und X). $X_1 \times \dots \times X_n$ ist das *Kartesische Produkt* der Mengen X_1, \dots, X_n , d.h. die Menge aller n -Tupel (x_1, \dots, x_n) mit $x_i \in X_i$ für $1 \leq i \leq n$. Gilt $n = 0$, dann setzen wir $X_1 \times \dots \times X_n$ *per definitionem* gleich $\{\emptyset\}$. Gilt $X_1 = \dots = X_n = X$, dann wird $X_1 \times \dots \times X_n$ auch kurz mit X^n bezeichnet.

Um Mengen explizit anzugeben, verwenden wir die Schreibweise mit Mengenklammern: durch sogenannte *Mengenkompression* (z.B. ist $\{x \in X \mid E(x)\}$ die Menge aller Elemente x in X mit der Eigenschaft $E(x)$), oder *aufzählend* (z.B. sind $\{0, 1, 2, 3\}$ die Menge der ersten vier natürlichen Zahlen und $\{x_1, x_2, \dots\}$ die Menge aller x_j mit $j = 1, 2, \dots$).

Wichtige Mengen sind die Menge der *natürlichen Zahlen* $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ und die Menge der *ganzen Zahlen* $\mathbb{Z} = \{0, 1, -1, 2, -2, 3, -3, \dots\}$. $\{0, \dots, n-1\}$ ist die Menge der ersten n natürlichen Zahlen; gilt $n = 0$, dann setzen wir *per definitionem* $\{0, \dots, n-1\}$ gleich \emptyset .

Beispiel:

$$\begin{aligned}
& 3 \in \{0, 1, 2, 3\}, \quad 3 \notin \{0, 1, 2\}, \quad 0 \in \{0, 1, 2\}, \quad \{0\} \notin \{0, 1, 2\} (!) \\
& \{0, 1, 2\} \cap \{0, 2, 3\} = \{0, 2\} \\
& \{0, 1, 2\} \cup \{0, 2, 3\} = \{0, 1, 2, 3\} \\
& \{0, 1, 2\} \setminus \{0, 2, 3\} = \{1\} \\
& \{0, 1, 2\} \not\subseteq \{0, 2, 3\}, \quad \{0\} \subseteq \{0, 2, 3\}, \quad 0 \not\subseteq \{0, 2, 3\} (!), \quad \{0, 1\} \subsetneq \{0, 1, 2\} \\
& \text{mit Grundmenge } \mathbb{N} \text{ ist } \overline{\{0, 1, 2\}} = \{3, 4, 5, \dots\}, \quad \overline{\{0, 2, 4, \dots\}} = \{1, 3, 5, \dots\}, \quad \overline{\emptyset} = \mathbb{N} \\
& \text{mit Grundmenge } \mathbb{Z} \text{ ist } \overline{\{0, 1, 2\}} = \{\dots, -3, -2, -1, 3, 4, 5, \dots\}, \quad \overline{\emptyset} = \mathbb{Z} \\
& 2^{\{0,1,2\}} = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\} \\
& \{0, 1\} \times \{0, 2\} = \{(0, 0), (0, 2), (1, 0), (1, 2)\} \\
& \{0\}^3 = \{(0, 0, 0)\}, \quad \emptyset^3 = \emptyset, \quad \emptyset^0 = \{\emptyset\} (!).
\end{aligned}$$

Ende des Beispiels**2.1.2 Relationen**

Eine (*binäre*) *Relation* ρ ist eine Teilmenge des Produkts $X \times Y$ zweier Mengen X und Y , d.h. $\rho \subseteq X \times Y$. Wir schreiben $(x, y) \in \rho$ oder auch $x \rho y$, um auszudrücken, dass die Elemente $x \in X$ und $y \in Y$ sich in der Relation ρ befinden.

Achtung! Die Mengen X und Y gehören zur Definition einer Relation mit dazu. Wenn Sie das Wort „Relation“ hören, sollten Sie sich immer auch die Frage „weiß ich denn genau, zwischen welchen beiden Mengen?“ beantworten (lassen).

Beispiel:

Wir betrachten zwei dreielementige Mengen: $X = \{0, 1, 2\}$ und $Y = \{0, 1, 3\}$. Abbildung 2.1 zeigt drei Relationen zwischen diesen beiden Mengen.

Ende des Beispiels

Der *Vorbereich* (*engl.*: domain) bzw. der *Nachbereich* (*engl.*: codomain) von ρ sind die Mengen $dom(\rho) = \{x \in X \mid \exists y \in Y : x \rho y\}$ und $cod(\rho) = \{y \in Y \mid \exists x \in X : x \rho y\}$. Das *Komplement* von ρ ist die Relation $\bar{\rho} = (X \times Y) \setminus \rho$, die *Inverse* von ρ ist eine Relation ρ^{-1} auf $Y \times X$, die durch $(y, x) \in \rho^{-1} \Leftrightarrow (x, y) \in \rho$ definiert ist. Die Relation ρ heißt *rechtseindeutig*, wenn gilt:

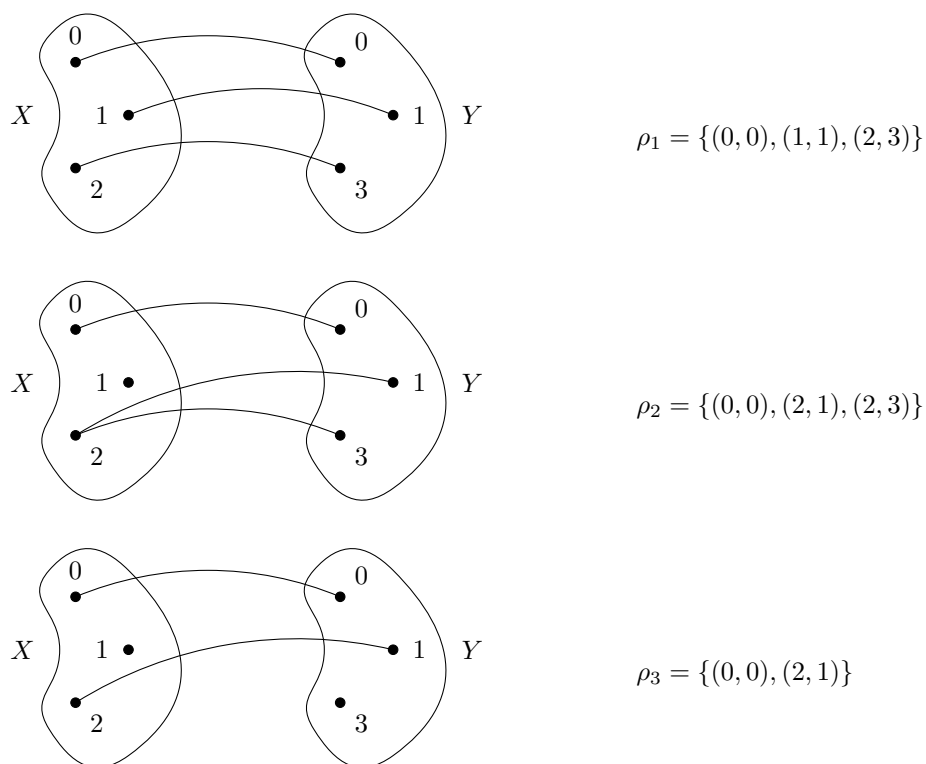
$$\forall x \in X \forall y, y' \in Y : ((x, y) \in \rho \wedge (x, y') \in \rho) \Rightarrow y = y',$$

rechtstotal, wenn gilt: $cod(\rho) = Y$, *linkseindeutig*, wenn ρ^{-1} rechtseindeutig ist, und *linkstotal*, wenn ρ^{-1} rechtstotal ist.

Es seien X, Y und Z drei Mengen und $\rho \subseteq X \times Y$ und $\tau \subseteq Y \times Z$ zwei Relationen. Die *relationale Komposition* $\rho \circ \tau \subseteq X \times Z$ von ρ und τ ist folgendermaßen definiert:

$$(x, z) \in \rho \circ \tau \text{ genau dann, wenn } \exists y \in Y : (x, y) \in \rho \wedge (y, z) \in \tau.$$

Alle vorgenannten Eigenschaften, von rechtseindeutig bis linkstotal, bleiben über die relationale Komposition erhalten. Sind zum Beispiel ρ und τ beide rechtseindeutig, dann ist auch $\rho \circ \tau$ rechtseindeutig.

Abbildung 2.1: Drei Relationen ρ_1 , ρ_2 und ρ_3 , alle $\subseteq X \times Y$

Beispiel: Für die Relationen aus Abbildung 2.1 gilt:

$$\text{cod}(\rho_1) = \{0, 1, 3\}, \text{ dom}(\rho_3) = \{0, 2\}$$

$$\overline{\rho_1} = \{(0, 1), (0, 3), (1, 0), (1, 3), (2, 0), (2, 1)\}$$

$$\rho_1^{-1} = \{(0, 0), (1, 1), (3, 2)\} \text{ Frage: Zwischen welchen Mengen?}$$

$$\rho_3^{-1} = \{(0, 0), (1, 2)\}$$

ρ_1 ist links- und rechtseindeutig und links- und rechtstotal

ρ_2 ist links-, aber nicht rechtseindeutig

ρ_3 ist weder links- noch rechtstotal

$$\rho_1 \circ (\rho_1^{-1}) = \{(0, 0), (1, 1), (2, 2)\} \text{ (eine Relation auf } X \times X)$$

$$\rho_1 \circ (\rho_3^{-1}) = \{(0, 0), (1, 2)\} \text{ (auch eine Relation auf } X \times X).$$

Ende des Beispiels

Sei $\rho \subseteq X \times Y$ eine Relation über den Mengen X und Y und seien $X' \subseteq X$ und $Y' \subseteq Y$ zwei Teilmengen von X bzw. von Y . Die *Einschränkung* $\rho|_{X' \times Y'} \subseteq X' \times Y'$ von ρ ist folgendermaßen definiert: $\rho|_{X' \times Y'} = \rho \cap (X' \times Y')$. Im Fall $Y' = Y$ schreiben wir manchmal auch kurz $\rho|_{X'}$ statt $\rho|_{X' \times Y}$.

Beispiel:

$$\rho_1|_{\{1,2\} \times \{0,1,3\}} = \{(1,1), (2,3)\}$$

$$\rho_1|_{\{1,2\} \times \{0,3\}} = \{(2,3)\}$$

$$\rho_2|_{\{1,2\} \times \{0,1,3\}} = \{(2,1), (2,3)\}$$

$$\rho_2|_{\{1,2\} \times \{0,3\}} = \{(2,3)\}$$

$$\rho_3|_{\{1,2\} \times \{0,1,3\}} = \{(2,1)\}$$

$$\rho_3|_{\{1,2\} \times \{0,3\}} = \emptyset$$

Ende des Beispiels

Nun nehmen wir zusätzlich $X = Y$ an.

Beispiel:

Wir betrachten wieder die Menge $X = \{0, 1, 2\}$. In Abbildung 2.2 ist eine Relation ρ über dieser Menge angegeben.

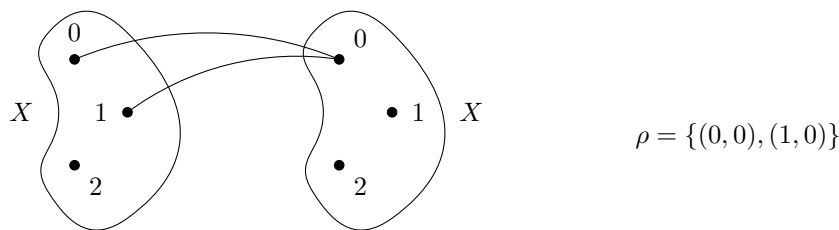


Abbildung 2.2: Eine Relation $\rho \subseteq X \times X$

Ende des Beispiels

Mit id_X bezeichnen wir die *Identitätsrelation* auf $X \times X$, die durch $(x, x') \in id_X \Leftrightarrow x = x'$ definiert ist. Eine Relation ρ auf X , d.h. $\rho \subseteq X \times X$, heißt:

<i>reflexiv</i> ,	wenn gilt: $id_X \subseteq \rho$.
<i>irreflexiv</i> ,	wenn gilt: $(\rho \cap id_X) = \emptyset$.
<i>transitiv</i> ,	wenn gilt: $(\rho \circ \rho) \subseteq \rho$
<i>symmetrisch</i> ,	wenn gilt: $\rho = \rho^{-1}$
<i>antisymmetrisch</i> ,	wenn gilt: $(\rho \cap \rho^{-1}) \subseteq id_X$.

Für $n \in \mathbb{N}$ ist die n te *Potenzierte* ρ^n von ρ induktiv folgendermaßen festgelegt:

$$\rho^0 = id_X \quad \text{und} \quad \rho^{n+1} = \rho \circ \rho^n.$$

Die vorgenannten Eigenschaften bleiben auch hier erhalten, denn id_X erfüllt alle, und ρ^n ist mit Hilfe der relationalen Komposition definiert; ist ρ z.B. rechtseindeutig und linkstotal, dann gilt das gleiche für ρ^n . Die *reflexive transitive Hülle* ρ^* und die *transitive Hülle* ρ^+ von ρ sind definiert durch

$$\rho^* = \bigcup_{n \in \mathbb{N}} \rho^n \quad \text{bzw. durch} \quad \rho^+ = \bigcup_{n \in (\mathbb{N} \setminus \{0\})} \rho^n.$$

Beispiel: Siehe Abbildung 2.2. Dort gilt:

$$\begin{aligned} \rho \circ \rho &= \{(0, 0), (1, 0)\} \\ \rho^{-1} &= \{(0, 0), (0, 1)\} \\ \rho^0 &= id_X = \{(0, 0), (1, 1), (2, 2)\} \\ \rho^1 &= \rho = \{(0, 0), (1, 0)\} \\ \rho^2 &= \rho \circ \rho = \{(0, 0), (1, 0)\} \\ \rho^3 &= \dots = \rho^n = \{(0, 0), (1, 0)\} \text{ für } n \geq 3 \\ \rho^* &= \{(0, 0), (1, 0), (1, 1), (2, 2)\} \\ \rho^+ &= \{(0, 0), (1, 0)\} \end{aligned}$$

ρ ist transitiv und antisymmetrisch, aber nicht reflexiv, nicht irreflexiv und nicht symmetrisch.

Ende des Beispiels

Eine Relation $\rho \subseteq X \times X$ heißt eine *Äquivalenzrelation*, wenn sie reflexiv, transitiv und symmetrisch ist. Die maximal großen Teilmengen $W \subseteq X$ mit $\forall x, y \in W: (x, y) \in \rho$ heißen die *Äquivalenzklassen* von ρ . Ein Element einer Äquivalenzklasse heißt ein *Repräsentant* dieser Klasse, weil aus seiner Kenntnis (und der Kenntnis der Relation) die gesamte Klasse zurückberechnet werden kann. Ist $x \in X$, dann bezeichnet man mit $[x]_\rho$ (oder einfach nur mit $[x]$, wenn ρ bekannt ist) diejenige Äquivalenzklasse von ρ , die x enthält. Es gilt $(x, y) \in \rho \Leftrightarrow [x] = [y]$.

Hat ρ endlich viele Äquivalenzklassen, heißt deren Anzahl der *Index* von ρ . Gilt $\rho_2 \subseteq \rho_1$, dann heißt ρ_2 eine *Verfeinerung* von ρ_1 . Der Name kommt daher, dass im Falle $\rho_2 \subseteq \rho_1$ alle Äquivalenzklassen von ρ_2 Teilmengen gewisser Äquivalenzklassen von ρ_1 sind.

Für beliebige Relationen $\rho \subseteq X \times X$ gilt: ρ^* ist reflexiv; ρ^* und ρ^+ sind transitiv; $(\rho \cup \rho^{-1})^*$ ist reflexiv, transitiv und symmetrisch. Die zuletzt genannte Relation heißt die von ρ *erzeugte* oder *generierte* Äquivalenzrelation.

Beispiel: Für Abbildung 2.2 gilt

$$\begin{aligned} (\rho \cup \rho^{-1})^* &= \{(0, 0), (0, 1), (1, 0), (1, 1), (2, 2)\} \\ \text{Die Äquivalenzklassen von } (\rho \cup \rho^{-1})^* &\text{ sind } \{0, 1\} \text{ und } \{2\} \\ \text{Es gilt } [0]_{(\rho \cup \rho^{-1})^*} &= \{0, 1\} = [1]_{(\rho \cup \rho^{-1})^*} \end{aligned}$$

$id_{\{0,1,2\}}$ ist ebenfalls eine Äquivalenzrelation und außerdem eine Verfeinerung von $(\rho \cup \rho^{-1})^*$, denn es gilt

$$\underbrace{id_{\{0,1,2\}}}_{\text{mit Äquivalenzklassen } \{0\}, \{1\} \text{ und } \{2\}} \subseteq \underbrace{(\rho \cup \rho^{-1})^*}_{\text{mit Äquivalenzklassen } \{0, 1\} \text{ und } \{2\}}.$$

Ende des Beispiels

Eine Relation $\rho \subseteq X \times X$ heißt eine *Halb Ordnungsrelation* oder *partielle Ordnung*, wenn sie reflexiv, transitiv und antisymmetrisch ist.

Beispiel:

Die Relation \subseteq ist eine Halbordnung auf der Potenzmenge einer gegebenen Menge.

Ende des Beispiels

Die Bezeichnung „Hülle“ für ρ^* ist durch die folgende Aussage gerechtfertigt:

Lemma 2.1.1 ρ^* ist die kleinste (in Bezug auf \subseteq) reflexive und transitive Relation, die ρ umfasst.

Beweis: Es gibt reflexive und transitive Relationen, die ρ umfassen, nämlich (trivialerweise) $\rho_{\text{voll}} = X \times X$ und ρ^* ; denn es gilt $\rho \subseteq \rho^*$ wegen $\rho = \rho^1$. Sei nun σ eine beliebige reflexive und transitive Relation, die ρ umfasst: $\rho \subseteq \sigma$. Wir zeigen $\rho^* \subseteq \sigma$. Sei (x, y) ein beliebiges Element von ρ^* . Dann gibt es ein $n \in \mathbb{N}$ mit $(x, y) \in \rho^n$, und daher auch Elemente x_0, \dots, x_n in X mit

$$x=x_0, x_n=y \quad \text{und} \quad (x_0, x_1) \in \rho, (x_1, x_2) \in \rho, \dots, (x_{n-1}, x_n) \in \rho.$$

Wegen $\rho \subseteq \sigma$ gilt Letzteres auch, wenn man ρ durch σ ersetzt. Da σ reflexiv und transitiv ist, folgt daraus durch einen Induktionsschluss, dass $(x_0, x_n) \in \sigma$, da aber $x=x_0$ und $x_n=y$, folgt daraus $(x, y) \in \sigma$, und damit ist der Beweis von $\rho^* \subseteq \sigma$ beendet. □ 2.1.1

Man beachte, dass wir hier implizit auch gezeigt haben, dass es eine eindeutige kleinste reflexive und transitive Relation gibt, die ρ umfasst. Das muss durchaus nicht immer gelten, wenn man die Eigenschaften „reflexiv“ und „transitiv“ durch andere ersetzt. Genauso zeigt man allerdings Folgendes: ρ^+ ist die kleinste transitive Relation, die ρ umfasst, und die von ρ erzeugte Äquivalenzrelation ist die kleinste Äquivalenzrelation, die ρ umfasst.

2.1.3 Funktionen und Operationen

Wir fassen Funktionen als spezielle Relationen auf.

Seien X, Y Mengen. Eine Relation $f \subseteq X \times Y$ heißt *partielle Funktion* von X nach Y , wenn f rechtseindeutig ist; f heißt *Funktion* (oder zur genaueren Unterscheidung auch *totale Funktion*) von X nach Y , wenn f zusätzlich linkstotal ist. Die Tatsache, dass f partielle Funktion (bzw. Funktion) von X nach Y ist, wird kurz auch durch $f: X \xrightarrow{p} Y$ (bzw. $f: X \rightarrow Y$) ausgedrückt. Die Menge aller Funktionen von X nach Y wird auch mit $X \rightarrow Y$ bezeichnet, so dass $f: X \rightarrow Y$ gleichbedeutend mit $f \in (X \rightarrow Y)$ ist.

Achtung! Die Mengen X und Y gehören zur Definition einer Funktion mit dazu. Wenn Sie das Wort „Funktion“ hören, sollten Sie sich immer auch die Frage „weiß ich denn genau, von wo nach wo die Funktion führt?“ beantworten.

Um explizit anzugeben, dass f eine (partielle) Funktion von X nach Y ist, die das Element $x \in X$ in das Element $y \in Y$ abbildet, verwenden wir die Schreibweise:

$$f: \left\{ \begin{array}{l} X \xrightarrow{p} Y \\ x \mapsto y \end{array} \right. \quad \text{bzw.} \quad f: \left\{ \begin{array}{l} X \rightarrow Y \\ x \mapsto y \end{array} \right. \quad \begin{array}{l} \text{Grundbereiche} \\ \text{Wirkung auf Elemente.} \end{array}$$

Beispiel:

Wir betrachten wieder die beiden dreielementigen Mengen $X = \{0, 1, 2\}$ und $Y = \{0, 1, 3\}$. Abbildung

2.3 zeigt zwei Funktionen zwischen diesen beiden Mengen. Traditionell wird die Zuordnung von Elementen hier durch einen Pfeil (statt einer ungerichteten Kante) ausgedrückt; das hat aber keine weitere Bedeutung, als zu suggerieren, dass „ein Element aus X auf ein Element aus Y abgebildet“ wird.

Die Relation ρ_2 aus Abbildung 2.1 und die Relation ρ^{-1} aus Abbildung 2.2 sind Beispiele für Relationen, die keine partiellen Funktionen sind.

Ende des Beispiels

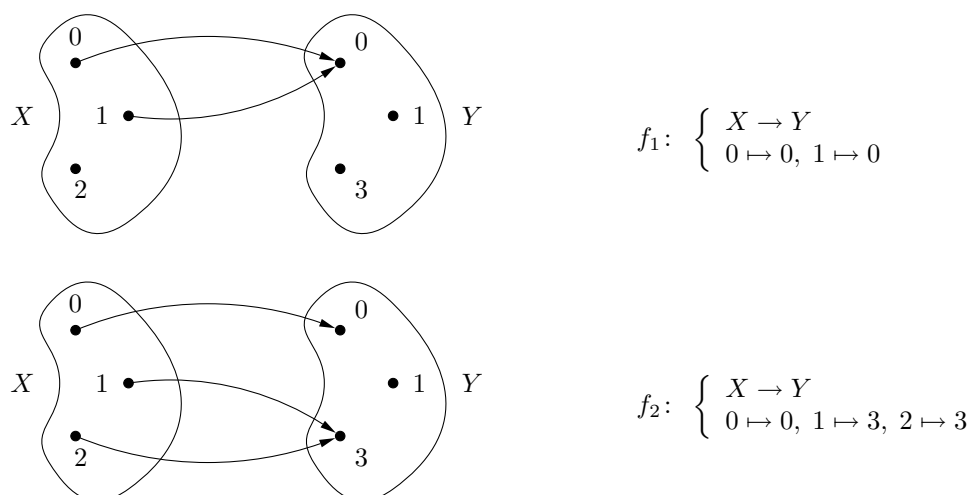


Abbildung 2.3: Zwei Funktionen f_1 und f_2 , beide $\in X \rightarrow Y$

Eine partielle Funktion f ordnet jedem $x \in X$ höchstens ein $y \in Y$ zu. Im Falle, dass x kein zugeordnetes y hat, heißt f auf x undefiniert. Im anderen Falle ist die Schreibweise $f(x)=y$ statt $(x,y) \in f$ üblich; y heißt dann der Funktionswert von f an der Stelle (oder dem Argument) x . Diese Schreibweise wird auch auf die Argumente x , an denen f undefiniert ist, ausgedehnt, und man schreibt $f(x)=undef$ für solche x (natürlich wird $undef \notin Y$ angenommen).

Beispiel: In Abbildung 2.3 ist f_1 eine partielle Funktion, aber keine (totale) Funktion. Es gilt

$$\begin{aligned} f_1(0) &= 0 \\ f_1(1) &= 0 \\ f_1(2) &= undef \end{aligned} \tag{2.1}$$

Hingegen ist f_2 sowohl partielle als auch totale Funktion, also eine Funktion. Es gilt

$$\begin{aligned} f_2(0) &= 0 \\ f_2(1) &= 3 \\ f_2(2) &= 3 \end{aligned}$$

Ende des Beispiels

Anmerkung:

Die letzte Überlegung zeigt, dass die partiellen Funktionen von X nach Y genau gleichwertig mit den Funktionen von X nach $Y \cup \{\text{undef}\}$ sind. Beispielsweise ist f_1 gleichwertig mit der Funktion von X nach $Y \cup \{\text{undef}\}$, die durch (2.1) definiert ist. Eine ähnliche Bemerkung kann über den Zusammenhang zwischen Relationen und Funktionen allgemein gemacht werden. Die Relationen zwischen X und Y sind genau gleichwertig mit den Funktionen von X nach 2^Y . Denn hat man eine Teilmenge ρ von $X \times Y$, gewinnt man eine Funktion, indem man jedem $x \in X$ die Menge seiner durch ρ zugeordneten Elemente von Y zuordnet. Hat man umgekehrt eine Funktion f von X nach 2^Y , gewinnt man eine Relation, indem man $x \in X$ genau mit den Elementen in $f(x)$ in Beziehung setzt.

Beispiel: Die Relation $\rho_2 = \{(0, 0), (2, 1), (2, 3)\}$ in Abbildung 2.1 ist gleichwertig mit der Funktion

$$f_{\rho_2}: \begin{cases} X \rightarrow 2^Y \\ 0 \mapsto \{0\}, 1 \mapsto \emptyset, 2 \mapsto \{1, 3\} \end{cases}$$

Ende des Beispiels**Ende der Anmerkung**

Davon zu unterscheiden ist die *Erweiterung* \bar{f} einer beliebigen (aber nicht partiellen) Funktion $f: X \rightarrow Y$, die definiert ist als eine Funktion von 2^X nach 2^Y :

$$\bar{f}(A) = \bigcup_{a \in A} f(a), \quad \text{für } A \subseteq X.$$

Dann gilt auf Einermengen $\{a\} \subseteq X$: $\bar{f}(\{a\}) = \{f(a)\}$. Es handelt sich deswegen um eine sogenannte *konservative* (d.h.: „das Alte respektierende“) Erweiterung. Deswegen wird auch häufig der Querstrich weggelassen, d.h., \bar{f} wird einfach wieder mit f bezeichnet.

Beispiel: Die Erweiterung \bar{f}_2 der Funktion f_2 in Abbildung 2.3 ist die folgende Funktion:

$$\bar{f}_2: \begin{cases} 2^X \rightarrow 2^Y \\ \emptyset \mapsto \emptyset, \{0\} \mapsto \{0\}, \{3\} \mapsto \{3\}, \{2\} \mapsto \{2\} \\ \{0, 1\} \mapsto \{0, 3\}, \{0, 2\} \mapsto \{0, 3\}, \{1, 2\} \mapsto \{3\}, \{0, 1, 2\} \mapsto \{0, 3\} \end{cases}$$

Ende des Beispiels

Eine Funktion f von X nach Y heißt *injektiv*, wenn sie zusätzlich linkseindeutig, *surjektiv*, wenn sie rechtstotal, und *bijektiv*, wenn sie sowohl injektiv als auch surjektiv ist.

Beispiel:

Die Funktion f_2 in Abbildung 2.3 ist weder injektiv noch surjektiv, also auch nicht bijektiv. Zwischen den beiden dreielementigen Mengen X und Y , die wir als Beispiele gewählt haben, gibt es überhaupt keine injektive Funktion, die nicht auch gleichzeitig surjektiv (und damit bijektiv) wäre.

Bei endlichen Mengen kann man eine surjektive, nicht injektive Funktion von X nach Y nur dann konstruieren, wenn X echt mehr Elemente als Y hat, und eine injektive, nicht surjektive Funktion nur dann, wenn Y echt mehr Elemente als X hat.

Bei unendlichen Mengen hat man zunächst keinen Begriff von „echt mehr“. Man kann aber obige Idee verwenden, um gerade einen solchen Begriff zu definieren. Dazu kommen wir noch im Detail.

Ende des Beispiels

Oft kommt der Fall $X = X_1 \times \dots \times X_n$ vor, d.h., die Elemente von X sind n -Tupel von Elementen (x_1, \dots, x_n) mit $x_i \in X_i$. Wenn $f: X_1 \times \dots \times X_n \xrightarrow{p} Y$, heißt f n -stellig. Um die Stelligkeit zu betonen, verwendet man oft einen eingeklammerten Hochindex: $f^{(n)}$ bedeutet, dass f eine n -stellige (partielle) Funktion ist.

Eine (partielle) Funktion $\otimes: X \times X \xrightarrow{p} X$ heißt eine (partielle) binäre *Operation* oder eine *Verknüpfung*. Die relationale Komposition \circ ist zum Beispiel eine partielle Operation auf der Menge der Relationen. Generell schreibt man statt $\otimes(x, y) = w$ auch $x \otimes y = w$. Eine Operation \otimes heißt *kommutativ*, wenn $\forall x, y \in X: x \otimes y = y \otimes x$ gilt, und *assoziativ*, wenn $\forall x, y, z \in X: (x \otimes y) \otimes z = x \otimes (y \otimes z)$ gilt. Analog definiert man *unäre* oder dreistellige Operationen usw.

2.1.4 Zahlentheoretische Grundfunktionen

Eine (partielle) Funktion f heißt *zahlentheoretisch*, wenn sie ein n -Tupel natürlicher Zahlen auf ein k -Tupel natürlicher Zahlen abbildet. Meist gilt $k = 1$ und f ist dann eine Funktion $f^{(n)}: \mathbb{N}^n \xrightarrow{p} \mathbb{N}$. Wir definieren einige einfache zahlentheoretische Funktionen.

Die Nachfolgefunktion.

Die Nachfolgefunktion ist einstellig und total, $S: \mathbb{N} \rightarrow \mathbb{N}$, und ist definiert durch $S(x) = x+1$ für alle $x \in \mathbb{N}$. Eine andere Schreibweise dieser Definition ist:

$$S: \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ x & \mapsto & x+1. \end{cases}$$

Die konstante Funktion.

Sei $m \in \mathbb{N}$. Die konstante Funktion $C_m^{(n)}$ ist n -stellig und total. Sie ordnet jedem Tupel die feste Zahl m zu:

$$C_m^{(n)}: \begin{cases} \mathbb{N}^n & \rightarrow & \mathbb{N} \\ (x_1, \dots, x_n) & \mapsto & m. \end{cases}$$

Die Projektionsfunktion.

Die Projektionsfunktion $P_i^{(n)}$ (für ein i mit $1 \leq i \leq n$) ist ebenfalls n -stellig und total und wählt das i te Argument aus:

$$P_i^{(n)}: \begin{cases} \mathbb{N}^n & \rightarrow & \mathbb{N} \\ (x_1, \dots, x_n) & \mapsto & x_i. \end{cases}$$

Die überall undefinierte Funktion.

Die Funktion $O^{(n)}$ (manchmal auch „Nullfunktion“ genannt, nicht jedoch mit der Funktion $C_0^{(n)}$, die jedem Argument die Zahl 0 zuordnet, zu verwechseln) ist eine n -stellige, nicht-totale Grundfunktion. Sie ist auf keinem Argument definiert:

$$O^{(n)}: \begin{cases} \mathbb{N}^n & \xrightarrow{p} & \mathbb{N} \\ (x_1, \dots, x_n) & \mapsto & \text{undef.} \end{cases}$$

Die Auswahl- oder Selektionsfunktion.

Die vierstellige totale Funktion $A: \mathbb{N}^4 \rightarrow \mathbb{N}$ wählt ihr drittes Argument aus, wenn die beiden ersten Argumente gleich sind, andernfalls das vierte:

$$A: \begin{cases} \mathbb{N}^4 & \rightarrow \mathbb{N} \\ (x, y, u, v) & \mapsto \begin{cases} u & \text{falls } x=y \\ v & \text{falls } x \neq y. \end{cases} \end{cases}$$

Benutzung der Grundfunktionen zur Definition weiterer zahlentheoretischer Funktionen.

Eine nahe liegende Schreibweise ist es, k n -stellige Funktionen $f_1^{(n)}, \dots, f_k^{(n)}: \mathbb{N}^n \rightarrow \mathbb{N}$ zu Vektoren zusammenzufassen: $(f_1^{(n)}, \dots, f_k^{(n)})$ ist, *per definitionem*, eine Funktion von \mathbb{N}^n nach \mathbb{N}^k mit

$$(x_1, \dots, x_n) \mapsto (f_1^{(n)}(x_1, \dots, x_n), \dots, f_k^{(n)}(x_1, \dots, x_n)).$$

Wir betrachten als Beispiel:

$$sg = (P_1^{(1)}, C_0^{(1)}, C_0^{(1)}, C_1^{(1)}) \circ A.$$

Das ist eine Funktion, die ein 1-Tupel (d.h., eine Zahl x) zuerst durch $(P_1^{(1)}, C_0^{(1)}, C_0^{(1)}, C_1^{(1)})$ in ein Viertupel überführt, das als Argument der Funktion A dient und letztlich eine Zahl liefert, insgesamt also eine Funktion von \mathbb{N} nach \mathbb{N} . Inhaltlich wird 0 ausgewählt, wenn x gleich 0 ist, sonst 1. Es handelt sich also um eine etwas ungewöhnliche Schreibweise für die nichtnegative Signum- (d.h. Vorzeichen-)funktion

$$sg: \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ x & \mapsto \begin{cases} 0 & \text{falls } x=0 \\ 1 & \text{falls } x \neq 0. \end{cases} \end{cases}$$

2.1.5 Mächtigkeit und Abzählbarkeit

Wir verwenden den Funktionsbegriff, um Größenvergleiche zwischen Mengen anzustellen. Seien X und Y zwei Mengen. X und Y heißen *gleichmächtig* (in Zeichen $X \sim Y$) wenn es eine Bijektion $\beta: X \rightarrow Y$ gibt. X heißt *höchstens so mächtig wie* Y , oder Y *mindestens so mächtig wie* X ($X \preceq Y$), wenn $\exists Y' \subseteq Y: X \sim Y'$, d.h., X ist gleichmächtig mit einer Teilmenge von Y . Y ist *mächtiger* als X ($X \prec Y$), falls $X \preceq Y$ und $\neg(X \sim Y)$. X ist (höchstens) *abzählbar*, falls $X \preceq \mathbb{N}$. X ist *überabzählbar* oder *nicht abzählbar*, falls $\mathbb{N} \prec X$. X ist *endlich*, falls $X \prec \mathbb{N}$. X ist *unendlich*, falls $\mathbb{N} \preceq X$.

Beispiel:

Die Mengen $\{0, 1, 2\}$ und $\{0, 1, 3\}$ sind gleichmächtig.

Die Menge $\{0, 1, 2\}$ ist mächtiger als die Menge $\{0, 3\}$.

Die Mengen \mathbb{N} und \mathbb{Z} sind gleichmächtig (s.u.).

Die Mengen $\{0, 1, 2\}$ und $\{0, 1, 3\}$ sind abzählbar.

Die Menge \mathbb{N} ist abzählbar unendlich (s.u.).

Die Mengen $2^{\mathbb{N}}$ und $\mathbb{N} \rightarrow \mathbb{N}$ sind überabzählbar (s.u.).

Ende des Beispiels

Es gibt einige Charakterisierungen des Begriffs der Abzählbarkeit einer Menge X :

Lemma 2.1.2 ÄQUIVALENTE DEFINITIONEN VON ABZÄHLBARKEIT

Sei X eine Menge. Folgende Eigenschaften sind äquivalent:

- (a) $X \preceq \mathbb{N}$, d.h., X ist abzählbar;
- (b) $X = \emptyset$ oder es gibt eine surjektive Funktion von \mathbb{N} nach X ;
- (c) es gibt eine injektive Funktion von X nach \mathbb{N} .

Beweis: (a) \Rightarrow (b): Falls $X = \emptyset$, ist (b) direkt erfüllt. Sei $X \neq \emptyset$ und sei $z \in X$. Laut (a) gibt es eine Bijektion $\beta: X \rightarrow A$ mit $A \subseteq \mathbb{N}$. Wir definieren eine Funktion $f: \mathbb{N} \rightarrow X$ mit:

$$f(n) = \begin{cases} \beta^{-1}(n) & \text{falls } n \in A \\ z & \text{falls } n \notin A. \end{cases}$$

Dann ist f eine Surjektion von \mathbb{N} nach X .

(b) \Rightarrow (c): Falls $X = \emptyset$, ist $g = \emptyset$ eine Injektion von X nach \mathbb{N} . Sei $X \neq \emptyset$ und sei $f: \mathbb{N} \rightarrow X$ eine laut (b) existierende Surjektion. Für jedes $x \in X$ definiere $A_x = \{n \in \mathbb{N} \mid f(n) = x\}$. Dann gilt $A_x \neq \emptyset$ für alle $x \in X$ und $A_x \cap A_y = \emptyset$ für $x \neq y$. Wähle $n_x \in A_x$ beliebig¹. Definiere $g: X \rightarrow \mathbb{N}$ mit $g(x) = n_x$. Offensichtlich ist g eine Injektion.

(c) \Rightarrow (a): Sei $g: X \rightarrow \mathbb{N}$ eine Injektion. Wähle $A = \text{cod}(g)$. Dann gilt $A \subseteq \mathbb{N}$ und $\beta = \{(x, g(x)) \mid x \in X\}$ ist eine Bijektion von X nach A . □ 2.1.2

Teil (b) dieses Lemmas charakterisiert die Abzählbarkeit einer nicht leeren Menge X durch tatsächliches „Abzählen“, d.h., durch die Angabe einer Funktion $\beta: \mathbb{N} \rightarrow X$ mit $X = \{\beta(0), \beta(1), \beta(2), \dots\}$, d.h., der Surjektivität von β .

Auch die Unendlichkeit von X kann (wieder mit Hilfe des Auswahlaxioms) anders charakterisiert werden: durch die Existenz einer Bijektion von X auf eine echte Teilmenge von X . (Ohne Beweis.)

Speziell ist \emptyset eine endliche Menge. Ist X eine endliche Menge, bezeichnen wir mit $|X|$ die Anzahl der Elemente in X . Speziell gilt $|\emptyset| = 0$, $|\{0, \dots, n-1\}| = n$, $|X^n| = |X|^n$ und $|X_1 \times \dots \times X_n| = |X_1| \cdot \dots \cdot |X_n|$, vorausgesetzt, dass alle beteiligten Mengen endlich sind.

Die Relation \sim ist reflexiv, transitiv und symmetrisch, d.h. eine Äquivalenzrelation. Auch ist klar, dass die Relation \preceq reflexiv und transitiv ist; letzteres führt man leicht auf den Erhalt der Linkseindeutigkeit über die relationale Komposition zurück. Eine Art Antisymmetrieeigenschaft der Relation \preceq wird durch den folgenden Satz geregelt.

Satz 2.1.3 Satz von Bernstein-Cantor-Dedekind-Schröder

Aus $X \preceq Y$ und $Y \preceq X$ folgt $X \sim Y$.

¹Hierzu benötigen wir das Auswahlaxiom der Mengenlehre, das zwar unabhängig von den anderen Axiomen ist, üblicherweise (und auch hier) aber als gegeben angenommen wird.

Beweis: Für $A, B \subseteq X$ gilt $A \subseteq B \Leftrightarrow (X \setminus B) \subseteq (X \setminus A)$, ein rein mengentheoretisch zu beweisendes Faktum. Sei $f: X \rightarrow Y$ eine beliebige Funktion. Wir betrachten die Erweiterung von f auf Teilmengen $A \in 2^X$ und verwenden den Buchstaben f (statt \bar{f}) weiter. Aus $A \subseteq B$ folgt $f(A) \subseteq f(B)$, denn seien $A \subseteq B$ und $y \in f(A)$, dann gibt es ein $x \in A$, also auch ein $x \in B$, mit $y = f(x)$, und daher auch $y \in f(B)$. Man sagt, dass eine Funktion, die diese Eigenschaft hat, *monoton* bezüglich \subseteq ist².

Es gelte nun sowohl $X \preceq Y$ als auch $Y \preceq X$. Dann gibt es eine Injektion $f: X \rightarrow Y$ und eine Injektion $g: Y \rightarrow X$. Wir definieren eine Funktion $h: 2^X \rightarrow 2^X$ durch $h(A) = X \setminus g(Y \setminus f(A))$ für alle $A \in 2^X$. Auch h ist monoton, denn

$$\begin{aligned} A \subseteq B &\Rightarrow (\text{Monotonie (der Erweiterung) von } f) \\ &\quad f(A) \subseteq f(B) \\ &\Rightarrow (\text{Mengenlehre}) \\ &\quad (Y \setminus f(A)) \supseteq (Y \setminus f(B)) \\ &\Rightarrow (\text{Monotonie (der Erweiterung) von } g) \\ &\quad g(Y \setminus f(A)) \supseteq g(Y \setminus f(B)) \\ &\Rightarrow (\text{Mengenlehre}) \\ &\quad (X \setminus g(Y \setminus f(A))) \subseteq (X \setminus g(Y \setminus f(B))). \end{aligned}$$

Wir definieren nun eine spezielle Teilmenge $H \subseteq X$ durch die Formel $H = \bigcap \{A \in 2^X \mid h(A) \subseteq A\}$, d.h., H ist der Durchschnitt aller Teilmengen von X , auf die angewendet h „kontrahierend“ (im Sinne von \subseteq) wirkt. Man kann leicht nachweisen, dass H ein Fixpunkt von h ist, d.h. $h(H) = H$ (siehe unten). Jetzt definieren wir unter Zuhilfenahme von H eine Funktion $\beta: X \rightarrow Y$, die zusammengesetzt ist aus f (auf H) beziehungsweise aus g^{-1} (auf $X \setminus H$):

$$\beta(x) = \begin{cases} f(x) & \text{falls } x \in H \\ g^{-1}(x) & \text{falls } x \notin H. \end{cases}$$

Falls x nicht in H enthalten ist, folgt aus der Definition von H , dass x in $g(Y \setminus f(H))$ liegt; deswegen und wegen der Injektivität von g ist in der zweiten Zeile der vorangegangenen Definition $g^{-1}(x)$ in der Tat ein wohldefiniertes Element von Y . Es ist nicht schwer, zu zeigen, dass β eine Bijektion von X nach Y ist (siehe unten). Also gilt $X \sim Y$, was zu zeigen war.

Ad „ $H = h(H)$ “: wir zeigen zuerst $h(H) \subseteq H$ und dann $H \subseteq h(H)$. Setze $\mathcal{A} = \{A \in 2^X \mid h(A) \subseteq A\}$. Sei $B \in \mathcal{A}$ beliebig. Dann gilt $h(B) \subseteq B$, und weiter, da H als Durchschnitt aller $A \in \mathcal{A}$ definiert ist:

$$\begin{aligned} H \subseteq B &\Rightarrow (h \text{ ist monoton}) \\ &\quad h(H) \subseteq h(B) \\ &\Rightarrow (h(B) \subseteq B \text{ und } \subseteq \text{ transitiv}) \\ &\quad h(H) \subseteq B. \end{aligned}$$

Da $B \in \mathcal{A}$ beliebig war, gilt also $h(H) \subseteq B$ für alle $B \in \mathcal{A}$, und damit auch $h(H) \subseteq H$, denn H ist die größte Menge, die Teilmenge jeder Menge aus \mathcal{A} ist.

Aus $h(H) \subseteq H$ und der Monotonie von h folgt $h(h(H)) \subseteq h(H)$, und damit auch $h(H) \in \mathcal{A}$ (nach Definition von \mathcal{A}) sowie $H \subseteq h(H)$ (da H Teilmenge jeder Menge aus \mathcal{A} ist).

²Monotonie ist nicht eine Eigenschaft nur einer Funktion, sondern bezieht sich immer auch auf eine Ordnungsrelation wie \subseteq in unserem Fall. Die gleiche Funktion kann monoton bezüglich einer Ordnungsrelation und gleichzeitig nicht-monoton bezüglich einer anderen Ordnungsrelation sein.

Ad „ β ist bijektiv“: wir zeigen zuerst, dass β surjektiv ist, und dann, dass β injektiv ist.

β ist surjektiv: Sei $y \in Y$. Falls y in $f(H)$ liegt, gibt es ein $x \in H$ mit $y = f(x)$ und damit auch $y = \beta(x)$, da die erste Zeile der Definition von β anwendbar ist. Es gelte $y \in Y \setminus f(H)$. Definiere $x = g(y)$ (dann ist x wohldefiniert, weil g eine Funktion ist). Es gilt $x \in g(Y \setminus f(H))$, also $x \notin X \setminus g(Y \setminus f(H))$ und somit $x \notin H$ wegen $H \subseteq X \setminus g(Y \setminus f(H))$. Also $y = \beta(x)$, da die zweite Zeile der Definition von β anwendbar ist.

β ist injektiv: Es gelte $y = \beta(x)$ und $y = \beta(x')$; zu zeigen ist $x = x'$. Falls $x \in H$ und $x' \in H$, gilt $f(x) = y = f(x')$, und $x = x'$ folgt aus der Injektivität von f . Falls $x \notin H$ und $x' \notin H$, gilt $g^{-1}(x) = y = g^{-1}(x')$, und $x = x'$ folgt daraus, dass g eine Funktion ist. Es gelte $x \in H$ und $x' \notin H$. Dann $f(x) = y = g^{-1}(x')$, also $x' = g(f(x))$ und damit $x' \in g(f(H))$. Andererseits gilt auch $x' \in g(Y \setminus f(H))$ wegen $x' \notin H$ und $H \supseteq X \setminus g(Y \setminus f(H))$. Das widerspricht der Injektivität von g . Genauso zeigt man, dass $x \notin H$ und $x' \in H$ auf einen Widerspruch führt.

Für den Nachweis der Surjektivität von β wurde der Teil $H \subseteq X \setminus g(Y \setminus f(H))$, für den Beweis der Injektivität jedoch der Teil $H \supseteq X \setminus g(Y \setminus f(H))$ der Fixpunktgleichung $H = h(H)$ benötigt. \square 2.1.3

Korollar 2.1.4 CHARAKTERISIERUNG VON ABZÄHLBAR UNENDLICHEN MENSCHEN

X ist abzählbar unendlich genau dann, wenn $X \sim \mathbb{N}$. \square 2.1.4

Die abzählbaren Mengen sind also entweder endlich oder *abzählbar unendlich*, d.h. gleichmächtig mit \mathbb{N} , und jede überabzählbare Menge ist unendlich.

Beispiel:

Die folgenden Mengen sind abzählbar unendlich: \mathbb{N} ; \mathbb{Z} (eine scheinbar „größere“ Menge als \mathbb{N}); $\{n \in \mathbb{N} \mid n \text{ ist gerade}\}$ (eine scheinbar „kleinere“ Menge als \mathbb{N}); und $\mathbb{N} \times \mathbb{N}$ (eine scheinbar „viel größere“ Menge als \mathbb{N}). Dagegen sind die Mengen $2^{\mathbb{N}}$ und $\mathbb{N} \rightarrow \mathbb{N}$ überabzählbar. Die drei letzten Behauptungen beweisen wir in den folgenden Lemmata.

Ende des Beispiels

Lemma 2.1.5 ABZÄHLBARKEIT VON $\mathbb{N} \times \mathbb{N}$

Die Menge $\mathbb{N} \times \mathbb{N}$ ist abzählbar unendlich.

Beweis: In diesem Beweis erläutern wir ein bekanntes *Diagonalschema*. Man stellt sich die Paare in $\mathbb{N} \times \mathbb{N}$ in Matrixform aufgeschrieben vor:

$\mathbb{N} \times \mathbb{N}$	0	1	2	...
0	(0, 0)	(0, 1)	(0, 2)	...
1	(1, 0)	(1, 1)	(1, 2)	...
2	(2, 0)	(2, 1)	(2, 2)	...
⋮	⋮	⋮	⋮	⋮

Die Abzählbarkeit zu zeigen, bedeutet nichts anderes, als die Elemente der unteren rechten Vierelebene dieses Schemas in eine Reihenfolge zu bringen. Der Versuch (0, 0), (0, 1), (0, 2), ..., (1, 0), (1, 1), (1, 2), ..., etc., schlägt fehl, weil nicht „zwischendurch“ Unendlichkeiten auftauchen dürfen. Deshalb zählt man z.B. folgendermaßen ab: (0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), (0, 3), (1, 2), (2, 1), ..., d.h., bildlich gesprochen, in Diagonalform „von rechts oben nach links unten“ durch die Matrix, oder, arithmetisch gesprochen, erst alle Paare mit der Summe (der beiden Einzelzahlen) 0 (davon gibt es nur eins), dann alle Paare mit Summe 1 (davon gibt es zwei), dann alle Paare mit Summe 2 (davon gibt es drei), usw., in

lexikografischer Reihenfolge bei gleichen Summen. Es ist klar, dass in dieser Abzählung jedes Paar genau einmal vorkommt, und deswegen gibt es eine Surjektion von \mathbb{N} nach $\mathbb{N} \times \mathbb{N}$, die in diesem Fall sogar eine Bijektion ist. ☒ 2.1.5

Wie könnte diese Abzählung algorithmisch „realisiert“ werden?

Auflösung: siehe Abschnitt 2.5.

Eine ähnliche Idee liegt dem Beweis des folgenden Satzes zu Grunde, der zeigt, dass abzählbare Vereinigungen abzählbarer Mengen nicht aus der Klasse der abzählbaren Mengen hinaus führen.

Satz 2.1.6 DIE ABZÄHLBARE VEREINIGUNG ABZÄHLBARER MENGEN IST ABZÄHLBAR

Seien X_0, X_1, \dots abzählbare Mengen. Dann ist $\bigcup_{i \in \mathbb{N}} X_i$ abzählbar.

Beweis: Wenn einige Mengen X_i leer sind, streichen wir diese, da sie zur Vereinigung nichts beitragen. Falls unendlich viele X_i übrig bleiben, benutzen wir das Diagonalschema

$$\begin{array}{l|l} X_0 & x_{00}, x_{01}, x_{02}, \dots \\ X_1 & x_{10}, x_{11}, x_{12}, \dots \\ X_2 & x_{20}, x_{21}, x_{22}, \dots \\ \vdots & \vdots \end{array}$$

wobei, wenn X_i nicht leer und endlich ist, eines der Elemente von X_i in der entsprechenden Zeile unendlich oft wiederholt wird. Wir konstruieren eine Surjektion f von \mathbb{N} nach $\bigcup_{i \in \mathbb{N}} X_i$ nach dem Muster $f(0) = x_{00}$, $f(1) = x_{01}$, $f(2) = x_{10}$, $f(3) = x_{02}$, $f(4) = x_{11}$ usw. und wenden Lemma 2.1.2(b) an.

Falls nach Streichen leerer X_i nur endlich viele übrig bleiben, kann ein ähnliches Schema verwendet werden, wobei der Unterschied nur darin besteht, dass es endlich viele Zeilen gibt. ☒ 2.1.6

Lemma 2.1.7 ÜBERABZÄHLBARKEIT VON $2^{\mathbb{N}}$

Die Menge $2^{\mathbb{N}}$ ist überabzählbar.

Beweis: Wir nehmen an, dass $2^{\mathbb{N}}$ doch abzählbar ist. Dann gibt es, da $2^{\mathbb{N}}$ nicht leer ist, nach Lemma 2.1.2 eine Surjektion $f: \mathbb{N} \rightarrow 2^{\mathbb{N}}$. Wir betrachten die Menge $A = \{x \mid x \notin f(x)\}$. Man beachte, dass dies eine wohldefinierte Menge ist: $f(x)$ ist eine Menge von natürlichen Zahlen, so dass $x \notin f(x)$ eine wohldefinierte Eigenschaft ist. Nun benutzen wir die Surjektivität von f ; daraus folgt, dass es eine Zahl n mit $A = f(n)$ gibt. Jetzt gilt aber:

$$\begin{aligned} n \in A &\Leftrightarrow (\text{Definition von } A) \\ &\quad n \notin f(n) \\ &\Leftrightarrow (A = f(n)) \\ &\quad n \notin A. \end{aligned}$$

D.h., n ist ein Element von A genau dann, wenn n kein Element von A ist. Dies ist ein Widerspruch. Unsere Annahme muss also falsch gewesen sein, und es gilt stattdessen: $2^{\mathbb{N}}$ ist nicht abzählbar, also überabzählbar. ☒ 2.1.7

Lemma 2.1.8 ÜBERABZÄHLBARKEIT VON $\mathbb{N} \rightarrow \mathbb{N}$

Die Menge aller Funktionen $\mathbb{N} \rightarrow \mathbb{N}$ ist überabzählbar.

Beweis: Wir nehmen an, dass $\mathbb{N} \rightarrow \mathbb{N}$ doch abzählbar ist. Dann gibt es, da $\mathbb{N} \rightarrow \mathbb{N}$ nicht leer ist, eine Surjektion $f: \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$. Wir bilden eine Funktion $g \in (\mathbb{N} \rightarrow \mathbb{N})$ durch $g(x) = (f(x))(x) + 1$ für alle $x \in \mathbb{N}$. Man beachte, dass dies eine wohldefinierte Funktion ist: $f(x)$ ist eine Funktion von \mathbb{N} nach \mathbb{N} , so dass $(f(x))(x)$, und damit auch $(f(x))(x) + 1$, wohldefinierte natürliche Zahlen sind. Nun benutzen wir die Surjektivität von f ; daraus folgt, dass es eine Zahl n mit $g = f(n)$ gibt. Jetzt gilt aber:

$$\begin{aligned} g(n) &= (\text{Definition von } g) \\ &\quad (f(n))(n) + 1 \\ &= (g = f(n)) \\ &\quad g(n) + 1. \end{aligned}$$

Dies ist ein Widerspruch. Unsere Annahme muss also falsch gewesen sein, und es gilt stattdessen: $\mathbb{N} \rightarrow \mathbb{N}$ ist überabzählbar. □ 2.1.8

Beide zuletzt verwendeten Überabzählbarkeitsbeweise benutzen ein *Selbstanwendungs-* oder *Diagonalisierungsargument*, das im Kern auf G. Cantor zurückgeht (der es für den Nachweis der Überabzählbarkeit der Menge der reellen Zahlen zuerst verwendet hat) und das vom Diagonalverfahren des Abzählbarkeitsbeweises zu unterscheiden ist. Im letzten Beweis wird dieses Argument besonders deutlich. Es werden die Werte von $(f(x))(y)$ auf der „Diagonalen“ $x=y$ betrachtet, und g wird so definiert (durch die Addition des Wertes 1; es könnte dort auch +2 oder +3 usw. stehen), dass ein Widerspruch entstehen muss, sobald die g entsprechende Stelle n der Diagonalen betrachtet wird. Dass eine solche Stelle existiert, wird durch die Surjektivität garantiert. Im Beweis der Überabzählbarkeit von $2^{\mathbb{N}}$ wird ein ähnliches Argument benutzt; dabei ist die Negation \notin in der Definition von A die Analogie des Zusatzes +1 in der Definition von g .

2.2 Graphen und Bäume

Ein (*gerichteter*) *Graph* ist ein Paar (V, E) mit $E \subseteq V \times V$. Grafisch zeichnet man die Elemente von V als Punkte und ein Element $e = (v, w)$ als Pfeil von v nach w . Ein Graph heißt *endlich*, wenn V eine endliche Menge ist, ansonsten *unendlich*. Die Elemente von V heißen *Knoten* (vertices), die Elemente von E *Kanten* (edges).

Ein Graph ist also nichts anderes als eine Menge V zusammen mit einer Relation E auf V . Man zeichnet einen Graphen jedoch nicht durch Verdopplung der Menge V (so wie wir Relationen bisher, z.B. in Abbildung 2.2, veranschaulicht haben), sondern durch eine Menge von Pfeilen E auf einer einzigen Menge V .

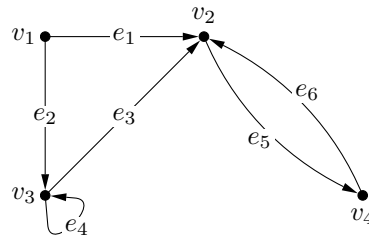
Anmerkung:

Wir benötigen den Begriff eines *ungerichteten Graphen*, in dem die Kanten keine ausgezeichnete Richtung haben, zunächst nicht.

Ende der Anmerkung

Beispiel: Der Graph $G_1 = (V_1, E_1)$ mit

$$\begin{aligned} V_1 &= \{v_1, v_2, v_3, v_4\}, \quad E_1 = \{e_1, e_2, e_3, e_4, e_5, e_6\} \\ \text{und } e_1 &= (v_1, v_2), \quad e_2 = (v_1, v_3), \quad e_3 = (v_3, v_2), \quad e_4 = (v_3, v_3), \quad e_5 = (v_2, v_4), \quad e_6 = (v_4, v_2) \end{aligned}$$

Abbildung 2.4: Ein Graph $G_1 = (V_1, E_1)$

ist in Abbildung 2.4 abgebildet.

Ende des Beispiels

Sei $v \in V$ ein Knoten eines Graphen. Die Menge $E^{-1}(v)$ heißt die Menge der *Eingangsknoten* von v und die Menge $E(v)$ heißt die Menge der *Ausgangsknoten* von v . Ist $E^{-1}(v)$ eine endliche Menge, dann heißt ihre Kardinalität $|E^{-1}(v)|$ der *Eingangsgrad* von v , ist $E(v)$ endlich, dann heißt die Kardinalität $|E(v)|$ der *Ausgangsgrad* von v .

Eine Folge $v_0 e_1 v_1 \dots e_m v_m$ mit $m \geq 0$ und $e_j = (v_{j-1}, v_j)$ für alle $0 < j \leq m$ heißt ein (*endlicher*) *Weg* (von v_0 nach v_m). Ein solcher Weg heißt *Zyklus*, wenn $m > 0$ und $v_0 = v_m$. Ein Graph heißt *azyklisch*, wenn es in ihm keine Zyklen gibt. Eine unendliche Folge $v_0 e_1 v_1 e_2 v_2 \dots$ mit $e_j = (v_{j-1}, v_j)$ für alle $0 < j$ heißt ein (*unendlicher*) *Weg* (ausgehend von v_0).

Beispiel: In Abbildung 2.4 gilt

$$E^{-1}(v_1) = \emptyset, \quad E^{-1}(v_2) = \{v_1, v_3, v_4\}$$

$$E(v_1) = \{v_3\}, \quad E(v_2) = \{v_4\}, \quad E(v_3) = \{v_2, v_3\}$$

$v_1 e_1 v_2 e_5 v_4 e_6 v_2$ ist ein endlicher Weg

$v_2 e_5 v_4 e_6 v_2$ und $v_3 e_4 v_3 e_4 v_3$ sind Zyklen

$v_1 e_2 v_3 e_4 v_3 e_3 v_2 e_5 v_4 e_6 v_2 e_5 v_4 e_6 v_2 \dots$ ist ein unendlicher Weg.

Ende des Beispiels

Ein *Baum* ist ein azyklischer Graph (V, E) mit genau einem Element $w \in V$, das den Eingangsgrad 0 hat (dieses Element heißt seine *Wurzel*), so dass es zu jedem Knoten $v \in V$ genau einen Weg von w nach v gibt. Als Folge dieser Definition hat jeder Knoten v eines Baumes entweder den Eingangsgrad 0 (falls $v = w$) oder den Eingangsgrad 1 (falls $v \neq w$). Falls $(v, v') \in E$, nennt man v' auch *Kind* (oder *unmittelbaren Nachfolger*) von v und v *Vater* (oder *unmittelbaren Vorgänger*) von v' . Falls $(v, v') \in E^*$, nennt man v' *Nachfolger* von v und v *Vorgänger* von v' . Ein Knoten mit Ausgangsgrad 0 heißt *Blatt* des Baumes.

Beispiel: Der Baum $G_2 = (V_2, E_2)$ mit

$$V_2 = \{w, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}, \quad E_2 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$$

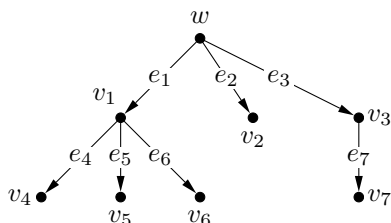
$$\text{und } e_1 = (w, v_1), \quad e_2 = (w, v_2), \quad e_3 = (w, v_3), \quad e_4 = (v_1, v_4), \quad e_5 = (v_1, v_5), \quad e_6 = (v_1, v_6), \quad e_7 = (v_3, v_7)$$

ist in Abbildung 2.5 abgebildet. Es gilt hier:

v_1 ist ein Kind von w , v_1 ist ein Vater von v_5 ,

w ist ein Vorgänger (aber kein Vater) von v_5 ,

$\{v_4, v_5, v_6, v_7\}$ ist die Menge der Blätter.

Ende des BeispielsAbbildung 2.5: Ein Baum $G_2 = (V_2, E_2)$ **Lemma 2.2.1** LEMMA VON KÖNIG

Ein unendlicher Baum, dessen Knoten endlichen Ausgangsgrad haben, hat einen unendlichen Weg.

Beweis: Sei (V, E) ein Baum mit Wurzel w . Wir setzen $v_0 = w$. Da v_0 endlichen Ausgangsgrad hat, gibt es wegen der Unendlichkeit des Baumes unter den endlich vielen Kindern von v_0 mindestens eines, das unendlich viele Nachfolger hat. Wähle ein solches Kind als v_1 und setze $e_1 = (v_0, v_1)$. Da v_1 wieder endlichen Ausgangsgrad, aber unendlich viele Nachfolger hat, kann das Verfahren wiederholt werden und wir gewinnen ein Kind v_2 von v_1 mit unendlich vielen Nachfolgern; setze $e_2 = (v_1, v_2)$. So fortfahrend, erhalten wir einen unendlichen Weg $v_0 e_1 v_1 e_2 v_2 \dots$. \square 2.2.1

2.3 Alphabete, Wörter und Sprachen

Ein *Alphabet* ist eine beliebige, nichtleere, endliche Menge. Die Elemente eines Alphabets heißen *Zeichen* (oder *Buchstaben* oder *Symbole*). Wir benutzen Σ, Γ (oder auch A, B) als typische Namen für Alphabete. Später wollen wir Abzählbarkeitsargumente benutzen und wollen deswegen die *Anzahl* der Alphabete abzählbar halten. Um trotzdem genügend (abzählbar unendlich viele) Buchstaben zur Verfügung zu haben, betrachten wir eine abzählbar unendliche Menge \mathbb{A} von *zulässigen Buchstaben*. Darunter mögen sich alle Buchstaben aller Alphabete der Sprachen der Erde, alle Ziffern, alle Sonderzeichen, alle daraus formbaren Paare und Tupel, sowie noch ein unendlich großer Rest-Zeichenvorrat befinden. Alle Alphabete Σ, Γ usw. schränken wir nunmehr so ein, dass es nur nicht leere, endliche Teilmengen von \mathbb{A} sein dürfen. In \mathbb{A} gibt es mehr als genug Zeichen, so dass kein Mangel an Zeichenauswahl herrschen wird.

Beispiel:

$$\begin{aligned} \Sigma_1 &= \{a, \dots, z, A, \dots, Z\} \\ \Sigma_2 &= \{0, 1\} \\ \Sigma_3 &= \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x \leq 5 \wedge y \leq 9\} \end{aligned}$$

sind drei Alphabete.

Ende des Beispiels

Ein *Wort* (*Kette*, *String*) über einem Alphabet Σ ist eine endliche Folge von Zeichen aus Σ . Ist w ein Wort, bezeichnet $|w|$ seine *Länge*, d.h., die Anzahl der Zeichen im Wort.

Ein Wort w ist also eine Aneinanderreihung von genau $|w|$ Buchstaben. Es ist überaus sinnvoll, auch die Aneinanderreihung von genau 0 Buchstaben zuzulassen. Das dann entstehende Wort heißt das *leere Wort* und wird mit ε bezeichnet.

Beispiel:

<i>Liebe</i>	(Wort über Σ_1)	$ Liebe = 5$
011	(Wort über Σ_2)	$ 011 = 3$
(0,0)(0,9)	(Wort über Σ_3)	$ (0,0)(0,9) = 2$ (!)
ε	(Wort über jedem Alphabet)	$ \varepsilon = 0$

Ende des Beispiels

Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet.

Beispiel:

$$\begin{aligned} \{a, \dots, z\}^* &= \{ \underbrace{\varepsilon, a, \dots, z}_{\text{Länge 1}}, \underbrace{aa, ab, \dots, az, ba, bb, \dots, zz}_{\text{Länge 2}}, aaa, \dots, \dots \} \\ \{0, 1\}^* &= \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots, \dots \}. \end{aligned}$$

Ende des Beispiels

Die Menge Σ stimmt genau mit den Wörtern der Länge 1 über Σ überein und wird deswegen als eine Teilmenge von Σ^* aufgefasst. Man beachte aber, dass Σ *per definitionem* stets endlich ist, während Σ^* aufgrund der Forderung $\Sigma \neq \emptyset$ stets unendlich ist. Es gilt jedoch:

Lemma 2.3.1 ABZÄHLBARKEIT VON Σ^*

Σ^* ist abzählbar unendlich.

Beweis: Wir verallgemeinern die oben für $\{a, \dots, z\}^*$ angewandte Abzählungsweise. Für jede Zahl $n \in \mathbb{N}$ gibt es genau $|\Sigma|^n$ Wörter der Länge n über Σ . Man zähle Σ^* so ab, dass zuerst das leere Wort, dann alle Wörter der Länge 1, danach alle Wörter der Länge 2, usw., vorkommen; da diese Teilmengen jedesmal endlich sind, liefert dies eine vernünftige Abzählung.

Des Weiteren ist Σ^* eine unendliche Menge, weil Σ nicht leer ist. □ 2.3.1

Eine grundlegende binäre Operation auf Wörtern ist die *Verkettung* oder *Konkatenation*, die für zwei Worte $v = a_1 \dots a_n$ und $w = b_1 \dots b_m$ in Σ^* als das Wort

$$v \cdot w = vw = a_1 \dots a_n b_1 \dots b_m \text{ in } \Sigma^*,$$

also einfach als die „Hintereinanderschreibung“ von v und w , definiert ist. Das leere Wort ε ist beidseitig neutral bezüglich dieser Operation, d.h. $\varepsilon v = v = v \varepsilon$. Außerdem ist die Konkatenation assoziativ, d.h. $u(vw) = (uv)w$ für beliebige Wörter u , v und w . Wir definieren die n -fache Hintereinanderschreibung eines Wortes v induktiv: $v^0 = \varepsilon$ und $v^{n+1} = vv^n$. Beispielsweise ist $(01)^3 = 010101$. Ebenso kann die Menge der Wörter einer gegebenen Länge n induktiv definiert werden:

$$\begin{aligned} \Sigma^0 &= \{\varepsilon\} \\ \Sigma^{n+1} &= \{av \mid a \in \Sigma, v \in \Sigma^n\}. \end{aligned}$$

Beispielsweise ist $\{0, 1\}^2 = \{00, 01, 10, 11\}$. Es gilt $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$. Mit Σ^+ bezeichnen wir die *Menge aller nicht leeren Wörter* über Σ , d.h. $\Sigma^+ = \bigcup_{n \in (\mathbb{N} \setminus \{0\})} \Sigma^n$.

Anmerkung:

Man beachte die Analogie zu entsprechenden Definitionen bei Relationen $\rho \subseteq X \times X$ (siehe Abschnitt 2.1.2): ρ^* entspricht Σ^* und ρ^+ entspricht Σ^+ . Diese Analogie ist jedoch keine vollständige. Man kann die Menge Σ^+ zwar äquivalent folgendermaßen definieren: $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$, aber die Relation $\rho^* \setminus id_X$ ist irreflexiv, während die Relation ρ^+ durchaus reflexiv sein kann. Es stimmen also die Mengen Σ^+ und $\Sigma^* \setminus \{\varepsilon\}$ überein, während die Relationen ρ^+ und $\rho^* \setminus id_X$ unterschiedlich sein können.

Ende der Anmerkung

Ein Wort v heißt *Teilwort* eines Wortes w , falls gilt: $\exists u_1, u_2 \in \Sigma^* : w = u_1 v u_2$. Es gibt hiervon zwei wichtige Spezialfälle. Ein Wort w heißt *Präfix* (bzw. *Suffix*) von v , wenn gilt: $\exists u \in \Sigma^* : v = w u$ (beziehungsweise $\exists u \in \Sigma^* : v = u w$). Beispielsweise kommt das Wort 0101 in dem Wort 01010101 dreimal als Teilwort, einmal als Präfix und einmal als Suffix vor.

Das zu v *reverse Wort* (oder *Spiegelwort*) v^R ist induktiv definiert: $\varepsilon^R = \varepsilon$ und $(av)^R = v^R a$ mit $a \in \Sigma$. Beispielsweise ist $(0101)^R = 1010$.

Eine *Sprache* über einem gegebenen Alphabet Σ ist eine Teilmenge von Σ^* , d.h. $L \subseteq \Sigma^*$. Wir verwenden L (für *language*) als typischen Buchstaben für Sprachen.

Achtung! Die Menge Σ gehört zur Definition einer Sprache mit dazu. Wenn Sie das Wort „Sprache“ hören, sollten Sie sich immer auch die Frage „weiß ich denn genau, über welchem Alphabet?“ beantworten.

Beispiel: Wir betrachten das Alphabet $\Sigma_2 = \{0, 1\}$.

L_1	$= \emptyset$	(die <i>leere Sprache</i>)
L_2	$= \{\varepsilon\}$	(die <i>Einheitssprache</i>)
L_3	$= \{0, 1\}^*$	(= $\{\varepsilon, 0, 1, 00, 01, \dots\}$, die <i>volle Sprache</i> über $\{0, 1\}$)
L_4	$= \{0^n \mid n \in \mathbb{N}\}$	(= $\{\varepsilon, 0, 00, 000, \dots\}$)
L_5	$= \{w0w \mid w \in \{1\}^*\}$	(= $\{0, 101, 11011, \dots\}$)
L_6	$= \{0^n 1^n \mid n \in \mathbb{N}\}$	(= $\{\varepsilon, 01, 0011, 000111, \dots\}$)
L_7	$= \{(01)^n \mid n \in \mathbb{N}\}$	(= $\{\varepsilon, 01, 0101, 010101, \dots\}$)

sind Sprachen über diesem Alphabet.

Ende des Beispiels

Für eine Sprache $L \subseteq \Sigma^*$ ist die *charakteristische Funktion von L*, χ_L , folgendermaßen definiert:

$$\chi_L : \begin{cases} \Sigma^* & \mapsto \{0, 1\} \\ w & \mapsto \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{falls } w \notin L. \end{cases} \end{cases}$$

Diese Funktion heißt „charakteristisch“, weil man aus ihr die Sprache L zurückrechnen kann: die Wörter, für die die Funktion 1 liefert, sind in der Sprache, diejenigen, für die die Funktion 0 liefert, nicht.

Die *halbe (oder partielle) charakteristische Funktion* ist eine partielle Funktion χ_L^+ , die folgendermaßen definiert ist:

$$\chi_L^+ : \begin{cases} \Sigma^* & \xrightarrow{p} \{1\} \\ w & \mapsto \begin{cases} 1 & \text{falls } w \in L \\ \text{undef} & \text{falls } w \notin L. \end{cases} \end{cases}$$

Für zwei Sprachen L über Σ und K über Γ sind die folgenden Verknüpfungen definiert:

$$\begin{aligned}
 L \cap K &= \{w \mid w \in L \wedge w \in K\} && \text{(der } \textit{Durchschnitt} \text{ von } L \text{ und } K \text{)} \\
 L \cup K &= \{w \mid w \in L \vee w \in K\} && \text{(die } \textit{Vereinigung} \text{ von } L \text{ und } K \text{)} \\
 L \setminus K &= \{w \mid w \in L \wedge w \notin K\} && \text{(die } \textit{Differenz} \text{ von } L \text{ und } K \text{)} \\
 \overline{L} &= \Sigma^* \setminus L && \text{(das } \textit{Komplement} \text{ von } L \text{)} \\
 L \cdot K = LK &= \{uv \mid u \in L \wedge v \in K\} && \text{(die } \textit{Konkatenation} \text{ von } L \text{ und } K \text{)} \\
 L^0 &= \{\varepsilon\} && \text{(die } \textit{Einheitssprache} \text{ oder } \textit{nullte Iterierte} \text{ einer Sprache } L \text{)} \\
 L^n &= L \cdot L^{n-1} && \text{(die } \textit{nte Iterierte} \text{ einer Sprache } L \text{)} \\
 L^* &= \bigcup_{n \in \mathbb{N}} L^n && \text{(der } \textit{„Sternabschluss“} \text{ einer Sprache } L \text{)} \\
 L^+ &= \bigcup_{n \in (\mathbb{N} \setminus \{0\})} L^n && \text{(der } \textit{„Plusabschluss“} \text{ einer Sprache } L \text{)} \\
 L^R &= \{w^R \mid w \in L\} && \text{(die zu einer Sprache } L \text{ „} \textit{reverse Sprache“} \text{ oder „} \textit{Spiegelsprache“} \text{).}
 \end{aligned}$$

Durchschnitt, Vereinigung und Differenz entsprechen den mengentheoretischen Operationen, ebenso das Komplement, das bei Sprachen immer in Bezug auf die zu Grunde liegende Menge Σ^* gemeint ist. Die sprachenspezifischen Operationen sind die Konkatenation, die – wie die analoge Operation auf Wörtern – assoziativ ist und die einelementige Sprache $\{\varepsilon\}$ als beidseitiges Neutrales besitzt (d.h., $\{\varepsilon\} \cdot L = L = L \cdot \{\varepsilon\}$), und übrigens auch die leere Sprache als „Null“: $\emptyset \cdot L = \emptyset = L \cdot \emptyset$, sowie die iterierte Konkatenation L^0, L^1, L^2 usw. Eine weitere wichtige sprachenspezifische Operation ist die Spiegelsprachenbildung.

Beispiel: Für $\Sigma = \{0, 1\}$, $L_1 = \{1, 00\}$ und $L_2 = \{00, 01, 11\}$ gilt:

$$\begin{aligned}
 L_1 \cap L_2 &= \{00\} \\
 L_1 \cup L_2 &= \{1, 00, 01, 11\} \\
 L_1 \setminus L_2 &= \{1\} \\
 \overline{L_1} &= \{0, 1\}^* \setminus \{1, 00\} \\
 L_2^R &= \{00, 10, 11\} \\
 L_1 \cdot L_2 &= \{100, 101, 111, 0000, 0001, 0011\} \\
 L_2 \cdot L_1 &= \{001, 0000, 011, 0100, 111, 1100\} \\
 L_1 \cdot L_2^R &= \{100, 110, 111, 0000, 0010, 0011\} \\
 L_1^0 &= \{\varepsilon\} \\
 L_1^1 &= \{1, 00\} \\
 L_1^2 &= \{11, 100, 001, 0000\} \\
 L_1^3 &= \{111, 1100, 1001, 10000, 0011, 00100, 00001, 000000\} \\
 L_1^* &= \{\varepsilon, 1, 00, 11, \dots, 0000, 111, \dots, 000000, \dots\} \\
 L_2^+ &= \{00, 01, 11, 0000, 0001, 0011, 0100, 0101, 0111, 1100, 1101, 1111, 000000, 000001, \dots\}.
 \end{aligned}$$

Ende des Beispiels

Man sieht, dass $L \cup K$ und LK Sprachen über $\Sigma \cup \Gamma$, $L \cap K$ eine Sprache über $\Sigma \cap \Gamma$, und $L \setminus K, \overline{L}, L^n, L^*$ und L^+ Sprachen über Σ sind. Vor einiger Zeit haben wir Σ^n als Wörter der Länge n definiert, eben ist jedoch eine Definition von Σ^n hinzu gekommen, wenn Σ als spezielle Sprache (der Wörter der Länge 1) aufgefasst wird. Diese Doppeldeutigkeit von Σ^n ist aber erträglich, denn beide Definitionen kommen auf das Gleiche heraus. Analoges gilt für Σ^* und für Σ^+ .

Eine Klasse³ von Sprachen \mathcal{L} heißt *abgeschlossen* gegenüber einer Operation auf Sprachen, wenn mit den Argumenten auch immer das Ergebnis in \mathcal{L} liegt. Zum Beispiel heißt \mathcal{L} heißt abgeschlossen gegenüber der Konkatenation, wenn aus $L \in \mathcal{L} \wedge K \in \mathcal{L}$ folgt: $L \cdot K \in \mathcal{L}$.

Beispiel:

$\mathcal{E}\mathcal{N}\mathcal{D}\mathcal{L}$ sei die Klasse der endlichen Sprachen über Σ . Dann ist $\mathcal{E}\mathcal{N}\mathcal{D}\mathcal{L}$ abgeschlossen gegenüber der Konkatenation \cdot , aber nicht gegenüber dem Sternabschluss $*$.

Ende des Beispiels

2.4 Elemente einer Programmiersprache

Später werden wir Wörter über einem Alphabet in Wörter über einem anderen Alphabet oder in Zahlen umrechnen. Das wollen wir algorithmisch-konstruktiv tun, d.h., es soll eine Vorschrift angegeben werden, wie diese Umrechnungen zu bewerkstelligen sind. Auch für andere Zwecke benötigen wir verschiedene Algorithmen.

Wir geben Algorithmen und die Datenstrukturen, auf denen sie operieren, teils informell an, teils in einer Programmiernotation, die wir jetzt einführen. Die Notation ist an gängige Programmiersprachen wie Java und C angelehnt, enthält aber wesentlich weniger Möglichkeiten. Wir werden z.B. auf Prozedur- und Blockmechanismen verzichten.

2.4.1 Deklarationen und Datentypen

Programme dienen der Manipulation von Daten, die in Form von Variablenwerten vorliegen. Für jede in einem Programm verwendete Variable stellen wir sicher, dass ihr Wertebereich (auch der *Typ* der Variablen genannt) stets wohldefiniert ist. Dieser Bereich ergibt sich manchmal aus dem Kontext, manchmal verwenden wir eine Klausel wie etwa $x \in \mathbb{N}$, um anzuzeigen, dass die Variable x nur Werte aus \mathbb{N} haben darf, manchmal verwenden wir eine explizite Deklaration der Form:

var x : *Wertebereich*.

Diese Deklaration bedeutet, dass x nur Werte aus dem angegebenen *Wertebereich* haben darf und dass auf x nur die in diesem Bereich gültigen Operationen angewendet werden dürfen. Wahlweise kann eine **init**-Klausel hinzugefügt werden, um den Anfangswert einer Variablen anzugeben.

Uns interessieren in erster Linie die folgenden Wertebereiche:

Natürliche Zahlen

Mit **var** $x : \mathbb{N}$ wird x als natürlichzahlige Variable deklariert. Konstanten sind alle c in \mathbb{N} . Als grundlegende Operation erlauben wir das Addieren der Zahl 1. D.h.: $x+1$ ist ein wohldefinierter Ausdruck, dessen Wert $S(x)$ (siehe Abschnitt 2.1.4) ist. Wir werden später sehen, dass sich andere Operationen (z.B. $x+c$, wobei c eine beliebige Konstante aus \mathbb{N} ist) ableiten lassen.

³D.h. eine Menge. Man verwendet das Wort „Klasse“ statt „Menge“ oft dann, wenn es sich um „große“ Mengen handelt. Wir verwenden oft kalligrafische Buchstaben wie \mathcal{L} für Klassen.

Der Boolesche Bereich

var $b : \{\mathbf{false}, \mathbf{true}\}$ bedeutet, dass die Variable b einen der zwei angegebenen Werte annehmen kann. Konstanten sind die beiden Werte **false**, **true**. Als grundlegende Operationen sind die Negation \neg (einstellig) und die Konjunktion \wedge (zweistellig) mit ihren üblichen Bedeutungen zugelassen.

Wörter

Mit **var** $w : \Sigma^*$ wird eine Wortvariable w über dem Alphabet Σ deklariert. Konstanten sind das leere Wort ε und die einelementigen Wörter $a \in \Sigma$. Ein Wort w wird von 0 bis $|w|-1$ indiziert. Mit $w[i]$, für $0 \leq i \leq |w|-1$, ist der i te Buchstabe von w gemeint; $w[0]$ ist also der linke, $w[|w|-1]$ der rechte Buchstabe eines nicht leeren Wortes w . An grundlegenden Operationen sind folgende zugelassen: die Berechnung der Wortlänge $|w|$, die Katenation ww' zweier Wortvariablen w, w' und der *chop*-Operator $chop(w, i)$, der nur anwendbar ist für $0 \leq i \leq |w|$ und ein Wortpaar liefert, nämlich $(w[0] \dots w[i-1], w[i] \dots w[|w|-1])$. Ist z.B. $w = aba$, dann ist

$$\begin{aligned} chop(w, 0) &= (\varepsilon, aba) \\ chop(w, 1) &= (a, ba) \\ chop(w, 2) &= (ab, a) \\ chop(w, 3) &= (aba, \varepsilon). \end{aligned}$$

Außerdem ist $chop(\varepsilon, 0) = (\varepsilon, \varepsilon)$.

2.4.2 Aufbau eines Programms

Wir benutzen die folgende Struktur für Programme:

Input/Output-Teil;
Deklaration lokaler Variablen;
Kommandoteil.

Der Input-/Output-Teil beschreibt die Ein-/Ausgabeschnittstelle des Programms. Er kann Klauseln der Art:

input *Variable* (eventuell noch mit \in Wertebereich)
output *Variable* (eventuell noch mit \in Wertebereich oder mit Initialisierung)

enthalten. Dabei führt die **input**-Klausel Variablen auf, die global zum Programm definiert sind und die als Eingabeparameter dienen. Die **output**-Klausel führt Variablen auf, die als Ausgabe des Programms an seine Umgebung abgeliefert werden (das können teilweise die gleichen Variablen sein, die auch in der **input**-Klausel vorkommen). Sofern die Typen dieser Variablen nicht klar aus dem Kontext hervorgehen, werden sie hier aufgeführt.

Nach dem Input-/Output-Teil kann eine Reihe von Deklarationen von Variablen kommen, die rein lokal im Programm benötigt werden und keine Auswirkungen nach außen haben. Keine Variable darf mehrfach deklariert werden, so dass bei jeder Verwendung eines Variablennamens klar ist, welche Variable gemeint ist. Der Kommandoteil enthält einen Algorithmus. Zu seiner Beschreibung stehen eine Reihe von Sprachelementen zur Verfügung: primitive Kommandos und Kommandoverknüpfungen.

2.4.3 Primitive Kommandos

Das **Leerkommando skip**. Dieses Kommando terminiert stets und bewirkt außerdem nichts weiter (insbesondere keine Veränderungen von Variablenwerten).

Das **Nichtterminierungskommando loop**. Dieses Kommando terminiert nie und bewirkt außerdem nichts weiter (insbesondere keine Veränderungen von Variablenwerten).

Das **Terminierungskommando stop**. Dieses Kommando terminiert stets (ohne Veränderungen von Variablenwerten) und bewirkt zudem den Stopp des ganzen Programms.

Die **Zuweisung** $x := E$. Dieses Kommando bewirkt eine Neuberechnung (und normalerweise Änderung) des Wertes der Variablen x . Dazu wird der momentane Wert des Ausdrucks E berechnet und als neuer Wert der Variablen x zugewiesen. Wir fordern, dass der Ausdruck E in endlicher Zeit ausgewertet werden kann, dass keine Wertänderungen von Variablen ungleich x involviert sind und dass der errechnete Wert stets im Wertebereich der Variablen x liegt.

Beispiel: Mit $\text{var } x, y : \mathbb{N}; b : \{\text{false}, \text{true}\}$ gilt Folgendes:

$$\begin{array}{l} \{x=5, y=7, b=\text{true}\} \quad x := (y-3) \quad \{x=4, y=7, b=\text{true}\} \\ \{x=5, y=7, b=\text{true}\} \quad x := (y-x) \quad \{x=2, y=7, b=\text{true}\} \\ \{x=5, y=7, b=\text{true}\} \quad b := (x=2) \quad \{x=5, y=7, b=\text{false}\} \\ \{x=5, y=7, b=\text{true}\} \quad x := (x-y) \quad \text{unzulässig, da der Wert } -2 \text{ nicht im Typ von } x \text{ ist.} \end{array}$$

Dabei bedeutet $\{\alpha\}Z\{\beta\}$: wenn α die Werte der Variablen direkt vor der Ausführung der Zuweisung Z beschreibt, dann beschreibt β deren Werte nach der Ausführung von Z .

Ende des Beispiels

Ebenfalls unzulässig sind die Zuweisungen $b := 2$ und $x := b$, da die Typen rechts und links des Zeichens $:=$ nicht übereinstimmen, und $b := (y=b)$, da die Typen rechts und links des Gleichheitszeichens im Ausdruck nicht übereinstimmen.

2.4.4 Kommandoverknüpfungen

Die Hintereinanderausführung zweier Kommandos

Diese Verknüpfung wird mit Hilfe des Semikolons dargestellt: $K_1; K_2$ bedeutet: erst wird K_1 ausgeführt; wenn K_1 terminiert, wird K_2 ausgeführt.

Beispiel:

$$\begin{array}{l} \{x=5, y=7, b=\text{true}\} \quad b := (x=2); x := (y-x) \quad \{x=2, y=7, b=\text{false}\} \\ \{x=5, y=7, b=\text{true}\} \quad x := (y-x); b := (x=2) \quad \{x=2, y=7, b=\text{true}\} \\ \{x=5, y=7, b=\text{true}\} \quad x := (y-1); y := (x-1) \quad \{x=6, y=5, b=\text{true}\} \\ \{x=5, y=7, b=\text{true}\} \quad \text{skip}; y := (x-1) \quad \{x=5, y=4, b=\text{true}\} \\ \{x=5, y=7, b=\text{true}\} \quad y := (x-1); \text{loop} \quad \text{Werte werden } x=5, y=4, b=\text{true}; \text{ terminiert nicht} \\ \{x=5, y=7, b=\text{true}\} \quad \text{loop}; y := (x-1) \quad \text{terminiert nicht; Werte bleiben } x=5, y=7, b=\text{true}. \end{array}$$

Ende des Beispiels

Das allgemeine Alternativkommando

Dieses Kommando hat die Form

$$\mathbf{if } B_1 \rightarrow K_1 \square B_2 \rightarrow K_2 \square \dots \square B_m \rightarrow K_m \mathbf{ fi.}$$

Dabei sind die K_j Kommandos und die B_j spezielle Ausdrücke, deren Wert stets im Bereich $\{\mathbf{false}, \mathbf{true}\}$ liegt. Solche Ausdrücke werden *Boolesch* genannt. Man liest **if** als „falls“, \rightarrow als „dann“ und \square als „oder falls“. Zur Ausführung eines Alternativkommandos wird ein K_j gewählt, dessen zugehöriges B_j zu **true** ausgewertet werden kann, und ausgeführt. Gibt es mehrere solcher $B_j \rightarrow K_j$, wird eines davon beliebig gewählt. Gibt es kein solches $B_j \rightarrow K_j$, wirkt das Alternativkommando wie ein **loop**. Als B_m darf auch das Wort **else** stehen, was wie „andernfalls“ gelesen wird; K_m wird in diesem Fall (nur) dann ausgeführt, wenn alle B_1 bis B_{m-1} zu **false** ausgewertet werden.

Beispiel:

Betrachte **if** $x=0 \rightarrow y := 0 \square x \neq 0 \rightarrow y := 1$ **fi**. Dieses Alternativkommando weist y den Signumwert von x zu (siehe hierzu Abschnitt 2.1.4). Das Kommando **if** $x=0 \rightarrow y := 0 \square \mathbf{else} \rightarrow y := 1$ **fi** leistet das Gleiche.

Betrachte andererseits **if** $x=5 \rightarrow \mathbf{skip} \square x \geq 3 \rightarrow x := 1$ **fi**. Dann sind bei einem ursprünglichen Wert $x=5$ zwei Ausführungen möglich: nach Ausführung der **if**-Kommandos gilt entweder $x=5$ oder $x=1$. Bei einem ursprünglichen Wert $x=6$ ist nur eine Ausführung möglich: nach Ausführung des **if**-Kommandos gilt $x=1$. Bei einem ursprünglichen Wert $x=0$ ist keine Ausführung möglich: die Ausführung des **if**-Kommandos terminiert nicht.

Ende des Beispiels

Das IF-Kommando

Wir definieren zwei Kommandos, die häufig in Programmiersprachen vorkommen, als Spezialfälle des allgemeinen Alternativkommandos:

$$\begin{aligned} \mathbf{IF } B \mathbf{ THEN } K \mathbf{ ENDIF} &= \mathbf{if } B \rightarrow K \square \neg B \rightarrow \mathbf{skip} \mathbf{ fi} \\ \mathbf{IF } B \mathbf{ THEN } K \mathbf{ ELSE } K' \mathbf{ ENDIF} &= \mathbf{if } B \rightarrow K \square \neg B \rightarrow K' \mathbf{ fi.} \end{aligned}$$

Diese Spezialfälle sind nicht echt einschränkend, denn eine „First Come First Served“-Version des allgemeinen Alternativkommandos

$$\mathbf{if } B_1 \rightarrow K_1 \square B_2 \rightarrow K_2 \square \dots \square B_m \rightarrow K_m \mathbf{ fi}$$

kann unter Zuhilfenahme des IF-Kommandos folgendermaßen implementiert werden:

```

var nochmal : {false, true} (init false);
IF  $B_1 \wedge \neg \mathit{nochmal}$  THEN  $K_1$ ; nochmal := true ENDIF;
...
IF  $B_m \wedge \neg \mathit{nochmal}$  THEN  $K_m$ ; nochmal := true ENDIF;
IF  $\neg \mathit{nochmal}$  THEN loop ENDIF.

```

Das allgemeine Schleifenkommando

Dieses Kommando hat die Form

$$\mathbf{do} B_1 \rightarrow K_1 \square B_2 \rightarrow K_2 \square \dots \square B_m \rightarrow K_m \mathbf{od},$$

mit B_j und K_j wie beim Alternativkommando. Man liest **do** als „wiederhole falls“, \rightarrow als „dann“ und \square als „oder falls“. Zur Ausführung eines Alternativkommandos wird wiederholt ein K_j gewählt, dessen zugehöriges B_j zu **true** ausgewertet werden kann, und ausgeführt. Gibt es mehrere solcher $B_j \rightarrow K_j$, wird eines davon beliebig gewählt. Wenn es kein solches $B_j \rightarrow K_j$ gibt, wirkt das Schleifenkommando wie ein **skip**.

Beispiel:

```

input  $n : \mathbb{N}$ ;
output  $x : \mathbb{N}$  (init 0);
var  $i : \mathbb{N}$  (init 0);
do  $i < n \rightarrow i := i+1; x := x+i$  od.

```

Dieses Programm liest einen **input**-Parameter n , berechnet die Summe der Zahlen $0, \dots, n$ und liefert das Ergebnis in der **output**-Variablen x ab. Zum Beispiel wird der Input $\{n = 4, x = 0\}$ in den Output $\{(n = 4, x = 10)\}$ transformiert.

Ende des Beispiels

Im Folgenden definieren wir mehrere Kommandos, die häufig in Programmiersprachen vorkommen, als Spezialfälle des allgemeinen Schleifenkommandos.

Das WHILE-Kommando

$$\mathbf{WHILE} B \mathbf{DO} K \mathbf{ENDWHILE} = \mathbf{do} B \rightarrow K \mathbf{od}.$$

Dieser Spezialfall ist nicht echt einschränkend, denn eine „First Come First Served“-Version des allgemeinen Schleifenkommandos

$$\mathbf{do} B_1 \rightarrow K_1 \square B_2 \rightarrow K_2 \square \dots \square B_m \rightarrow K_m \mathbf{od}$$

kann unter Zuhilfenahme der WHILE- und IF-Kommandos folgendermaßen implementiert werden:

```

var nochmal : {false, true} (init true);
WHILE nochmal DO nochmal := false;
                    IF  $B_1 \wedge \neg \textit{nochmal}$  THEN  $K_1; \textit{nochmal} := \mathbf{true}$  ENDIF;
                    ...
                    IF  $B_m \wedge \neg \textit{nochmal}$  THEN  $K_m; \textit{nochmal} := \mathbf{true}$  ENDIF
ENDWHILE

```

Das LOOP-Kommando

Unter der Annahme, dass die Variable x den Typ \mathbb{N} hat und in K nicht auf der linken Seite einer Zuweisung vorkommt, ist das LOOP-Kommando folgendermaßen definiert:

$$\mathbf{LOOP} x \mathbf{DO} K \mathbf{ENDLOOP} = \mathbf{WHILE} x > 0 \mathbf{DO} K; x := x-1 \mathbf{ENDWHILE}.$$

Das Programm LOOP x DO K ENDLOOP wirkt also wie eine x -malige Wiederholung von K :

$$\underbrace{K; K; \dots; K}_{x\text{-mal}}$$

Das LOOP-Kommando ist offensichtlich einschränkend, denn es kann z.B. nicht *per se* (höchstens durch eine Nichtterminierung innerhalb von K) zur Nichtterminierung führen. Die Eigenschaft, dass x nicht auf der linken Seite einer Zuweisung in K vorkommen kann, ist hierfür entscheidend; dürfte man K z.B. zu $x := x+1$ setzen, wäre eine unendlich laufende Schleife programmiert.

2.4.5 WHILE- und LOOP-Registerprogramme

Wir definieren eine Teilklasse von Programmen mit starken syntaktischen Einschränkungen:

- Nur Variablen x, y, \dots mit Typ \mathbb{N} („Register“) werden zugelassen;
- das Programm **loop** wird nicht erlaubt;
- an Ausdrücken E auf der rechten Seite einer Zuweisung werden nur zugelassen:
 - eine Konstante $c \in \mathbb{N}$, oder x , oder $x+1$, wobei x eine Variable ist;
- an Booleschen Ausdrücken werden nur zugelassen: **true**, $x=c$, $x=y$, **false**, $x \neq c$, $x \neq y$;
- nur das IF-Kommando wird als Auswahlkommando zugelassen;
- nur das WHILE-Kommando wird als Schleifenkommando zugelassen.

Programme mit diesen Einschränkungen nennen wir WHILE-Registerprogramme, oder kürzer: WHILE-Programme. Wenn nur LOOP- statt WHILE-Kommandos zugelassen werden, sprechen wir von LOOP-Registerprogrammen, oder kürzer von LOOP-Programmen.

Wir werden später sehen, dass von diesen Einschränkungen nur diejenige wirklich wesentlich ist, die von WHILE- auf LOOP-Programme reduziert. Weil das Kommando **loop** ausgeschlossen ist und wegen der Eigenschaften des LOOP-Kommandos terminiert jedes LOOP-Programm. Das Kommando **loop** kann jedoch als ein WHILE-Programm WHILE **true** DO **skip** ENDWHILE implementiert werden.

2.4.6 WHILE- und LOOP-berechenbare zahlentheoretische Funktionen

Definition 2.4.1 WHILE- UND LOOP-BERECHENBARKEIT ZAHLENTHEORETISCHER FUNKTIONEN

Eine Funktion $f: \mathbb{N}^n \rightarrow \mathbb{N}^k$ heißt *WHILE-berechenbar* bzw. *LOOP-berechenbar*, wenn es ein WHILE-Programm (bzw. ein LOOP-Programm) mit den folgenden Eigenschaften gibt:

- Die **input**-Zeile hat n Variablen x_1, \dots, x_n , die **output**-Zeile hat k Variablen z_1, \dots, z_k .
- Das Programm terminiert bei gegebenen Eingabewerten $x_1=c_1, \dots, x_n=c_n$ mit den Ausgabewerten $z_1=d_1, \dots, z_k=d_k$ genau dann, wenn $f(c_1, \dots, c_n)$ definiert ist und $f(c_1, \dots, c_n) = (d_1, \dots, d_k)$ gilt.

□ 2.4.1

Demnach sind alle LOOP-berechenbaren Funktionen total, da LOOP-Programme stets terminieren. Wir überprüfen die Berechenbarkeit der zahlentheoretischen Funktionen aus Abschnitt 2.1.4. In Abbildung

2.6 wird nachgewiesen, dass die Funktionen S , $C_m^{(n)}$, $P_i^{(n)}$ und A LOOP-berechenbar sind. (Man benötigt dazu keine Schleifen.) Die partielle Funktion $O^{(n)}$ ist nicht LOOP-berechenbar, wird aber durch das nie terminierende WHILE-Programm

input $x_1, \dots, x_n : \mathbb{N}$; **output** $z : \mathbb{N}$; WHILE true DO skip ENDWHILE

berechnet.

S : input x_1 ; output z ; $z := x_1 + 1$	$C_m^{(n)}$: input x_1, \dots, x_n ; output z ; $z := m$
$P_i^{(n)}$: input x_1, \dots, x_n ; output z ; $z := x_i$	A : input x_1, x_2, x_3, x_4 ; output z ; IF $x_1 = x_2$ THEN $z := x_3$ ELSE $z := x_4$ ENDIF

Abbildung 2.6: LOOP-Berechenbarkeit der Grundfunktionen S , $C_m^{(n)}$, $P_i^{(n)}$ und A

Beispiel:

Einfache totale zahlentheoretische Funktionen wie Addition, (nichtnegative) Subtraktion und Multiplikation, sowie Vergleiche zwischen Zahlen sind LOOP-berechenbar. Dies beweisen wir hier jedoch nicht streng, sondern geben nur zwei Beispiele:

```

input  $x_1, x_2$ ;
output  $z$ ;
 $z := x_1$ ;
LOOP  $x_2$  DO    $z := z + 1$ ; ENDLLOOP

```

Dieses Programm berechnet die Summe $x_1 + x_2$.

```

input  $x$ ;
output  $z$  (init 0);
var  $a : \mathbb{N}$  (init 0);
LOOP  $x$  DO   IF  $a = 1$  THEN  $z := z + 1$  ENDIF;
             IF  $a = 0$  THEN  $a := 1$  ENDIF
ENDLOOP

```

Dieses Programm berechnet den Ausdruck $x \dot{-} 1$, der definiert ist als **if** $x=0 \rightarrow 0$ **fi** $x > 0 \rightarrow x-1$ **fi** (hier verwenden wir eine offensichtliche und mit der **if**-Kommandoschreibweise konsistente Schreibweise für bedingte Ausdrücke).

Ende des Beispiels

2.5 Codierungen

Oft möchte man die Wörter einer Sprache über einem Alphabet in Wörter über einem anderen Alphabet umcodieren. Ein Beispiel ist die Zahldarstellung in verschiedenen Zahlssystemen. Die Zahl Neunzehn wird

zum Beispiel im Zehnersystem (d.h. als Wort über dem Alphabet $\{0, 1, \dots, 9\}$) als 19, im Zweiersystem (d.h. als Wort über $\{0, 1\}$) als 10011 dargestellt. Die Übersetzung von (Wörtern über) einem Alphabet in (Wörter über) einem anderen Alphabet wird auch in der Kryptographie gebraucht, zum Beispiel zur Übermittlung „geheimer Botschaften“:

$$\begin{aligned} \text{ich} \sqcup \text{liebe} \sqcup \text{dich} & \quad (\text{Wort über } \{a, \dots, z\} \cup \{\sqcup\}) \\ \rightsquigarrow & \\ 0903080012090502050004090308 & \quad (\text{Wort über } \{0, \dots, 9\}). \end{aligned}$$

Formal werden solche Übersetzungen durch Funktionen von Σ^* nach Γ^* beschrieben, wobei Σ und Γ die beiden Alphabete sind.

Eine wichtige Eigenschaft von Codierungen ist die effektive Umrechenbarkeit. Beispielsweise werden in Vorlesungen der „Technischen Informatik“ effiziente Algorithmen vorgestellt, um eine im Zehnersystem dargestellte Zahl ins Zweiersystem und umgekehrt umzuformen. In der Vorlesung „Kryptographie“ werden Algorithmen besprochen, die der Ver- und Entschlüsselung von Texten oder anderen Daten dienen.

Beispiel:

Wir besuchen den Beweis des Lemmas 2.1.5, in dem die Abzählbarkeit der Menge $\mathbb{N} \times \mathbb{N}$ festgestellt wurde, und fassen ihn algorithmisch. Die dort angegebene Bijektion zwischen $\mathbb{N} \times \mathbb{N}$ und \mathbb{N} kann – in beiden Richtungen – algorithmisch berechnet und somit als eine effektive Codierung von Paaren natürlicher Zahlen als natürliche Zahlen betrachtet werden. Um dies zu sehen, definieren wir zwei Algorithmen A_1 und A_2 :

$$\begin{aligned} A_1 : & \quad \mathbf{input} \ k, m \in \mathbb{N}; \\ & \quad \mathbf{output} \ n \in \mathbb{N}; \\ & \quad n := k \cdot (k+3)/2 + k \cdot m + m \cdot (m+1)/2. \end{aligned}$$

Es ist nicht allzu schwer zu erkennen, dass A_1 mit einem Paar (k, m) als Eingabe diejenige natürliche Zahl n produziert, die dem Schema im Beweis von Lemma 2.1.5 entspricht. Nicht so leicht zu sehen ist, dass der folgende Algorithmus die Umkehrformel berechnet:

$$\begin{aligned} A_2 : & \quad \mathbf{input} \ n \in \mathbb{N}; \\ & \quad \mathbf{output} \ (k, m) \in (\mathbb{N} \times \mathbb{N}) \ (\mathbf{init} \ (0, 0)); \\ & \quad \mathbf{var} \ i, s : \mathbb{N} \ (\mathbf{init} \ 0); \\ & \quad \mathbf{do} \ i \neq n \rightarrow \mathbf{if} \ (k, m) = (s, 0) \rightarrow s := s+1; (k, m) := (0, s); \\ & \quad \quad \square \ (k, m) \neq (s, 0) \rightarrow (k, m) := (k+1, m-1) \\ & \quad \quad \mathbf{fi}; \\ & \quad \quad i := i+1 \\ & \quad \mathbf{od} \end{aligned}$$

A_1 und A_2 liefern also eine umkehrbar effektive (sogar bijektive), Codierung von \mathbb{N} nach $\mathbb{N} \times \mathbb{N}$.

Ende des Beispiels

Wir betrachten einige weitere Beispiele.

Beispiel $\mathbb{N} \rightsquigarrow \{\}\^*$:

Wir wollen die natürlichen Zahlen \mathbb{N} in das einelementige „Strich-“ oder „Bierdeckel-“Alphabet $\{\}$ umrechnen. Es gibt eine auf der Hand liegende Bijektion zwischen \mathbb{N} und $\{\}\^*$: 0 wird als ε codiert, 1 als

|, 2 als || usw. Die Codierung von natürlichen Zahlen durch die entsprechende Anzahl von Strichen ist effektiv, weil man Algorithmen finden kann, die sie bewerkstelligen: links die Umrechnung von n nach $|\dots|$ (n Striche) und rechts die Umkehrung davon.

input $n : \mathbb{N}$; output $w : \{\}\^*$ (init ε); LOOP n DO $w := w $ ENDLLOOP	input $w : \{\}\^*$; output $n : \mathbb{N}$ (init 0); LOOP $ w $ DO $n := n + 1$ ENDLLOOP
---	---

Wir bezeichnen mit $un(n)$ die Codierung von $n \in \mathbb{N}$ als n Striche, und wenn ein Wort $w \in \{\}\^*$ gegeben ist, bezeichnen wir mit $nat(w)$ die entsprechende Zahl in \mathbb{N} .

Anmerkung:

Manchmal findet man auch eine andere Codierung von \mathbb{N} in $\{\}\^*$: 0 wird durch | dargestellt, 1 durch || usw., d.h., n durch $n+1$ Striche („Bierdeckelsystem mit einem Bier zu viel“). Diese Darstellung hat fast die gleichen Eigenschaften wie die vorige, mit einem Unterschied: sie ist in einer der beiden Richtungen nicht surjektiv. Denn das leere Wort entspricht keiner natürlichen Zahl. Die fehlende Surjektivität ist oft nicht wirklich ein Problem, solange man für Elemente algorithmisch bestimmen kann, ob sie Code-Elemente sind oder nicht. Hier ist eine solche Bestimmung sehr leicht möglich; dazu genügt die Abfrage, ob ein Wort leer ist oder nicht.

Ende der Anmerkung

Ende des Beispiels $\mathbb{N} \rightsquigarrow \{\}\^*$

Beispiel $\mathbb{N} \rightsquigarrow \{0, 1\}^*$:

Auch für die Codierung von natürlichen Zahlen in Wörter über dem Binäralphabet $\{0, 1\}$ gibt es unterschiedliche (effektive) Möglichkeiten. Vielleicht am bekanntesten ist die dyadische Zahldarstellung:

$$0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 10, 3 \mapsto 11, 4 \mapsto 100, \text{ usw.}$$

Diese Codierung hat (wie die Bierdeckeldarstellung mit einem zusätzlichen Strich) den Nachteil, in einer Richtung nicht surjektiv zu sein, denn für das leere Wort und für Wörter mit führenden Nullen (außer der 0) gibt es keine Zahl n , die darauf abgebildet wird. Trotzdem ist sie effektiv in folgendem Sinn: für beide Übersetzungsrichtungen gibt es Algorithmen, und zusätzlich gibt es einen Algorithmus, der für Wörter in $\{0, 1\}^*$ bestimmt, ob diese Bilder der Übersetzung sind oder nicht. Wir nennen $bin(n)$ diese binäre Darstellung von n , und für Wörter w im Bildbereich bezeichnen wir mit $nat(n)$ diejenige Zahl, die auf sie abgebildet wird.

Eine andere Möglichkeit, natürliche Zahlen binär zu codieren, orientiert sich am Abzählbarkeitsbeweis für $\{0, 1\}^*$. Wir ordnen die Wörter in $\{0, 1\}^*$ primär ihrer Länge nach und sekundär lexikografisch, also folgendermaßen:

$$\{\varepsilon, 0, 1, 00, 01, 10, \dots\},$$

und konstruieren dadurch eine Bijektion von \mathbb{N} in die Menge $\{0, 1\}^*$. Dies hat der dyadischen Zahldarstellung gegenüber den Vorteil, bijektiv zu sein (und außerdem in beiden Richtungen effektiv), aber den Nachteil, dass die Umrechnungsalgorithmen etwas komplizierter und gewöhnungsbedürftiger sind.

Ende des Beispiels $\mathbb{N} \rightsquigarrow \{0, 1\}^*$

Anmerkung:

Beide Codierungen lassen sich sehr leicht auf Alphabete Σ mit $|\Sigma| = p > 2$ übertragen. Einerseits kann $n \in \mathbb{N}$ in Σ p -adisch codiert werden, indem die Elemente von Σ als Ziffern aufgefasst werden.

Diese Darstellung hat die gleichen Eigenschaften bezüglich Effektivität und Nicht-Surjektivität wie die dyadische. Andererseits können die Wörter von Σ^* der Länge nach systematisch aufgeschrieben werden und den Zahlen aus \mathbb{N} umkehrbar effektiv und sogar bijektiv zugeordnet werden.

Ende der Anmerkung

Beispiel $\Sigma_1^* \times \Sigma_2^* \rightsquigarrow \mathbb{N}$:

Es seien Σ_1 und Σ_2 zwei beliebige Alphabete. Wir wollen Tupel in $\Sigma_1^* \times \Sigma_2^*$ in die natürlichen Zahlen umrechnen. Dazu können wir zunächst Σ_1^* effektiv umkehrbar nach \mathbb{N} , dann Σ_2^* effektiv umkehrbar nach \mathbb{N} codieren und anschließend die effektiv umkehrbare Codierung von $\mathbb{N} \times \mathbb{N}$ nach \mathbb{N} verwenden. Diese Codierungen können leicht auch bijektiv definiert werden. Ganz genau so codiert man $\Sigma_1^* \times \Sigma_2^* \times \Sigma_3^*$ in \mathbb{N} und allgemeiner, beliebige (Kartesische Produkte von) Mengen der Art Σ^* oder \mathbb{N} in beliebige andere (Kartesische Produkte von) Mengen dieser Art.

Ende des Beispiels $\Sigma_1^* \times \Sigma_2^* \rightsquigarrow \mathbb{N}$

Allgemein sehen wir eine Codierung als eine Relation $\rho \subseteq X \times Y$ zwischen zwei abzählbaren Mengen X und Y an. Wir nennen eine solche Codierung *umkehrbar effektiv*, wenn es jeweils einen Algorithmus gibt, der:

- für $x \in X$ entscheidet, ob $x \notin \text{dom}(\rho)$ oder $x \in \text{dom}(\rho)$;
- für $y \in Y$ entscheidet, ob $y \notin \text{cod}(\rho)$ oder $y \in \text{cod}(\rho)$;
- für $x \in \text{dom}(\rho)$ ein Bild $y \in \rho(x)$ berechnet;
- für $y \in \text{cod}(\rho)$ ein Urbild $x \in \rho^{-1}(y)$ berechnet.

Je nach Bedarf kommen noch weitere Eigenschaften hinzu, vor allem manchmal die Injektivität oder die Surjektivität oder beides. Wie wir weiter oben gesehen haben, gibt es zwischen den Grundmengen, die uns im weiteren Verlauf interessieren werden, nämlich den (Kreuzprodukten) von Mengen der Art Σ^* und \mathbb{N} , immer umkehrbar effektive (sogar bijektive) Codierungen.

2.6 Übungsaufgaben

1. Man zeige: wenn X und Y endliche Mengen sind, dann gibt es genau $|X| \cdot |Y|$ Paare in $X \times Y$, $|X|^n$ Elemente in X^n , $|Y|^{|X|}$ Funktionen von X nach Y und $2^{|X| \cdot |Y|}$ Relationen über X und Y .
2. Seien $\rho \subseteq (X \times Y)$ und $\tau \subseteq (Y \times Z)$ zwei Relationen. Man zeige $(\rho \circ \tau)^{-1} = (\tau^{-1} \circ \rho^{-1})$.
3. Man finde eine Menge X und eine irreflexive Relation ρ auf $X \times X$ derart, dass ρ^+ reflexiv ist.
4. Sei $\rho \subseteq (X \times X)$ eine Relation auf X . Man zeige $(\rho^{-1})^* = (\rho^*)^{-1}$.
5. Man zeige, dass $f: X \rightarrow Y$ bijektiv genau dann ist, wenn sowohl f als auch f^{-1} Funktionen sind; und dann ist auch f^{-1} bijektiv.
6. Seien ρ_1 und ρ_2 zwei Äquivalenzrelationen mit endlichem Index. Man zeige, dass aus $\rho_1 \subseteq \rho_2$ folgt, dass der Index von ρ_2 kleiner oder gleich dem Index von ρ_1 ist.
7. Sei X eine Menge mit $|X| = m \in \mathbb{N}$, d.h. X enthalte m Elemente. Im folgenden wird nach der Anzahl von speziellen Relationen auf X gefragt, womit die *maximale* Anzahl von *voneinander verschiedenen, zweistelligen* Relationen gemeint ist.

- a) Wieviele reflexive Relationen gibt es auf X ?
 b) Wieviele symmetrische Relationen gibt es auf X ?
 c) Eine reflexive und transitive Relation \preceq auf X beschreibt eine totale Ordnung, wenn

$$\forall a, b \in M: (a \preceq b) \vee (b \preceq a)$$

gilt. Wieviele Relationen gibt es auf X , die eine totale Ordnung beschreiben?

8. a) Geben Sie eine Menge X zusammen mit einer irreflexiven Relation $\rho \subseteq X \times X$ an, so dass ρ^+ reflexiv ist. Geben Sie auch ρ^+ an.
 b) Gegeben seien zwei Mengen X und Y , sowie zwei Funktionen $f : X \rightarrow Y$ und $g : Y \rightarrow X$. Zeigen Sie: Ist $f \circ g = id|_X$ und $g \circ f = id|_Y$, dann sind f und g bijektiv.
9. Gegeben seien zwei gleichmächtige Mengen $X = \{1, 2, 3\}$ und $Y = \{4, 5, 6\}$, sowie zwei offensichtlich injektive Funktionen

$$f : \begin{cases} X & \rightarrow & Y \\ x & \mapsto & x + 3 \end{cases} \quad \text{und} \quad g : \begin{cases} Y & \rightarrow & X \\ y & \mapsto & \begin{cases} 1 & , \text{ falls } y = 4 \\ 3 & , \text{ falls } y = 5 \\ 2 & , \text{ falls } y = 6. \end{cases} \end{cases}$$

- a) Geben Sie für jedes $A \in 2^X$ den Wert von $h(A) = X \setminus g(Y \setminus f(A))$ an.
 b) Berechnen Sie $H = \bigcap \{A \in 2^X \mid h(A) \subseteq A\}$.
 c) Kann im allgemeinen Fall *immer* etwas über die wie oben definierte Menge $H \subseteq X$ ausgesagt werden, wenn die gleichmächtigen Mengen X und Y *endlich* sind? Beweisen Sie Ihre Antwort!
10. Sei Σ ein Alphabet. Man zeige $\bigcup_{n \in (\mathbb{N} \setminus \{0\})} \Sigma^n = \Sigma^* \setminus \{\varepsilon\}$. Man gebe eine Sprache $L \subseteq \Sigma^*$ an, so dass $\bigcup_{n \in (\mathbb{N} \setminus \{0\})} L^n \neq L^* \setminus \{\varepsilon\}$.
11. Ist die Menge aller Sprachen über einem gegebenen Alphabet abzählbar oder überabzählbar unendlich? (Begründung oder Beweis.)
12. Man zeige: Sei X eine unendliche Menge; dann ist 2^X nicht abzählbar.
13. Man zeige: „Die abzählbare Vereinigung abzählbarer Mengen ist wieder abzählbar“, d.h.: Sei I eine abzählbare Menge und seien, für jedes $i \in I$, die X_i abzählbare Mengen. Dann ist auch die Vereinigung $X = \bigcup_{i \in I} X_i$ eine abzählbare Menge.
14. Man zeige: χ_L^+ ist überall undefiniert genau dann, wenn $L = \emptyset$, und überall definiert – also eine Funktion – genau dann, wenn $L = \Sigma^*$.
15. Man zeige: Ist \mathcal{L} abgeschlossen gegenüber Komplement und Vereinigung (Durchschnitt), dann auch gegenüber Durchschnitt (bzw. Vereinigung).
16. Gegeben sei ein Alphabet Σ , eine Sprache $L \subseteq \Sigma^*$ und ein Wort $w \in L$. Beweisen Sie durch vollständige Induktion, dass für jedes $n \in \mathbb{N}$

$$|w^n| = n * |w| \quad \wedge \quad |\Sigma^n| = |\Sigma|^n.$$

gilt. Ein Hinweis zur Arbeitserleichterung: $\Sigma \subseteq \Sigma^*$.

Kapitel 3

Ersetzungssysteme und Grammatiken

Sprachen über Σ können dadurch erzeugt werden, dass die Wörter in ihnen schrittweise durch sogenannte *Ersetzungsregeln*, die „mechanisch“ angewendet werden können, produziert werden. Zuerst gehen wir allgemein auf Ersetzungsregeln ein (Abschnitt 3.1) und dann speziell auf grammatikalische Regeln (Abschnitt 3.2).

3.1 Ersetzungssysteme

Definition 3.1.1 ERSETZUNGSSYSTEM

Sei Σ ein Alphabet. Ein *Ersetzungssystem* oder *Semi-Thue-System* (nach dem norwegischen Mathematiker A. Thue) über Σ ist ein Paar $E = (\Sigma, P)$, wobei $P \subseteq \Sigma^* \times \Sigma^*$ eine endliche Relation ist. Die Elemente von P heißen auch *Produktionen* oder (*Ersetzungs-*)*Regeln* (engl. *rewrite rules*). ☒ 3.1.1

Der Vorsatz „Semi-“ in dieser Definition ist so zu verstehen, dass zu einem Thue-System (die von Thue in erster Linie betrachteten Struktur) noch eine weitere Eigenschaft hinzu kommt: die Relation P muss symmetrisch sein. Wir interessieren uns jedoch hauptsächlich für Semi-Thue-Systeme.

Regeln $(u, v) \in P$ werden oft $u \rightarrow v$ geschrieben, um anzuzeigen, dass u der Ursprung ist, aus dem v „in einem Schritt“ produziert wird. Die Arbeitsweise eines Ersetzungssystems $E = (\Sigma, P)$ wird durch folgenden Ableitungsbegriff beschrieben.

Definition 3.1.2 ABLEITBARKEIT

E induziert eine Relation $\vdash_E \subseteq \Sigma^* \times \Sigma^*$, die folgendermaßen definiert ist: für $w, v \in \Sigma^*$ gilt $w \vdash_E v$ (v ist *unmittelbar* aus w *ableitbar*), falls es Wörter $w_1, w_2 \in \Sigma^*$ und eine Regel $u_1 \rightarrow u_2 \in P$ gibt, so dass $w = w_1 u_1 w_2$ und $v = w_1 u_2 w_2$. Die Regel $u_1 \rightarrow u_2$ wird also auf das Wort w angewandt, indem eines der (eventuell mehreren) Vorkommen des Teilwortes u_1 durch u_2 ersetzt wird. Statt $w \vdash_E v$ wird auch $w \rightarrow_E v$ oder (wenn E aus dem Kontext bekannt ist) auch $w \vdash v$ bzw. $w \rightarrow v$ geschrieben. ☒ 3.1.2

Gilt $w \vdash^* v$, d.h. $(w, v) \in \vdash^*$, dann gibt es eine Folge $w = z_0 \vdash \dots \vdash z_n = v$. Diese Folge heißt auch eine *Ableitung* von v aus w (oder von w nach v) der *Länge* n . Insbesondere gilt $w \vdash^* w$ mit einer Ableitung der Länge 0.

Wir definieren zu einem gegebenen Wort $w \in \Sigma^*$ die Menge aller daraus mit Hilfe von E ableitbaren Wörter, die Sprache $L(w, E)$ (L für „language“):

$$L(w, E) = \{v \in \Sigma^* \mid w \vdash_E^* v\}.$$

Die Menge der Ableitungen aus w kann in baumförmiger Art beschrieben werden. Die Wurzel des Baumes ist w und die Kinder eines Knoten des Baumes sind die daraus mittels \vdash_E unmittelbar ableitbaren Wörter (die eventuell mehrfach in einem solchen Baum vorkommen). Dieser Baum hat endlichen Ausgangsgrad, denn wegen der Endlichkeit von P ist für jedes Wort $w \in \Sigma^*$ die Menge der Kinder von w bezüglich der unmittelbaren Ableitbarkeit \vdash_E , d.h. $\{v \in \Sigma^* \mid w \vdash_E v\}$, endlich. Wir nennen ihn den *Baum der Ableitungen* (von w aus, oder mit Wurzel w).

Besonders wichtig sind die Blätter dieses Baumes. Seien $w, z \in \Sigma^*$. Dann heißt z *terminal* (oder auch *in Normalform*), wenn es kein $v \in \Sigma^*$ mit $z \vdash_E v$ gibt. Mit anderen Worten, bei z endet die unmittelbare Ableitbarkeit. Wir sagen, dass w *die Normalform* z *hat*, wenn z in Normalform ist und $w \vdash_E^* z$ gilt. Es kann zu einem w mehrere verschiedene Normalformen geben.

Es folgen einige Beispiele. Alle benutzen das Alphabet $\Sigma = \{\sqcup, a, b, \dots, z\}$, wobei \sqcup das Leerzeichen bedeuten soll.

Beispiel E_1 :

Zuerst betrachten wir $E_1 = (\Sigma, \{(hasse, liebe)\})$. Die maximal langen Ableitungen in diesem Ersetzungssystem wandeln einen Eingabetext so um, dass alle Vorkommen von „hasse“ durch „liebe“ ersetzt werden, z.B. in:

$$\begin{aligned} &ich \sqcup hasse \sqcup gti \vdash_{E_1} ich \sqcup liebe \sqcup gti \\ &\text{und} \\ &ich \sqcup hasse \sqcup gti \sqcup und \sqcup ich \sqcup hasse \sqcup logik \vdash_{E_1}^* ich \sqcup liebe \sqcup gti \sqcup und \sqcup ich \sqcup liebe \sqcup logik. \end{aligned}$$

Ende des Beispiels E_1

Beispiel E_2 :

Sei $E_2 = (\Sigma, \{(a, aba)\})$. Im Gegensatz zu E_1 hat dieses Ersetzungssystem unendliche Ableitungen, z.B.

$$cac \vdash_{E_2} cabac \vdash_{E_2} cababac \dots$$

Im zweiten Schritt dieser Ableitung können beide a ersetzt werden, was (zufälligerweise) auf das gleiche Wort führt. Dies ändert sich bei einem anderen Anfangswort, wie zum Beispiel $w = caac$. Wenn immer das erste a ersetzt wird, ergibt sich die Ableitung

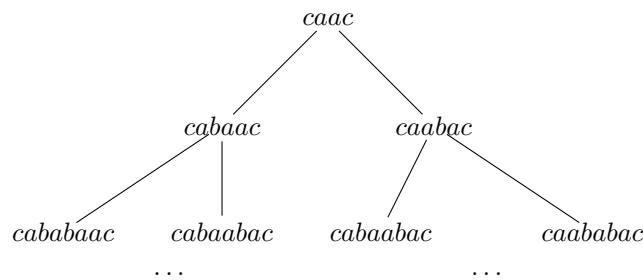
$$caac \vdash_{E_2} cabaac \vdash_{E_2} cababaac \dots,$$

wenn immer das letzte a ersetzt wird, die Ableitung

$$caac \vdash_{E_2} caabac \vdash_{E_2} caababac \dots$$

Dazwischen gibt es unendlich viele weitere Ableitungen. Die Sprachen $L(cac, E_2)$ und $L(caac, E_2)$ sind beide unendlich. In Abbildung 3.1 ist der Baum der Ableitungen von $L(caac, E_2)$ gezeigt bzw. angedeutet.

Ende des Beispiels E_2

Abbildung 3.1: Baum von E_2 mit Wurzel $caac$ **Beispiel E_3 :**

Wir betrachten $E_3 = (\Sigma, \{(a, b), (a, c)\})$. Hier tritt der Fall auf, dass ein Wort keine eindeutige Normalform hat. Zum Beispiel gilt

$$aa \vdash_{E_3}^* bb \text{ und } aa \vdash_{E_3}^* bc \text{ und } aa \vdash_{E_3}^* cb \text{ und } aa \vdash_{E_3}^* cc,$$

und alle vier erreichten Wörter sind terminal.

Ende des Beispiels E_3 **Beispiel E_4 :**

Es sei $E_4 = (\Sigma, \{(ba, ab)\})$. Eine maximal lange Ableitung von $baacba$ aus ist

$$baacba \vdash_{E_4} baacab \vdash_{E_4} abacab \vdash_{E_4} aabcab.$$

Es gibt auch noch andere, gleich lange, Ableitungen vom gleichen Wort $baacab$, allerdings nur endlich viele. Die Sprache $L(baacba, E_4)$ ist also endlich.

Ende des Beispiels E_4

Als nächste Beispiele betrachten wir zwei Spezialfälle, in denen es (wie bei E_3) nur auf die Ersetzung von Buchstaben ankommt.

Beispiel E_5 :

Sei $E_5 = (\Sigma, \{(a, b), (a, c), (c, d), (b, e), (d, e)\})$. Eine Ableitung ist zum Beispiel

$$a \vdash_{E_5} c \vdash_{E_5} d \vdash_{E_5} e.$$

E_5 hat ersichtlich keine unendlichen Ableitungen, auch nicht, wenn mit mehrbuchstabigen Wörtern gestartet wird.

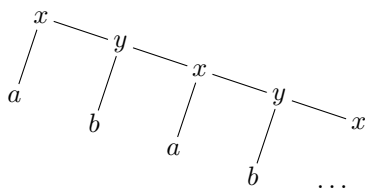
Ende des Beispiels E_5 **Beispiel E_6 :**

Sei zuletzt $E_6 = (\Sigma, \{(x, y), (y, x), (x, a), (y, b)\})$ betrachtet. E_6 hat unendliche Ableitungen, zum Beispiel

$$x \vdash_{E_6} y \vdash_{E_6} x \vdash_{E_6} y \vdash_{E_6} \dots,$$

aber auch maximale endliche, z.B.

$$x \vdash_{E_6} y \vdash_{E_6} x \vdash_{E_6} y \vdash_{E_6} b.$$

Abbildung 3.2: Baum von E_6 mit Wurzel x

Trotz der existierenden unendlichen Ableitung ist die Sprache $L(x, E_6)$ endlich. Der Baum der Ableitungen von E_6 mit x als Wurzel hat unendlich viele Knoten, aber nur endlich viele Wörter kommen in ihm vor. Der Baum mit Wurzel x ist in Abbildung 3.2 gezeigt.

Ende des Beispiels E_6

Wir untersuchen nun die Frage der Existenz von eindeutigen Normalformen für die Wörter eines gegebenen Ersetzungssystems. Um Kriterien hierfür zu finden, und zwecks einer Untersuchung der entstehenden Spezialfälle, definieren wir einige Eigenschaften von Ersetzungssystemen.

Definition 3.1.3 SPEZIELLE ERSETZUNGSSYSTEME

Sei $E = (\Sigma, P)$ ein Ersetzungssystem.

- (i) E heißt *Noethersch* (nach der Mathematikerin Emmy Noether) oder *terminierend*, wenn es keine unendlichen Ableitungen

$$w_0 \vdash_E w_1 \vdash_E w_2 \vdash_E \dots$$

gibt.

- (ii) E heißt *deterministisch*, wenn die Relation \vdash_E rechtseindeutig ist.

- (iii) E heißt *stark konfluent*, falls

aus:  stets folgt: $\exists v \in \Sigma^*$: 

- (iv) E heißt *konfluent* oder *besitzt die Church-Rosser-Eigenschaft* oder kurz: *ist Church-Rosser*, falls

aus:  stets folgt: $\exists v \in \Sigma^*$: 

(v) E heißt *lokal konfluent*, falls



Die drei zuletzt genannten Eigenschaften können auch relational formuliert werden: E ist

(iii) *stark konfluent*, wenn $(\vdash^{-1} \circ \vdash) \setminus id_{\Sigma^*} \subseteq (\vdash \circ \vdash^{-1})$;

(iv) *konfluent*, wenn $((\vdash^{-1})^* \circ \vdash^*) \subseteq (\vdash^* \circ (\vdash^{-1})^*)$;

(v) *lokal konfluent*, wenn $(\vdash^{-1} \circ \vdash) \subseteq (\vdash^* \circ (\vdash^{-1})^*)$.

□ 3.1.3

Wenn ein Ersetzungssystem E über Σ Noethersch ist, dann ist für jedes Wort $w \in \Sigma^*$ die Sprache $L(w, E)$ endlich. Dies folgt aus der Gradendlichkeit des mit w beginnenden Baums der Ableitungen, zusammen mit Lemma 2.2.1 (dem Lemma von König). Die Sprache $L(w, E)$ kann höchstens so viele Elemente haben wie dieser Baum Knoten hat. Insbesondere gibt es für ein Noethersches Ersetzungssystem auch immer eine maximale Länge für Ableitungen aus einem Wort w , nämlich die Länge eines maximalen Pfades in dem Baum der Ableitungen von w aus.

Determinismus bedeutet, dass es keine Auswahl zwischen zwei oder mehr verschiedenen (unmittelbaren) Ableitungsschritten geben kann. D.h., wenn das Ergebnis eines Ableitungsschrittes $w \vdash_E \dots$ überhaupt existiert, dann ist es bestimmt durch das eindeutige v mit $w \vdash_E v$.

Die vorigen Beispiele sind so gewählt, dass sie die teilweise Unterschiedlichkeit der soeben definierten Begriffe, vor allem der Konfluenzbegriffe, zeigen. Zum Beispiel sind E_2 und E_6 nicht Noethersch, aber alle anderen sind Noethersch; wir werden dies gleich ausführlich für E_4 zeigen. E_3 besitzt keine der drei Konfluenzeigenschaften, d.h., es ist nicht stark konfluent, nicht lokal konfluent und auch nicht konfluent. Für alle drei Eigenschaften sind die beiden Ableitungen $a \vdash_{E_3} b$ und $a \vdash_{E_3} c$ ein Gegenbeispiel, denn von b und c aus gibt es kein gemeinsames ableitbares Wort. E_4 besitzt dagegen alle drei Konfluenzeigenschaften; wir zeigen dies gleich für die starke Konfluenz. E_5 ist lokal konfluent und konfluent, aber nicht stark konfluent. E_6 ist lokal konfluent, aber nicht konfluent. Dass diese Beispiele maximal diskriminierend zwischen den betrachteten Konfluenzeigenschaften sind, folgt aus einem gleich zu beweisenden Satz.

Lemma 3.1.4 *Das Ersetzungssystem E_4 ist Noethersch und stark konfluent.*

Beweis: Wir zeigen zuerst, dass E_4 Noethersch ist. Die Idee: bei jedem Schritt wandert eines der b 's im Wort nach rechts, bis kein b mehr wandern kann; die Wortlänge ändert sich aber nicht. Genauer: man definiere eine Funktion $\tau(w)$, die über alle b im Wort w die Anzahl der rechts von b stehenden a -Zeichen summiert. Z.B. ist $\tau(baacba) = 4$, weil rechts des ersten b drei a s stehen und rechts des zweiten b ein a steht. Offensichtlich gibt es keine Ableitungen von w , die länger als $\tau(w)$ sind.

Wir zeigen dann, dass E_4 stark konfluent ist. Gilt $w \vdash_{E_4} w_1$ und $w \vdash_{E_4} w_2$ mit $w_1 \neq w_2$, dann muss es in w zwei Vorkommen des Teilwortes ba geben. Da es in jedem Wort der Länge ≤ 3 höchstens ein

solches Vorkommen geben kann, können sich die beiden Vorkommen von ba in w nicht überlappen¹ und w ist von der Form $w'ba w''ba w'''$ mit Wörtern w', w'', w''' . Deswegen verbleibt nach Ersetzung des einen Vorkommens von ba durch ab das andere Vorkommen noch im Wort und kann weiter ersetzt werden. ☒ 3.1.4

Der folgende Satz gibt einen ersten Zusammenhang zwischen den drei Konfluenzeigenschaften:

Satz 3.1.5 ERSTER KONFLUENZSATZ

Wenn ein Ersetzungssystem stark konfluent ist, dann ist es Church-Rosser.

Wenn ein Ersetzungssystem Church-Rosser ist, dann ist es lokal konfluent.

Die drei Konfluenzbegriffe bilden also eine lineare Hierarchie.

Beweis: Sei E stark konfluent. Sei w ein Wort mit $w \vdash_E^* w_1$ und $w \vdash_E^* w_2$. Dann gibt es Zahlen n, m und Wörter w_1^0, \dots, w_1^n sowie w_2^0, \dots, w_2^m mit $w_1^0 = w = w_2^0$, $w_1^n = w_1$, $w_2^m = w_2$ sowie $w_1^{i-1} \vdash_E w_1^i$ ($0 < i \leq n$) und $w_2^{j-1} \vdash_E w_2^j$ ($0 < j \leq m$). Die dadurch aufgespannte „große Raute“ kann nach und nach mit „kleinen Rauten“ gemäß starker Konfluenz aufgefüllt werden, bis ein Element entsteht, das sowohl aus w_1 als auch aus w_2 ableitbar ist (siehe Abbildung 3.3).

Dass die Church-Rosser-Eigenschaft lokale Konfluenz impliziert, ist unmittelbar klar. ☒ 3.1.5

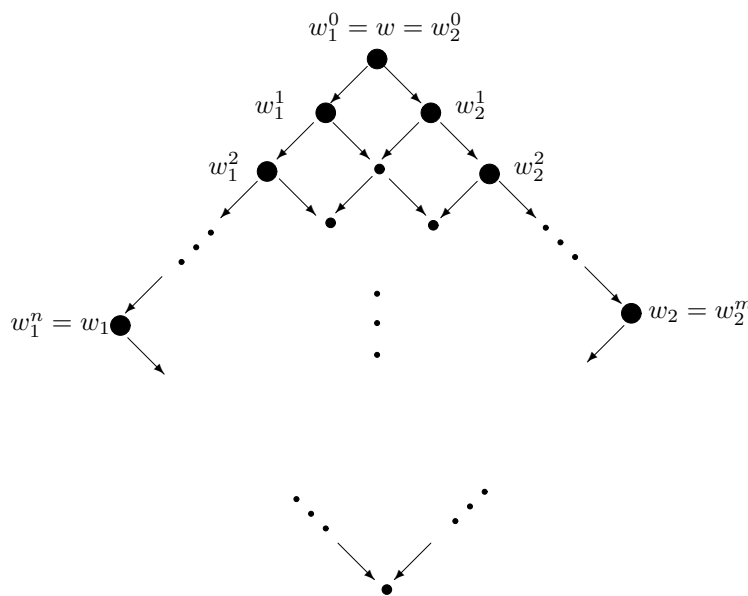


Abbildung 3.3: Zum Beweis von Satz 3.1.5

Die Implikationen des letzten Satzes können nicht umgedreht werden, wie aus den vorangegangenen Beispielen E_5 und E_6 folgt. Lokale Konfluenz ist allerdings stark genug, die Church-Rosser-Eigenschaft nach sich zu ziehen, wenn nur Terminierung vorausgesetzt wird:

¹Ein Beispiel für Teilwörter, die sich überlappen können, ist $abc bca$ mit zwei Teilwörtern, $bc b$ und nochmals $bc b$.

Satz 3.1.6 ZWEITER KONFLUENZSATZ / SATZ VON NEWMANN

Sei E ein Noethersches und lokal konfluentes Ersetzungssystem. Dann ist E Church-Rosser.

Demnach dürfte E_6 , das den Unterschied zwischen Konfluenz und lokaler Konfluenz aufzeigt, nicht Noethersch sein. In der Tat gibt es die gezeigte unendliche Ableitung.

Beweis: Da E Noethersch ist, ist für jedes Wort w der Baum der Ableitungen mit w als Wurzel endlich. Es existiert also eine Funktion $l: \Sigma^* \rightarrow \mathbb{N}$, die $w \in \Sigma^*$ die Länge einer längsten von w ausgehenden Ableitung zuordnet. Wir beweisen den Satz per Induktion über $l(w)$.

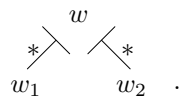
- **Induktionsbeginn:** $l(w) = 0$. In diesem Fall gibt es von w aus keine Ableitung, und die Konfluenz ist trivialerweise erfüllt.
- **Induktionsschritt:** $l(w) > 0$. Es gelte $w \vdash_E^* w_1$ und $w \vdash_E^* w_2$. Dann gibt es Zahlen n, m und Wörter $w_1^{(0)}, \dots, w_1^{(n)}$ sowie $w_2^{(0)}, \dots, w_2^{(m)}$ mit $w_1^{(0)} = w = w_2^{(0)}$, $w_1^{(n)} = w_1$, $w_2^{(m)} = w_2$ sowie $w_1^{(i-1)} \vdash_E w_1^{(i)}$ ($0 < i \leq n$) und $w_2^{(j-1)} \vdash_E w_2^{(j)}$ ($0 < j \leq m$). Wenn $n=0$ oder $m=0$, ist die Konfluenz erfüllt, es gelte also $n > 0$ und $m > 0$. Aufgrund der lokalen Konfluenz existiert ein Wort $t \in \Sigma^*$ mit $w_1^{(1)} \vdash_E^* t$ und $w_2^{(1)} \vdash_E^* t$. Da $w \vdash_E w_1^{(1)}$ und $w \vdash_E w_2^{(1)}$, gilt $l(w_1^{(1)}) \leq l(w) - 1$ und $l(w_2^{(1)}) \leq l(w) - 1$. Aus der Induktionsvoraussetzung folgt, dass es Wörter $u_1, u_2 \in \Sigma^*$ gibt, so dass $w_1 \vdash_E^* u_1$, $t \vdash_E^* u_1$, $t \vdash_E^* u_2$ und $w_2 \vdash_E^* u_2$ gilt. Da auch $l(t) \leq l(w) - 1$ gilt, kann wieder die Induktionsvoraussetzung angewendet werden, und danach existiert ein $v \in \Sigma^*$, so dass $u_1 \vdash_E^* v$ und $u_2 \vdash_E^* v$ und somit auch $w_1 \vdash_E^* v$ und $w_2 \vdash_E^* v$. □ 3.1.6

Dies führt zu einem Kriterium dafür, dass jedes Wort eines Ersetzungssystems eine eindeutige Normalform hat.

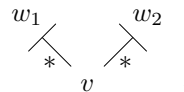
Satz 3.1.7 NORMALFORMSATZ

Sei $E = (\Sigma, P)$ ein Noethersches und lokal konfluentes Ersetzungssystem. Dann hat jedes Wort $w \in \Sigma^*$ eine eindeutige Normalform.

Beweis: Betrachte $w \in \Sigma^*$. Die Existenz einer Normalform von w folgt direkt daraus, dass E terminiert (Noethersch ist). Zu zeigen ist noch die Eindeutigkeit. Seien dazu w_1 und w_2 terminal mit



Da E laut Satz 3.1.6 die Church-Rosser-Eigenschaft hat, gibt es ein Wort v mit



Da w_1 und w_2 in Normalform sind, folgt $w_1 = v = w_2$. □ 3.1.7

3.2 Grammatiken

Formale Grammatiken werden im Folgenden als Ersetzungssysteme mit zusätzlicher Struktur angesehen. Diese zusätzliche Struktur ist geprägt durch die Interpretation von einigen Zeichen als sogenannte *Nichtterminalsymbole*, die nur zum Erzeugen von Wörtern benötigt werden und in diesen Wörtern selbst keine Rolle mehr spielen. Darunter ist ein spezielles Symbol, das *Startsymbol* S , von dem aus alle zu betrachtenden Wörter generiert werden. Die Buchstaben der zu erzeugenden Wörter heißen demgegenüber *Terminalsymbole*. Da wir an der Erzeugung von Sprachen über einem Alphabet Σ interessiert sind, verwenden wir Σ weiter für die Menge der Terminalsymbole. Für die Menge der Nichtterminalsymbole verwenden wir die Bezeichnung N . Das Gesamtalphabet des zu betrachtenden Ersetzungssystems ist damit $\Sigma \cup N$ (und nicht mehr nur Σ wie in Abschnitt 3.1).

Formale Grammatiken wurden ca. 1959 von dem Linguisten Noam Chomsky eingeführt und heißen deshalb auch Chomsky–Grammatiken. Die Unterscheidung zwischen Nichtterminalsymbolen und Terminalsymbolen hat eine direkte linguistische Entsprechung. Linguistische Begriffe wie „Prädikat“, „Subjekt“, „Objekt“ entsprechen Nichtterminalsymbolen, die Buchstaben bzw. Wörter des (deutschen, englischen, etc.) Alphabets bzw. Wörterbuchs dagegen den Terminalsymbolen.

Definition 3.2.1 FORMALE GRAMMATIKEN

Sei Σ ein Alphabet. Eine *formale Grammatik* über Σ ist ein Quadrupel $G = (N, \Sigma, P, S)$, wobei gilt:

- (i) N ist ein Alphabet von *Nichtterminalsymbolen*;
- (ii) Σ ist ein Alphabet von *Terminalsymbolen*, und es gilt $N \cap \Sigma = \emptyset$;
- (iii) $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ ist eine endliche Menge von *Produktionen* oder *Regeln*, wobei wir für $(u, v) \in P$ fordern, dass u mindestens ein Nichtterminalsymbol enthält;
- (iv) $S \in N$ ist das *Startsymbol*. ☒ 3.2.1

Ist $G = (N, \Sigma, P, S)$ eine Grammatik, so ist $E(G) = (N \cup \Sigma, P)$ ein Ersetzungssystem, das G zugeordnete *Ersetzungssystem*. Wir schreiben $u \rightarrow v$ für Regeln in P und \vdash_G, \rightarrow_G oder auch (um Ableitungen von den Regeln zu unterscheiden) \Rightarrow_G für den durch $E(G)$ bestimmten Ableitungsbegriff $\vdash_{E(G)}$ und $\vdash_G^*, \rightarrow_G^*$ oder \Rightarrow_G^* für die reflexive, transitive Hülle $\vdash_{E(G)}^*$ von $\vdash_{E(G)}$.

Die Nichtterminalsymbole werden normalerweise nur als Hilfssymbole innerhalb von Ableitungen benutzt. Durch die Forderung, dass in einer Regel $(u, v) \in P$ das Wort u mindestens ein Nichtterminalsymbol enthält, erreichen wir erstens, dass aus dem „Nichts“ (dem leeren Wort) nichts erzeugt werden kann und zweitens, dass ein einmal erzeugtes nur aus Terminalzeichen bestehendes Wort nicht weiter verändert werden kann. Die Rolle des Startsymbols S ist die des einzig interessierenden Startwortes des Ersetzungssystems $E(G)$. D.h., es interessieren überhaupt nur die Wörter in $L(S, E(G))$. Darunter sind einige besonders interessant:

Definition 3.2.2 DURCH EINE GRAMMATIK ERZEUGTE SPRACHE

Die *von G erzeugte Sprache*, $L(G)$, ist definiert als die Wortmenge

$$L(G) = \{w \in L(S, E(G)) \mid w \in \Sigma^*\},$$

d.h. diejenigen aus S ableitbaren Wörter, die nur aus Terminalsymbolen bestehen. ☒ 3.2.2

Beispiel G_1 : Wir betrachten die Grammatik $G_1 = (N_1, \Sigma_1, P_1, S_1)$ mit

$$\begin{aligned} N_1 &= \{S_1\} \\ \Sigma_1 &= \{a, b\} \\ P_1 &= \{S_1 \rightarrow \varepsilon, S_1 \rightarrow aS_1b\} \end{aligned}$$

und fragen, welche Wortmenge über Σ_1 dadurch generiert wird. In diesem einfachen Beispiel ist es leicht möglich, den ganzen Baum der Ableitungen mit Wurzel S_1 anzugeben (siehe Abbildung 3.4). Die Ableitungen nach links stammen von der Produktion $S_1 \rightarrow \varepsilon$, die nach rechts von der Produktion $S_1 \rightarrow aS_1b$. Diejenigen Wörter im Baum, die Elemente von Σ_1^* sind, sind genau seine Blätter, d.h. die Wörter ε , ab , $aabb$ etc. Die von G_1 erzeugte Sprache ist demnach $L(G_1) = \{a^n b^n \mid n \in \mathbb{N}\} = \{\varepsilon, ab, aabb, \dots\}$.

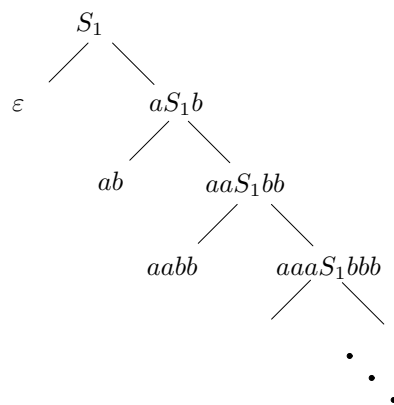


Abbildung 3.4: Baum der Ableitungen (von S_1 aus) für G_1

Ende des Beispiels G_1

Beispiel G_2 : Wir wollen Wörter der Form

$$w = ((()))(())$$

über dem Alphabet $\Sigma_2 = \{(\,)\}$ beschreiben, d.h. die Sprache der korrekten Klammerungen:

$$KL = \{w \in \Sigma^* \mid \text{Anzahl } (\text{ in } w = \text{Anzahl }) \text{ in } w \\ \text{und für alle Präfixe } u \text{ von } w \text{ gilt: Anzahl } (\text{ in } u \geq \text{Anzahl }) \text{ in } u \}.$$

Wir wählen die Grammatik $G_2 = (N_2, \Sigma_2, P_2, S_2)$ mit

$$\begin{aligned} N_2 &= \{S_2\} \\ \Sigma_2 &= \{(\,)\} \\ P_2 &= \{S_2 \rightarrow \varepsilon, S_2 \rightarrow (S_2), S_2 \rightarrow S_2S_2\}. \end{aligned}$$

Ähnlich wie oben lässt sich zeigen, dass $L(G_2) = KL$ gilt.

Ende des Beispiels G_2

Wir kommen nun zu einigen strukturellen Anforderungen bzw. Einschränkungen, die an die Form der Produktionen $x \rightarrow y$ einer Grammatik gestellt werden. Durch die *Chomsky-Hierarchie* werden die Grammatiken und vor allem auch die durch sie erzeugten Sprachen in Klassen eingeteilt. Es gibt hauptsächlich vier Sprachklassen, die folgendermaßen durchnummeriert werden: von Chomsky-0 (uneingeschränkt) bis Chomsky-3 (am meisten eingeschränkt). Bei der Definition sind einige Details in Bezug auf Produktionen $X \rightarrow \varepsilon$, durch die das leere Wort aus einem Nichtterminalsymbol $X \in N$ erzeugt wird, zu beachten. Wir nennen solche Produktionen ε -Produktionen.

Sei $u \rightarrow v$ eine Produktion der Grammatik $G = (N, \Sigma, P, S)$. Dann heißt $u \rightarrow v$

<i>linkslin</i>	wenn $u \in N$ und $v \in (N\Sigma^*) \cup \Sigma^*$ (d.h. links steht genau ein Nichtterminalzeichen, rechts höchstens eins, und wenn in v überhaupt eins vorkommt, dann ganz links im Wort)
<i>rechtslin</i>	wenn $u \in N$ und $v \in (\Sigma^*N) \cup \Sigma^*$ (d.h. links steht genau ein Nichtterminalzeichen, rechts höchstens eins, und wenn in v überhaupt eins vorkommt, dann ganz rechts im Wort)
<i>kontextfrei</i>	wenn $u \in N$ (d.h. links steht nur ein Nichtterminalzeichen; $v = \varepsilon$ ist zugelassen)
<i>kontextsensitiv</i>	wenn $u = w_1Xw_2$ und $v = w_1ww_2$ mit $X \in N, w \in (N \cup \Sigma)^+$ (d.h. ein Nichtterminalzeichen X wird im Kontext $w_1..w_2$ durch das nichtleere Wort w ersetzt)
<i>monoton</i>	wenn $ u \leq v $ (d.h. die Produktion wirkt nicht wortverkürzend).

Beispiel: Wir nehmen an, dass S und X Nichtterminalsymbole, a und b Terminalsymbole sind:

$S \rightarrow \varepsilon$	ist (links-, rechts-)linear; kontextfrei; nicht kontextsensitiv; nicht monoton.
$S \rightarrow a$	ist (links-, rechts-)linear; kontextfrei; kontextsensitiv; monoton.
$S \rightarrow aX$	ist rechts-, aber nicht linkslin; kontextfrei; kontextsensitiv; monoton.
$S \rightarrow Sb$	ist links-, aber nicht rechtslin; kontextfrei; kontextsensitiv; monoton.
$S \rightarrow aSb$	ist weder links- noch rechtslin; kontextfrei; kontextsensitiv; monoton.
$aSb \rightarrow aXaXb$	ist nicht kontextfrei, aber kontextsensitiv und monoton.
$aXb \rightarrow bSXa$	ist nicht kontextsensitiv, aber monoton.
$aXb \rightarrow Xa$	besitzt keine der genannten fünf Eigenschaften.

Ende des Beispiels

Es ist klar, dass jede linkslinere Produktion auch kontextfrei ist, dass jede rechtslinere Produktion auch kontextfrei ist, dass jede kontextfreie Produktion außer einer ε -Produktion auch kontextsensitiv ist, und dass jede kontextsensitive Produktion auch monoton ist.

Wir nennen eine Grammatik *linkslin*, wenn alle ihre Produktionen linkslin sind, *rechtslin*, wenn alle ihre Produktionen rechtslin sind, und *kontextfrei*, wenn alle ihre Produktionen kontextfrei sind. Wir nennen eine Grammatik *kontextsensitiv*, wenn alle ihre Produktionen kontextsensitiv sind, außer eventuell die Produktion $S \rightarrow \varepsilon$ (dann aber S nicht auf der rechten Seite einer Produktion vorkommt), und *monoton* oder *vom Erweiterungstyp*, wenn alle ihre Produktionen monoton sind, außer eventuell die Produktion $S \rightarrow \varepsilon$ (dann aber S nicht auf der rechten Seite einer Produktion vorkommt).

Wir lassen also in der Definition kontextsensitiver bzw. monotoner Grammatiken als einzige nicht-kontextsensitive bzw. nicht-monotone Ausnahme die Produktion $S \rightarrow \varepsilon$ zu. Mit dieser Produktion kann

das leere Wort erzeugt werden, das sonst kontextsensitiv und monoton nicht erzeugt werden könnte. Dies lassen wir allerdings nur unter der Voraussetzung zu, dass S nicht auch auf der rechten Seite einer Regel vorkommt, denn sonst könnte man indirekt eine verkürzende Produktion „einschuggeln“, etwa nach dem Muster: $S \rightarrow \varepsilon$ in Verbindung mit $aXb \rightarrow aSb$.

In einer kontextfreien Grammatik darf die Produktion $X \rightarrow \varepsilon$ auftreten, wobei $X \neq S$ ist. Deshalb ist nach obiger Definition nicht jede kontextfreie Grammatik auch kontextsensitiv. Allerdings kann man jede kontextfreie Grammatik ohne Änderung der erzeugten Sprache so umwandeln, dass sie sowohl kontextfrei als auch kontextsensitiv wird. Dies geschieht durch einen Eliminationsalgorithmus, der alle Produktionen $X \rightarrow \varepsilon$ mit $X \neq S$ aus der Grammatik entfernt. Eine Grammatik G heißt *eingeschränkt kontextfrei* oder *ε -frei*, wenn alle Produktionen kontextfrei sind und es entweder überhaupt keine ε -Produktion gibt oder $S \rightarrow \varepsilon$ die einzige ε -Produktion ist und S dann nicht auf der rechten Seite einer Produktion vorkommt. Im ersten Fall gilt $\varepsilon \notin L(G)$, im zweiten Fall gilt $\varepsilon \in L(G)$.

Lemma 3.2.3 EINGESCHRÄNKT KONTEXTFREI IST ÄQUIVALENT MIT KONTEXTFREI

Sei G eine kontextfreie Grammatik über Σ mit $L(G)$. Dann kann aus G eine eingeschränkt kontextfreie Grammatik G' über Σ mit $L(G) = L(G')$ konstruiert werden.

Beweis: Sei $G = (N, \Sigma, P, S)$. Wir zerlegen N so in zwei Teilmengen N_1 und $N_2 = N \setminus N_1$, dass N_1 genau die Variablen X mit $X \Rightarrow_G^* \varepsilon$ enthält. Die Menge N_1 ist induktiv folgendermaßen konstruierbar:

$N_1 := \{X \mid (X \rightarrow \varepsilon) \in P\};$
do $\exists X: ((X \rightarrow X_1 \dots X_m) \in P \text{ und } X \notin N_1 \text{ und } \forall j, 1 \leq j \leq m: X_j \in N_1) \rightarrow$
 füge ein solches X zur Menge N_1 hinzu
od.

Diese Konstruktion bricht ab, weil es nur endlich viele Nichtterminalsymbole und Produktionen gibt. Variablen, die nach dieser Konstruktion in N_2 liegen, können das leere Wort nicht produzieren, weil jede Anwendung einer Produktion mindestens ein Terminal- oder Nichtterminalsymbol hinzufügt. Dann werden die Regeln folgendermaßen verändert:

Entferne alle ε -Produktionen der Form $(X \rightarrow \varepsilon)$;
do \exists Regel $(X \rightarrow uYv)$ mit $X \in N, Y \in N_1, uv \in (N \cup \Sigma)^+$ und $\neg \exists$ Regel $(X \rightarrow uv) \rightarrow$
 füge eine Regel $(X \rightarrow uv)$ zur Regelmenge hinzu
od.

Die neuen Regeln $(X \rightarrow uv)$ sind kontextfrei, und außerdem wird wegen $uv \in (N \cup \Sigma)^+$ keine ε -Produktion neu hinzugefügt. Für die neu hinzugefügten Regeln kann es vorkommen, dass die Eingangsbedingung der Schleife erneut wahr wird, die Schleife also erneut betreten wird. Die Konstruktion bricht trotzdem ab, weil die Längen der rechten Seiten der neu hinzu gefügten Regeln echt kleiner werden.

Wenn $S \notin N_1$, sei die so erhaltene Grammatik G' . Wenn $S \in N_1$, wird ein neues Startsymbol S' mit den beiden Regeln $S' \rightarrow \varepsilon$ und $S' \rightarrow S$ hinzugefügt; S wird ein normales Nichtterminalsymbol. Die so erhaltene Grammatik sei G' . Es gilt $L(G) = L(G')$. □ 3.2.3

Wir geben zwei Beispiele für diese Konstruktion.

Beispiel G_3 :

Seien $G_3 = (N_3, \{a, \dots, z, 0, \dots, 9\}, P_3, \langle \text{id} \rangle)$ mit $N_3 = \{\langle \text{id} \rangle, \langle \text{tail} \rangle, \langle \text{letter} \rangle, \langle \text{digit} \rangle\}$ und P_3 wie folgt:

$$\begin{aligned} \langle \text{id} \rangle &\rightarrow \langle \text{letter} \rangle \langle \text{tail} \rangle \\ \langle \text{tail} \rangle &\rightarrow \varepsilon \mid \langle \text{letter} \rangle \langle \text{tail} \rangle \mid \langle \text{digit} \rangle \langle \text{tail} \rangle \\ \langle \text{letter} \rangle &\rightarrow a \mid \dots \mid z \\ \langle \text{digit} \rangle &\rightarrow 0 \mid \dots \mid 9. \end{aligned}$$

Dabei ist $\langle \text{tail} \rangle \rightarrow \varepsilon \mid \langle \text{letter} \rangle \langle \text{tail} \rangle \mid \langle \text{digit} \rangle \langle \text{tail} \rangle$ eine gebräuchliche Abkürzung für die drei Regeln:

$$\begin{aligned} \langle \text{tail} \rangle &\rightarrow \varepsilon \\ \langle \text{tail} \rangle &\rightarrow \langle \text{letter} \rangle \langle \text{tail} \rangle \\ \langle \text{tail} \rangle &\rightarrow \langle \text{digit} \rangle \langle \text{tail} \rangle, \end{aligned}$$

und eine ähnliche Art von Abkürzung wurde für $\langle \text{letter} \rangle$ und $\langle \text{digit} \rangle$ benutzt. Ein Wort, das aus G_3 generiert werden kann, ist z.B. `summe1`. Hier ist (andeutungsweise) die Erzeugung:

$$\langle \text{id} \rangle \vdash_{G_3} \langle \text{letter} \rangle \langle \text{tail} \rangle \vdash_{G_3} \dots \vdash_{G_3} \text{summe1}$$

Generell sind alle aus G_3 erzeugbaren Wörter auch *identifier* im Sinne einer Programmiersprache wie etwa `C`, `Pascal` oder `Java`.

Die Grammatik G_3 ist kontextfrei (und sogar „versteckt“ rechtslinear), aber nicht ε -frei, da die Produktion $\langle \text{tail} \rangle \rightarrow \varepsilon$ enthalten ist. Wir formen sie gemäß der Konstruktion des obigen Lemmas um. Die Menge der Variablen aus N_3 , aus denen ε produziert werden kann, ergibt sich zu $\{\langle \text{tail} \rangle\}$. Da das ursprüngliche Startsymbol $\langle \text{id} \rangle$ nicht in dieser Menge enthalten ist, benötigen wir kein neues Startsymbol S' . Wir löschen daher die Produktion $\langle \text{tail} \rangle \rightarrow \varepsilon$, fügen die Produktionen $\langle \text{id} \rangle \rightarrow \langle \text{letter} \rangle$, $\langle \text{tail} \rangle \rightarrow \langle \text{letter} \rangle$ und $\langle \text{tail} \rangle \rightarrow \langle \text{digit} \rangle$ hinzu und erhalten G'_3 :

$$\begin{aligned} \langle \text{id} \rangle &\rightarrow \langle \text{letter} \rangle \langle \text{tail} \rangle \mid \langle \text{letter} \rangle \\ \langle \text{tail} \rangle &\rightarrow \langle \text{letter} \rangle \langle \text{tail} \rangle \mid \langle \text{digit} \rangle \langle \text{tail} \rangle \mid \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \\ \langle \text{letter} \rangle &\rightarrow a \mid \dots \mid z \\ \langle \text{digit} \rangle &\rightarrow 0 \mid \dots \mid 9. \end{aligned}$$

Die ε -Produktion wird durch diese Konstruktion sozusagen „vorweggenommen“.

Ende des Beispiels G_3

Beispiel G_1 (Fortsetzung):

Wir erinnern uns an G_1 mit seinem Startsymbol S_1 und den beiden Produktionen $S_1 \rightarrow \varepsilon$ und $S_1 \rightarrow aS_1b$. Hier gilt $S_1 \Rightarrow_{G_1}^* \varepsilon$, und S_1 kommt auf einer rechten Seite vor. Deswegen löschen wir die Produktion $S_1 \rightarrow \varepsilon$, fügen die Produktion $S_1 \rightarrow ab$ hinzu und führen ein neues Startsymbol S'_1 mit den neuen Produktionen $S'_1 \rightarrow \varepsilon$ und $S'_1 \rightarrow S_1$ ein:

$$\begin{aligned} P'_1 : S'_1 &\rightarrow \varepsilon \\ S'_1 &\rightarrow S_1 \\ S_1 &\rightarrow ab \\ S_1 &\rightarrow aS_1b. \end{aligned}$$

Ende des Beispiels G_1 (Fortsetzung)

Wegen Lemma 3.2.3 gilt allgemein:

$$\begin{aligned} &\{ L \subseteq \Sigma^* \mid \text{es gibt eine kontextfreie Grammatik } G \text{ über } \Sigma \text{ mit } L = L(G) \} \\ = &\{ K \subseteq \Sigma^* \mid \text{es gibt eine eingeschränkt kontextfreie Grammatik } G' \text{ über } \Sigma \text{ mit } K = L(G') \}. \end{aligned} \quad (3.1)$$

In Bezug auf die Klasse der generierten Sprachen ist es also keine Einschränkung, eine Grammatik als eingeschränkt kontextfrei, anstelle von kontextfrei, anzunehmen. In formaler Hinsicht sind die eingeschränkt kontextfreien Grammatiken günstig, denn eine eingeschränkt kontextfreie Grammatik ist auch eine kontextsensitive Grammatik. Für den „täglichen Gebrauch“ sind die uneingeschränkt kontextfreien Grammatiken besser geeignet, da sie mehr Definitionsspielraum lassen.

Wir kommen nun zur Definition der Chomsky-Hierarchie. Dazu ändern wir unsere Sichtweise von den Grammatiktypen hin zu den erzeugten Sprachklassen.

Definition 3.2.4 CHOMSKY-SPRACHKLASSEN

Sei Σ ein Alphabet. Eine Sprache $L \subseteq \Sigma^*$ heißt

- *linkslin*ear, wenn es eine linkslinare Grammatik G mit $L = L(G)$ gibt;
- *rechtslin*ear oder *Chomsky-3*, wenn es eine rechtsliniare Grammatik G mit $L = L(G)$ gibt;
- *kontextfrei* oder *Chomsky-2*, wenn es eine kontextfreie Grammatik G mit $L = L(G)$ gibt;
- *kontextsensitiv* oder *Chomsky-1*, wenn es eine kontextsensitive Grammatik G mit $L = L(G)$ gibt;
- *vom Erweiterungstyp*, wenn es eine monotone Grammatik G mit $L = L(G)$ gibt;
- *Chomsky-0*, wenn es eine Grammatik G mit $L = L(G)$ gibt. ☒ 3.2.4

Beispiel L_4, G_4 :

Wir nehmen zunächst $\Sigma = \{a, b\}$ an und betrachten die Sprache

$$L_4 = \{a^n b \mid n \in \mathbb{N}\} = \{b, ab, aab, \dots\}.$$

Eine Grammatik, die diese Sprache erzeugt, ist $G_4 = (\{S\}, \{a, b\}, P_4, S)$ mit $P_4 = \{S \rightarrow aS, S \rightarrow b\}$. Dies zeigt, dass L_4 rechtslinear ist. L_4 ist auch linkslinear, wie durch die Grammatik $G'_4 = (\{A, S\}, \{a, b\}, P'_4, S)$ mit $P'_4 = \{S \rightarrow Ab, A \rightarrow \varepsilon, A \rightarrow Aa\}$ gezeigt wird.

Ende des Beispiels L_4, G_4

Beispiel L_1, G_1 :

Wieder mit $\Sigma = \{a, b\}$ betrachten wir die Sprache

$$L_1 = \{a^n b^n \mid n \in \mathbb{N}\} = \{\varepsilon, ab, aabb, \dots\}.$$

Wie wir schon gesehen haben (Beispiel G_1), kann diese Sprache durch eine Grammatik mit den Produktionen $S \rightarrow \varepsilon$ und $S \rightarrow aSb$ generiert werden. Diese Grammatik ist kontextfrei, also ist auch L_1 kontextfrei.

Ende des Beispiels L_1, G_1

Wir werden später sehen, dass die Sprache L_1 durch keine links- oder rechtslineare Grammatik erzeugt werden kann, d.h. weder linkslinear noch rechtslinear ist. Für einen solchen Unmöglichkeitbeweis benötigen wir ein Argument über die Menge *aller* (rechts- und linkslinärer) Grammatiken, was etwas schwieriger ist, als eine geeignete (kontextfreie) Grammatik für L_1 anzugeben.

Beispiel L_5, G_5 :

Nun betrachten wir $\Sigma = \{a, b, c\}$ und die Sprache

$$L_5 = \{a^n b^n c^n \mid n \in \mathbb{N}\} = \{\varepsilon, abc, aabbcc, aaabbccc, \dots\}.$$

Diese Sprache ist monoton, denn sie wird von der folgenden monotonen Grammatik erzeugt: $G_5 = (\{S, R, B\}, \{a, b, c\}, P_5, S)$ mit

$$P_5 : \begin{array}{l} S \rightarrow \varepsilon \mid R \\ R \rightarrow aRb \mid abc \\ cB \rightarrow Bc \\ bB \rightarrow bb. \end{array}$$

Ein Ableitungsbeispiel:

$$S \vdash_{G_5} R \vdash_{G_5} aRb \vdash_{G_5} aabcBc \vdash_{G_5} aabBcc \vdash_{G_5} aabbcc.$$

Später werden wir zeigen, dass die Sprache L_5 kontextsensitiv, aber nicht kontextfrei ist.

Ende des Beispiels L_5, G_5

Dass tatsächlich eine Hierarchie von Sprachklassen vorliegt, zeigt der nächste Satz.

Satz 3.2.5 CHOMSKY-HIERARCHIE

- (a) Jede linkslineare Sprache ist kontextfrei.
- (b) Jede rechtslineare (Chomsky-3) Sprache ist kontextfrei (Chomsky-2).
- (c) Jede kontextfreie (Chomsky-2) Sprache ist kontextsensitiv (Chomsky-1).
- (d) Jede kontextsensitive (Chomsky-1) Sprache ist vom Erweiterungstyp.
- (e) Jede Sprache vom Erweiterungstyp ist eine Chomsky-0-Sprache.

Beweis: (a) und (b): Sei $L = L(G)$ mit einer links- bzw. rechtslinearen Grammatik G . Da G auch kontextfrei ist, ist L eine kontextfreie Sprache.

(c): Sei L kontextfrei, d.h. $L = L(G)$ mit einer kontextfreien Grammatik G . Nach (3.1) gilt auch $L = L(G')$ mit einer eingeschränkt kontextfreien Grammatik G' . Da G' auch kontextsensitiv ist, ist L eine kontextsensitive Sprache.

(d): Jede kontextsensitive Grammatik ist auch monoton.

(e): Jede monotone Grammatik ist überhaupt eine Grammatik. ☒ 3.2.5

Später zeigen wir, dass jede linkslineare Sprache auch rechtslinear ist (und umgekehrt) und dass jede Sprache vom Erweiterungstyp auch kontextsensitiv ist. Damit reduziert sich die Chomsky-Hierarchie auf die vier genannten Sprachklassen Chomsky-3 (links- bzw. rechtslineare Sprachen), Chomsky-2 (kontextfreie Sprachen), Chomsky-1 (kontextsensitive Sprachen, bzw. Sprachen vom Erweiterungstyp) und Chomsky-0 (allgemein durch Grammatiken erzeugbare Sprachen). Jede Chomsky- i -Sprache ist auch Chomsky- $(i-1)$, für $0 < i \leq 3$. Dass die Mengen der Chomsky-3- und Chomsky-2-Sprachen sowie die Mengen der Chomsky-2- und Chomsky-1-Sprachen auseinanderfallen, zeigen die beiden Beispiele L_1 und L_5 (wofür der strenge Nachweis allerdings noch fehlt). Dafür, dass die beiden Sprachklassen Chomsky-1 und Chomsky-0 auseinanderfallen (d.h., dass es eine Chomsky-0-Sprache gibt, die nicht Chomsky-1 ist), werden wir erst später ein Argument finden. Der Theorie der Sprachen vom Typ 3 und vom Typ 2 werden zwei eigene Kapitel dieses Skripts gewidmet (Kapitel 4 resp. 5).

3.3 Übungsaufgaben

1. Sei $\Sigma = \{a, b\}$. Untersuchen Sie, ob die folgenden Ersetzungssysteme Noethersch sind und ob sie die Church-Rosser Eigenschaft besitzen. Führen Sie explizite Beweise oder geben Sie entsprechende Gegenbeispiele an.

a) $E_1 = (\Sigma, \{(ab, baa)\})$

b) $E_2 = (\Sigma, \{(a, b), (aba, bbaab)\})$

Hinweis: In Teil a) spielt die Position der b 's, in Teil b) die Anzahl der a 's eine wichtige Rolle.

2. Sei $\Sigma = \{a, b, c\}$. Gegeben sei die Grammatik $G = (\{S, X\}, \Sigma, P, S)$ mit

$$P = \left\{ \begin{array}{ll} S & \rightarrow a, \\ S & \rightarrow abSX, \\ X & \rightarrow ab, \\ baX & \rightarrow Xc, \\ bXc & \rightarrow a, \\ aXc & \rightarrow bX \end{array} \right\}.$$

- a) Überprüfen Sie, welches der Wörter bab , caa und $abaabab$ zu $L(G)$ gehört und welches nicht (begründen Sie Ihre Antwort!).

- b) Ist die partielle Funktion $f : \Sigma^* \xrightarrow{p} \mathbb{N}$ mit

$$f(w) = \begin{cases} 1 & , \text{ falls } w \in L(G) \\ \text{undef} & , \text{ falls } w \notin L(G) \end{cases}$$

berechenbar? Falls ja: Erläutern Sie ein Verfahren zur Berechnung von f . Falls nein: Warum ist f nicht berechenbar?

3. Gegeben seien das Alphabet $\Sigma = \{a, b, c\}$ und die Sprachen

$$L_1 = \{w \mid w \text{ ist ein Palindrom über } \Sigma\} \text{ und}$$

$$L_2 = \{a^n b^m c^n \mid m, n \in \mathbb{N} \wedge m > n\}.$$

Dabei ist die Menge aller Palindrome über einem Alphabet Σ folgendermaßen definiert:

- (i) ε ist ein Palindrom.
- (ii) Für alle $a \in \Sigma$ ist a ein Palindrom.
- (iii) Ist $a \in \Sigma$ und w ein Palindrom, so ist auch awa ein Palindrom.
- (iv) Sonst ist nichts ein Palindrom über Σ .

- a) Konstruieren Sie eine Grammatik G_1 mit $L(G_1) = L_1$.

- b) Konstruieren Sie eine Grammatik G_2 mit $L(G_2) = L_2$.

4. Zeigen Sie, dass die Sprache

$$\{a^{2^i} \mid i \in \mathbb{N}\} = \{a, aa, aaaa, aaaaaaaaa, \dots\}$$

vom Typ Chomsky-1 ist.

5. Seien $G_1 = (N_1, \Sigma_1, P_1, S_1)$ und $G_2 = (N_2, \Sigma_2, P_2, S_2)$ zwei kontextfreie Grammatiken, welche die Sprachen $L_1 = L(G_1)$ und $L_2 = L(G_2)$ erzeugen. Beweisen Sie, dass auch

a) $L_1 \cup L_2$,

b) $L_1 \circ L_2$ und

c) L_1^*

kontextfreie Sprachen sind.

6. Welche Sprache wird von der kontextfreien Grammatik $G = (\{S, X, Y, Z\}, \{a, b\}, P, S)$ mit

$$P = \left\{ \begin{array}{l} S \rightarrow aX \mid Ybb \\ X \rightarrow Zb \mid aXb \\ Y \rightarrow aaZ \mid aYb \\ Z \rightarrow \varepsilon \mid aZb \end{array} \right\}$$

erzeugt? Beweisen Sie Ihre Behauptung!

Kapitel 4

Endliche Automaten und reguläre Sprachen

In diesem Kapitel untersuchen wir ein eingeschränktes, aber sehr wichtiges Maschinenmodell, dessen Leistungsfähigkeit durch Chomsky-3-Sprachen charakterisiert werden kann: den endlichen Automaten. Endliche Automaten können in der Informatik vielseitig eingesetzt werden und die von endlichen Automaten akzeptierten Sprachen besitzen viele interessante Struktureigenschaften. Endliche Automaten können als Tabellen in Programmen und als Schaltungen realisiert bzw. implementiert werden.

4.1 Endliche Automaten: Definition und Beispiele

Wir können uns einen endlichen Automaten vorstellen als eine Maschine, die Wörter über einem Alphabet Σ entweder *akzeptiert* oder nicht akzeptiert. Zu diesem Zweck wird mit einem „Lesekopf“ ein Eingabewort $w \in \Sigma^*$, dessen Akzeptabilität untersucht werden soll, von links nach rechts Buchstabe für Buchstabe eingelesen. Beim Lesen eines Zeichens befindet sich der Automat in einem bestimmten Zustand, der sich nach dem Lesen auch ändern kann. Wenn der Automat das Eingabewort w ganz lesen kann und gelesen hat, gibt der dann vorliegende Zustand darüber Auskunft, ob w akzeptiert wird. Insgesamt hat der Automat nur eine endliche Menge Q von Zuständen zur Verfügung.

Wegen dieser Interpretation wird bei endlichen Automaten eine Überführungsrelation $\delta \subseteq (Q \times \Sigma) \times Q$ definiert. Q und δ bilden die *endliche Kontrolle* des Automaten. Ein Element $(q, a, q') \in \delta$ bedeutet: Im Zustand q wird gerade a gelesen und ein möglicher Folgezustand ist q' . Die Abbildung 4.1 zeigt zum Beispiel einen Automaten, der gerade das erste „T“ des Eingabewortes AUTOMAT im aktuellen Zustand q abarbeitet.

Definition 4.1.1 ENDLICHE AUTOMATEN

Ein *nichtdeterministischer endlicher Automat* über Σ , kurz NFA (für nondeterministic finite automaton), ist eine Struktur $A = (Q, \Sigma, \delta, Q_0, F)$ mit folgenden Komponenten:

Q ist eine endliche Menge von *Zuständen*.

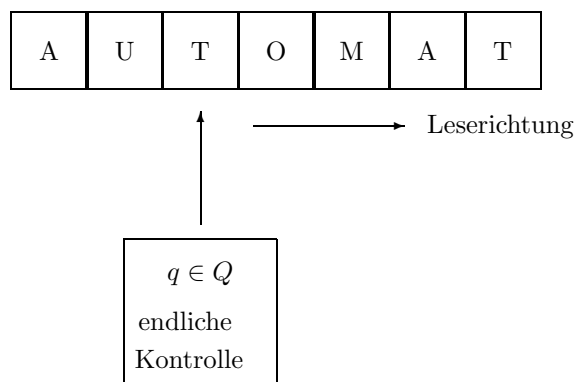


Abbildung 4.1: Skizze für die Wirkungsweise eines endlichen Automaten

Σ ist das *Alphabet* der Maschine. Intuitiv sind die Zeichen in Σ die „Eingabezeichen“ von A .

$\delta \subseteq (Q \times \Sigma) \times Q$ ist die *Übergangsrelation*.

$Q_0 \subseteq Q$ ist eine Menge von *Anfangszuständen*.

$F \subseteq Q$ ist eine Menge von *Endzuständen*.

Ein endlicher Automat heißt *deterministisch* (kurz DFA), wenn $Q_0 = \{q_0\}$ eine Einermenge ist und δ (als Relation $\delta \subseteq (Q \times \Sigma) \times Q$) rechtseindeutig und linkstotal (d.h. eine Funktion von $Q \times \Sigma$ nach Q) ist. Statt $(Q, \Sigma, \delta, \{q_0\}, F)$ wird dann auch einfacher $(Q, \Sigma, \delta, q_0, F)$ geschrieben. \boxtimes 4.1.1

Die Elemente $(q, a, q') \in \delta$ heißen auch manchmal *Transitionen* und werden als $q \xrightarrow{a} q'$ geschrieben, manchmal auch als $q \xrightarrow{a}_A q'$, um anzudeuten, dass es sich um den Automaten A handelt. Manchmal wird deswegen δ selbst als \longrightarrow bzw. als \longrightarrow_A bezeichnet.

Ein NFA kann graphisch durch ein *Zustandsdiagramm* dargestellt werden. Das ist ein gerichteter Graph, dessen Knotenmenge Q ist und der für jede Transition $(q, a, q') \in \delta$ eine gerichtete, mit a beschriftete Kante von q nach q' enthält. Anfangszustände q_0 werden durch eingehende Pfeile markiert, Endzustände q sind durch einen zusätzlichen Kreis gekennzeichnet.

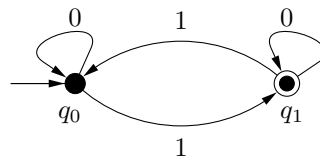
Beispiel A_1 :

$$A_1 = (\{q_0, q_1\}, \{0, 1\}, \delta_1, \{q_0\}, \{q_1\})$$

mit $\delta_1 = \{(q_0, 0, q_0), (q_0, 1, q_1), (q_1, 0, q_1), (q_1, 1, q_0)\}$. Dann wird A_1 durch das in Abbildung 4.2 gezeigte Zustandsdiagramm dargestellt.

Ende des Beispiels A_1

Zur Definition der von einem endlichen Automaten A akzeptierten Sprache $L(A)$ betrachten wir die Relationen $\xrightarrow{a} \subseteq Q \times Q$ (eine für jedes Zeichen a aus Σ) mit $\xrightarrow{a} = \{(q, q') \mid (q, a, q') \in \delta\}$ und erweitern

Abbildung 4.2: Zustandsdiagramm von A_1

sie folgendermaßen:

$$\begin{aligned} \xrightarrow{\varepsilon} &= id_Q \\ \xrightarrow{wa} &= (\xrightarrow{w} \circ \xrightarrow{a}) \quad (w \in \Sigma^*, a \in \Sigma). \end{aligned}$$

Demzufolge gilt für $a_1, \dots, a_n \in \Sigma$ die Beziehung

$$q \xrightarrow{a_1} \circ \dots \circ \xrightarrow{a_n} q'$$

genau dann, wenn es Zustände q_0, q_1, \dots, q_n gibt mit

$$q = q_0, q_0 \xrightarrow{a_1} q_1, q_1 \xrightarrow{a_2} q_2, \dots, q_{n-1} \xrightarrow{a_n} q_n, q_n = q'.$$

Die Folge $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \dots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n$ wird auch *Transitionenfolge* genannt.

Definition 4.1.2 AKZEPTANZ UND SPRACHÄQUIVALENZ

Die von einem NFA $A = (Q, \Sigma, \delta, Q_0, F)$ akzeptierte Sprache ist definiert als

$$L(A) = \{w \in \Sigma^* \mid \exists q_0 \in Q_0 \exists q \in F: q_0 \xrightarrow{w} q\}.$$

Mit anderen Worten, $a_1 \dots a_n$ wird von A akzeptiert, wenn aus einem Anfangszustand durch sukzessive Anwendung von $\xrightarrow{a_1}, \xrightarrow{a_2}, \dots, \xrightarrow{a_n}$ ein Endzustand erreicht werden kann.

Eine Sprache $L \subseteq \Sigma^*$ heißt *endlich akzeptierbar*, wenn es einen NFA A mit $L = L(A)$ gibt, und *deterministisch endlich akzeptierbar*, wenn es einen DFA A mit $L = L(A)$ gibt.

Zwei Automaten A_1 und A_2 heißen *sprachäquivalent* oder nur *äquivalent*, wenn gilt: $L(A_1) = L(A_2)$.

☒ 4.1.2

Beispiel A_1 (Fortsetzung):

Für den Automaten A_1 des letzten Beispiels gilt

$$L(A_1) = \{w \in \{0, 1\}^* \mid w \text{ enthält ungerade viele Symbole } 1\}.$$

Da A_1 deterministisch ist, ist diese Sprache deterministisch endlich akzeptierbar.

Ende des Beispiels A_1 (Fortsetzung)

Jede deterministisch endliche Sprache ist auch endlich akzeptierbar, da jeder DFA auch ein NFA ist. Wie wir bald sehen werden, ist umgekehrt auch jede endlich akzeptierbare Sprache deterministisch endlich

akzeptierbar, so dass zwischen den beiden Begriffen kein Unterschied besteht und man nur von „endlicher Akzeptierbarkeit“ spricht.

Beispiel (Suffix-Erkennung):

Gegeben seien ein Alphabet Σ und ein Wort $v = a_1 \dots a_n \in \Sigma^*$. Wir wollen die Sprache

$$L_v = \{wv \mid w \in \Sigma^*\}$$

aller Wörter über Σ , die den Suffix v haben, erkennen. Speziell betrachten wir $\Sigma = \{0, 1, 2\}$ und den Suffix $v = 01$. Zur Erkennung von $L_v = L_{01}$ betrachten wir den NFA

$$A_{01} = (\{q_0, q_1, q_2\}, \Sigma, \delta, \{q_0\}, \{q_2\}),$$

wobei δ durch das in Figur 4.3 gezeigte Zustandsdiagramm erklärt ist, d.h.:

$$\delta = \{(q_0, 0, q_0), (q_0, 1, q_0), (q_0, 2, q_0), (q_0, 0, q_1), (q_1, 1, q_2)\}.$$

A_{01} ist im Anfangszustand q_0 nichtdeterministisch: Beim Lesen eines Wortes kann sich A_{01} bei jedem Vorkommen von 0 entscheiden, entweder 0 zu „überlesen“ oder ab jetzt zu versuchen, den Suffix 01 zu akzeptieren. Zu letzterem Zweck geht A_{01} nach q_1 über und erwartet 1 als Rest. Sollte keine 1, sondern eine 0 oder eine 2 folgen, so „stoppt“ A_{01} , ohne zu akzeptieren. Es gilt $L_{01} = L(A_{01})$.

Frage: wie mag wohl ein deterministischer Automat aussehen, der L_{01} akzeptiert?

Auflösung: einen solchen DFA werden wir später angeben.

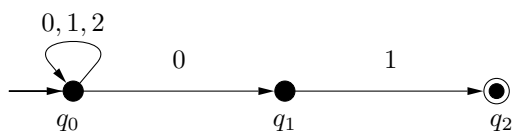


Abbildung 4.3: Zustandsdiagramm von A_{01} für Suffixerkennung: $L(A_{01}) = L_{01}$

Ende des Beispiels (Suffixerkennung)

4.2 Endlich akzeptierbare Sprachen und Chomsky-3-Sprachen

In diesem Abschnitt beweisen wir vier Sätze, nämlich

- dass jede Chomsky-3-Sprache (siehe Definition 3.2.4) endlich akzeptierbar ist,
- dass jede endlich akzeptierbare Sprache deterministisch endlich akzeptierbar ist,
- dass jede deterministisch endlich akzeptierbare Sprache Chomsky-3 ist,
- und dass eine Sprache genau dann rechtslinear (d.h. Chomsky-3) ist, wenn sie linkslinear ist.

Damit hat man die vier genannten, für Sprachen definierten Begriffe (rechtslinear, d.h. Chomsky-3, linkslinear, endlich akzeptierbar, deterministisch endlich akzeptierbar) als äquivalent nachgewiesen. Für den ersten dieser Sätze benutzen wir ein Lemma, das die Erzeugbarkeit von Chomsky-3-Sprachen durch einen besonders einfachen Typ rechtslinearer Grammatiken charakterisiert.

Lemma 4.2.1 EINE CHARAKTERISIERUNG VON RECHTSLINEAREN SPRACHEN

Sei $L \subseteq \Sigma^*$ eine rechtslineare Sprache. Dann gibt es eine Grammatik $G = (N, \Sigma, P, S)$ mit $L = L(G)$, so dass alle Produktionen in P entweder die Form $X \rightarrow aY$ (mit $X, Y \in N, a \in \Sigma$) oder die Form $X \rightarrow a$ (mit $X \in N, a \in \Sigma$) oder die Form $S \rightarrow \varepsilon$ haben.

Beweis: Durch geeignete Umformung und eventuelle Einführung neuer Variablen.

Eine Produktion $A \rightarrow a_1 \dots a_n B$ ($n \geq 2$) kann durch Einführen von $n - 1$ neuen Variablen auf die geforderte Form gebracht werden. Z.B. kann die Produktion $A \rightarrow abB$ durch Einführen einer Variablen A' in die beiden Produktionen $A \rightarrow aA'$ und $A' \rightarrow bB$ überführt werden.

Eine Produktion der Form $A \rightarrow B$ (die Kettenproduktion genannt wird) kann redundant gemacht – und dann gelöscht – werden, indem überall dort, wo B auf der linken Seite einer Produktion vorkommt, eine Regel hinzugefügt wird, indem B durch A ersetzt wird.

Eine ε -Produktion $X \rightarrow \varepsilon$ ($X \neq S$) kann redundant gemacht – und dann gelöscht – werden, indem auf allen rechten Seiten X durch ε ersetzt und die entstehende Produktion hinzugefügt wird. \square 4.2.1

Beispiel:

Wir betrachten die Menge

$$\{S \rightarrow abS, S \rightarrow aY, Y \rightarrow Z, Z \rightarrow \varepsilon\}$$

von Produktionen. Nach dem ersten Schritt ergibt sich

$$\{S \rightarrow aR, R \rightarrow bS, S \rightarrow aY, Y \rightarrow Z, Z \rightarrow \varepsilon\},$$

nach dem zweiten Schritt

$$\{S \rightarrow aR, R \rightarrow bS, S \rightarrow aY, Y \rightarrow \varepsilon, Z \rightarrow \varepsilon\}$$

und nach dem dritten Schritt

$$\{S \rightarrow aR, R \rightarrow bS, S \rightarrow aY, S \rightarrow a\}.$$

Die entstandene unnötige Produktion $S \rightarrow aY$ kann, wenn gewünscht, noch gelöscht werden (das gehört aber nicht zur Konstruktion im Beweis).

Ende des Beispiels**Satz 4.2.2** CHOMSKY-3-SPRACHEN SIND ENDLICH AKZEPTIERBAR

Sei $L \subseteq \Sigma^*$ eine Chomsky-3-Sprache. Dann ist L endlich akzeptierbar.

Beweis: Sei $G = (N, \Sigma, P, S)$ eine rechtslineare Grammatik, die L erzeugt, d.h. $L = L(G)$. Wegen Lemma 4.2.1 dürfen wir ohne Beschränkung der Allgemeinheit annehmen, dass die Produktionen von P von der Form $A \rightarrow aB$ oder von der Form $A \rightarrow a$ mit $A, B \in N$ und $a \in \Sigma$, oder von der Form $S \rightarrow \varepsilon$ sind. Wir definieren aus G einen NFA $A(G) = (Q, \Sigma, \delta, Q_0, F)$ über dem gleichen Alphabet Σ folgendermaßen:

$$\begin{aligned} Q &= N \cup \{X\} \quad (\text{wobei } X \notin N) \\ \delta &= \{(A, a, B) \mid (A \rightarrow aB) \in P\} \cup \{(A, a, X) \mid (A \rightarrow a) \in P\} \\ Q_0 &= \{S\} \\ F &= \begin{cases} \{S, X\} & \text{falls } (S \rightarrow \varepsilon) \in P \\ \{X\} & \text{falls } (S \rightarrow \varepsilon) \notin P. \end{cases} \end{aligned}$$

Wegen der Definition von F gilt $\varepsilon \in L(G)$ genau dann, wenn $\varepsilon \in L(A(G))$. Außerdem gilt für $m \geq 1$:

$$\begin{aligned}
 a_1 a_2 \dots a_m \in L(G) &\Leftrightarrow (\text{Form von } G \text{ und Definition von } L(G)) \\
 &\text{es gibt eine Folge von Variablen } A_1, \dots, A_{m-1} \text{ mit} \\
 &S \vdash_G a_1 A_1 \vdash_G \dots \vdash_G a_1 \dots a_{m-1} A_{m-1} \vdash_G a_1 a_2 \dots a_{m-1} a_m \\
 &\Leftrightarrow (\text{Definition von } Q \text{ und } \delta) \\
 &\text{es gibt eine Folge von Zuständen } A_1, \dots, A_{m-1} \text{ mit} \\
 &(S, a_1, A_1) \in \delta, (A_1, a_2, A_2) \in \delta, \dots, (A_{m-1}, a_m, X) \in \delta \\
 &\Leftrightarrow (\text{Definition von } F) \\
 &a_1 a_2 \dots a_m \in L(A(G)). \quad \square \text{ 4.2.2}
 \end{aligned}$$

Beispiel:

Als Anwendungsbeispiele für diese Konstruktion betrachten wir zwei rechtslineare Grammatiken G_1 und G_2 über $\Sigma = \{a, b\}$:

$$G_1 : \begin{cases} S \rightarrow \varepsilon \\ S \rightarrow aR \\ R \rightarrow bS \end{cases} \qquad G_2 : \begin{cases} S \rightarrow a \\ S \rightarrow aR \\ R \rightarrow bS. \end{cases}$$

Es gilt $L(G_1) = \{(ab)^n \mid n \in \mathbb{N}\}$ und $L(G_2) = \{(ab)^n a \mid n \in \mathbb{N}\}$. Die Abbildung 4.4 zeigt die beiden Automaten $A(G_1)$ und $A(G_2)$.

Ende des Beispiels



Abbildung 4.4: Automaten für G_1 (links) und G_2 (rechts)

Satz 4.2.3 RABIN UND SCOTT, 1959

Sei $L \subseteq \Sigma^*$ eine endlich akzeptierbare Sprache. Dann ist L deterministisch endlich akzeptierbar.

Beweis: Gegeben sei ein NFA $A = (Q, \Sigma, \delta, Q_0, F)$ mit $L = L(A)$. Wir konstruieren einen DFA $B = (Q_B, \Sigma, \delta_B, q_{0B}, F_B)$, für den $L(A) = L(B)$ gilt. Diese Konstruktion ist auch als *Potenzmengen-Konstruktion* bekannt geworden.

$$\begin{aligned}
 Q_B &= 2^Q \\
 (S, a, S') \in \delta_B &\Leftrightarrow S' = \{q' \in Q \mid \exists q \in S: (q, a, q') \in \delta\} \quad \text{für } S, S' \subseteq Q \\
 q_{0B} &= Q_0 \\
 F_B &= \{S \subseteq Q \mid S \cap F \neq \emptyset\}.
 \end{aligned}$$

B hat also für jede Teilmenge S der Zustandsmenge Q von A einen eigenen Zustand. Dabei kann S' aus S durch einen a -Übergang genau dann erreicht werden, wenn S' die Menge der Folgezustände ist, die

von S aus durch a -Übergänge erreicht werden können. Gibt es keine solchen, dann ist $S' = \emptyset$. Also ist δ_B immer linkstotal, selbst wenn das für δ nicht gilt. Offenbar ist δ_B auch rechtseindeutig, denn zu S und a ist S' mit $(S, a, S') \in \delta_B$ eindeutig definiert. Außerdem hat B genau einen Anfangszustand q_{0B} . Also ist B in der Tat ein DFA. Ferner gilt für alle $q, q' \in Q$, $S, S' \subseteq Q$ und $a \in \Sigma$:

- (i) Wenn $(q, a, q') \in \delta$ und $q \in S$, dann existiert ein $S' \subseteq Q$ mit $(S, a, S') \in \delta_B$ und $q' \in S'$.
(Denn das oben zu S und a definierte S' erfüllt diese Eigenschaft.)
- (ii) Wenn $(S, a, S') \in \delta_B$ und $q' \in S'$, dann existiert ein $q \in Q$ mit $(q, a, q') \in \delta$ und $q \in S$.
(Denn so ist S' definiert.)

Damit lässt sich $L(A) = L(B)$ zeigen. Sei $w = a_1 \dots a_n \in \Sigma^*$. Dann gilt:

$$\begin{aligned}
 w \in L(A) &\Leftrightarrow (\text{Definition von } L(A)) \\
 &\quad \exists q_0, \dots, q_n \in Q: (q_0, a_1, q_1) \in \delta, \dots, (q_{n-1}, a_n, q_n) \in \delta \text{ mit } q_0 \in Q_0 \text{ und } q_n \in F \\
 &\Leftrightarrow (\text{„}\Rightarrow\text{“ wegen (i) und „}\Leftarrow\text{“ wegen (ii) und wegen } q_n \in S_n) \\
 &\quad \exists S_1, \dots, S_n \subseteq Q: (Q_0, a_1, S_1) \in \delta_B \dots (S_{n-1}, a_n, S_n) \in \delta_B \text{ mit } q_n \in (S_n \cap F) \\
 &\Leftrightarrow (\text{Definition von } L(B); \text{ für „}\Leftarrow\text{“ definiere } q_n \text{ als beliebiges Element von } S_n \cap F) \\
 &\quad w \in L(B).
 \end{aligned}$$

$L(A) = L(B) = L$ ist also deterministisch endlich akzeptierbar. □ 4.2.3

Beispiel A_{01} (Fortsetzung):

Wir betrachten nochmals die Suffix-Erkennung von $v = 01$ in Wörtern über dem Alphabet $\Sigma = \{0, 1, 2\}$. Die Sprache $L_{01} = \{w01 \mid w \in \Sigma^*\}$ wird vom NFA A_{01} , dessen Zustandsdiagramm in Abbildung 4.3 angegeben ist, erkannt. Wir wenden jetzt auf A_{01} die Potenzmengen-Konstruktion aus dem Satz von Rabin und Scott an. Das Ergebnis ist der DFA B_{01} , der in Abbildung 4.5 gezeigt ist.

Die umkasteten Teile (d.h., die Zustände \emptyset , $\{q_1\}$, $\{q_2\}$, $\{q_1, q_2\}$ und $\{q_0, q_1, q_2\}$) sind vom Anfangszustand $\{q_0\}$ von B_{01} nicht erreichbar; sie sind „überflüssig“. B_{01} kann deshalb zu dem in Abbildung 4.6 gezeigten sprachäquivalenten DFA B_{01}^{min} vereinfacht werden. B_{01}^{min} lässt sich bezüglich seiner Zustandszahl nicht weiter minimieren.

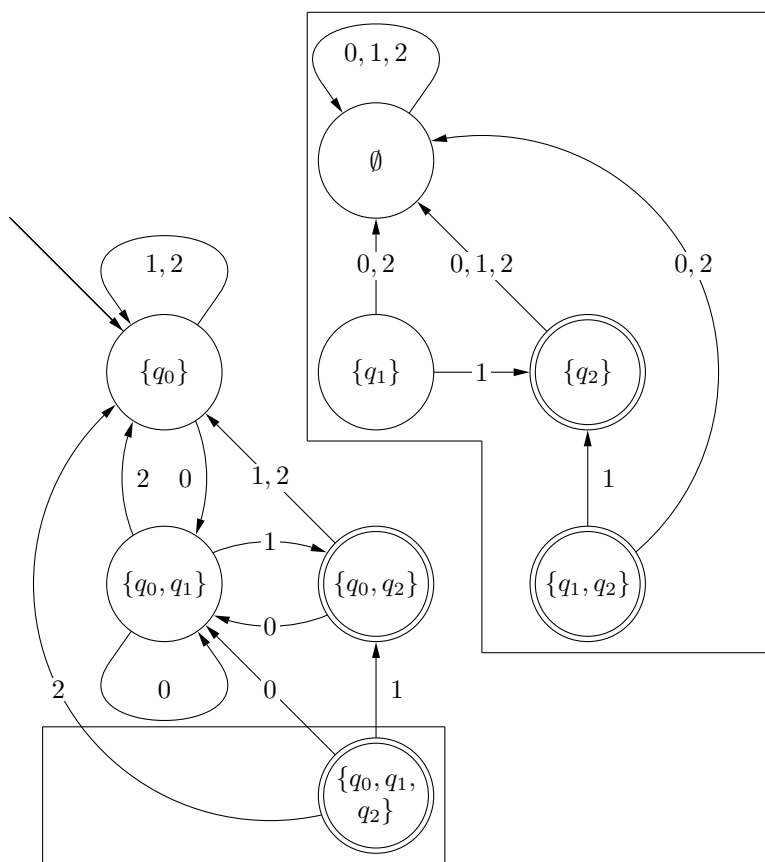
Ende des Beispiels A_{01} (Fortsetzung)

Die im Beweis des Satzes von Rabin und Scott angegebene Konstruktion ist von *exponentiellem* Aufwand, weil der Automat B exponentiell viele Zustände hat, gemessen an der Zustandszahl von A . Es kann gefragt werden, ob es eventuell nach Minimierung noch wesentlich „kleinere“ deterministische Automaten gibt, die die gleiche Sprache wie A akzeptieren. Die Antwort lautet jedoch „im Allgemeinen nein“, denn es gibt Beispiele, wo der im Beweis entstandene DFA nicht weiter vereinfacht werden kann; d.h., wenn der gegebene NFA ungefähr n Zustände besitzt, so benötigt ein äquivalenter DFA im schlechtesten Fall tatsächlich exponentiell viele, also nicht wesentlich weniger als 2^n Zustände.

Beispiel $L_{nondet}(n)$:

Eine deterministisch sehr aufwändig zu akzeptierende Sprache ist $L_{nondet}(n)$ über dem Alphabet $\Sigma = \{0, 1\}$, die für ein beliebiges, aber festes $n \in \mathbb{N}$ folgendermaßen definiert ist: die Wörter von $L_{nondet}(n)$ sind diejenigen 0, 1-Folgen, die an der n ten Stelle von rechts eine 1 stehen haben (und sonst beliebig sind).

Für $L_{nondet}(n)$ kann ein akzeptierender NFA mit $n+1$ Zuständen gefunden werden. Jedoch hat kein DFA, der $L_{nondet}(n)$ akzeptiert, weniger als 2^n Zustände. Der Grund dafür liegt darin, dass ein Automat nicht

Abbildung 4.5: Ein DFA B_{01} mit $L(B_{01}) = L(A_{01}) = L_{01}$

wissen kann, ob eine Eins, die gerade gelesen wird, die „richtige“ Eins ist (genau n Stellen vom Wortende weg). Ein NFA kann einfach „raten“, ob es sich um die richtige Eins handelt. Ein DFA muss sich aber an die letzten n Zeichen, die er gelesen hat, erinnern, um zu entscheiden, ob es sich um die richtige Eins gehandelt hat. Das geht nur mit mindestens 2^n Zuständen.

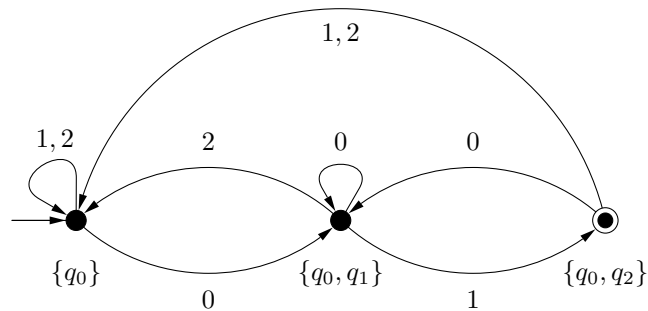
Ende des Beispiels $L_{\text{nondet}}(n)$

Satz 4.2.4 ENDLICH AKZEPTIERBARE SPRACHEN SIND CHOMSKY-3

Sei $L \subseteq \Sigma^*$ eine endlich akzeptierbare Sprache. Dann ist L Chomsky-3.

Beweis: Gegeben sei ein NFA $A = (Q, \Sigma, \delta, q_0, F)$ mit $L = L(A)$. Wir konstruieren eine Grammatik $G = (N, \Sigma, P, S)$ folgendermaßen:

$$\begin{aligned} N &= Q \\ P &= \{q \rightarrow aq' \mid (q, a, q') \in \delta\} \cup \{q \rightarrow \varepsilon \mid q \in F\} \\ S &= q_0. \end{aligned}$$

Abbildung 4.6: Minimierte Version B_{01}^{min} von B_{01}

G ist rechtslinear und es gilt $L(G) = L(A) = L$, und damit ist L eine Chomsky-3-Sprache. ☒ 4.2.4

Beispiel:

Zur Erläuterung dieser Konstruktion mag der in Abbildung 4.4 (rechts) angegebene Automat $A(G_2)$ dienen. Wendet man auf diesen Automaten die im Beweis angegebene Konstruktion an, erhält man die Grammatik G'_2 mit den Produktionen

$$P'_2 = \{S \rightarrow aR, R \rightarrow bS, S \rightarrow aX, X \rightarrow \varepsilon\},$$

also nicht wieder genau die Grammatik G_2 . Durch weitere Umformungen (wie z.B. in Lemma 3.2.3) kann man erreichen, dass als einzige ε -Produktion $S \rightarrow \varepsilon$ vorkommt.

Ende des Beispiels

Als Hauptresultat aus den bis jetzt bewiesenen Sätzen hat man folgende Äquivalenzen: Für eine Sprache $L \subseteq \Sigma^*$ sind alle diese Eigenschaften gleichbedeutend:

- L ist rechtslinear (Chomsky-3);
- L ist endlich akzeptierbar;
- L ist deterministisch endlich akzeptierbar.

Um zu beweisen, dass die rechtslinearen genau die linkslinearen Sprachen linkslinear sind, kann man mit Vorteil die Tatsache verwenden, dass die endlich akzeptierbaren Sprachen unter Spiegelung abgeschlossen sind:

Satz 4.2.5 ENDLICH AKZEPTIERBARE SPRACHEN SIND UNTER SPIEGELUNG ABGESCHLOSSEN

Sei L endlich akzeptierbar. Dann ist auch L^R endlich akzeptierbar.

Beweis: Da L endlich akzeptierbar ist, gibt es einen NFA A , der L akzeptiert. Aus A konstruiere man einen Automaten A' durch Vertauschen der Anfangs- mit den Endzuständen. Offenbar akzeptiert A' die Sprache L^R . ☒ 4.2.5

Satz 4.2.6 RECHTSLINEAR \Leftrightarrow LINKSLINEAR

Eine Sprache $L \subseteq \Sigma^*$ ist rechtslinear genau dann, wenn sie linkslinear ist.

Beweis: „ L rechtslinear $\Rightarrow L$ linkslinear“:

Sei L rechtslinear. Dann gibt es eine rechtslineare Grammatik G mit $L = L(G)$ und der in Lemma 4.2.1 angegebenen speziellen Form, d.h.: alle Produktionen haben entweder die Form $X \rightarrow aY$ oder die Form $X \rightarrow a$ oder die Form $S \rightarrow \varepsilon$.

Man konstruiere daraus eine linkslineare Grammatik G' durch Ersetzen aller Produktionen $X \rightarrow aY$ durch $X \rightarrow Ya$. Offenbar erzeugt G' die Sprache L^R .

Nach Satz 4.2.5 ist L^R endlich akzeptierbar; also gibt es eine rechtslineare Grammatik G'' , die L^R erzeugt. Wie eben gewinnt man daraus eine linkslineare Grammatik G''' , die $(L^R)^R = L$ erzeugt.

„ L linkslinear $\Rightarrow L$ rechtslinear“:

Sei L linkslinear. Dann gibt es eine linkslineare Grammatik G mit $L = L(G)$, so dass alle Produktionen entweder die Form $X \rightarrow Ya$ oder die Form $X \rightarrow a$ oder die Form $S \rightarrow \varepsilon$ haben. (Man benötigt dazu eine Version von Lemma 4.2.1 für linkslineare Sprachen.)

Man konstruiere daraus eine rechtslineare Grammatik G' durch Ersetzen aller Produktionen $X \rightarrow Ya$ durch $X \rightarrow aY$. Offenbar erzeugt G' die Sprache L^R , also ist L^R auch endlich akzeptierbar.

Nach Satz 4.2.5 ist $(L^R)^R$ endlich akzeptierbar; also gibt es eine rechtslineare Grammatik G'' , die $L = (L^R)^R$ erzeugt. □ 4.2.6

Damit sind die vier Spracheigenschaften „rechtslinear“ (d.h. „Chomsky-3“), „linkslinear“, „endlich akzeptierbar“ und „deterministisch endlich akzeptierbar“ als äquivalent nachgewiesen.

4.3 Abschlusseigenschaften

Wir untersuchen jetzt, unter welchen Operationen die Klasse der endlich akzeptierbaren Sprachen abgeschlossen ist. Dazu betrachten wir die Sprachoperationen Vereinigung, Komplement, Durchschnitt, Differenz, Konkatenation und Sternabschluss bzw. Iteration (vergleiche Abschnitt 2.3 sowohl zur Definition dieser Operationen als auch zum Begriff der Abgeschlossenheit einer Sprachklasse).

Satz 4.3.1 ABGESCHLOSSENHEIT ENDLICH AKZEPTIERBARER SPRACHEN

Die Klasse der endlich akzeptierbaren Sprachen ist abgeschlossen unter den Operationen Vereinigung, Komplement, Durchschnitt, Differenz, Konkatenation und Sternabschluss.

Beweis:

Seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ endlich akzeptierbar. Dann gibt es einen DFA $A_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$ mit $L_1 = L(A_1)$ und einen DFA $A_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$ mit $L_2 = L(A_2)$. Ohne Beschränkung der Allgemeinheit kann $Q_1 \cap Q_2 = \emptyset$ angenommen werden. Wir zeigen, dass die Sprachen $L_1 \cup L_2$, $\overline{L_1} = \Sigma_1^* \setminus L_1$, $L_1 \cap L_2$, $L_1 \setminus L_2$, $L_1 \cdot L_2$ und L_1^* allesamt endlich akzeptierbar sind.

Vereinigung: Der neue Automat soll akzeptieren, wenn entweder A_1 oder A_2 akzeptiert. Daher liegt es nahe, einen nichtdeterministischen Automaten A mit zwei Anfangszuständen q_{01} und q_{02} zu konstruieren:

$A_{\cup} = (Q_{\cup}, \Sigma_{\cup}, \delta_{\cup}, q_{0\cup}, F_{\cup})$ mit

$$\begin{aligned} Q_{\cup} &= Q_1 \cup Q_2 \\ \Sigma_{\cup} &= \Sigma_1 \cup \Sigma_2 \\ \delta_{\cup} &= \delta_1 \cup \delta_2 \\ q_{0\cup} &= \{q_{01}, q_{02}\} \\ F_{\cup} &= F_1 \cup F_2. \end{aligned}$$

Die beiden Automaten werden, bildlich gesprochen, „nebeneinander gelegt“. Es ist klar, dass $L(A_{\cup}) = L(A_1) \cup L(A_2)$ gilt.

Komplement: Der neue Automat soll genau dann akzeptieren, wenn der ursprüngliche (d.h. A_1) nicht akzeptiert. Wir konstruieren deshalb einen DFA $A_{compl} = (Q_1, \Sigma_1, \delta_1, q_{01}, F_{compl})$ mit

$$F_{compl} = Q_1 \setminus F_1.$$

A_{compl} ist deterministisch, weil A_1 deterministisch ist. Es gilt für alle $w \in \Sigma_1^*$:

$$\begin{aligned} w \in L(A_{compl}) &\Leftrightarrow (\text{Definition von } L(A_{compl}) \text{ und von } F_{compl}) \\ &\quad \exists q \in Q_1 \setminus F_1 : q_{01} \xrightarrow{w} q \\ &\Leftrightarrow (A_{compl} \text{ ist deterministisch: } \xrightarrow{w} \text{ linkstotal für „}\Leftarrow\text{“, } \xrightarrow{w} \text{ rechtseindeutig für „}\Rightarrow\text{“}) \\ &\quad \neg \exists q \in F_1 : q_{01} \xrightarrow{w} q \\ &\Leftrightarrow (\text{Definition von } L(A_1)) \\ &\quad w \notin L(A_1) \\ &\Leftrightarrow (L(A_1) = L_1, \overline{L_1} = \Sigma_1^* \setminus L_1) \\ &\quad w \in \overline{L_1}. \end{aligned}$$

Also gilt $L(A_{compl}) = \overline{L_1}$.

Die Korrektheit dieser Konstruktion hängt wesentlich davon ab, dass A_1 deterministisch ist. Hat man zuerst einen nichtdeterministischen Automaten A und möchte einen Automaten konstruieren, der die Komplementmenge von $L(A)$ erkennt, sind dazu im Allgemeinen zwei Schritte nötig: erst muss A gemäß Satz 4.2.3 in einen deterministischen Automaten umgewandelt werden und dann erst kann die obige Konstruktion angewandt werden. Der erste dieser beiden Schritte ist im Allgemeinen sehr (d.h.: exponentiell) zeitaufwändig.

Durchschnitt: Es gilt $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Die endliche Akzeptierbarkeit von $L_1 \cap L_2$ folgt also aus den beiden bereits bewiesenen Behauptungen. Auch diese Konstruktion kann wegen der (sogar dreifachen, davon im schlimmsten Fall zweimal exponentiellen) Benutzung der Komplementkonstruktion sehr aufwändig sein. Wir geben weiter unten eine direkte und effizientere Konstruktion an. Da $L_1 \cap L_2$ eine Sprache über $\Sigma_1 \cap \Sigma_2$ ist, dürfen aus dem sich ergebenden Automaten alle Pfeile \xrightarrow{a} mit $a \notin \Sigma_1 \cap \Sigma_2$ entfernt werden; der dann resultierende Automat hat das Alphabet $\Sigma_1 \cap \Sigma_2$.

Differenz: Es gilt $L_1 \setminus L_2 = L_1 \cap \overline{L_2} = \overline{\overline{L_1} \cup L_2}$. Die endliche Akzeptierbarkeit von $L_1 \setminus L_2$ folgt also aus den bereits bewiesenen Behauptungen. Hier geht die Komplementkonstruktion wesentlich ein.

Konkatenation $L_1 \cdot L_2$:

Die Grundidee ist hier, die beiden Automaten „in Serie zu schalten“. Dazu hängen wir an die Endzustände des ersten Automaten Ausgangspfeile an, die sich genau so verhalten, wie die Pfeile, die vom

Anfangszustand des zweiten Automaten ausgehen: $A_{konk} = (Q_{konk}, \Sigma_{konk}, \delta_{konk}, Q_{0konk}, F_{konk})$ mit

$$\begin{aligned} Q_{konk} &= Q_1 \cup Q_2 \\ \Sigma_{konk} &= \Sigma_1 \cup \Sigma_2 \\ \delta_{konk} &= \delta_1 \cup \{(q, a, q') \mid q \in F_1 \wedge (q_{02}, a, q') \in \delta_2\} \cup \delta_2 \\ Q_{0konk} &= \{q_{01}\} \\ F_{konk} &= \begin{cases} F_2 & \text{falls } \varepsilon \notin L(A_2) \\ F_1 \cup F_2 & \text{falls } \varepsilon \in L(A_2) \end{cases} \end{aligned}$$

A_{konk} ist im Normalfall nichtdeterministisch, da (außer im Fall $F_1 = \emptyset$) Pfeile zu einem deterministischen Automaten hinzugefügt werden, und es gilt $L(A_{konk}) = L(A_1) \cdot L(A_2)$.

Sternabschluss:

Der neue Automat soll Wörter akzeptieren, die Konkationen von Wörtern sind, die von A_1 akzeptiert werden. Es kann die gleiche Idee wie bei der Konkation verwendet werden, außer dass dieses Mal A_1 mit sich selbst in Serie geschaltet wird. Wenn A_1 nicht schon selbst das leere Wort akzeptiert, muss noch dafür Sorge getragen werden, dass das leere Wort akzeptiert wird (denn $\varepsilon \in L(A_1)^*$, egal ob $\varepsilon \in L(A)$ oder nicht). Dazu addieren wir einen neuen Anfangszustand $q_{\varepsilon*} \notin Q_1$, der auch Endzustand ist und dessen einzige Funktion es ist, das leere Wort zu akzeptieren: $A_* = (Q_*, \Sigma_*, \delta_*, Q_{0*}, F_*)$ mit

$$\begin{aligned} Q_* &= Q_1 \cup \{q_{\varepsilon*}\} \\ \Sigma_* &= \Sigma_1 \\ \delta_* &= \delta_1 \cup \{(q, a, q') \mid q \in F_1 \wedge (q_{01}, a, q') \in \delta_1\} \\ Q_{0*} &= \{q_{01}, q_{\varepsilon*}\} \\ F_* &= F_1 \cup \{q_{\varepsilon*}\}. \end{aligned}$$

Es gilt $L(A_*) = L_1^*$.

□ 4.3.1

Bemerkung: Für die Vereinigung und den Durchschnitt sind die oben angegebenen Konstruktionen nicht optimal, da – sofern ein resultierender *deterministischer* Automat gewünscht wird – in beiden Fällen die aufwändige Konstruktion aus Satz 4.2.3 benutzt werden muss. Wenn die beiden Ursprungsautomaten auf dem gleichen Alphabet definiert sind, gibt es eine einfache alternative und in vielen Fällen „schnellere“ direkte Konstruktion von akzeptierenden DFAs.

Seien dazu A_1 und A_2 wie im obigen Beweis, mit Alphabeten $\Sigma_1 = \Sigma_2 = \Sigma$. Dann betrachten wir auf dem Kartesischen Produkt $Q = Q_1 \times Q_2$ der beiden Zustandsmengen die folgende Transitionsrelation $\longrightarrow_{\times} \subseteq (Q \times \Sigma) \times Q$: für alle $q_1, q'_1 \in Q_1$ und $q_2, q'_2 \in Q_2$ und $a \in \Sigma$ gelte

$$(q_1, q_2) \xrightarrow{a}_{\times} (q'_1, q'_2) \quad \text{genau dann, wenn} \quad q_1 \xrightarrow{a}_{1} q'_1 \quad \text{und} \quad q_2 \xrightarrow{a}_{2} q'_2.$$

Die Relation \xrightarrow{a}_{\times} modelliert das *synchrone parallele Fortschreiten* der beiden Einzelautomaten A_1 und A_2 beim Einlesen des Symbols a . Nun betrachte man die beiden DFAs

$$\begin{aligned} B_{\cup} &= (Q, \Sigma, \longrightarrow_{\times}, (q_{01}, q_{02}), F_{\cup}) \quad \text{mit} \quad F_{\cup} = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \text{ oder } q_2 \in F_2\} \\ \text{und } B_{\cap} &= (Q, \Sigma, \longrightarrow_{\times}, (q_{01}, q_{02}), F_{\cap}) \quad \text{mit} \quad F_{\cap} = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \text{ und } q_2 \in F_2\}. \end{aligned}$$

Dann gilt $L(B_\cup) = L_1 \cup L_2$ und $L(B_\cap) = L_1 \cap L_2$. Wir beweisen die zweite dieser Behauptungen: Für beliebiges $w \in \Sigma^*$ gilt:

$$\begin{aligned}
 w \in L(B_\cap) &\Leftrightarrow (\text{Definition von } L(B_\cap)) \\
 &\quad \exists (q_1, q_2) \in F_\cap: (q_{01}, q_{02}) \xrightarrow{w}_\times (q_1, q_2) \\
 &\Leftrightarrow (\text{Definitionen von } F_\cap \text{ und von } \xrightarrow{w}_\times) \\
 &\quad \exists q_1 \in F_1, q_2 \in F_2: q_{01} \xrightarrow{w}_1 q_1 \text{ und } q_{02} \xrightarrow{w}_2 q_2 \\
 &\Leftrightarrow (\text{Definitionen von } L(A_1) \text{ und von } L(A_2)) \\
 &\quad w \in L(A_1) \text{ und } w \in L(A_2) \\
 &\Leftrightarrow (L(A_1) = L_1 \text{ und } L(A_2) = L_2) \\
 &\quad w \in L_1 \cap L_2.
 \end{aligned}$$

Mit etwas Mehraufwand kann diese Konstruktion auf den Fall $\Sigma_1 \neq \Sigma_2$ verallgemeinert werden. Für B_\cap wird dazu $\Sigma = \Sigma_1 \cap \Sigma_2$ festgelegt; die obige Konstruktion kann direkt verwendet werden. Für B_\cup können für Buchstaben $a \in \Sigma_1 \setminus \Sigma_2$ geeignete Senken in A_2 , für Buchstaben $a \in \Sigma_2 \setminus \Sigma_1$ geeignete Senken in A_1 angelegt werden. (Senken sind Zustände, die nie wieder verlassen werden können.) Danach kann $\Sigma = \Sigma_1 \cup \Sigma_2$ festgelegt und die obige Konstruktion verwendet werden.

4.4 Reguläre Ausdrücke

Reguläre Ausdrücke sind Formeln, mit denen Sprachen auf endliche (oft sehr kompakte) Art beschrieben werden können. Formal ist ein regulärer Ausdruck ein syntaktisches Objekt, dem eine Sprache eindeutig zugeordnet wird. Demzufolge führen wir reguläre Ausdrücke in zwei Schritten ein: zuerst ihre *Syntax*, dann ihre *Semantik*. Die Zuordnung einer Sprache $L(\rho)$ (semantisches Objekt) zu einem regulären Ausdruck ρ (syntaktisches Objekt) geschieht durch syntaktische Induktion.

Es wird sich herausstellen, dass die so charakterisierten Sprachen wieder genau die Chomsky-3-Sprachen sind.

Definition 4.4.1 SYNTAX REGULÄRER AUSDRÜCKE ÜBER Σ

Sei Σ ein Alphabet. Induktiv definieren wir:

- *Beginn*: \emptyset und ε sind reguläre Ausdrücke über Σ .
Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck über Σ .
- *Schritt*: Wenn ρ, ρ_1, ρ_2 reguläre Ausdrücke über Σ sind, so auch $(\rho_1 \cup \rho_2)$, $(\rho_1 \cdot \rho_2)$ und ρ^* .
- *Abschluss*: Sonst ist nichts ein regulärer Ausdruck über Σ . ☒ 4.4.1

Definition 4.4.2 SEMANTIK REGULÄRER AUSDRÜCKE

Sei ρ ein regulärer Ausdruck über Σ . Wir ordnen ρ eine Sprache $L(\rho)$ zu:

$$\begin{aligned}
 L(\emptyset) &= \emptyset \\
 L(\varepsilon) &= \{\varepsilon\} \\
 L(a) &= \{a\} \quad (\text{für alle } a \in \Sigma) \\
 L(\rho_1 \cup \rho_2) &= L(\rho_1) \cup L(\rho_2) \\
 L(\rho_1 \cdot \rho_2) &= L(\rho_1) \cdot L(\rho_2) \\
 L(\rho^*) &= (L(\rho))^*.
 \end{aligned}
 \tag*{☒ 4.4.2}$$

In diesen definierenden Gleichungen sind links die syntaktischen Möglichkeiten für ρ aufgeführt, rechts aber Operationen auf Sprachen (siehe Abschnitt 2.3). Die syntaktischen Zeichen wurden gerade so gewählt, dass sie den semantischen Operationen entsprechen. Manchmal findet man statt $\rho_1 \cup \rho_2$ auch die syntaktischen Formen $\rho_1 \mid \rho_2$ (so im **Unix**-Kommando **grep**) oder $\rho_1 + \rho_2$ oder gar (ρ_1, ρ_2) (wie bei **Ebay**), und statt $\rho_1 \cdot \rho_2$ die syntaktische Form $\rho_1\rho_2$ (ebenfalls im Kommando **grep**).

Definition 4.4.3 REGULÄRE SPRACHEN

Eine Sprache $L \subseteq \Sigma^*$ heißt *regulär*, falls es einen regulären Ausdruck ρ über Σ gibt mit $L = L(\rho)$.

☐ 4.4.3

Zur Klammereinsparung bei regulären Ausdrücken vereinbaren wir:

* bindet stärker als \cdot und \cdot bindet stärker als \cup .

Außerdem lassen wir äußere Klammern weg und nutzen die Assoziativität von \cdot und \cup aus.

Beispiel:

1. Sei $\Sigma = \{0, 1, 2\}$. Dann wird die Sprache $L_{01} = \{w01 \mid w \in \{0, 1, 2\}^*\}$ der Wörter mit Suffix 01 durch den regulären Ausdruck

$$\rho_{01} = (0 \cup 1 \cup 2)^* \cdot 0 \cdot 1$$

beschrieben: $L(\rho_{01}) = L_{01}$.

2. Jede endliche Sprache ist regulär, denn für $L = \{w_1, \dots, w_m\}$ ist $\rho = (w_1 \cup \dots \cup w_m)$ ein regulärer Ausdruck mit $L(\rho) = L$ (wobei für $m=0$, *per definitionem*, $(w_1 \cup \dots \cup w_m) = \emptyset$ gesetzt wird).

Ende des Beispiels

Wir zeigen allgemein:

Satz 4.4.4 KLEENE

Eine Sprache ist genau dann regulär, wenn sie endlich akzeptierbar ist.

Beweis: Wir betrachten eine Sprache $L \subseteq \Sigma^*$.

„ \Rightarrow “: Es gelte $L = L(\rho)$ für einen regulären Ausdruck ρ über Σ . Wir zeigen durch Induktion über die Struktur von ρ : $L(\rho)$ ist endlich akzeptierbar.

Induktionsanfang: Die Sprachen $L(\emptyset)$, $L(\varepsilon)$ und $L(a)$ für $a \in \Sigma$ sind endlich akzeptierbar: $L(\emptyset)$ durch einen Automaten ohne Endzustand; $L(\varepsilon)$ durch einen Automaten mit nur einem Zustand, der gleichzeitig Anfangs- und Endzustand ist (und ohne Zustandsübergänge); und $L(a)$ durch einen Automaten mit zwei Zuständen und einem einzigen Pfeil.

Induktionsschritt: Seien $L(\rho)$, $L(\rho_1)$ und $L(\rho_2)$ bereits endlich akzeptierbar. Dann sind auch die Sprachen $L(\rho_1 \cup \rho_2)$, $L(\rho_1 \cdot \rho_2)$ und $L(\rho^*)$ endlich akzeptierbar, weil die Klasse der endlich akzeptierbaren Sprachen gegenüber Vereinigung, Konkatenation und Sternabschluss abgeschlossen ist (Satz 4.3.1).

„ \Leftarrow “: Es gelte $L = L(A)$ für einen NFA A mit n Zuständen. O.B.d.A. gelte $A = (\Sigma, Q, \delta, q_0, F)$ mit $Q = \{1, \dots, n\}$, $q_0 = 1$ und $F = \{j_1, \dots, j_m\}$. Wir „bauen“ einen regulären Ausdruck ρ mit $L(A) = L(\rho)$, indem wir die (willkürliche aber feste) Nummerierung der Zustandsmenge ausnutzen.

Für $i, j \in \{1, \dots, n\}$ und $k \in \{0, 1, \dots, n\}$ definieren wir folgende Sprachen $L_{i,j}^k$:

$$L_{i,j}^k = \left\{ w \in \Sigma^* \mid \begin{array}{l} i \xrightarrow{w} j \\ \wedge \forall u \in \Sigma^* \forall l \in Q: ((\exists v: \varepsilon \neq v \neq w \wedge uv = w) \wedge i \xrightarrow{u} l) \Rightarrow l \leq k \end{array} \right\}.$$

Die Sprache $L_{i,j}^k$ besteht also aus allen Wörtern w , die beim Einlesen den Automaten A vom Zustand i in den Zustand j überführen, so dass zwischendurch nur Zustände auftreten, deren Nummer höchstens k ist. Speziell mit $i = 1$ und $k = n$ kann $L(A)$ so ausgedrückt werden:

$$L = L(A) = L_{1,j_1}^n \cup \dots \cup L_{1,j_m}^n.$$

Wir zeigen nun, dass die Sprachen $L_{i,j}^k$ äquivalent auch durch Induktion über k definiert werden können. Es gilt nämlich:

$k = 0$: Für Wörter aus $L_{i,j}^0$ darf überhaupt kein Zwischenzustand benutzt werden. Also gilt:

$$L_{i,j}^0 = \begin{cases} \{a \in \Sigma \mid i \xrightarrow{a} j\} & \text{falls } i \neq j \\ \{\varepsilon\} \cup \{a \in \Sigma \mid i \xrightarrow{a} j\} & \text{falls } i = j. \end{cases}$$

$k \rightsquigarrow k+1 \leq n$: Dann gilt für alle $i, j \in \{1, \dots, n\}$:

$$L_{i,j}^{k+1} = L_{i,j}^k \cup (L_{i,k+1}^k \cdot (L_{k+1,k+1}^k)^* \cdot L_{k+1,j}^k),$$

denn um vom Zustand i aus den Zustand j zu erreichen, wird entweder der Zwischenzustand $k+1$ nicht benötigt, dann reicht $L_{i,j}^k$ zur Beschreibung aus; oder der Zustand $k+1$ wird ein- oder mehrfach als Zwischenzustand durchlaufen, dann wird $(L_{i,k+1}^k \cdot (L_{k+1,k+1}^k)^* \cdot L_{k+1,j}^k)$ zur Beschreibung verwendet.

Man zeigt leicht mit Induktion nach k , dass die Sprachen $L_{i,j}^k$ alle regulär sind.

$k = 0$: Die Sprachen $L_{i,j}^0$ sind endlich und damit regulär.

$k \rightsquigarrow k+1 \leq n$: Seien für alle $i, j \in \{1, \dots, n\}$ die Sprachen $L_{i,j}^k$ regulär, und sei $\rho_{i,j}^k$ ein regulärer Ausdruck mit $L(\rho_{i,j}^k) = L_{i,j}^k$. Dann ist – nach der Syntax regulärer Ausdrücke – die Formel

$$\rho_{i,j}^{k+1} = \rho_{i,j}^k \cup (\rho_{i,k+1}^k \cdot (\rho_{k+1,k+1}^k)^* \cdot \rho_{k+1,j}^k)$$

ein regulärer Ausdruck für $L_{i,j}^{k+1}$ und nach obiger Formel für $L_{i,j}^{k+1}$ gilt $L(\rho_{i,j}^{k+1}) = L_{i,j}^{k+1}$.

Nach der Syntax regulärer Ausdrücke ist auch

$$\rho = \rho_{1,j_1}^n \cup \dots \cup \rho_{1,j_m}^n$$

ein regulärer Ausdruck und aus $L = L(A) = L_{1,j_1}^n \cup \dots \cup L_{1,j_m}^n$ folgt $L(\rho) = L = L(A)$. □ 4.4.4

Beispiel:

Zur Konstruktion im zweiten Teil dieses Beweises betrachten wir den Automaten

$$A_{Kleene_bsp} = (\{1, 2\}, \{a, b\}, \{(1, a, 2), (2, b, 1)\}, 1, \{2\}),$$

der in Abbildung 4.7 gezeigt ist.

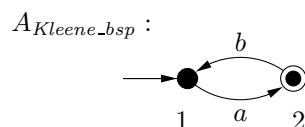


Abbildung 4.7: Ein Beispielautomat zur Konstruktion der $L_{i,j}^k$ im Beweis von Satz 4.4.4.

Dann berechnen wir zunächst:

$$\begin{array}{llll}
 L_{1,1}^0 = \{\varepsilon\} & L_{1,2}^0 = \{a\} & L_{2,2}^0 = \{\varepsilon\} & L_{2,1}^0 = \{b\} \\
 L_{1,1}^1 = \{\varepsilon\} & L_{1,2}^1 = \{a\} & L_{2,2}^1 = \{\varepsilon, ba\} & L_{2,1}^1 = \{b\} \\
 L_{1,1}^2 = \{\varepsilon, ab, abab, \dots\} & L_{1,2}^2 = \{a, aba, ababa, \dots\} & L_{2,2}^2 = \{\varepsilon, ba, baba, \dots\} & L_{2,1}^2 = \{b, bab, \dots\}.
 \end{array}$$

Da wegen $F = \{2\}$ in A_{Kleene_bsp} die Gleichung $L(A_{Kleene_bsp}) = L(\rho_{1,2}^2)$ gilt, betrachten wir:

$$\rho_{1,2}^2 = \rho_{1,2}^1 \cup (\rho_{1,2}^1 (\rho_{2,2}^1)^* \rho_{2,2}^1) = a \cup a(\varepsilon \cup ba)^*(\varepsilon \cup ba).$$

Also ist $a \cup a(\varepsilon \cup ba)^*(\varepsilon \cup ba)$ ein regulärer Ausdruck, der die von A_{Kleene_bsp} akzeptierte Sprache beschreibt. Diesen kann man allerdings wegen:

$$L(a \cup a(\varepsilon \cup ba)^*(\varepsilon \cup ba)) = L(a \cup a(ba)^*) = L(a(ba)^*),$$

was leicht zu sehen ist, noch vereinfachen. Insgesamt gilt:

$$L(A_{Kleene_bsp}) = L(a(ba)^*) = \{a(ba)^n \mid n \in \mathbb{N}\}.$$

Ende des Beispiels

Das Hauptresultat dieses Abschnitts fügt zu den Äquivalenzen, die am Ende des Abschnitts 4.2 zusammengefasst wurden, eine weitere hinzu. Wegen guter Einprägsamkeit verwenden wir ab jetzt den Terminus „reguläre Sprache“ statt „endlich akzeptierbare“ bzw. „rechtslineare“ bzw. „linkslineare“ Sprache.

4.5 Struktureigenschaften regulärer Sprachen

Bis hierher haben wir auf verschiedene Weisen die regulären Sprachen charakterisiert. Jetzt wenden wir uns der Frage zu, welche Sprachen *nicht* regulär sind. Beispielsweise ist – wie wir bald zeigen werden – die kontextfreie Sprache

$$L_1 = \{a^n b^n \mid n \in \mathbb{N}\},$$

in der in jedem Wort genau gleich viele *as* und *bs* vorkommen, nicht regulär.

Die Aussage, dass L_1 nicht regulär ist, bedeutet ja eine Aussage über alle möglichen Erzeugendensysteme für L_1 . Z.B. ist damit ausgesagt, dass es keinen endlichen Automaten gibt, der diese Sprache erzeugt. Um dies streng zu beweisen, ist es notwendig, die regulären Sprachen strukturell zu analysieren.

Wir nutzen zunächst die Endlichkeit der Zustandsmengen von endlichen Automaten aus, um wichtige strukturelle Eigenschaften regulärer Sprachen abzuleiten. Im Abschnitt 4.5.1 wird das sogenannte *Pumping-Lemma* (oder *Schleifenlemma*) betrachtet. Dieses Lemma liefert eine notwendige Bedingung dafür, dass eine Sprache regulär ist. In Abschnitt 4.5.2 wird eine – nur auf Sprachen definierte – exakte Bedingung dafür angegeben, dass eine Sprache regulär ist. Beide Bedingungen, die notwendige und die exakte, liefern auch Beweise für die Nicht-Regularität der Sprache L_1 . Schließlich betrachten wir in Abschnitt 4.5.3 eine wichtige Konsequenz aus der exakten Strukturbedingung, nämlich die Tatsache, dass man DFAs minimieren kann und dass die Minimalform eines DFA eindeutig ist.

Für den ganzen Abschnitt sei Σ ein beliebiges, festes Alphabet.

4.5.1 Das Pumping-Lemma für reguläre Sprachen

Das folgende Lemma behauptet die Existenz einer natürlichen Zahl p mit den angegebenen Eigenschaften für jede reguläre Sprache L . Dabei hängt p von L ab. Die genaue Art der Erzeugung von L geht nicht in die Formulierung des Lemmas ein.

Lemma 4.5.1 PUMPING-LEMMA (SCHLEIFENLEMMA) FÜR REGULÄRE SPRACHEN

Zu jeder regulären Sprache $L \subseteq \Sigma^*$ existiert eine Zahl $p \in \mathbb{N}$, so dass es für alle Wörter $x \in L$ mit $|x| \geq p$ eine Zerlegung $x = uvw$ mit den folgenden drei Eigenschaften gibt:

- (1) $1 \leq |v|$
- (2) $|uv| \leq p$
- (3) $\forall i \in \mathbb{N}: uv^i w \in L$.

D.h.: sofern x „lang genug“ ist ($|x| \geq p$), kann das nicht leere (1) Teilwort v , das in einem Präfix der Länge $\leq p$ von x liegt (2), beliebig oft „aufgepumpt“ werden ((3): $\forall i \dots v^i \dots$), ohne dass die reguläre Sprache L verlassen wird (3).

Beweis: Sei $L \subseteq \Sigma^*$ regulär. Dann gibt es einen DFA $A = (Q, \Sigma, \delta, q_0, F)$ mit $L = L(A)$. Wir wählen $p = |Q|$ und betrachten ein Wort $x \in L$ mit $|x| \geq p$. Dann muss A beim Akzeptieren von x mindestens einen Zustand zweimal durchlaufen, denn ein Pfad mit $\geq p$ Kanten kann nicht durch einen Graphen mit p Knoten laufen, ohne mindestens einen Knoten doppelt zu berühren. Genauer: sei $x = a_1 \dots a_r$ mit $r \geq p$. Dann wird x durch eine Folge

$$q_0 \xrightarrow{a_1} q^1, q^1 \xrightarrow{a_2} q^2, \dots, q^{r-1} \xrightarrow{a_r} q^r \in F$$

akzeptiert, wobei es Indices $j < r$ und $k \leq r$ mit $j < k$ und $q^j = q^k$ gibt. Wir wählen j und k derart, dass die Zustände q^0 bis q^{k-1} alle verschieden sind (dazu wählen wir das erste Vorkommen eines wiederholten Zustands).

Wir setzen nun $u = a_1 \dots a_j$, $v = a_{j+1} \dots a_k$ und $w = a_{k+1} \dots a_r$.

Dann gilt $v \neq \varepsilon$ wegen $j < k$ und $|uv| \leq p$ weil die q^0 bis q^{k-1} alle verschieden sind; d.h., es gelten die Eigenschaften (1) und (2) des Lemmas. Außerdem kann der Automat A die $q^j = q^k$ -Schleife beim Akzeptieren beliebig oft (inklusive keinmal) durchlaufen. Also gilt für alle $i \in \mathbb{N}$ (inklusive $i = 0$): $uv^i w \in L$; d.h., es gilt die Eigenschaft (3) des Lemmas. ☒ 4.5.1

Dieses Lemma kann man typischerweise für den Nachweis benutzen, dass eine bestimmte Sprache nicht regulär ist. Ein Beispiel ist die Sprache L_1 :

Behauptung: Die Sprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ ist nicht regulär.

Beweis: Durch Widerspruch. Nehmen wir an, dass L_1 regulär ist. Dann gibt es nach dem Pumping-Lemma eine Zahl $p \in \mathbb{N}$ mit den dort genannten Eigenschaften (1) bis (3). Betrachte jetzt speziell das Wort $x = a^p b^p$, das nach der Definition von L_1 in L_1 liegt. Da $|x| \geq p$ gilt, lässt sich x zerlegen in $x = uvw$ mit (1) $v \neq \varepsilon$ und (2) $|uv| \leq p$, so dass (3) für alle $i \in \mathbb{N}$ gilt: $uv^i w \in L_1$. Wegen $|uv| \leq p$ und $x = uvw = a^p b^p$ bestehen u und v nur aus Buchstaben a . Das Wort $uv^0 w$, das nach (3) in L_1 liegt, ist deswegen das Wort $uw = a^{n-|v|} b^n$, das wegen (1) mindestens ein a weniger als b s enthält. Nach Definition von L_1 liegt dieses Wort jedoch nicht in L_1 , Widerspruch! Also war die Annahme falsch, und L_1 ist nicht regulär. ☒ Behauptung

Man hätte in diesem Beweis statt $uv^0 w$ auch das Wort $uv^2 w$ betrachten können, das mindestens ein a mehr als b s enthält, deswegen auch nicht in L_1 liegt und so den Widerspruch zu (3) liefert.

Um die Wörter der Sprache L_1 zu erkennen, muss man, intuitiv gesprochen, die Anzahl der a s abzählen und dann mit der Anzahl der b s vergleichen. Dass diese Sprache nicht regulär ist, beruht intuitiv darauf, dass endliche Automaten nicht „unbeschränkt“ (sondern nur „von 0 bis $|Q|-1$ “) zählen können.

4.5.2 Sprachkongruenzen

Eine charakteristische, exakte, also notwendige und hinreichende, Bedingung für die Regularität von Sprachen $L \subseteq \Sigma^*$ erhalten wir durch Betrachtung der folgenden Relation:

Definition 4.5.2 NERODE-RECHTSKONGRUENZ

Sei $L \subseteq \Sigma^*$ eine Sprache. Die *Nerode-Rechtskongruenz* von L ist eine Relation $\equiv_L \subseteq (\Sigma^* \times \Sigma^*)$, die wie folgt definiert ist: für $u, v \in \Sigma^*$:

$$u \equiv_L v \text{ genau dann, wenn für alle } w \in \Sigma^* \text{ gilt: } uw \in L \Leftrightarrow vw \in L.$$

☒ 4.5.2

Es gilt also $u \equiv_L v$, wenn sich u und v bezüglich Rechtsverlängerungen gleich verhalten: entweder macht die Verlängerung aus beiden Wörter in L , oder sie macht aus beiden Wörter nicht in L .

Der Name Rechtskongruenz ist durch folgende leicht zu beweisende Eigenschaften gerechtfertigt:

1. \equiv_L ist eine Äquivalenzrelation auf Σ^* , also reflexiv, symmetrisch und transitiv.
2. \equiv_L ist verträglich mit der Konkatination von rechts, d.h. aus $u \equiv_L v$ folgt $ux \equiv_L vx$ für alle $x \in \Sigma^*$.

Beispiel: Wir betrachten nochmals $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$. Folgendes sind die Äquivalenzklassen von \equiv_{L_1} :

$$\begin{aligned} [\varepsilon] &= \{\varepsilon\} \\ [a] &= \{a\} \\ [aab] &= \{aab, aaabb, \dots\} \\ [aa] &= \{aa\} \\ [aaab] &= \{aaab, aaaabb, \dots\} \\ &\vdots \\ [a^k] &= \{a^k\} \\ [a^{k+1}b] &= \{a^{l+1}b^{l+1-k} \mid l \geq k\} \\ &\vdots \\ [ab] &= L_1 \setminus \{\varepsilon\} \\ [b] &= \text{alle übrigen Wörter.} \end{aligned}$$

L_1 hat also unendlich viele \equiv_{L_1} -Äquivalenzklassen.

Ende des Beispiels

Besonders interessant ist der Fall, dass \equiv_L nur endlich viele Äquivalenzklassen hat. Nach Abschnitt 2.1.2 nennen wir die Anzahl dieser Klassen den Index von \equiv_L .

Satz 4.5.3 MYHILL UND NERODE

Eine Sprache $L \subseteq \Sigma^*$ ist genau dann regulär, wenn \equiv_L einen endlichen Index hat.

Beweis:

„ \Rightarrow “: Sei L regulär. Dann gilt $L = L(A)$ für einen DFA $A = (Q, \Sigma, \delta, q_0, F)$. Wir führen die folgende Relation $\equiv_A \subseteq (\Sigma^* \times \Sigma^*)$ ein:

für $u, v \in \Sigma^*$ sei $u \equiv_A v$ genau dann, wenn es ein $q \in Q$ mit $q_0 \xrightarrow{u} q$ und $q_0 \xrightarrow{v} q$ gibt,

d.h. falls die Eingaben u und v den Automaten A von q_0 aus in den selben Zustand q überführen. Weil A deterministisch ist, ist q (durch q_0 und u bzw. durch q_0 und v) eindeutig bestimmt. Daher ist \equiv_A eine Äquivalenzrelation auf Σ^* . Aber \equiv_A ist auch eine Verfeinerung von \equiv_L , d.h., es gilt $\equiv_A \subseteq \equiv_L$. Sei nämlich $u \equiv_A v$. Dann gilt für ein beliebiges Wort $w \in \Sigma^*$:

$$\begin{aligned} uw \in L &\Leftrightarrow (\text{Definition von } L) \\ &\quad \exists q \in Q \exists q' \in F: q_0 \xrightarrow{u} q \xrightarrow{w} q' \\ &\Leftrightarrow (\text{wegen } u \equiv_A v) \\ &\quad \exists q \in Q \exists q' \in F: q_0 \xrightarrow{v} q \xrightarrow{w} q' \\ &\Leftrightarrow (\text{Definition von } L) \\ &\quad vw \in L. \end{aligned}$$

D.h.: es gilt auch $u \equiv_L v$. Es gibt also mindestens so viele Äquivalenzklassen von \equiv_A wie von \equiv_L . Andererseits hat \equiv_A einen endlichen Index, da \equiv_A so viele Äquivalenzklassen hat, wie es Zustände gibt,

die von q_0 aus erreichbar sind. Insgesamt gilt:

$$\text{Index von } \equiv_L \leq \text{Index von } \equiv_A \leq |Q|$$

und damit hat auch \equiv_L einen endlichen Index.

„ \Leftarrow “: Sei $L \subseteq \Sigma^*$ eine Sprache und sei $k \in \mathbb{N}$ der endliche Index von \equiv_L . Wir wählen k Wörter $u_1, \dots, u_k \in \Sigma^*$ als Repräsentanten der k Äquivalenzklassen von \equiv_L . Dann ist Σ^* als disjunkte Vereinigung folgendermaßen darstellbar:

$$\Sigma^* = [u_1] \cup \dots \cup [u_k] \quad \text{mit} \quad [u_i] \cap [u_j] = \emptyset \quad \text{für} \quad i \neq j.$$

Insbesondere gibt es für jedes Wort $w \in \Sigma^*$ genau ein $i \in \{1, \dots, k\}$ mit $[w] = [u_i]$.

Wir konstruieren jetzt einen DFA $A_L = (Q_L, \Sigma, \delta_L, q_{0L}, F_L)$ mit:

$$\begin{aligned} Q_L &= \{[u_1], \dots, [u_k]\}, \\ \delta_L([u_i], a) &= [u_i a] \quad \text{für alle } i \text{ mit } 1 \leq i \leq k \text{ und für alle } a \in \Sigma \\ q_{0L} &= [\varepsilon], \\ F_L &= \{[u_j] \mid u_j \in L\}. \end{aligned}$$

A_L heißt der *Äquivalenzklassen-Automat* von L .

Wir zeigen zunächst durch Induktion über $|w|$, dass für alle Wörter $w \in \Sigma^*$ gilt:

$$[\varepsilon] \xrightarrow{w}_L [w]. \tag{4.1}$$

Falls $|w| = 0$, dann $w = \varepsilon$ und $[\varepsilon] \xrightarrow{\varepsilon}_L [\varepsilon]$ nach Definition eines endlichen Automaten. Falls $|w| \neq 0$, dann $w = w'a$, wobei die Induktionshypothese auf w' anwendbar ist. Demnach gilt $[\varepsilon] \xrightarrow{w'}_L [w']$. Sei u_i derjenige Repräsentant mit $[w'] = [u_i]$. Laut Definition von δ_L gilt $\delta_L([u_i], a) = [u_i a]$. Wegen $w' \equiv_L u_i$ und der Rechtskongruenzeigenschaft von \equiv_L gilt $w'a \equiv_L u_i a$. Da $w'a = w$, folgt daraus $[w] = [u_i a]$ und damit auch $[\varepsilon] \xrightarrow{w'}_L [w'] \xrightarrow{a}_L [w]$, mithin $[\varepsilon] \xrightarrow{w}_L [w]$.

Damit gilt dann:

$$\begin{aligned} w \in L(A_L) &\Leftrightarrow (\text{Definition von } L(A_L)) \\ &\quad \exists [u_j] \in F_L: [\varepsilon] \xrightarrow{w}_L [u_j] \\ &\Leftrightarrow (\Rightarrow \text{ wegen (4.1) und Determinismus, } \Leftarrow \text{ wegen (4.1) und } [u_j] = [w]) \\ &\quad \exists u_j \in L: [u_j] = [w] \\ &\Leftrightarrow (\text{Definition von } F_L; \text{ und wegen } w \equiv_L u_j \text{ gilt } w \in L \Leftrightarrow u_j \in L) \\ &\quad w \in L. \end{aligned}$$

Also akzeptiert A_L die Sprache L . Daher ist L regulär. ☒ 4.5.3

Intuitiv gesprochen, speichert der Äquivalenzklassen-Automat in seinen Zuständen Information darüber, welche \equiv_L -Äquivalenzklasse gerade erreicht worden ist. Der Satz 4.5.3 liefert einen zweiten Beweis dafür, dass die Sprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ nicht regulär ist, denn die Relation \equiv_{L_1} hat keinen endlichen Index. Wollte man so etwas Ähnliches wie den Äquivalenzklassen-Automaten für die Sprache L_1 konstruieren, benötigte man unendlich viele Zustände.

4.5.3 Deterministische Minimalautomaten

Wir wenden die Beweistechnik des Satzes von Myhill und Nerode an, um die Zustandszahl von deterministischen Automaten zu minimieren. Wir werden zeigen, dass der Äquivalenzklassen-Automaten A_L aus dem Beweis dieses Satzes sowohl minimal als auch in gewisser – noch zu präzisierender – Weise eindeutig ist.

Korollar 4.5.4 DER ÄQUIVALENZKLASSEN-AUTOMAT HAT MINIMALE ZUSTANDSANZAHL

Sei $L \subseteq \Sigma^*$ regulär und sei k der Index von \equiv_L . Dann besitzt jeder DFA, der L akzeptiert, wenigstens k Zustände. Die Minimalzahl k wird von dem DFA A_L erreicht.

Beweis: Im Beweis des Satzes 4.5.3 haben wir unter „ \Rightarrow “ gezeigt, dass jeder DFA, der L akzeptiert, wenigstens k Zustände besitzt: $k = \text{Index von } \equiv_L \leq |Q|$. Unter „ \Leftarrow “ haben wir einen DFA A_L mit k Zuständen konstruiert, der L akzeptiert. □ 4.5.4

Es kann durchaus NFAs mit (sehr viel) weniger als k Zuständen geben, die L akzeptieren. Ein Beispiel hierfür haben wir nach Satz 4.2.3 gegeben: die Sprache $L_{\text{non-det}}(n)$ hat einen akzeptierenden NFA mit $n+1$ Zuständen, aber der minimale akzeptierende DFA hat nicht weniger als 2^n Zustände.

Wir zeigen nun, dass jeder DFA, der L akzeptiert und minimale Zustandszahl k hat, sich von A_L nur durch eine bijektive Umbenennung der Zustände unterscheidet. Das bedeutet, dass alle solche Automaten im Wesentlichen (nämlich in ihrer Struktur) mit A_L übereinstimmen. Zunächst definieren wir diese Übereinstimmungseigenschaft:

Definition 4.5.5 ISOMORPHIE VON DFAS

Zwei DFAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ und $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ über dem gleichen Alphabet Σ heißen *isomorph*, falls es eine Bijektion $\beta: Q_1 \rightarrow Q_2$ mit den folgenden Eigenschaften gibt:

$$\begin{aligned} \beta(q_{01}) &= q_{02} \\ \beta(F_1) &= F_2, \quad \text{wobei } \beta(F_1) = \{\beta(q) \mid q \in F_1\} \\ \forall q, q' \in Q_1 \forall a \in \Sigma: q \xrightarrow{a}_1 q' &\Leftrightarrow \beta(q) \xrightarrow{a}_2 \beta(q'). \end{aligned}$$

Die Bijektion β heißt *Isomorphismus* von A_1 auf A_2 . □ 4.5.5

Isomorphie ist eine Äquivalenzrelation auf der Klasse der DFAs. Für die Reflexivität braucht man nur die Identität als β zu betrachten, für die Symmetrie kann man das Inverse β^{-1} eines Isomorphismus β benutzen, für die Transitivität betrachtet man die Komposition $\beta_1 \circ \beta_2$ zweier Isomorphismen β_1 und β_2 .

Wir zeigen jetzt die angekündigte Aussage.

Satz 4.5.6 DETERMINISTISCHE MINIMALAUTOMATEN SIND ZUEINANDER ISOMORPH

Sei $L \subseteq \Sigma^*$ regulär und sei k der Index von \equiv_L .

Dann ist jeder DFA A , der L akzeptiert und k Zustände besitzt, zu A_L isomorph.

Beweis: Sei L eine Sprache und sei $A_L = (Q_L, \Sigma, \delta_L, q_{0L}, F_L)$ der Äquivalenzklassen-Automat des Satzes von Myhill und Nerode mit $Q_L = \{[u_1], \dots, [u_k]\}$. Betrachte einen beliebigen L akzeptierenden DFA $A = (Q, \Sigma, \delta, q_{0A}, F)$ mit k Zuständen, d.h., $L(A) = L$ und $|Q| = k$. Wir definieren eine Funktion $\beta: Q_L \rightarrow Q$ und zeigen, dass diese Funktion ein Isomorphismus ist:

$$\beta([u_i]) = \text{dasjenige } q \in Q \text{ mit } q_{0A} \xrightarrow{u_i} q.$$

Dies ist tatsächlich eine Funktion, denn A ist deterministisch und q ist deswegen eindeutig bestimmt. Wir zeigen nun, dass β ein Isomorphismus von A_L auf A ist.

- β ist injektiv:

Es gelte $\beta([u_i]) = q = \beta([u_j])$. Dann gilt nach der Definition von β sowohl $q_{0A} \xrightarrow{u_i} q$ als auch $q_{0A} \xrightarrow{u_j} q$. Also gilt, da A L akzeptiert: $\forall w \in \Sigma^*: u_i w \in L \Leftrightarrow u_j w \in L$, und deswegen nach Definition von $\equiv_L: u_i \equiv_L u_j$, d.h. $[u_i] = [u_j]$.

- β ist surjektiv:

Diese Eigenschaft folgt aus der Injektivität und der Tatsache, dass $|Q_L| = k = |Q|$.

- β erfüllt die erste Isomorphieeigenschaft aus Definition 4.5.5:

$$\beta(q_{0L}) = \beta([\varepsilon]) = \text{dasjenige } q \in Q \text{ mit } q_{0A} \xrightarrow{\varepsilon} q = q_{0A}.$$

- β erfüllt die zweite Isomorphieeigenschaft aus Definition 4.5.5:

$[u_i] \in F_L \Leftrightarrow u_i \in L \Leftrightarrow \beta([u_i]) \in F$, wobei die erste Äquivalenz aus der Definition von F_L stammt, die zweite aus der Definition von β und der Tatsache, dass A die Sprache L akzeptiert.

- β erfüllt die dritte Isomorphieeigenschaft aus Definition 4.5.5:

$$\begin{aligned} [u_i] \xrightarrow{a}_L [u_j] &\Leftrightarrow (\text{Definition von } \xrightarrow{a}_L) \\ & [u_i a] = [u_j] \\ &\Leftrightarrow (\Rightarrow \text{weil } \beta \text{ Funktion ist, } \Leftarrow \text{weil } \beta \text{ injektiv ist}) \\ & \beta([u_i a]) = \beta([u_j]) \\ &\Leftrightarrow (\text{Definition von } \beta([u_i a]) \text{ und } \beta([u_j])) \\ & q_{0A} \xrightarrow{u_i a}_A \beta([u_j]) \\ &\Leftrightarrow (\text{Definition von } \xrightarrow{u_i a}_A \text{ und von } \beta([u_i])) \\ & \beta([u_i]) \xrightarrow{a}_A \beta([u_j]). \end{aligned}$$

Also sind A_L und A isomorph. ☒ 4.5.6

Definition 4.5.7 MINIMALAUTOMAT

Der Minimalautomat für eine reguläre Sprache $L \subseteq \Sigma^*$ ist der (laut Satz 4.5.6) bis auf Isomorphie eindeutige DFA, der L akzeptiert und dessen Zustandszahl gleich dem Index der Nerode-Rechtskongruenz \equiv_L ist. ☒ 4.5.7

Der Minimalautomat für L kann aus einem beliebigen DFA A , der L akzeptiert, systematisch durch zwei Reduktionsschritte konstruiert werden:

1. *Eliminieren nicht erreichbarer Zustände.*

Ein Zustand $q \in Q$ heißt erreichbar, falls es ein Wort $w \in \Sigma^*$ mit $q_0 \xrightarrow{w} q$ gibt. Die Teilmenge der erreichbaren Zustände von Q ist berechenbar, weil man sich bei den Wörtern w mit $q_0 \xrightarrow{w} q$ auf solche der Länge $\leq |Q|$ beschränken kann.

2. *Zusammenfassen äquivalenter Zustände.*

Für $q \in Q, w \in \Sigma^*$ und $S \subseteq Q$ schreiben wir $q \xrightarrow{w} S$, falls es ein $q' \in S$ mit $q \xrightarrow{w} q'$ gibt. Zwei Zustände $q_1, q_2 \in Q$ heißen *äquivalent*, abgekürzt $q_1 \sim q_2$, falls für alle $w \in \Sigma^*$ gilt:

$$q_1 \xrightarrow{w} F \Leftrightarrow q_2 \xrightarrow{w} F,$$

d.h. von q_1 und q_2 führen die selben Wörter zu akzeptierenden Endzuständen. Es besteht ein enger Zusammenhang zwischen Äquivalenz und der Nerode-Rechtskongruenz \equiv_L : Sei $q_0 \xrightarrow{u} q_1$ und $q_0 \xrightarrow{v} q_2$; dann gilt

$$q_1 \sim q_2 \Leftrightarrow u \equiv_L v.$$

Die Begründung hierfür ist die gleiche, die oben bereits beim Beweis der Injektivität von β angegeben worden ist.

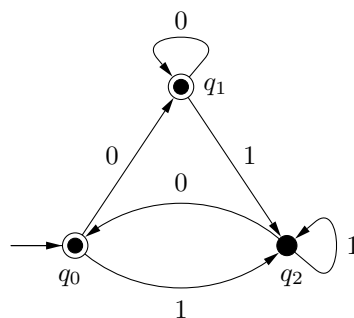
Durch die Relation \sim identifiziert man Äquivalenzklassen von Zuständen, die im Minimalautomaten zusammenfallen müssen. Die Zustandsmenge des aus A konstruierten reduzierten Automaten besteht aus \sim -Äquivalenzklassen, und die Übergangsfunktion ist hierauf genau so definierbar wie auf dem Äquivalenzklassen-Automaten A_L .

Beispiel 1:

Der Übergang von Abbildung 4.5 nach Abbildung 4.6. Hier führt speziell der erste Schritt (Eliminierung nicht erreichbarer Zustände) zum Minimalautomaten.

Ende von Beispiel 1**Beispiel 2:**

Die beiden Zustände q_0 und q_1 in Abbildung 4.8 sind \sim -äquivalent. Keine anderen Zustandspaare sind äquivalent. Wir fassen also die Äquivalenzklassen $\{q_0, q_1\}$ und $\{q_2\}$ zu zwei neuen Zuständen r_0 und r_2 zusammen und erhalten die Abbildung 4.9.

Ende von Beispiel 2Abbildung 4.8: Ein nicht-minimaler DFA A

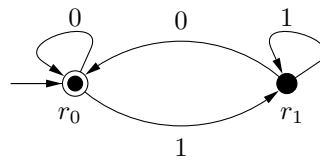


Abbildung 4.9: Minimierte Version von A aus Abbildung 4.8 ($r_0 = \{q_0, q_1\}$ und $r_1 = \{q_2\}$)

Für den Test, ob zwei Zustände \sim -äquivalent sind oder nicht, gibt es einen effizienten Algorithmus (mit quadratischem Aufwand, gemessen in der Größe $|Q|$ der Zustandsmenge). Im *Table-Filling-Algorithmus* wird zunächst eine Tabelle der Zustandspaare $\{q, q'\}$ mit $q \neq q'$ angelegt. Dann werden sukzessive alle Zustandspaare markiert, die schon als inäquivalent erkannt sind. Zum Schluss werden die äquivalenten Zustandsmengen zusammengefasst.

Bei einer Zustandsmenge $Q = \{q_0, \dots, q_{n-1}\}$ kann eine Dreieckstabelle mit $\frac{1}{2} \cdot n \cdot (n - 1)$ Einträgen angelegt werden, um die Zustandspaare zu beschreiben. In Abbildung 4.8 gilt zum Beispiel $n = 3$, und eine entsprechende Tabelle ist in Abbildung 4.10 gezeigt. Dort gibt es einen Eintrag für jedes geordnete Zustandspaar (q_i, q_j) mit $i < j$, der als Eintrag für das (ungeordnete) Zustandspaar $\{q_i, q_j\}$ gilt.

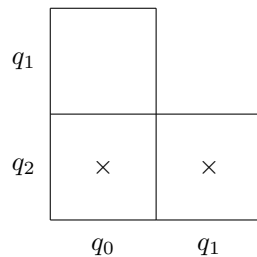


Abbildung 4.10: Beispiel zum Table-Filling-Algorithmus

Auf die anfänglich unmarkierte Tabelle wird der folgende Algorithmus angewendet:

Markiere alle $\{q, q'\}$ mit $(q \in F \wedge q' \notin F) \vee (q \notin F \wedge q' \in F)$;
 (dann sind q, q' von vornherein inäquivalent)
do \exists unmarkiertes Paar $\{q, q'\}, a \in \Sigma: \delta(q, a) \neq \delta(q', a) \wedge \{\delta(q, a), \delta(q', a)\}$ ist markiert \rightarrow
 (solche Paare sind ebenfalls inäquivalent)
 wähle ein solches Paar und markiere es
od;
 Bilde maximale Mengen paarweise unmarkierter Zustände
 (solche Mengen von Zuständen können zu einem zusammengefasst werden)

Im Beispiel (Abbildung 4.10) werden zuerst die beiden Einträge (q_2, q_0) und (q_2, q_1) markiert, weil q_0 und q_1 Endzustände sind, q_2 aber nicht. Beim Betreten der Schleife wird nur noch (q_1, q_0) untersucht. Mit $a = 0$ ergeben sich gleiche Folgezustände (q_0) , mit $a = 1$ ebenfalls (q_2) . Demzufolge wird die Schleife nicht betreten (kein sehr gutes Beispiel für den Algorithmus!), und das Zustandspaar $\{q_0, q_1\}$ bleibt

unmarkiert. Es handelt sich um eine maximale Menge paarweise unmarkierter Zustände, weshalb q_0 und q_1 zusammengefasst werden können. Das Resultat der Zusammenfassung ist in Abbildung 4.9 gezeigt.

4.6 Entscheidbarkeitsfragen

Wir stellen zunächst fest, dass die folgenden Konstruktionen alle effektiv (d.h. algorithmisch berechenbar) sind:

- Chomsky-3-Grammatik \rightsquigarrow NFA \rightsquigarrow DFA \rightsquigarrow Chomsky-3-Grammatik (Abschnitt 4.2).
- Die Abschlussoperationen des Abschnitts 4.3.
- Regulärer Ausdruck \rightsquigarrow NFA \rightsquigarrow DFA \rightsquigarrow regulärer Ausdruck (Abschnitt 4.4).
- DFA \rightsquigarrow Minimalautomat (Abschnitt 4.5).

Damit können wir uns einigen algorithmischen Fragen zuwenden. Wir beschränken dabei uns auf DFAs als Eingaben, denn andere Eingaben (NFAs, reguläre Ausdrücke, rechtslineare oder linkslineare Grammatiken) lassen sich durch die genannten Konstruktionen algorithmisch auf DFAs zurückführen. Die folgenden Probleme definieren zu bestimmten (durch „Gegeben“ gekennzeichneten) Eingaben jeweils eine mit „ja“ oder „nein“ beantwortbare Frage. Gesucht ist jeweils ein Algorithmus, der diese Frage automatisch beantwortet. Existiert ein solcher, heißt das Problem *entscheidbar*.

<i>DFA-Wortproblem</i>	Gegeben: Ein DFA A über Σ und ein Wort $w \in \Sigma^*$. Frage: Gilt $w \in L(A)$?
<i>DFA-Leerheitsproblem</i>	Gegeben: Ein DFA A . Frage: Gilt $L(A) = \emptyset$?
<i>DFA-Endlichkeitsproblem</i>	Gegeben: Ein DFA A . Frage: Ist $L(A)$ endlich?
<i>DFA-Äquivalenzproblem</i>	Gegeben: DFAs A_1 und A_2 . Frage: Gilt $L(A_1) = L(A_2)$?
<i>DFA-Inklusionsproblem</i>	Gegeben: DFAs A_1 und A_2 . Frage: Gilt $L(A_1) \subseteq L(A_2)$?
<i>DFA-Schnittproblem</i>	Gegeben: DFAs A_1 und A_2 . Frage: Gilt $L(A_1) \cap L(A_2) = \emptyset$?

Satz 4.6.1 ENTSCHEIDBARKEIT

Die Entscheidungsprobleme *DFA-Wortproblem*, *DFA-Leerheitsproblem*, *DFA-Endlichkeitsproblem*, *DFA-Äquivalenzproblem*, *DFA-Inklusionsproblem*, *DFA-Schnittproblem* sind alle entscheidbar.

Beweis:**DFA-Wortproblem:**

Man setzt den gegebenen DFA A auf das vorgelegte Wort w an und stellt fest, ob ein Endzustand von A erreicht wird. Falls ja, wird „ $w \in L(A)$ “ ausgegeben, andernfalls „ $w \notin L(A)$ “. A ist also selbst das Entscheidungsverfahren.

DFA-Leerheitsproblem:

Sei p die Anzahl der Zustände des eingegebenen DFA A , und damit auch eine nach dem Pumping-Lemma zur regulären Sprache $L(A)$ gehörige Zahl. Wir behaupten, dass gilt:

$$L(A) = \emptyset \Leftrightarrow \neg \exists w \in L(A) : |w| \leq p. \quad (4.2)$$

Beweis: „ \Rightarrow “ ist klar. „ \Leftarrow “ beweisen wir durch Kontraposition. Sei $L(A) \neq \emptyset$. Dann gibt es ein Wort $w \in L(A)$. Falls $|w| < p$, ist schon alles gezeigt. Falls $|w| \geq p$, erhalten wir durch sukzessives Anwenden des Pumping-Lemmas, jeweils mit $i = 0$, ein Wort $w_0 \in L(A)$ mit $|w_0| \leq p$.

Aus (4.2) kann nun ein mögliches Entscheidungsverfahren für $L(A) = \emptyset$ abgeleitet werden. Für jedes Wort w über dem Eingabealphabet von A mit $|w| \leq p$ (das sind nur endlich viele) wird das Wortproblem „Gilt $w \in L(A)$?“ gelöst. Falls die Antwort stets „nein“ lautet, wird „ $L(A) = \emptyset$ “ ausgegeben. Sonst wird „ $L(A) \neq \emptyset$ “ ausgegeben:

```

input  $A$ ;
output „ $L(A) \neq \emptyset$ “ bzw. „ $L(A) = \emptyset$ “;
var  $p : \mathbb{N}$ ,  $w : \Sigma^*$ ;
 $p :=$  Anzahl der Zustände von  $A$ ;
 $w := \varepsilon$ ;
do  $|w| \leq p \rightarrow$ 
  if  $w \in L(A) \rightarrow$  output(„ $L(A) \neq \emptyset$ “); stop
   $\square$   $w \notin L(A) \rightarrow w :=$  nächstes  $w$  in lexikografischer Reihenfolge
fi
od; output(„ $L(A) = \emptyset$ “)

```

DFA-Endlichkeitsproblem:

Sei p wie oben. Wir behaupten:

$$L(A) \text{ endlich} \Leftrightarrow \neg \exists w \in L(A) : p \leq |w| \leq 2 \cdot p. \quad (4.3)$$

Beweis: „ \Rightarrow “: Durch Kontraposition. Sei w ein Wort in $L(A)$ mit $|w| \geq p$; dann ist $L(A)$ nach dem Pumping-Lemma unendlich. „ \Leftarrow “ ebenfalls durch Kontraposition: Sei $L(A)$ unendlich. Dann gibt es Wörter beliebig großer Länge in $L(A)$, insbesondere ein Wort w mit $|w| \geq 2 \cdot p$. Sukzessives Anwenden des Pumping-Lemmas, jeweils mit $i = 0$, liefert ein Wort w_0 mit $p \leq |w_0| \leq 2 \cdot p$, weil sich mit $i = 0$ das vorgelegte Wort um maximal p Buchstaben verkürzt.

Mit Hilfe von (4.3) kann das Endlichkeitsproblem durch endlichmaliges Lösen des Wortproblems entschieden werden.

DFA-Äquivalenzproblem:

Gegeben seien zwei DFAs A_1 und A_2 . Man konstruiere zunächst einen DFA A mit folgender Eigenschaft:

$$L(A) = (L(A_1) \cap \overline{L(A_2)}) \cup (L(A_2) \cap \overline{L(A_1)}).$$

Offenbar gilt:

$$L(A_1) = L(A_2) \Leftrightarrow L(A) = \emptyset. \quad (4.4)$$

Damit wird das Äquivalenzproblem für Automaten auf das Leerheitsproblem für A zurückgeführt.

Die obige Konstruktion von A ist jedoch sehr aufwändig. Es gibt Alternativen. Wir nehmen an, dass Σ das gemeinsame Alphabet von $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ und $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ ist.

Eine erste Alternative ist die Konstruktion eines Produktautomaten A analog zur Bemerkung nach Satz 4.3.1, außer dass die Endzustandsmenge des Produktautomaten folgendermaßen definiert ist:

$$\{(q_1, q_2) \in Q_1 \times Q_2 \mid q_1 \in F_1 \Leftrightarrow q_2 \notin F_2\}.$$

Hierdurch wird (4.4) für diesen DFA A gewährleistet, denn A akzeptiert genau dann, wenn A_1 und A_2 unterschiedlich akzeptieren (einer akzeptiert, der andere nicht).

Eine zweite Alternative ist, die beiden Automaten A_1 und A_2 neben einander zu legen und zu prüfen, ob die beiden Anfangszustände q_{01} und q_{02} \sim -äquivalent im Sinne der nach Definition 4.5.7 angegebenen Konstruktion sind. Falls ja, gilt $L(A_1) = L(A_2)$, andernfalls $L(A_1) \neq L(A_2)$.

DFA-Inklusionsproblem:

Gegeben seien zwei DFAs A_1 und A_2 . Man konstruiere einen DFA A , so dass $L(A) = L(A_1) \cap \overline{L(A_2)}$ gilt. Wegen

$$L(A_1) \subseteq L(A_2) \Leftrightarrow L(A) = \emptyset$$

kann das Inklusionsproblem für A_1 und A_2 auf das Leerheitsproblem von A zurückgeführt werden.

DFA-Schnittproblem:

Gegeben seien zwei DFAs A_1 und A_2 . Man konstruiere einen DFA A , so dass $L(A) = L(A_1) \cap L(A_2)$ gilt (z.B. mit Hilfe der Produktkonstruktion A_{\cap} aus der Bemerkung nach Satz 4.3.1). Hiermit kann das Schnittproblem A_1 und A_2 auf das Leerheitsproblem von A zurückgeführt werden. \square 4.6.1

4.7 Übungsaufgaben

1. In Texteditoren gibt es üblicherweise eine Möglichkeit, in einem Text nach einer bestimmten Zeichenfolge zu suchen.
 - a) Geben Sie das Diagramm eines deterministischen endlichen Automaten an, der die Ausführung eines solchen Suchbefehls für die beiden Zeichenfolgen *gegeben* und *gegenüberlegen* modelliert. Markieren Sie Start- und Endzustände!
 - b) Welche Sprache erkennt der von Ihnen konstruierte Automat?
2. Gegeben sei ein deterministischer endlicher Automat M mit n Zuständen. Zeigen Sie, dass die von M erkannte Sprache genau dann endlich ist, wenn M kein Wort w mit $|w| \geq n$ erkennt.
3. a) Konstruieren Sie einen deterministischen endlichen Automaten, der die Sprache

$$L_1 = \{w \in \{a, b\}^* \mid w \text{ enthält nicht das Teilwort } bab\}$$

akzeptiert.

- b) Wieviel Zustände muss ein deterministischer endlicher Automat notwendigerweise haben, um eine Sprache L_2 zu erkennen, die nur ein einziges Wort enthält, d.h. $L_2 = \{w\}$, für ein beliebiges, aber fest gewähltes $w \in \Sigma^*$? Begründen Sie Ihre Antwort!
4. Gegeben sei eine Sprache L_B , welche alle Bitfolgen der Länge ≥ 3 enthält, bei denen an *drittletzter* Stelle eine 0 steht, d.h.

$$L_B = \{w0ab \mid w \in \{0, 1\}^* \wedge a, b \in \{0, 1\}\}.$$

- a) Geben Sie einen nichtdeterministischen endlichen Automaten (NFA) über dem Alphabet $\{0, 1\}$ an, der die Sprache L_B erkennt.
- b) Konstruieren Sie aus dem NFA aus Aufgabenteil a) einen deterministischen endlichen Automaten (DFA), der die Sprache L_B erkennt.
- c) Geben Sie einen regulären Ausdruck an, der die vorher erwähnte Sprache $L_B = \{w0ab \mid w \in \{0, 1\}^* \wedge a, b \in \{0, 1\}\}$ beschreibt.
5. a) Zeigen Sie, dass es sich bei der Sprache $L_1 = \{a^n a^n \mid n \in \mathbb{N}\}$ um eine reguläre Sprache handelt.
- b) Zeigen Sie mit Hilfe des Pumping-Lemmas, dass es sich bei der Sprache $L_2 = \{a^n b a^n \mid n \in \mathbb{N}\}$ *nicht* um eine reguläre Sprache handelt.
6. a) Zeigen Sie mit Hilfe des Pumping-Lemmas, dass die Sprache $L = \{a^{2^n} \mid n \in \mathbb{N}\}$ nicht regulär ist (ein Tipp: $\forall p \in \mathbb{N}: 2^p > p$).
- b) Für jedes $n \in \mathbb{N}$ sei L_n eine reguläre Sprache. Die Sprache L sei definiert durch $L = \bigcup_{n \in \mathbb{N}} L_n$. Beweisen Sie *entweder*, dass auch L eine reguläre Sprache ist, *oder* zeigen Sie mit Hilfe eines Beispiels, dass die Sprache L nicht regulär sein muss.

7. Zu einer Sprache L bezeichne $\text{repr}(L)$ die kleinstmögliche Anzahl von Endzuständen eines deterministischen endlichen Automaten, der L erkennt. Zeigen Sie, dass für eine reguläre Sprache $L \subseteq \{a\}^*$ das zugehörige $\text{repr}(L)$ beliebig groß werden kann. (Mit anderen Worten: Zu jedem $n \in \mathbb{N}$ gibt es eine Sprache L_n , die von einem deterministischen endlichen Automaten nur dann erkannt werden kann, wenn dieser mindestens $n + 1$ Endzustände hat.)

8. Für eine Sprache $L \subseteq \Sigma^*$ sei der Halbierungsoperator *half* folgendermaßen definiert:

$$\text{half}(L) := \{w \in \Sigma^* \mid \exists w' \in \Sigma^* : ww' \in L \wedge |w| = |w'|\}.$$

Zeigen Sie, dass die Klasse der regulären Sprachen gegenüber der Halbierungsoperation abgeschlossen ist, d.h.: L ist regulär $\Rightarrow \text{half}(L)$ ist regulär .

9. Sei $L \subseteq \Sigma^*$ eine Sprache. Zeigen Sie, dass \equiv_L eine Rechtskongruenz ist.
10. Sei $L \subseteq \Sigma^*$ eine Sprache, sei $u \in \Sigma^*$ und sei $[u]$ eine \equiv_L -Äquivalenzklasse. Zeigen Sie, dass $[u] \cap L = \emptyset$ oder $[u] \subseteq L$ gilt (in anderen Worten: \equiv_L verfeinert die durch die Äquivalenzklasseneinteilung $\Sigma^* = L \cup \overline{L}$ gegebene Äquivalenzrelation).

Kapitel 5

Kellerautomaten und kontextfreie Sprachen

Reguläre Sprachen finden in der Informatik vielfach Anwendungen (z.B. zur lexikalischen Analyse und zur Teilworterkennung) und sind besonders leicht zu handhaben (Darstellbarkeit durch endliche Automaten und reguläre Ausdrücke, angenehme Abschluss- und Entscheidbarkeitseigenschaften). Dennoch reichen sie für eine wichtige Aufgabe der Informatik nicht aus: die *Syntaxbeschreibung von Programmiersprachen*. Ein Grund ist, dass Programmiersprachen Klammerstrukturen beliebiger Schachtelungstiefe zulassen, zum Beispiel:

- arithmetische Ausdrücke der Form $3 * (4 - (x + 1))$,
- Anweisungen der Form WHILE ... DO WHILE ... DO ... ENDWHILE ENDWHILE

Im Abschnitt 4.5 hatten wir gezeigt, dass bereits das einfachste Muster einer solchen Klammerstruktur, nämlich die Sprache

$$L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$$

nicht mehr Chomsky-3 ist. Für die Syntaxbeschreibung von Programmiersprachen hat sich die nächstmächtigere Sprachklasse in der Chomsky-Hierarchie erfolgreich angeboten: die Chomsky-2- oder kontextfreien Sprachen.

5.1 Kontextfreie Grammatiken und kontextfreie Sprachen

5.1.1 Notation und Beispiele

Jede Produktion einer kontextfreien Grammatik $G = (N, \Sigma, P, S)$ hat die Form $A \rightarrow u$, wobei u das leere Wort sein kann. Falls mehrere Produktionen die selbe linke Seite besitzen, etwa

$$A \rightarrow u_1, A \rightarrow u_2, \dots, A \rightarrow u_k,$$

so schreiben wir dies oft kürzer als eine einzige „Metaregel“

$$A \rightarrow u_1 \mid u_2 \mid \dots \mid u_k.$$

Beispiel G_1 :

Die nicht-reguläre Sprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ lässt sich durch die kontextfreie Grammatik

$$G_1 = (N_1, \Sigma_1, P_1, S_1) \quad \text{mit} \quad \begin{aligned} N_1 &= \{S_1\} \\ \Sigma_1 &= \{a, b\} \\ P_1 &= \{S_1 \rightarrow \varepsilon \mid aS_1b\} \end{aligned}$$

generieren.

Ende von Beispiel G_1

Beispiel G_2 :

Eine Klasse arithmetischer Ausdrücke mit Variablen a, b, c und Operatoren $+$ und $*$ wird durch die Grammatik $G_2 = (\{S\}, \{a, b, c, +, *, (,)\}, P_2, S)$ mit folgender Regelmengemenge P_2 erzeugt:

$$P_2: \quad S \rightarrow a \mid b \mid c \mid S + S \mid S * S \mid (S).$$

Zum Beispiel ist der Ausdruck $(a + b) * c$ in $L(G_2)$, denn es gilt:

$$\begin{aligned} S &\xrightarrow{G_2} S * S && \xrightarrow{G_2} (S) * S && \xrightarrow{G_2} (S + S) * S \\ &\xrightarrow{G_2} (a + S) * S && \xrightarrow{G_2} (a + b) * S && \xrightarrow{G_2} (a + b) * c. \end{aligned} \quad (5.1)$$

Auch $a + (b * c)$ und $a + b * c$ sind in $L(G_2)$, mit ähnlichen Ableitungen aus S .

Ende von Beispiel G_2

5.1.2 Links- und Rechtsableitungen

Wir untersuchen die Erzeugung von Wörtern in einer kontextfreien Grammatik G genauer.

Definition 5.1.1 (LINKS-, RECHTS-)ABLEITUNGEN

Eine *Ableitung* von A nach w in G der Länge $n \geq 0$ ist eine Folge von Ableitungsschritten

$$A = z_0 \xrightarrow{G} z_1 \xrightarrow{G} \dots \xrightarrow{G} z_n = w. \quad (5.2)$$

Eine solche Ableitung heißt *Linksableitung*, falls in jedem Ableitungsschritt $z_{i-1} \xrightarrow{G} z_i$ das in z_{i-1} am weitesten links stehende Nichtterminalsymbol ersetzt wird, falls also z_{i-1} und z_i ($1 \leq i \leq n$) stets von der Form

$$z_{i-1} = w_1 B w_2 \quad \text{und} \quad z_i = w_1 u w_2 \quad \text{mit} \quad w_1 \in \Sigma^* \quad \text{und} \quad (B \rightarrow u) \in P$$

sind. Entsprechend sind *Rechtsableitungen* definiert (dann gilt $w_2 \in \Sigma^*$).

☒ 5.1.1

Beispiel G_2 (Fortsetzung):

$$\begin{aligned} \text{(i)} \quad S &\rightarrow S + S \rightarrow a + S && \rightarrow a + S * S \rightarrow a + b * S \rightarrow a + b * c \\ \text{(ii)} \quad S &\rightarrow S * S \rightarrow S + S * S \rightarrow a + S * S \rightarrow a + b * S \rightarrow a + b * c \\ \text{(iii)} \quad S &\rightarrow S + S \rightarrow S + S * S \rightarrow S + S * c \rightarrow S + b * c \rightarrow a + b * c. \end{aligned} \quad (5.3)$$

Hier sind (i) und (ii) zwei verschiedene Linksableitungen von S nach $a + b * c$, und (iii) ist eine Rechtsableitung von S nach $a + b * c$, bezogen auf die oben angegebene Grammatik G_2 . Alle drei Ableitungen haben die Länge 5.

Ende des Beispiels G_2 (Fortsetzung)

5.1.3 Parsebäume

Wir ordnen jeder Ableitung eindeutig einen Parsebaum zu. Zuerst jedoch definieren wir, was unter einem solchen Baum zu verstehen ist.

Definition 5.1.2 PARSEBAUM

Ein *Parsebaum* von A nach w in G ist ein Baum mit folgenden Eigenschaften:

- Jeder Knoten ist mit einem Symbol aus $N \cup \Sigma \cup \{\varepsilon\}$ beschriftet. Die Wurzel ist mit A beschriftet und jeder innere Knoten ist mit einem Symbol aus N beschriftet.
- Wenn ein mit B beschrifteter innerer Knoten k Nachfolgeknoten besitzt, die in der Reihenfolge von links nach rechts mit den Symbolen β_1, \dots, β_k beschriftet sind, dann gilt

entweder $k = 1$ und $\beta_1 = \varepsilon$ und $(B \rightarrow \varepsilon) \in P$
 oder $k \geq 1$ und $\beta_1, \dots, \beta_k \in N \cup \Sigma$ und $(B \rightarrow \beta_1 \dots \beta_k) \in P$.

- Das Wort w entsteht, indem man die Symbole an den Blättern von links nach rechts konkateniert.

☒ 5.1.2

Abbildung 5.1 veranschaulicht diese Definition.

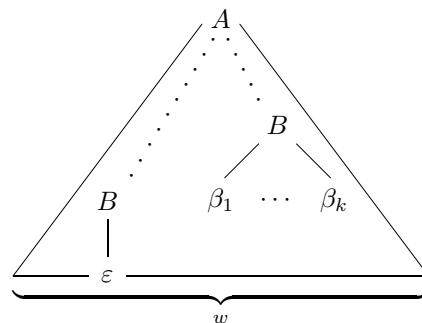


Abbildung 5.1: Veranschaulichung von Definition 5.1.2

Sei

$$A = z_0 \rightarrow_G z_1 \rightarrow_G \dots \rightarrow_G z_n = w.$$

eine Ableitung von A nach w der Form (5.2). Den dazu gehörigen Parsebaum von A nach w konstruieren wir durch Induktion nach der Länge n der Ableitung:

- $n = 0$: Zur trivialen Ableitung von A nach A gehört der Parsebaum mit einzigem Knoten (Wurzel und Blatt) A .
- $n \rightsquigarrow n + 1$: Sei eine Ableitung

$$\underbrace{A}_{z_0} \rightarrow \dots \rightarrow \underbrace{w_1 B w_2}_{z_n} \rightarrow \underbrace{w_1 u w_2}_{z_{n+1}}$$

gegeben und sei t der zur Ableitung $A \rightarrow \dots \rightarrow w_1 B w_2$ gehörige Parsebaum. Wir erweitern diesen Baum gemäß $w_1 B w_2 \rightarrow w_1 u w_2$:

Falls $u = \varepsilon$, so ist der Gesamtparsebaum auf der linken Seite von Abbildung 5.2 gezeigt.

Falls $u = \beta_1 \dots \beta_k$ mit $\beta_1, \dots, \beta_k \in N \cup \Sigma$, so ist der Gesamtparsebaum auf der rechten Seite von Abbildung 5.2 gezeigt.

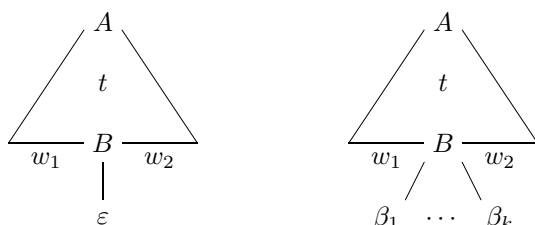


Abbildung 5.2: Zur induktiven Konstruktion des zu einer Ableitung gehörigen Baums

Achtung! Der Parsebaum wird oft auch *Ableitungsbaum* genannt. Der Parsebaum und der Baum der Ableitungen, der in Kapitel 2 eingeführt worden ist, sind jedoch zwei verschiedene Dinge. Beim Baum der Ableitungen aus einem Wort w steht dieses Wort an der Wurzel und der Baum gibt *alle* Ableitungen aus w an. Beim Parse- oder Ableitungsbaum steht ein Wort w dagegen zeichenweise an den Blättern, S an der Wurzel, und der Baum beschreibt *einige* (mindestens eine und eventuell nicht alle) Ableitungen von S nach w .

Die Abbildung 5.3 zeigt links einen Baum, der zur Ableitung

$$S \rightarrow_{G_1} aSb \rightarrow_{G_1} aaSbb \rightarrow_{G_1} aa\varepsilon bb = aabb$$

in G_1 gehört, und rechts einen Baum, der zur Ableitung (5.1), d.h.

$$\begin{array}{ccccccc} S & \rightarrow_{G_2} & S * S & \rightarrow_{G_2} & (S) * S & \rightarrow_{G_2} & (S + S) * S \\ & \rightarrow_{G_2} & (a + S) * S & \rightarrow_{G_2} & (a + b) * S & \rightarrow_{G_2} & (a + b) * c. \end{array}$$

in G_2 gehört.

Bemerkung:

Es gelten folgende Beziehungen zwischen Ableitungen und Parsebäumen:

$A \rightarrow_G^* w$ genau dann, wenn es einen Parsebaum von A nach w in G gibt.

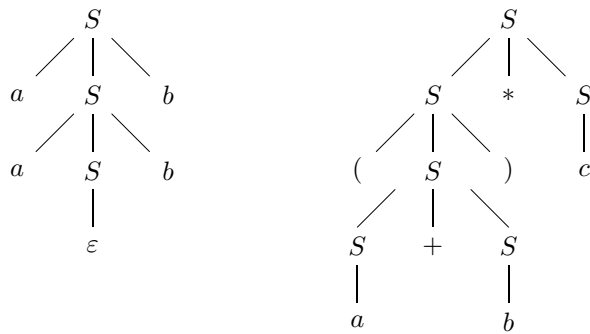


Abbildung 5.3: Zwei Beispiel-Parsebäume

Einer gegebenen Ableitung $A \rightarrow_G^* w$ entspricht genau ein Parsebaum von A nach w .

Einem gegebenen Parsebaum von A nach w entsprechen im Allgemeinen *mehrere* Ableitungen von A nach w , jedoch nur genau eine Linksableitung und genau eine Rechtsableitung.

Ende der Bemerkung

Beispiel G_2 (Fortsetzung):

Wir zeigen, dass zwei verschiedene Ableitungen den gleichen Parsebaum ergeben können:

$$\begin{aligned}
 & S \rightarrow_{G_2} S + S \rightarrow_{G_2} a + S \rightarrow_{G_2} a + b \\
 \text{und } & S \rightarrow_{G_2} S + S \rightarrow_{G_2} S + b \rightarrow_{G_2} a + b
 \end{aligned}$$

sind zwei verschiedene Ableitungen, die beide dem in Abbildung 5.4 dargestellten Baum entsprechen. Parsebäume abstrahieren von unwesentlichen Reihenfolgen bei den Regelanwendungen, die möglich werden, wenn mehrere Nichtterminale gleichzeitig auftreten. Bei Festlegung auf Linksableitungen bzw. Rechtsableitungen sind solche Wahlmöglichkeiten ausgeschlossen.

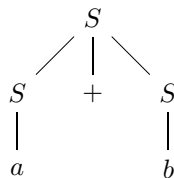


Abbildung 5.4: Ein Parsebaum von $a + b$ in G_2

Ende des Beispiels G_2 (Fortsetzung)

Nebenbemerkung:

Sei die Syntax einer Programmiersprache PL durch eine kontextfreie Grammatik G gegeben. Ein Übersetzer für PL erstellt in der Phase der Syntaxanalyse für jedes gegebene PL-Programm einen Parsebaum in G . Die *Bedeutung* oder *Semantik* des PL-Programms hängt von der Struktur des erstellten Parsebaums ab. Der Übersetzer erzeugt nämlich den Maschinencode des PL-Programms anhand des Parsebaums. Für

einen Benutzer der Programmiersprache PL ist es wichtig, dass jedes PL-Programm eine eindeutige Semantik hat. Daher sollte es zu jedem PL-Programm genau einen Parsebaum geben.

Ende der Nebenbemerkung

5.1.4 Mehrdeutigkeit

Es ist im Allgemeinen nicht klar, ob es zu jedem Wort $w \in L(G)$ einen eindeutigen Parsebaum von S nach w gibt, weil es durchaus mehrere Ableitungen (auch mehrere Linksableitungen) von S nach w geben kann. Betrachten wir dazu wieder G_2 und das Wort $a + b * c \in L(G_2)$. Es gibt die beiden verschiedenen, in Abbildung 5.5 gezeigten Parsebäume. Der linke Baum gehört zu der Linksableitung (5.3(i)), der rechte Baum jedoch zu der Linksableitung (5.3(ii)) des Wortes $a + b * c$. Diese entsprechen den beiden inhaltlich verschiedenen Klammerungen des Ausdrucks $a + b * c$: entweder $a + (b * c)$ oder $(a + b) * c$. Man sagt dazu, dass die Grammatik G_2 für arithmetische Ausdrücke „mehrdeutig“ ist.

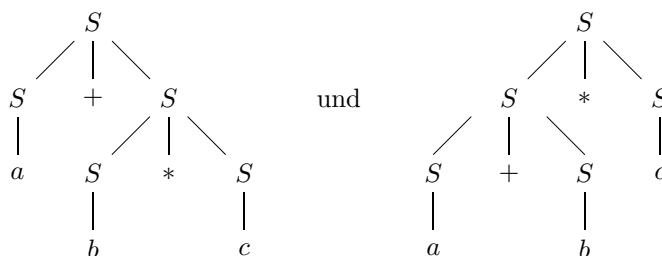


Abbildung 5.5: Zwei Parsebäume für $a + b * c$ in G_2

Definition 5.1.3 EIN-/MEHRDEUTIGKEIT VON KONTEXTFREIEN GRAMMATIKEN/SPRACHEN

- (i) Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ heißt *eindeutig*, wenn es zu jedem Wort $w \in \Sigma^*$ höchstens einen Parsebaum bzw. höchstens eine Linksableitung von S nach w in G gibt. Andernfalls heißt G *mehrdeutig*.
- (ii) Eine kontextfreie Sprache $L \subseteq \Sigma^*$ heißt *eindeutig*, wenn es eine eindeutige kontextfreie Grammatik G mit $L = L(G)$ gibt. Andernfalls heißt L (*inhärent*) *mehrdeutig*. □ 5.1.3

Beispiel G_3 :

Dass unsere Beispielgrammatik G_2 mehrdeutig ist, bedeutet noch nicht, dass $L(G_2)$ inhärent mehrdeutig ist. Denn es kann eine Grammatik existieren, die die gleiche Sprache generiert und eindeutig ist. In der Tat ist dies bei $L(G_2)$ der Fall.

Um die Eindeutigkeit von Grammatiken für arithmetische Ausdrücke zu erreichen, wählt man eine syntaktisch festgelegte Auswertungsstrategie:

- Die Auswertung erfolgt von links nach rechts. Damit wird z.B. $a + b + c$ wie $(a + b) + c$ ausgewertet.
- Die Multiplikation $*$ erhält eine höhere Priorität als $+$. Damit wird z.B. $a + b * c$ wie $a + (b * c)$ ausgewertet.

- Um andere Auswertungsreihenfolgen zu spezifizieren, müssen explizit Klammern gesetzt werden (z.B. $a + (b + c)$ oder $(a + b) * c$).

Wir betrachten also $G_3 = (\{S, T, F\}, \{a, b, c, +, *, (,)\}, P_3, S)$ mit folgender Regelmenge P_3 :

$$\begin{aligned} S &\rightarrow T \mid S + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow (S) \mid a \mid b \mid c. \end{aligned}$$

Dabei stehen der Buchstabe T für (*arithmetischer*) *Term* und der Buchstabe F für (*arithmetischer*) *Faktor*. G_3 ist eindeutig, und es gilt $L(G_3) = L(G_2)$. Der (eindeutige) Parsebaum von $a + b * c$ in G_3 ist in Abbildung 5.6 gezeigt. Damit ist $-$ in G_3 , aber nicht in G_2 – klar, dass mit $a + b * c$ der Ausdruck $a + (b * c)$ und nicht $(a + b) * c$ gemeint ist.

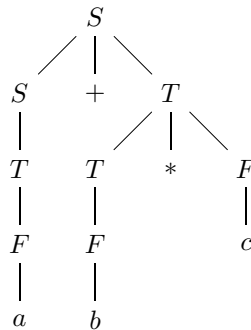


Abbildung 5.6: Parsebaum von $a + b * c$ in G_3

Ende von Beispiel G_3

Es erhebt sich die Frage, ob es überhaupt eine inhärent mehrdeutige Sprache gibt. Diese Frage wurde von N. Chomsky 1964 positiv beantwortet:

Satz 5.1.4 (OHNE BEWEIS)

Die Sprache $L = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ und } (i=j \vee j=k)\} \subseteq \{a, b, c\}^*$ ist inhärent mehrdeutig.

Der Beweis dieser Aussage ist recht aufwändig. Im Wesentlichen zeigt man, dass genügend lange Wörter mit genau gleich vielen Buchstaben a , b und c zwei verschiedene Linksableitungen haben *müssen*.

5.1.5 Normalformen kontextfreier Grammatiken

Für Untersuchungen über kontextfreie Sprachen ist es günstig, wenn die Regeln der zugrundegelegten kontextfreien Grammatiken von möglichst einfacher Bauart sind. Wir stellen deshalb in diesem Abschnitt Transformationen vor, die gegebene kontextfreie Grammatiken in äquivalente (d.h.: sprachäquivalente) Grammatiken überführen, deren Regeln zusätzlichen Bedingungen genügen. Eine solche Transformation wurde bereits in Abschnitt 3.2 angegeben, wo gezeigt wurde, dass jede kontextfreie Grammatik effektiv

äquivalent zu einer eingeschränkt kontextfreien Grammatik ist. (Zur Erinnerung: eine kontextfreie Grammatik ist eingeschränkt kontextfrei, wenn es entweder überhaupt keine ε -Produktionen $A \rightarrow \varepsilon$ gibt, oder als einzige ε -Produktion $S \rightarrow \varepsilon$ vorhanden ist, dann aber S auf keiner rechten Seite einer Produktion vorkommt.)

Hier betrachten wir zwei weitere Normalformen.

Definition 5.1.5 CHOMSKY-NORMALFORM

Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ ist in *Chomsky-Normalform*, wenn folgendes gilt:

- G ist eingeschränkt kontextfrei (so dass höchstens $(S \rightarrow \varepsilon) \in P$ erlaubt ist, dann aber S nicht auf einer rechten Seite auftaucht),
- jede Produktion in P anders als $S \rightarrow \varepsilon$ ist von der Form

$$A \rightarrow a \quad \text{oder} \quad A \rightarrow BC,$$

wobei $A, B, C \in N$ und $a \in \Sigma$ sind. □ 5.1.5

Satz 5.1.6 KONTEXTFREIE GRAMMATIK \rightsquigarrow CHOMSKY-NORMALFORM

Jede kontextfreie Grammatik lässt sich effektiv in eine äquivalente Grammatik in Chomsky-Normalform transformieren.

Beweis: (Skizze.)

In einem ersten Schritt erreichen wir, dass alle Produktionen anders als $(S \rightarrow \varepsilon)$ entweder von der Form $A \rightarrow a$ oder von der Form $A \rightarrow A_1 \dots A_m$ (mit $m \geq 2$) sind.

In einem zweiten Schritt ersetzen wir z.B. $A \rightarrow A_1 A_2 A_3 A_4$ mit Hilfe neuer Variablen D_1, D_2 durch $A \rightarrow A_1 D_1$, $D_1 \rightarrow A_2 D_2$ und $D_2 \rightarrow A_3 A_4$. □ 5.1.6

Definition 5.1.7 GREIBACH-NORMALFORM

Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ ist in *Greibach-Normalform*, wenn folgendes gilt:

- G ist eingeschränkt kontextfrei (so dass höchstens $(S \rightarrow \varepsilon) \in P$ erlaubt ist, dann aber S nicht auf einer rechten Seite auftaucht),
- jede Produktion in P anders als $S \rightarrow \varepsilon$ ist von der Form

$$A \rightarrow aB_1 \dots B_k,$$

wobei $k \geq 0$, $A, B_1, \dots, B_k \in N$ und $a \in \Sigma$ sind. □ 5.1.7

Satz 5.1.8 KONTEXTFREIE GRAMMATIK \rightsquigarrow GREIBACH-NORMALFORM

Jede kontextfreie Grammatik lässt sich effektiv in eine äquivalente Grammatik in Greibach-Normalform transformieren.

Beweis: (*Skizze.*)

Wir starten mit der Chomsky-Normalform und erreichen (unter Umständen durch die Einführung neuer Variablen), dass die Variablen $A_1, A_2, \dots, A_{n-1}, A_n$ so nummeriert sind, dass gilt: falls $(A_i \rightarrow A_j \gamma) \in P$ (mit $\gamma \in N^+$), dann $j > i$. Dieser Schritt ist nicht trivial.

Danach hat die Variable A_n auf ihren rechten Seiten nur Ausdrücke der Form $a\gamma$, die mit einem Terminalzeichen a und nicht mit einer Variablen anfangen.

Die Variable A_{n-1} hat auf ihren rechten Seiten ebenfalls solche Ausdrücke, oder eventuell auch Ausdrücke der Form $A_n \gamma$, aber keine Ausdrücke mit *anderen* Variablen als A_n . Wir ersetzen dann auf den rechten Seiten der Variable A_{n-1} die A_n durch die rechten Seiten der A_n ; danach beginnen alle rechten Seiten von A_{n-1} auch nur mit Terminalzeichen.

So fortfahrend, erhält man eine Menge von Produktionen (für die gleiche Sprache), deren rechte Seiten (außer evtl. $S \rightarrow \varepsilon$) alle nur mit – einem oder mehreren – Terminalzeichen beginnen. Danach kann, gegebenenfalls durch die Einführung neuer Variablen, die gewünschte Form hergestellt werden, so dass alle rechten Seiten mit *genau* einem Terminalzeichen beginnen. □ 5.1.8

5.2 Das Pumping-Lemma für kontextfreie Sprachen

Analog Abschnitt 4.5.1 beschreiben wir im Folgenden eine notwendige Bedingung dafür, dass eine gegebene Sprache kontextfrei ist.

Lemma 5.2.1 PUMPING-LEMMA FÜR KONTEXTFREIE SPRACHEN, ODER $uvwxy$ -LEMMA

Zu jeder kontextfreien Sprache $L \subseteq \Sigma^*$ existiert eine Zahl $p \in \mathbb{N}$, so dass es für alle Wörter $z \in L$ mit $|z| \geq p$ eine Zerlegung $z = uvwxy$ mit den folgenden drei Eigenschaften gibt:

- (1) $1 \leq |vx|$
- (2) $|vwx| \leq p$
- (3) $\forall i \in \mathbb{N}: u v^i w x^i y \in L$.

D.h., die Teilwörter v und x können beliebig oft „aufgepumpt“ werden, ohne dass die kontextfreie Sprache L verlassen wird.

Zum Beweis des Pumping-Lemmas benötigen wir ein allgemeines Lemma über Bäume, das wir dann speziell für Parsebäume anwenden werden. Sei t ein endlicher Baum. Der *Verzweigungsgrad* von t ist der maximale Ausgangsgrad, d.h.: die maximale Anzahl von Nachfolgeknoten, die ein Knoten in t besitzt. Ein *Pfad* der Länge m in t ist eine Kantenfolge von der Wurzel bis zu einem Blatt von t mit m Kanten. Der Fall $m = 0$ ist zugelassen, wenn die Wurzel auch Blatt ist.

Lemma 5.2.2 OBERE SCHRANKE FÜR DIE ANZAHL DER BLÄTTER EINES BAUMES

Sei t ein endlicher Baum mit dem Verzweigungsgrad $\leq k$, in dem jeder Pfad die Länge $\leq m$ hat. Dann ist in t die Anzahl der Blätter $\leq k^m$.

Beweis: Durch Induktion über $m \in \mathbb{N}$.

- $m = 0$: Dann besteht t nur aus $k^0 = 1$ Knoten.
- $m \rightsquigarrow m + 1$: t besitzt j Unterbäume t_1, \dots, t_j mit $j \leq k$, in denen die Pfade die Länge $\leq m$ haben (siehe Abbildung 5.7). Nach Induktionsvoraussetzung ist für jeden der Unterbäume t_1, \dots, t_j die Anzahl der Blätter $\leq k^m$. Damit ist in t die Anzahl der Blätter $\leq j \cdot k^m \leq k \cdot k^m = k^{m+1}$.

□ 5.2.2

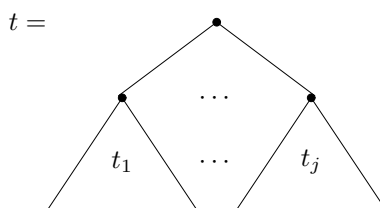


Abbildung 5.7: Zum Induktionsschritt des Beweises von Lemma 5.2.2

Wir sind jetzt vorbereitet für den Beweis des Pumping-Lemmas.

Beweis: Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik mit $L(G) = L$. Wir setzen:

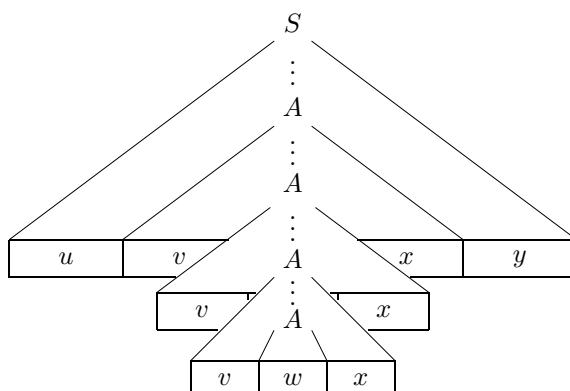
- $k =$ Wortlänge der längsten rechten Seite einer Produktion aus P , wenigstens jedoch 2;
- $m = |N|$;
- und $p = k^{m+1}$.

Sei jetzt z ein Wort in L mit $|z| \geq p$. Dann gibt es einen Parsebaum t von S nach z in G , in dem es kein Teilstück gibt, das einer Ableitung der Form $B \rightarrow_G^* B$ (siehe Abbildung 5.8) entspricht. Jedes solche Teilstück könnte nämlich aus t entfernt werden, ohne dass das abgeleitete Wort z verändert würde.

Wegen der Wahl von k und $|z|$ hat t einen Verzweigungsgrad $\leq k$ und $\geq k^{m+1}$ Blätter. Also gibt es nach dem vorangegangenen Lemma (bzw. dessen logischer Kontraposition) in t einen Pfad der Länge $\geq m + 1$. Auf diesem Pfad liegen $\geq m + 1 > |N|$ innere Knoten (die Wurzel mitgezählt), so dass es eine Wiederholung eines Nichtterminalsymbols bei der Beschriftung dieser Knoten gibt. Wir benötigen diese Wiederholung in einer speziellen Lage.

Unter einem *Wiederholungsbaum* in t verstehen wir einen Unterbaum von t , in dem sich die Beschriftung der Wurzel bei einem weiteren Knoten wiederholt. Wir wählen jetzt in t einen minimalen Wiederholungsbaum t_0 , d.h. einen solchen, der keinen weiteren Wiederholungsbaum als echten Unterbaum enthält. In t_0 hat jeder Pfad eine Länge $\leq m + 1$ (gerade wegen der Minimalität). Konkret kann man einen solchen Wiederholungsbaum finden, indem man ein Blatt wählt, dessen Abstand zur Wurzel $\geq m + 1$ ist, und den entsprechenden Pfad von unten nach oben durchläuft, bis man zum ersten Mal ein Nichtterminalzeichen doppelt auffindet.

Sei A die Wurzelbeschriftung von t_0 . Dann hat t die in Abbildung 5.9 gezeigte Struktur.

Abbildung 5.10: Parsebaum für $i = 3$

Wie bei regulären Sprachen kann das eben bewiesene Pumping-Lemma zum Nachweis benutzt werden, dass eine bestimmte Sprache *nicht* kontextfrei ist.

Behauptung: Die Sprache $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist nicht kontextfrei.

Beweis: Durch Widerspruch. Nehmen wir an, dass L kontextfrei ist. Dann gibt es nach dem Pumping-Lemma ein $p \in \mathbb{N}$ mit den dort genannten Eigenschaften. Betrachten wir jetzt $z = a^p b^p c^p$. Da $|z| = 3 \cdot p \geq p$ gilt, lässt sich z zerlegen in $z = uvwxy$ mit $1 \leq |vx|$ und $|vwx| \leq p$, so dass für alle $i \in \mathbb{N}$ gilt: $uv^i wx^i y \in L$. Wegen $|vwx| \leq p$ kommen im Teilwort vwx von z keine a 's oder keine c 's vor. Deshalb werden beim Aufpumpen zu $uv^i wx^i y$ höchstens zwei der Buchstaben a, b, c berücksichtigt, und mindestens einer davon wird wegen $1 \leq |vx|$ berücksichtigt. Solche aufgepumpten Wörter liegen aber nicht in L , Widerspruch. Also war die Annahme falsch, und L ist nicht kontextfrei. \square Behauptung

Diese Anwendung des Pumping-Lemmas (wie auch die entsprechende Anwendung des Pumping-Lemmas für reguläre Sprachen) kann als ein *Zweipersonenspiel* zwischen *Proponent* und *Opponent* interpretiert werden.

Das Pumping-Lemma-Spiel:

Gegeben: Eine Sprache L . (Oben: $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$.)

Schritt 1a: Der Proponent behauptet, dass L nicht kontextfrei ist.

Schritt 1b: Der Opponent behauptet das Gegenteil und darf eine Zahl p wählen, von der er meint, dass sie die Bedingung des Lemmas 5.2.1 erfüllt.

Schritt 2a: Der Proponent darf z wählen (in Kenntnis und auch in Abhängigkeit von p).
(Oben: $z = a^p b^p c^p$.)

Schritt 2b: Der Opponent darf z in $uvwxy$ aufteilen, wobei die Beschränkungen $1 \leq |vx|$ und $|vwx| \leq p$ erfüllt sein müssen.

Schritt 3a: Wenn der Proponent jetzt ein i so wählen kann, dass das Wort $uv^i wx^i y$ *nicht* in L liegt, hat er gewonnen. (Oben: wähle $i = 0$ oder i beliebig, aber ≥ 2 .)

Gibt es für dieses Spiel eine Gewinnstrategie für den Proponenten, ist L nicht kontextfrei.

Ende des Pumping-Lemma-Spiels

Mit dem Pumping-Lemma lässt sich auch zeigen, dass kontextfreie Grammatiken nicht ausreichen, um die Syntax von höheren Programmiersprachen wie etwa Modula-2, C oder Java *vollständig* zu beschreiben. Obwohl die Grundstruktur der syntaktisch korrekten Programme oft mit Hilfe kontextfreier Grammatiken (entweder in – erweiterter – BNF-Notation oder durch Syntaxdiagramme) beschrieben wird, gibt es syntaktische Nebenbedingungen, die kontextsensitiv sind, zum Beispiel die Bedingung, dass jede Variable vor ihrem Gebrauch deklariert sein muss. Solche Bedingungen werden daher bei der Definition von Programmiersprachen zusätzlich zur kontextfreien Grundstruktur der Programme genannt. Ein Übersetzer überprüft diese kontextsensitiven Bedingungen zum Beispiel durch Anlegen geeigneter Tabellen zum Abspeichern der deklarierten Variablen.

5.3 Kellerautomaten

Bisher haben wir kontextfreie Sprachen über die Erzeugbarkeit von Grammatiken definiert. Wir wollen jetzt die Erkennbarkeit bzw. die Akzeptierbarkeit dieser Sprachen durch Automaten untersuchen. Unser Ziel ist eine Modifikation des Modells des endlichen Automaten, so dass genau die kontextfreien Sprachen erkannt werden können.

Die Schwäche der endlichen Automaten ist das Fehlen eines Speichers für unbeschränkt große Informationen. Intuitiv kann ein endlicher Automat die Klammersprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ deshalb nicht erkennen, weil Information über die Anzahl der bisher gelesenen Zeichen a nur in der (endlichen) Zustandsmenge gespeichert werden kann; wenn ein endlicher Automat z.B. p Zustände hat, scheitert er bei der Erkennung von L spätestens an einem Wort mit $p+1$ Buchstaben a und $p+1$ Buchstaben b . Wir benötigen also eine Methode, um unbeschränkt Information speichern zu können.

Dazu betrachten wir sogenannte *Kellerautomaten* oder *Pushdown-Automaten*. Das sind im Wesentlichen endliche Automaten, erweitert um einen Speicher, der zwar unbeschränkt viel Information aufnehmen kann, auf den aber nur sehr eingeschränkt zugegriffen werden kann. Der Speicher ist als *Keller* oder *Stack* oder *Pushdown-Liste* organisiert, bei dem nur auf das jeweils oberste Symbol zugegriffen werden kann. Die Transitionen eines Kellerautomaten hängen vom aktuellen Zustand, dem gelesenen Symbol des Eingabewortes und dem obersten Symbol des Kellers ab; sie verändern den Zustand und den Inhalt des Kellers (siehe Abbildung 5.11).

5.3.1 Definition und Beispiel

Definition 5.3.1 KELLERAUTOMAT

Ein (*nichtdeterministischer*) *Kellerautomat* (oder *Pushdown-Automat*), kurz PDA, ist ein 7-Tupel $K = (Q, \Sigma, \Gamma, Z_0, \delta, q_0, F)$ mit folgenden Komponenten:

1. Q ist eine endliche Menge von *Zuständen*.
2. Σ ist das *Eingabealphabet* der Maschine.
3. Γ ist das *Kelleralphabet*. Intuitiv enthält Γ alle Zeichen, die als Kellerelemente von K vorkommen.
4. $Z_0 \in \Gamma$ ist das *Startsymbol des Kellers*.
5. $\delta \subseteq (Q \times \Gamma \times (\Sigma \cup \{\varepsilon\})) \times (Q \times \Gamma^*)$ ist die *Übergangs- oder Transitionsrelation*.

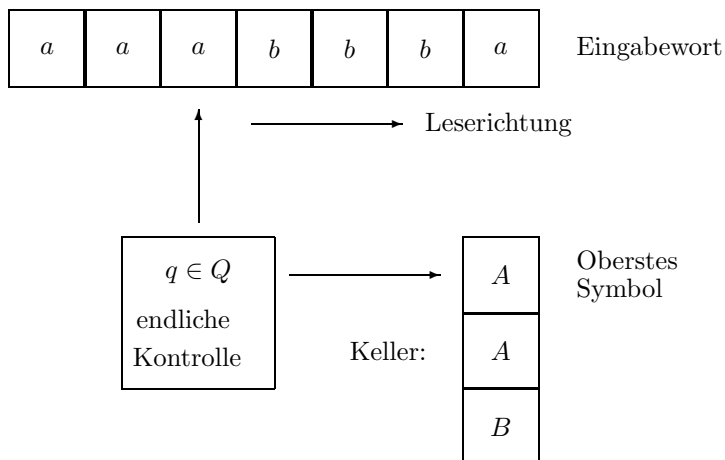


Abbildung 5.11: Skizze eines Kellerautomaten

6. $q_0 \in Q$ ist der *Anfangszustand*.

7. $F \subseteq Q$ ist die Menge der *Endzustände*.

☒ 5.3.1

In dieser Definition ist die Übergangsrelation

$$\delta \subseteq (Q \times \Gamma \times (\Sigma \cup \{\varepsilon\})) \times (Q \times \Gamma^*)$$

vor allem zu beachten. Im Vergleich mit endlichen Automaten:

$$\delta \subseteq (Q \times \Sigma) \times Q$$

taucht auf der linken Seite auch noch die Menge Γ auf (das bedeutet Abhängigkeit vom obersten Kellersymbol) und auf der rechten Seite auch die Menge Γ^* (das bedeutet eine Zeichenkette, die das oberste Kellersymbol ersetzt). Außerdem lassen wir neben den bislang bekannten Übergängen, durch die ein Zeichen eines Eingabewortes gelesen wird, auch sogenannte *spontane* Übergänge zu, die durch „Lesen eines ε “ umschrieben werden; deshalb steht auf der linken Seite die Menge $\Sigma \cup \{\varepsilon\}$ (und nicht nur Σ). Spontane Übergänge werden noch separat zu motivieren sein.

Im Zusammenhang mit Kellerautomaten benutzen wir folgende typische Buchstaben:

$$\begin{aligned} a, b, c &\in \Sigma, \\ u, v, w &\in \Sigma^*, \\ \alpha &\in \Sigma \cup \{\varepsilon\}, \\ q &\in Q, \\ A, B, Z &\in \Gamma, \\ \gamma &\in \Gamma^*. \end{aligned}$$

Die Elemente $((q, Z, \alpha), (q', B_1 \dots B_k)) \in \delta$ heißen *Transitionen*. Statt $((q, Z, \alpha), (q', B_1 \dots B_k)) \in \delta$ schreiben wir oft – wie früher –

$$(q, Z) \xrightarrow{\alpha} (q', B_1 \dots B_k).$$

Wir haben dabei folgende anschauliche Vorstellung:

- q ist der alte Zustand;
- Z ist das momentan gelesene oberste Kellersymbol, das durch den Übergang gelöscht wird;
- falls $\alpha = a \in \Sigma$, handelt es sich um einen Übergang, bei dem das Zeichen a des Eingabewortes gelesen wird;
- falls $\alpha = \varepsilon$, handelt es sich um einen spontanen Übergang, bei dem *kein* Zeichen des Eingabewortes gelesen wird;
- q' ist der neue Zustand;
- $B_1 \dots B_k$ ist das Wort, das – nach Löschen von Z – auf den Keller zuoberst geschrieben wird (mit B_1 als neuem oberstem Kellersymbol, falls $k \geq 1$).

Eine Transition $(q, Z) \xrightarrow{a} (q', B_1 \dots B_k)$ mit $a \in \Sigma$ besagt also: ist q der aktuelle Zustand und Z das oberste Kellersymbol, so liest der Kellerautomat das Eingabesymbol a , geht in den Zustand q' über und ersetzt an der Kellerspitze das Symbol Z durch das Wort $B_1 \dots B_k$. Es gibt dabei zwei Fälle: entweder ist $k \neq 0$, d.h. $B_1 \dots B_k \neq \varepsilon$, und dann ist B_1 das neue oberste Kellersymbol; oder es gilt $k = 0$, d.h. $B_1 \dots B_k = \varepsilon$, und dann ist das unter Z im Keller stehende Symbol das neue oberste Kellersymbol, oder (falls ein solches nicht existiert) der Keller ist leer.

Analog besagt eine Transition $(q, Z) \xrightarrow{\varepsilon} (q', B_1 \dots B_k)$: im Zustand q und Z als oberstem Kellersymbol kann der Kellerautomat selbstständig (spontan, d.h. ohne einen Eingabebuchstaben zu lesen) in den Zustand q' übergehen und an der Kellerspitze Z durch $B_1 \dots B_k$ ersetzen.

Ist $B_1 \dots B_k = \varepsilon$, so spricht man bei einer Transition $(q, Z) \xrightarrow{\alpha} (q', B_1 \dots B_k)$ von einem *pop-Schritt*, da das oberste Kellersymbol einfach nur entfernt wird. Ist $k \geq 2$ und $B_k = Z$, so spricht man bei einer Transition $(q, Z) \xrightarrow{\alpha} (q', B_1 \dots B_k)$ von einem *push-Schritt*, da das (nicht leere) Wort $B_1 \dots B_{k-1}$ an der Kellerspitze hinzugefügt wird (oberhalb von Z). Besonders interessant ist der Fall $k = 2$ (und $B_2 = Z$); dann handelt es sich um ein „einfaches push“: ein einziges Zeichen, B_1 , wird einfach nur auf den Keller gelegt. Beliebige Schritte eines Kellerautomaten können im Prinzip (und mit Hilfe zusätzlicher, spontaner Übergänge) durch eine Kette von pop- und (einfachen) push-Schritten simuliert werden.

Ist der Keller einmal vollständig leer, kann ein Kellerautomat keine weitere Bewegung machen, denn jeder Übergang setzt ein an der Kellerspitze befindliches Kellerzeichen voraus. Anfänglich ist ein Keller nie leer, er besteht allerdings nur aus dem Zeichen Z_0 .

Um das Verhalten und die Sprachakzeptanz von Kellerautomaten formal zu definieren, erweitern wir zuerst die Transitionsrelation. Dazu benötigen wir den Begriff der *Konfiguration* eines Kellerautomaten. Eine Konfiguration beschreibt einen Zustand und einen Kellerinhalt.

Definition 5.3.2 KONFIGURATIONEN UND ERWEITERTE TRANSITIONSRELATION

Sei $K = (Q, \Sigma, \Gamma, Z_0, \delta, q_0, F)$ ein Kellerautomat.

- Unter einer *Konfiguration* von K verstehen wir ein Paar $(q, \gamma) \in Q \times \Gamma^*$, das den momentanen Zustand q und den momentanen Kellerinhalt γ von K beschreibt.

- Für jedes $\alpha \in \Sigma \cup \{\varepsilon\}$ ist $\xrightarrow{\alpha}$ eine zweistellige Relation auf den Konfigurationen von K , die wie folgt definiert ist:

$$(q, \gamma) \xrightarrow{\alpha} (q', \gamma'), \quad \text{falls } \exists Z \in \Gamma \exists \gamma_0, \gamma_1 \in \Gamma^* : \\ \gamma = Z\gamma_0 \text{ und } \gamma' = \gamma_1\gamma_0 \text{ und } ((q, Z, \alpha), (q', \gamma_1)) \in \delta.$$

Wir nennen $\xrightarrow{\alpha}$ die α -*Transitionsrelation* auf den Konfigurationen.

- Für jedes Wort $w \in \Sigma^*$ ist \xRightarrow{w} eine zweistellige Relation auf den Konfigurationen von K , die induktiv definiert ist:

$$w = \varepsilon : \quad (q, \gamma) \xRightarrow{\varepsilon} (q', \gamma'), \quad \text{falls } \exists n \geq 0 : (q, \gamma) \xrightarrow{\varepsilon} \circ \dots \circ \xrightarrow{\varepsilon} (q', \gamma') \\ \text{\scriptsize } n\text{-mal}$$

$$w = av \text{ mit } a \in \Sigma, v \in \Sigma^* : \quad (q, \gamma) \xRightarrow{av} (q', \gamma'), \quad \text{falls } (q, \gamma) \xRightarrow{\varepsilon} \circ \xrightarrow{a} \circ \xRightarrow{v} (q', \gamma').$$

Wir nennen \xRightarrow{w} die *erweiterte w -Transitionsrelation* auf den Konfigurationen. ☒ 5.3.2

Es gelten (direkt nach Definition) die folgenden Eigenschaften. Für alle $\alpha \in \Sigma \cup \{\varepsilon\}, a_1, \dots, a_n \in \Sigma$ und $w_1, w_2 \in \Sigma^*$ gilt:

$$(q, \gamma) \xrightarrow{\alpha} (q', \gamma') \quad \Rightarrow \quad (q, \gamma) \xRightarrow{\alpha} (q', \gamma').$$

$$(q, \gamma) \xRightarrow{a_1 \dots a_n} (q', \gamma') \Leftrightarrow (q, \gamma) \xRightarrow{\varepsilon} \circ \xrightarrow{a_1} \circ \xRightarrow{\varepsilon} \dots \xRightarrow{\varepsilon} \circ \xrightarrow{a_n} \circ \xRightarrow{\varepsilon} (q', \gamma')$$

$$\Leftrightarrow (q, \gamma) \xRightarrow{a_1} \circ \dots \circ \xRightarrow{a_n} (q', \gamma').$$

$$(q, \gamma) \xRightarrow{w_1 w_2} (q', \gamma') \Leftrightarrow (q, \gamma) \xRightarrow{w_1} \circ \xRightarrow{w_2} (q', \gamma').$$

Für Kellerautomaten gibt es zwei Varianten von Sprachakzeptanz, die wir jetzt definieren können.

Definition 5.3.3 AKZEPTANZ

Sei $K = (Q, \Sigma, \Gamma, Z_0, \delta, q_0, F)$ ein Kellerautomat.

K akzeptiert ein Wort $w \in \Sigma^*$ mit Endzustand, falls $\exists q \in F \exists \gamma \in \Gamma^* : (q_0, Z_0) \xRightarrow{w} (q, \gamma)$. Die von K mit Endzustand akzeptierte Sprache ist

$$L(K) = \{w \in \Sigma^* \mid K \text{ akzeptiert } w \text{ mit Endzustand}\}.$$

K akzeptiert ein Wort $w \in \Sigma^*$ mit leerem Keller, falls $\exists q \in Q : (q_0, Z_0) \xRightarrow{w} (q, \varepsilon)$. Die von K mit leerem Keller akzeptierte Sprache ist

$$L_\varepsilon(K) = \{w \in \Sigma^* \mid K \text{ akzeptiert } w \text{ mit leerem Keller}\}.$$

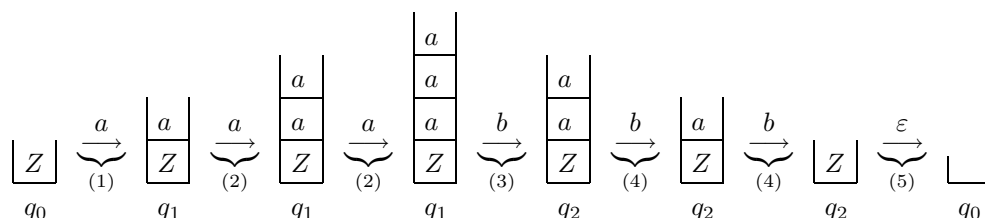
☒ 5.3.3

Beispiel L_1 (Fortsetzung):

Wir konstruieren einen Kellerautomaten, der die Klammersprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ mit Endzustand akzeptiert. Dazu setzen wir $K_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, Z\}, Z, \delta, q_0, \{q_0\})$, wobei δ aus den folgenden Transitionen besteht:

- (1) $(q_0, Z) \xrightarrow{a} (q_1, aZ)$
- (2) $(q_1, a) \xrightarrow{a} (q_1, aa)$
- (3) $(q_1, a) \xrightarrow{b} (q_2, \varepsilon)$
- (4) $(q_2, a) \xrightarrow{b} (q_2, \varepsilon)$
- (5) $(q_2, Z) \xrightarrow{\varepsilon} (q_0, \varepsilon)$.

Zum Beispiel wird das Wort $aaabbb$ folgendermaßen akzeptiert:



Im Allgemeinen akzeptiert K_1 das Wort $a^n b^n$ für $n \geq 1$ durch $2n+1$ Transitionen, die wir der Deutlichkeit halber mit ihren Nummern (1) bis (5) indiziert haben:

$$\begin{array}{l}
 (q_0, Z) \xrightarrow{a}_{(1)} (q_1, aZ) \xrightarrow{a}_{(2)} (q_1, aaZ) \xrightarrow{a}_{(2)} \dots \xrightarrow{a}_{(2)} (q_1, a^n Z) \\
 \xrightarrow{b}_{(3)} (q_2, a^{n-1} Z) \xrightarrow{b}_{(4)} (q_2, a^{n-2} Z) \xrightarrow{b}_{(4)} \dots \xrightarrow{b}_{(4)} (q_1, Z) \\
 \xrightarrow{\varepsilon}_{(5)} (q_0, \varepsilon).
 \end{array}$$

Damit gilt für $n \geq 1$ die Beziehung $(q_0, Z) \xRightarrow{a^n b^n} (q_0, \varepsilon)$. Für $n = 0$ gilt trivialerweise $(q_0, Z) \xRightarrow{\varepsilon} (q_0, Z)$. Da q_0 Endzustand von K_1 ist, folgt $L_1 \subseteq L(K_1)$.

Für die Inklusion $L(K_1) \subseteq L_1$ analysieren wir das Transitionsverhalten von K_1 . Dazu indizieren wir die Transitionen wie eben. Wir stellen zunächst fest, dass K_1 *deterministisch* ist, d.h., beim buchstabenweisen Lesen eines gegebenen Eingabewortes ist jeweils genau eine Transition anwendbar. Es sind genau die folgenden Transitionssequenzen möglich, wobei $n \geq m \geq 0$ gilt:

$$\begin{array}{l}
 (q_0, Z) \xrightarrow{a}_{(1)} \circ \left(\xrightarrow{a}_{(2)} \right)^n (q_1, a^{n+1} Z) \quad \text{einmal (1), gefolgt von } n\text{-mal (2)} \\
 (q_0, Z) \xrightarrow{a}_{(1)} \circ \left(\xrightarrow{a}_{(2)} \right)^n \circ \xrightarrow{b}_{(3)} \circ \left(\xrightarrow{b}_{(4)} \right)^m (q_2, a^{n-m} Z) \quad \text{nach (3) wird kein } a \text{ mehr gelesen} \\
 (q_0, Z) \xrightarrow{a}_{(1)} \circ \left(\xrightarrow{a}_{(2)} \right)^n \circ \xrightarrow{b}_{(3)} \circ \left(\xrightarrow{b}_{(4)} \right)^n \circ \xrightarrow{\varepsilon}_{(5)} (q_0, \varepsilon) \quad \text{(5) geht nur mit } n = m.
 \end{array}$$

Daher akzeptiert K_1 nur Wörter der Form $a^n b^n$. Insgesamt gilt also $L(K_1) = L_1$.

Wir bemerken ferner, dass bis auf den Fall $n = 0$ der Automat K_1 alle Wörter $a^n b^n$ mit dem leeren Keller akzeptiert; es gilt $L_\varepsilon(K_1) = \{a^n b^n \mid n \geq 1\} = L(K_1) \setminus \{\varepsilon\}$.

Ende des Beispiels L_1 (Fortsetzung)

Achtung! In vielen Lehrbüchern wird der Begriff „Konfiguration“ als Tripel (q, w, γ) gefasst, wobei die zusätzliche Komponente w das noch nicht gelesene Eingabewort bedeutet. Ein Schritt eines Kellerautomaten wird dann als

$$(q, av, \gamma) \xrightarrow{a} (q', v, \gamma') \quad (a \text{ wird gelesen; } v \in \Sigma^*) \\ \text{bzw. als } (q, v, \gamma) \xrightarrow{\varepsilon} (q', v, \gamma') \quad (\text{spontaner Übergang; } v \in \Sigma^*)$$

beschrieben. Akzeptanz wird dann so definiert, dass eine Konfiguration (q, ε, γ) erreicht wird, worin $q \in F$ (Akzeptanz durch Endzustand) oder $\gamma = \varepsilon$ (Akzeptanz durch leeren Keller) gilt.

Neben den formalen gibt es keine inhaltlichen Unterschiede zwischen diesen Definitionen. Schreibt man Konfigurationen als Tripel, sind die Zeichen oberhalb des \longrightarrow entbehrlich: statt $(q, av, \gamma) \xrightarrow{a} (q', v, \gamma')$ oder $(q, v, \gamma) \xrightarrow{\varepsilon} (q', v, \gamma')$ könnte man ohne Informationsverlust auch $(q, av, \gamma) \longrightarrow (q', v, \gamma')$ bzw. $(q, v, \gamma) \longrightarrow (q', v, \gamma')$ schreiben. In der Zweitupelschreibweise (q, γ) darf das Zeichen (a bzw. ε) über dem Pfeil \longrightarrow aber nicht weggelassen werden.

5.3.2 Äquivalenz der Akzeptanzbedingungen

Wir wollen jetzt zeigen, dass die beiden Akzeptanzbedingungen äquivalent in Bezug auf die akzeptierte Sprachklasse sind. Dazu benötigen wir das folgende Top-Lemma. Es besagt anschaulich, dass Veränderungen an der Spitze (dem Top) des Kellers unabhängig vom tieferen Inhalt des Kellers sind.

Lemma 5.3.4 TOP DES KELLERS

Sei $K = (Q, \Sigma, \Gamma, Z_0, \delta, q_0, F)$ ein Kellerautomat. Dann gilt für alle $w \in \Sigma^*$, $q, q' \in Q$, $Z \in \Gamma$ und $\gamma \in \Gamma^*$: wenn $(q, Z) \xrightarrow{w} (q', \varepsilon)$, so auch $(q, Z\gamma) \xrightarrow{w} (q', \gamma)$.

Beweis: Weggelassen. Der Beweis ist einfach: man zeigt (durch Induktion über die Länge von Übergangsfolgen), dass der Teil des Kellers, der unterhalb des obersten Elements steht, keine Rolle bei den jeweiligen Übergängen spielt. □ 5.3.4

Satz 5.3.5 AKZEPTANZ

- (1) Zu jedem Kellerautomaten A kann ein Kellerautomat B mit $L(A) = L_\varepsilon(B)$ konstruiert werden.
- (2) Zu jedem Kellerautomaten A kann ein Kellerautomat B mit $L_\varepsilon(A) = L(B)$ konstruiert werden.

Beweis: Sei $A = (Q, \Sigma, \Gamma, Z_0, \delta_A, q_0, F)$.

Zu (1): Die Beweisidee ist einfach: B arbeitet wie A und leert von Endzuständen aus den Keller. Es muss jedoch darauf geachtet werden, dass B durch Eingabewörter, die A nicht akzeptiert, die aber bei A zu leerem Keller führen, nicht auch schon einen leeren Keller erhält. Deshalb benutzt B ein zusätzliches Symbol $\#$ zur Markierung des Kellerbodens. Genauer konstruieren wir:

$$B = (Q \cup \{q_B, q_\varepsilon\}, \Sigma, \Gamma \cup \{\#\}, \#, \delta_B, q_B, \emptyset)$$

mit $q_B, q_\varepsilon \notin Q$ und $\# \notin \Gamma$ und folgender Transitionsrelation:

$$\begin{aligned} \rightarrow_B &= \{((q_B, \#, \varepsilon), (q_0, Z_0\#))\} && \text{„Starten von } A\text{“} \\ &\cup \rightarrow_A && \text{„Arbeiten wie } A\text{“} \\ &\cup \{((q, Z, \varepsilon), (q_\varepsilon, \varepsilon)) \mid q \in F, Z \in \Gamma \cup \{\#\}\} \\ &\cup \{((q_\varepsilon, Z, \varepsilon), (q_\varepsilon, \varepsilon)) \mid Z \in \Gamma \cup \{\#\}\} && \text{„Leeren des Kellers“} \end{aligned}$$

Dann gilt für alle $w \in \Sigma^*$, $q \in F$ und $\gamma \in \Gamma^*$:

$$(q_0, Z_0) \xrightarrow{w}_A (q, \gamma) \quad \text{gdw.} \quad (q_B, \#) \xrightarrow{\varepsilon}_B (q_0, Z_0\#) \xrightarrow{w}_B (q, \gamma\#) \xrightarrow{\varepsilon}_B (q_\varepsilon, \varepsilon).$$

(Hier wird das Top-Lemma angewandt: B kann A simulieren, weil das unterste Zeichen $\#$ dabei keine Rolle spielt.) Man erhält also $L(A) = L_\varepsilon(B)$.

Zu (2): Beweisidee: B arbeitet wie A , benutzt aber ein zusätzliches Symbol $\#$ zur Markierung des Kellerbodens. B hat das Zeichen $\#$ als Kellerstartsymbol und pusht zunächst Z_0 über $\#$. Wenn B das Zeichen $\#$ danach sieht, hat A seinen Keller geleert, und B geht in einen Endzustand über. Die genaue Konstruktion von B ist eine Übungsaufgabe. □ 5.3.5

Beispiel IE_ε (zu Teil (2) des Beweises):

Wir konstruieren einen **if/else**-Fehlererkennungsautomaten, der anzeigt, dass in einem Programm einem **else** (symbolisiert durch das Eingabezeichen e) kein passendes zuvor vorkommendes **if** (symbolisiert durch das Zeichen i) zugeordnet werden kann. Das ist dann der Fall, wenn für ein gerade gelesenes **else** die Anzahl der bislang gelesenen **if** um 1 kleiner ist als die Anzahl der bislang gelesenen **else**. Wir benutzen also einen Keller, um die Differenz zwischen den gelesenen **if** und den gelesenen **else** zu beobachten, und wir suchen das *erste* Vorkommen eines solchen Fehlers. Beim Lesen eines **if** wird ein Kellerzeichen gepusht, beim Lesen eines **else** wird der Keller gepopt, und es ist nur ein Zustand nötig:

$$IE_\varepsilon = (\{q\}, \{i, e\}, \{Z\}, Z, \delta, q, \emptyset)$$

$$\begin{aligned} \text{mit } \delta : \quad (q, Z) &\xrightarrow{i} (q, ZZ) \\ (q, Z) &\xrightarrow{e} (q, \varepsilon). \end{aligned}$$

Dann gilt, wie gewünscht,

$$L_\varepsilon(IE_\varepsilon) = \{w \in \{i, e\}^* \mid \neg \text{gut}(w), \text{ und für jeden Präfix } v \text{ von } w \text{ mit } v \neq w \text{ gilt } \text{gut}(v)\}.$$

Dabei soll das Prädikat $\text{gut}(x)$ für ein Wort $x \in \{i, e\}^*$ gelten, wenn die Anzahl der i s in x größer oder gleich der Anzahl der e s in x ist.

Wir wandeln diesen Automaten nun gemäß Punkt (2) des Beweises in einen Automaten IE mit $L_\varepsilon(IE_\varepsilon) = L(IE)$ um. IE hat ein neues Kellerzeichen $\#$ und zwei neue Zustände, einen Anfangszustand, von dem aus Z auf den Keller gelegt wird, und einen Endzustand, der erreicht wird, wenn IE_ε seinen Keller geleert hat:

$$\begin{aligned}
 IE &= (\{p, q, r\}, \{i, e\}, \{\#, Z\}, \#, \delta, p, \{r\}) \\
 \text{mit } \delta : & \quad (p, \#) \xrightarrow{\varepsilon} (q, Z\#) && (Z \text{ wird auf den Keller gelegt}) \\
 & \quad (q, Z) \xrightarrow{i} (q, ZZ) \\
 & \quad (q, Z) \xrightarrow{e} (q, \varepsilon) && \left. \vphantom{\begin{matrix} (q, Z) \xrightarrow{i} (q, ZZ) \\ (q, Z) \xrightarrow{e} (q, \varepsilon) \end{matrix}} \right\} (\text{arbeite wie } IE_\varepsilon) \\
 & \quad (q, \#) \xrightarrow{\varepsilon} (r, \varepsilon) && (IE_\varepsilon \text{ hat Keller geleert, also } \rightsquigarrow \text{ Endzustand})
 \end{aligned}$$

Ende des Beispiels IE_ε

Auf Grund von Satz 5.3.5 erhebt sich die Frage, warum beide Begriffe definiert worden sind und wieso nicht die Akzeptanz durch Endzustände (wie bei endlichen Automaten) genügt. Das hat hauptsächlich technische Gründe, denn für den Beweis des Hauptsatzes über Kellerautomaten (dass kontextfreie Grammatiken genau das Gleiche leisten wie Kellerautomaten, siehe den nächsten Abschnitt) ist die Akzeptanz durch leeren Keller viel günstiger als die Akzeptanz durch Endzustände.

5.3.3 Äquivalenz von Kellerautomaten und kontextfreien Grammatiken

Wir wollen jetzt zeigen, dass die nichtdeterministischen Kellerautomaten genau die kontextfreien Sprachen (mit leerem Keller) akzeptieren. Zunächst konstruieren wir zu einer gegebenen kontextfreien Grammatik G einen Kellerautomaten, der einen nichtdeterministischen „Top-down-Parser“ der Sprache $L(G)$ darstellt.

Satz 5.3.6 KONTEXTFREIE GRAMMATIK \rightsquigarrow KELLERAUTOMAT

Zu jeder kontextfreien Grammatik G kann man einen nichtdeterministischen Kellerautomaten K mit $L_\varepsilon(K) = L(G)$ konstruieren.

Beweis: Sei $G = (N, \Sigma, P, S)$. Wir konstruieren K (akzeptierend durch leeren Keller) so, dass die Linksableitungen in G simuliert werden. K hat nur einen Zustand:

$$K = (\{q\}, \Sigma, N \cup \Sigma, S, \delta, q, \emptyset),$$

wobei die Transitionsrelation δ aus den folgenden Transitionstypen besteht:

- (1) $(q, A) \xrightarrow{\varepsilon} (q, u)$ falls $(A \rightarrow u) \in P$,
- (2) $(q, a) \xrightarrow{a} (q, \varepsilon)$ falls $a \in \Sigma$.

Die Arbeitsweise von K ist folgendermaßen. Zunächst steht S im Keller. Eine Regelanwendung $(A \rightarrow u)$ der Grammatik wird im Keller nachvollzogen, indem das oberste Kellersymbol A durch u ersetzt wird. Stehen dann Terminalsymbole an der Kellerspitze, so werden sie mit den Symbolen des Eingabewortes verglichen und bei Übereinstimmung aus dem Keller entfernt. Auf diese Weise wird schrittweise eine

Linksableitung des Eingabewortes durch K hergestellt. Gelingt diese Ableitung, so akzeptiert K das Eingabewort mit dem leeren Keller. Die Anwendung der Transitionen vom Typ (1) ist nichtdeterministisch, wenn es mehrere Regeln mit der linken Seite A in P gibt. Der Zustand q spielt beim Transitionsverhalten von K keine Rolle, muss aber der Vollständigkeit halber bei der Definition von K angegeben werden.

Um zu zeigen, dass $L_\varepsilon(K) = L(G)$ gilt, untersuchen wir den Zusammenhang zwischen Linksableitungen in G und Transitionsfolgen in K genauer. Dabei benutzen wir für Wörter $w \in (N \cup \Sigma)^*$ folgende Abkürzungen:

- w_Σ ist der längste Präfix von w mit $w_\Sigma \in \Sigma^*$,
- w_R ist der Rest von w , definiert durch $w = w_\Sigma w_R$.

Behauptung 1:

Für alle $A \in N$, $w \in (N \cup \Sigma)^*$, $n \geq 0$ und Linksableitungen

$$A \underbrace{\rightarrow_G \dots \rightarrow_G}_{n\text{-mal}} w$$

der Länge n gilt $(q, A) \xRightarrow{w_\Sigma} (q, w_R)$.

Beweis mit Induktion über n :

- $n = 0$: Dann ist $w = A$, also $w_\Sigma = \varepsilon$ und $w_R = A$. Trivialerweise gilt $(q, A) \xRightarrow{\varepsilon} (q, A)$.
- $n \rightsquigarrow n + 1$: Wir analysieren den letzten Schritt einer Linksableitung der Länge $n + 1$:

$$A \underbrace{\rightarrow_G \dots \rightarrow_G}_{n\text{-mal}} \tilde{w} = \tilde{w}_\Sigma B v \rightarrow_G \tilde{w}_\Sigma u v = w$$

für $B \in N$ und $u, v \in (N \cup \Sigma)^*$ mit $(B \rightarrow u) \in P$. Nach Induktionsvoraussetzung gilt

$$(q, A) \xRightarrow{\tilde{w}_\Sigma} (q, B v).$$

Mit dem Transitionstyp (1) folgt

$$(q, B v) \xRightarrow{\varepsilon} (q, u v)$$

wegen $(B \rightarrow u) \in P$. Mit dem Transitionstyp (2) ($|(uv)_\Sigma|$ -mal angewendet) gilt außerdem

$$(q, u v) \xRightarrow{(uv)_\Sigma} (q, (u v)_R).$$

Da $w_\Sigma = (\tilde{w}_\Sigma u v)_\Sigma = \tilde{w}_\Sigma (u v)_\Sigma$ und $w_R = (\tilde{w}_\Sigma u v)_R = (u v)_R$ gilt, erhalten wir insgesamt

$$(q, A) \xRightarrow{w_\Sigma} (q, w_R).$$

□ Behauptung 1

Behauptung 2:

Für alle $A \in N$, $m \geq 0$, $\alpha_1, \dots, \alpha_m \in \Sigma \cup \{\varepsilon\}$, $\gamma_0, \dots, \gamma_m \in (N \cup \Sigma)^*$ und alle Transitionsfolgen

$$(q, A) = (q, \gamma_0) \xrightarrow{\alpha_1} (q, \gamma_1) \dots \xrightarrow{\alpha_m} (q, \gamma_m)$$

der Länge m gilt $A \xrightarrow*_G \alpha_1 \dots \alpha_m \gamma_m$.

Beweis mit Induktion über m :

- $m = 0$: Dann ist $\gamma_m = A$ und die Transitionsfolge besteht nur aus $(q, \gamma_0) = (q, A)$. Trivialerweise gilt $A \xrightarrow*_G A$.
- $m \rightsquigarrow m + 1$: Wir analysieren die letzte Transition

$$(q, \gamma_m) \xrightarrow{\alpha_{m+1}} (q, \gamma_{m+1}).$$

Nach Induktionsvoraussetzung gilt $A \xrightarrow*_G \alpha_1 \dots \alpha_m \gamma_m$.

Fall $\alpha_{m+1} = \varepsilon$:

Dann wurde Transitionstyp (1) angewandt und die Transition ist von der Form

$$(q, \gamma_m) = (q, Bv) \xrightarrow{\varepsilon} (q, uv) = (q, \gamma_{m+1})$$

für gewisse $B \in N$ und $u, v \in (N \cup \Sigma)^*$ mit $(B \rightarrow u) \in P$. Damit gilt:

$$\begin{aligned} A &\xrightarrow*_G \alpha_1 \dots \alpha_m Bv && (\text{wegen der Induktionsvoraussetzung}) \\ &\xrightarrow_G \alpha_1 \dots \alpha_m uv && (\text{wegen } (B \rightarrow u) \in P) \\ &= \alpha_1 \dots \alpha_m \alpha_{m+1} \gamma_{m+1} && (\text{wegen } \alpha_{m+1} = \varepsilon \text{ und } \gamma_{m+1} = uv). \end{aligned}$$

Fall $\alpha_{m+1} = a \in \Sigma$:

Dann wurde Transitionstyp (2) angewandt und die Transition ist von der Form

$$(q, \gamma_m) = (q, av) \xrightarrow{a} (q, v) = (q, \gamma_{m+1})$$

für ein gewisses $v \in (N \cup \Sigma)^*$. Dann gilt

$$\begin{aligned} A &\xrightarrow*_G \alpha_1 \dots \alpha_m av && (\text{wegen der Induktionsvoraussetzung}) \\ &= \alpha_1 \dots \alpha_m \alpha_{m+1} \gamma_{m+1} && (\text{wegen } \alpha_{m+1} = a \text{ und } \gamma_{m+1} = v). \end{aligned}$$

☒ Behauptung 2

Aus den Behauptungen 1 und 2 folgt insbesondere, dass für alle Wörter $w \in \Sigma^*$ gilt:

$$\begin{aligned} S \xrightarrow*_G w &\Leftrightarrow (q, S) \xRightarrow{w} (q, \varepsilon) \\ &\Leftrightarrow K \text{ akzeptiert } w \text{ mit leerem Keller.} \end{aligned}$$

Also gilt $L_\varepsilon(K) = L(G)$, wie behauptet.

☒ 5.3.6

Beispiel L_1 (Fortsetzung):

Wir betrachten noch einmal die Sprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$. Wir hatten bereits gesehen, dass diese Sprache durch die kontextfreie Grammatik $G_1 = (\{S\}, \{a, b\}, P_1, S)$, wobei P_1 aus den Produktionen

$$S \rightarrow \varepsilon \mid aSb$$

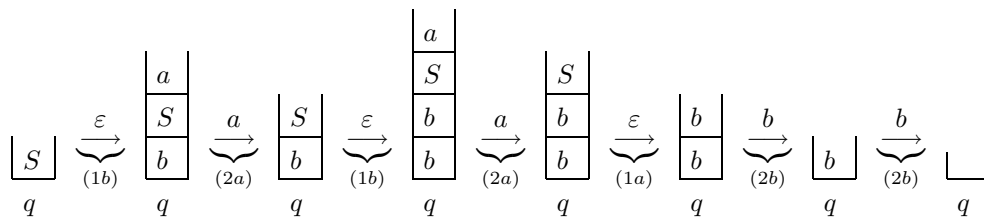
besteht, erzeugt wird: $L(G_1) = L_1$. Die im obigen Beweis benutzte Konstruktion liefert den Kellerautomaten

$$K_1 = (\{q\}, \{a, b\}, \{S, a, b\}, S, \delta, q, \emptyset),$$

wobei die Transitionsrelation δ aus folgenden Transitionen besteht:

- (1a) $(q, S) \xrightarrow{\varepsilon} (q, \varepsilon)$ (von $S \rightarrow \varepsilon$)
- (1b) $(q, S) \xrightarrow{\varepsilon} (q, aSb)$ (von $S \rightarrow aSb$)
- (2a) $(q, a) \xrightarrow{a} (q, \varepsilon)$ (a wird durch a -Übergang akzeptiert)
- (2b) $(q, b) \xrightarrow{b} (q, \varepsilon)$ (b wird durch b -Übergang akzeptiert).

Aus dem Beweis folgt: $L_\varepsilon(K_1) = L(G_1)$. Zur Veranschaulichung sei die Transitionsfolge von K_1 beim Akzeptieren von $aabb$ betrachtet:



Ende des Beispiels L_1 (Fortsetzung)

Jetzt konstruieren wir umgekehrt zu jedem gegebenen Kellerautomat eine passende kontextfreie Grammatik.

Satz 5.3.7 KELLERAUTOMAT \rightsquigarrow KONTEXTFREIE GRAMMATIK

Zu jedem Kellerautomaten K kann man eine kontextfreie Grammatik G mit $L(G) = L_\varepsilon(K)$ konstruieren.

Beweis: Sei $K = (Q, \Sigma, \Gamma, Z_0, \delta, q_0, F)$. Wir konstruieren $G = (N, \Sigma, P, S)$ mit

$$N = \{S\} \cup \{[qZq'] \mid q, q' \in Q \text{ und } Z \in \Gamma\}.$$

G soll die Rechenschritte von K durch Linksableitungen simulieren. Die Idee für die Nichtterminalsymbole $[qZq']$ ist folgende:

- (1) Von $[qZq']$ aus sollen in G alle Wörter $w \in \Sigma^*$ erzeugt werden, die K von der Konfiguration (q, Z) aus mit leerem Keller und dem dann erreichten Zustand q' akzeptieren kann: $(q, Z) \xRightarrow{w} (q', \varepsilon)$. Oder: „ K kann von q in q' übergehen und insgesamt Z vom Keller entfernen“.

- (2) Eine Transition $(q, Z) \xrightarrow{\alpha} (r_0, Z_1 \dots Z_k)$ ($k \geq 1$) von K führt deshalb in G zu folgenden Produktionen:

$$\underbrace{[qZr_k]}_{\text{„}Z \text{ wird entfernt“}} \rightarrow \underbrace{\alpha [r_0Z_1r_1] [r_1Z_2r_2] \dots [r_{k-1}Z_kr_k]}_{\text{„}Z_1 \dots Z_k \text{ wird aufgebaut, dann werden die } Z_i \text{ der Reihe nach entfernt“}}$$

wobei die r_1, \dots, r_k über ganz Q laufen. Von $[r_0Z_1r_1]$ aus werden die Wörter erzeugt, die von K bis zum Abbau des Symbols Z_1 akzeptiert werden, von $[r_1Z_2r_2]$ die Wörter, die von K bis zum Abbau des Symbols Z_2 akzeptiert werden, usw. Die Zwischenzustände r_1, \dots, r_{k-1} sind diejenigen, die K unmittelbar nach dem Abbau der Symbole Z_1, \dots, Z_{k-1} erreicht.

Für $k = 0$ wird eine entsprechende Regel eingeführt.

Insgesamt besteht P aus den folgenden Produktionen:

- **Typ (1):** $(S \rightarrow [q_0Z_0r]) \in P$,
für alle $r \in Q$.
- **Typ (2):** Für jede Transition $(q, Z) \xrightarrow{\alpha} (r_0, Z_1 \dots Z_k)$ mit $\alpha \in \Sigma \cup \{\varepsilon\}$ und $k \geq 1$ in K :
 $([qZr_k] \rightarrow \alpha [r_0Z_1r_1] [r_1Z_2r_2] \dots [r_{k-1}Z_kr_k]) \in P$,
mit allen Kombinationen von $r_1, \dots, r_k \in Q$.
- **Typ (3):** Für jede Transition $(q, Z) \xrightarrow{\alpha} (r, \varepsilon)$ in K :
 $([qZr] \rightarrow \alpha) \in P$.

Um zu zeigen, dass $L(G) = L_\varepsilon(K)$ gilt, untersuchen wir den Zusammenhang zwischen Ableitungen in G und Transitionfolgen in K .

Behauptung 1:

Für alle $q, r \in Q$, $Z \in \Gamma$, $w \in \Sigma^*$, $n \geq 1$ und Ableitungen

$$[qZr] \underbrace{\xrightarrow{G} \dots \xrightarrow{G}}_{n\text{-mal}} w$$

der Länge n gilt $(q, Z) \xRightarrow{w} (r, \varepsilon)$.

Beweis mit Induktion über n :

- $n = 1$: Aus $[qZr] \rightarrow_G w$ folgt wegen $w \in \Sigma^*$, dass es sich um den Produktionstyp (3) handelt, also $w = \alpha \in \Sigma \cup \{\varepsilon\}$ und $(q, Z) \xrightarrow{\alpha} (q', \varepsilon)$ gilt.
Also gilt $(q, Z) \xRightarrow{\alpha} (q', \varepsilon)$ nach Definition von $\xRightarrow{\alpha}$.
- $n \rightsquigarrow n + 1$: Wir analysieren den ersten Schritt einer Ableitung der Länge $n + 1$:

$$[qZr_k] \rightarrow_G \alpha [r_0Z_1r_1] \dots [r_{k-1}Z_kr_k] \underbrace{\xrightarrow{G} \dots \xrightarrow{G}}_{n\text{-mal}} \alpha w_1 \dots w_k = w,$$

wobei $\alpha \in \Sigma \cup \{\varepsilon\}$, $w_1, \dots, w_k \in \Sigma^*$ und

$$[r_{i-1}Z_i r_i] \underbrace{\rightarrow_G \dots \rightarrow_G}_{\leq n\text{-mal}} w_i$$

für $i = 1, \dots, k$ gilt, da es sich um einen Schritt des Produktionstyps (2) handelt. Nach Definition von P in G gilt

$$(q, Z) \xrightarrow{\alpha} (r_0, Z_1 \dots Z_k)$$

mit $k \geq 1$, und nach Induktionsvoraussetzung gilt

$$(r_{i-1}, Z_i) \xRightarrow{w_i} (r_i, \varepsilon)$$

für $i = 1, \dots, k$. Insgesamt erhalten wir mit dem Top-Lemma:

$$\begin{aligned} (q, Z) &\xRightarrow{\alpha} (r_0, Z_1 \dots Z_k) \\ &\xRightarrow{w_1} (r_1, Z_2 \dots Z_k) \\ &\vdots \\ &\xRightarrow{w_k} (r_k, \varepsilon) = (q', \varepsilon). \end{aligned}$$

Also $(q, Z) \xRightarrow{w} (r_k, \varepsilon)$, wie gewünscht.

☐ Behauptung 1

Behauptung 2:

Für alle $q, q' \in Q$, $Z \in \Gamma$, $n \geq 1$, $\alpha_1, \dots, \alpha_n \in \Sigma \cup \{\varepsilon\}$ und alle Transitionsfolgen

$$(q, Z) \xrightarrow{\alpha_1} \circ \dots \circ \xrightarrow{\alpha_n} (q', \varepsilon)$$

der Länge n gilt $[qZq'] \xrightarrow*_G \alpha_1 \dots \alpha_n$.

Beweis mit Induktion über n :

- $n = 1$: Dann gilt $(q, Z) \xrightarrow{\alpha_1} (q', \varepsilon)$.

Nach Definition von P in G (Produktionstyp (3)) folgt $[qZq'] \rightarrow_G \alpha_1$.

- $n \rightsquigarrow n + 1$: Wir analysieren den ersten Transitionsschritt einer Folge der Länge $n + 1$:

$$(q, Z) \xrightarrow{\alpha_1} (r_0, Z_1 \dots Z_k) \xrightarrow{\alpha_2} \circ \dots \circ \xrightarrow{\alpha_{n+1}} (q', \varepsilon),$$

wobei $k \geq 1$ gilt. Wir betrachten den Abbau des Kellerinhalts $Z_1 \dots Z_k$. Es gibt Transitionsfolgen

$$\begin{aligned} (r_0, Z_1) &\xrightarrow{\alpha_{11}} \circ \dots \circ \xrightarrow{\alpha_{1m_1}} (r_1, \varepsilon) \\ &\vdots \\ (r_{k-1}, Z_k) &\xrightarrow{\alpha_{k1}} \circ \dots \circ \xrightarrow{\alpha_{km_k}} (r_k, \varepsilon) = (q', \varepsilon) \end{aligned}$$

mit

$$\alpha_2 \dots \alpha_{n+1} = \alpha_{11} \dots \alpha_{1m_1} \dots \alpha_{k1} \dots \alpha_{km_k} \quad \text{und} \quad m_1, \dots, m_k \leq n. \quad (5.5)$$

Nach Definition von P in G (Produktionstyp (2)) gilt

$$[qZr_k] \rightarrow \alpha_1[r_0Z_1r_1] \dots [r_{k-1}Z_kr_k] \quad \text{mit} \quad r_k = q',$$

und nach Induktionsvoraussetzung gilt

$$[r_{i-1}Z_i r_i] \rightarrow_G^* \alpha_{i1} \dots \alpha_{im_i}$$

für $i = 1, \dots, k$. Zusammen mit (5.5) ergibt sich

$$[qZr_k] = [qZq'] \rightarrow_G^* \alpha_1 \alpha_2 \dots \alpha_{n+1},$$

wie gewünscht. ☐ Behauptung 2

Aus den Behauptungen 1 und 2 und der Definition des Produktionstyps (1) folgern wir: für alle $q \in Q$ und $w \in \Sigma^*$ gilt

$$S \rightarrow_G [q_0Z_0q] \rightarrow_G^* w \quad \text{genau dann, wenn} \quad (q_0, Z_0) \xRightarrow{w} (q, \varepsilon).$$

Damit gilt $L(G) = L_\varepsilon(K)$. ☐ 5.3.7

Beispiel IE_ε (Fortsetzung):

Wir starten mit dem Kellerautomaten

$$IE_\varepsilon = (\{q\}, \{i, e\}, \{Z\}, Z, \delta, q, \emptyset)$$

$$\text{mit} \quad \delta : \quad (q, Z) \xrightarrow{i} (q, ZZ) \\ (q, Z) \xrightarrow{e} (q, \varepsilon),$$

der die Sprache

$$L_\varepsilon(IE_\varepsilon) = \{w \in \{i, e\}^* \mid \neg \text{gut}(w), \text{ und für jeden Präfix } v \text{ von } w \text{ mit } v \neq w \text{ gilt } \text{gut}(v)\}$$

akzeptiert, und wenden die Konstruktion von Satz 5.3.7 an, um eine äquivalente kontextfreie Grammatik zu gewinnen. Das Beispiel ist besonders einfach, weil es nur einen Zustand gibt.

- Die Grammatik hat ein Startsymbol S , für das es eine einzige Produktion gibt:

$$S \rightarrow [qZq].$$

(Im Allgemeinen: hätte der Automat n Zustände q_0, \dots, q_{n-1} mit Anfangszustand q_0 , dann gäbe es für S genau n Produktionen $S \rightarrow [q_0Zq_0], \dots, S \rightarrow [q_0Zq_{n-1}]$.)

- Aus der Transition $(q, Z) \xrightarrow{i} (q, ZZ)$ des Automaten folgt eine einzige Produktion:

$$[qZq] \rightarrow i[qZq][qZq].$$

(Im Allgemeinen: hätte der Automat n Zustände q_0, \dots, q_{n-1} , dann gäbe es für die Transition

$(q, Z) \xrightarrow{i} (q, ZZ)$ genau n^2 Produktionen $[qZr_0] \rightarrow i[qZr_1][r_1Zr_0]$ wobei sowohl r_0 als auch r_1 über die ganze Zustandsmenge variieren.)

- Aus der Transition $(q, Z) \xrightarrow{e} (q, \varepsilon)$ des Automaten folgt eine einzige Produktion:

$$[qZq] \rightarrow e.$$

Insgesamt haben wir also die Grammatik

$$S \rightarrow R, \quad R \rightarrow iRR, \quad R \rightarrow e,$$

wobei wir vereinfachend R für $[qZq]$ gesetzt haben. Man sieht, dass S und R die gleichen Wörter produzieren, und dass die Grammatik deswegen noch etwas vereinfacht werden kann:

$$G: \quad S \rightarrow iSS, \quad S \rightarrow e.$$

Es gilt

$$L(G) = \{w \in \{i, e\}^* \mid \text{der letzte Buchstabe in } w \text{ ist } e \text{ und es gilt } \text{anz}(i, w) = \text{anz}(e, w) - 1\}$$

also $L(G) = L_\varepsilon(IE_\varepsilon)$, wie behauptet. Hier ist eine Beispiel-Herleitung des Wortes $ieiee \in L(G)$:

$$S \vdash_G iSS \vdash_G ieS \vdash_G ieiSS \vdash_G ieieS \vdash_G ieiee.$$

Ende des Beispiels IE_ε (Fortsetzung)

Beispiel L_1 (Fortsetzung):

Wir betrachten einen Kellerautomaten, der die Klammersprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ mit leerem Keller akzeptiert:

$$K_{1,\varepsilon} = (\{q_0, q_1\}, \{a, b\}, \{a, Z\}, Z, \delta, q_0, \emptyset)$$

$$\text{mit } \delta: \quad (1) \quad (q_0, Z) \xrightarrow{\varepsilon} (q_0, \varepsilon)$$

$$(2) \quad (q_0, Z) \xrightarrow{a} (q_0, a)$$

$$(3) \quad (q_0, a) \xrightarrow{a} (q_0, aa)$$

$$(4) \quad (q_0, a) \xrightarrow{b} (q_1, \varepsilon)$$

$$(5) \quad (q_1, a) \xrightarrow{b} (q_1, \varepsilon).$$

Nach Anwendung der Konstruktion von Satz 5.3.7 erhalten wir die folgende Grammatik:

$$(0): \quad S \rightarrow [q_0Zq_0], \quad S \rightarrow [q_0Zq_1],$$

$$(1): \quad [q_0Zq_0] \rightarrow \varepsilon,$$

$$(2): \quad [q_0Zq_0] \rightarrow a[q_0aq_0], \quad [q_0Zq_1] \rightarrow a[q_0aq_1],$$

$$(3): \quad [q_0aq_0] \rightarrow a[q_0aq_0][q_0aq_0], \quad [q_0aq_0] \rightarrow a[q_0aq_1][q_1aq_0], \\ [q_0aq_1] \rightarrow a[q_0aq_0][q_0aq_1], \quad [q_0aq_1] \rightarrow a[q_0aq_1][q_1aq_1],$$

$$(4): \quad [q_0aq_1] \rightarrow b,$$

$$(5): \quad [q_1aq_1] \rightarrow b,$$

oder etwas übersichtlicher aufgeschrieben:

- (0) : $S \rightarrow A, \quad S \rightarrow B,$
- (1) : $A \rightarrow \varepsilon,$
- (2) : $A \rightarrow aC, \quad B \rightarrow aD,$
- (3) : $C \rightarrow aCC, \quad C \rightarrow aDE,$
 $D \rightarrow aCD, \quad D \rightarrow aDF,$
- (4) : $D \rightarrow b,$
- (5) : $F \rightarrow b,$

die folgendermaßen vereinfacht werden kann (Begründung: $S \rightarrow B, B \rightarrow aD$ kann zu $S \rightarrow aD$ zusammengezogen werden, E und dann auch C sind unnötig, usw.):

$$G : \quad S \rightarrow \varepsilon \mid aD, \quad D \rightarrow aDF \mid b, \quad F \rightarrow b.$$

Hier ist eine Beispiel-Herleitung des Wortes $aabb \in L(G)$:

$$S \vdash_G aD \vdash_G aaDF \vdash_G aabF \vdash_G aabb.$$

Ende des Beispiels L_1 (Fortsetzung)

Zusammen genommen bedeuten die beiden Sätze 5.3.6 und 5.3.7 die effektive Äquivalenz von kontextfreien Grammatiken und Kellerautomaten.

Außerdem folgt aus den Sätzen, dass jeder Kellerautomat äquivalent umgebaut werden kann zu einem Automaten, der genau einen Zustand hat. Sei nämlich K ein beliebiger Kellerautomat. Durch Satz 5.3.7 gewinnen wir eine äquivalente kontextfreie Grammatik G und durch Satz 5.3.6 dann einen ebenfalls äquivalenten Kellerautomaten K' , der nur einen Zustand hat und mit leerem Keller akzeptiert. (Für Akzeptanz mit Endzustand gibt es keine allgemeine Konstruktion, die nur einen Zustand liefert.)

5.4 Abschlusseigenschaften

Wir untersuchen jetzt, unter welchen Operationen die Klasse der kontextfreien Sprachen abgeschlossen ist. Im Unterschied zu den regulären Sprachen haben wir folgendes Resultat.

Satz 5.4.1 ABSCHLUSSEIGENSCHAFTEN KONTEXTFREIER SPRACHEN

Die Klasse der kontextfreien Sprachen ist abgeschlossen unter den Operationen

- (i) *Vereinigung,*
- (ii) *Konkatenation,*
- (iii) *Iteration,*
- (iv) *Durchschnitt mit regulären Sprachen.*

Dagegen ist die Klasse der kontextfreien Sprachen nicht abgeschlossen unter den Operationen

- (v) *Durchschnitt,*
- (vi) *Komplement.*

Beweis:

Seien $L_1, L_2 \subseteq \Sigma^*$ zwei kontextfreie Sprachen. Dann gibt es kontextfreie Grammatiken $G_1 = (N_1, \Sigma, P_1, S_1)$ mit $L(G_1) = L_1$ und $G_2 = (N_2, \Sigma, P_2, S_2)$ mit $L(G_2) = L_2$. Ohne Beschränkung der Allgemeinheit kann man N_1 und N_2 so wählen, dass $N_1 \cap N_2 = \emptyset$ gilt. Wir zeigen zunächst, dass $L_1 \cup L_2$, $L_1 \cdot L_2$ und L_1^* kontextfrei sind, d.h. die Teile (i), (ii) und (iii) des Satzes. Sei dazu $S \notin N_1 \cup N_2$ ein neues Startsymbol.

Beweis von (i): Wir betrachten die kontextfreie Grammatik $G = (\{S\} \cup N_1 \cup N_2, \Sigma, P, S)$ mit

$$P = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2.$$

Offenbar gilt $L(G) = L_1 \cup L_2$.

Beweis von (ii): Wir betrachten die kontextfreie Grammatik $G = (\{S\} \cup N_1 \cup N_2, \Sigma, P, S)$ mit

$$P = \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2.$$

Offenbar gilt $L(G) = L_1 \cdot L_2$.

Beweis von (iii): Wir betrachten die kontextfreie Grammatik $G = (\{S\} \cup N_1, \Sigma, P, S)$ mit

$$P = \{S \rightarrow \varepsilon, S \rightarrow S_1 S\} \cup P_1.$$

Dann gilt $S \xrightarrow_G^* S_1^n$ für alle $n \geq 0$ und damit $L(G) = L_1^*$.

Beweis von (iv): Wir nutzen die Darstellung von kontextfreien bzw. regulären Sprachen durch Kellerautomaten bzw. endliche Automaten aus.

Seien $L_1 = L(K_1)$ für den Kellerautomaten $K_1 = (Q_1, \Sigma, \Gamma, Z_0, \delta_1, q_{01}, F_1)$ und $L_2 = L(A_2)$ für den DFA $A_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$. Wir konstruieren aus K_1 und A_2 den Kellerautomaten

$$K = (Q_1 \times Q_2, \Sigma, \Gamma, Z_0, \delta, (q_{01}, q_{02}), F_1 \times F_2),$$

wobei die Transitionsrelation von K so definiert ist: für $q_1, q'_1 \in Q_1$, $q_2, q'_2 \in Q_2$, $Z \in \Gamma$, $\alpha \in \Sigma \cup \{\varepsilon\}$ und $\gamma' \in \Gamma^*$,

$$((q_1, q_2), Z) \xrightarrow{\alpha} ((q'_1, q'_2), \gamma') \text{ in } K$$

genau dann, wenn

$$(q_1, Z) \xrightarrow{\alpha}_1 (q'_1, \gamma') \text{ in } K_1 \text{ und } q_2 \xrightarrow{\alpha}_2 q'_2 \text{ in } A_2.$$

Dabei soll $q_2 \xrightarrow{\varepsilon}_2 q'_2$ genau dann gelten, wenn $q_2 = q'_2$.

Die Relation $\xrightarrow{\alpha}$ von K modelliert also das synchrone parallele Fortschreiten der Einzelautomaten K_1 und A_2 . Im Fall $\alpha = \varepsilon$ schreitet nur K_1 durch einen spontanen ε -Übergang voran, während der DFA A_2 im aktuellen Zustand stehen bleibt.

Wir zeigen, dass für das Akzeptieren mit Endzuständen gilt: $L(K) = L(K_1) \cap L(A_2) = L_1 \cap L_2$. Sei dazu $w = a_1 \dots a_n \in \Sigma^*$ mit $n \geq 0$ und $a_i \in \Sigma$ für $i = 1, \dots, n$. Dann gilt:

$$\begin{aligned} w \in L(K) &\Leftrightarrow \exists (q_1, q_2) \in F, \gamma \in \Gamma^*: ((q_{01}, q_{02}), Z_0) \xrightarrow{a_1} \circ \dots \circ \xrightarrow{a_n} ((q_1, q_2), \gamma) \\ &\Leftrightarrow \exists q_1 \in F_1, q_2 \in F_2, \gamma \in \Gamma^*: (q_{01}, Z_0) \xrightarrow{a_1}_1 \circ \dots \circ \xrightarrow{a_n}_1 (q_1, \gamma) \text{ und } q_{02} \xrightarrow{a_1}_2 \circ \dots \circ \xrightarrow{a_n}_2 q_2 \\ &\Leftrightarrow w \in L(K_1) \cap L(A_2). \end{aligned}$$

Beweis von (v): Wir betrachten die beiden Sprachen

$$\begin{aligned} L &= \{a^m b^n c^n \mid m, n \geq 0\} \\ \text{und } L' &= \{a^m b^m c^n \mid m, n \geq 0\}. \end{aligned}$$

Es ist leicht, nachweisen, dass L und L' kontextfrei sind. Zum Beispiel kann L durch die kontextfreie Grammatik $G = (\{S, A, B\}, \{a, b, c\}, P, S)$ mit folgender Produktionsmenge erzeugt werden:

$$\begin{aligned} S &\rightarrow AB, \\ A &\rightarrow \varepsilon \mid aA, \\ B &\rightarrow \varepsilon \mid bBc. \end{aligned}$$

Der Durchschnitt von L und L' liefert jedoch die Sprache

$$L \cap L' = \{a^n b^n c^n \mid n \in \mathbb{N}\},$$

von der wir mit Hilfe des Pumping-Lemmas bereits gezeigt haben, dass sie nicht kontextfrei ist.

Eine analoge Konstruktion wie in Teil (iv) des Beweises funktioniert hier nicht, weil auch in der konstruierten Maschine nur ein einziger Keller zur Verfügung steht. Es ist dem zweiten Kellerautomaten also nicht möglich, gleichzeitig mit dem ersten synchronisiert zu sein, aber auch zu warten, bis die Buchstaben c (statt b) zurückgezählt werden können.

Beweis von (vi): Diese Aussage folgt aus (i) und (v): wären die kontextfreien Sprachen unter Komplement abgeschlossen, so wären sie auch gegenüber Durchschnitt abgeschlossen, weil die Beziehung $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ gilt. ☒ 5.4.1

5.5 Deterministisch kontextfreie Sprachen

Mit Satz 5.3.6 haben wir gezeigt, dass jede kontextfreie Sprache von einem (im allgemeinen nichtdeterministischen) Kellerautomaten akzeptiert werden kann. Es stellt sich die Frage, ob wie bei endlichen Automaten der Nichtdeterminismus beseitigt werden kann, d.h., ob zu einem nichtdeterministischen Kellerautomaten stets auch ein (sprach-)äquivalenter deterministischer Kellerautomat konstruiert werden kann. Diese Frage ist von großer praktischer Bedeutung für die Konstruktion von Parsern für gegebene kontextfreie Sprachen. Wir präzisieren zunächst den Begriff Determinismus für Kellerautomaten und kontextfreie Sprachen.

Definition 5.5.1 DETERMINISTISCHER KELLERAUTOMAT

Ein Kellerautomat $K = (Q, \Sigma, \Gamma, Z_0, \delta, q_0, F)$ heißt *deterministisch* (oder *DPDA*), falls für alle $q \in Q$, $Z \in \Gamma$ und $a \in \Sigma$ gilt:

$$|\delta(q, Z, a)| + |\delta(q, Z, \varepsilon)| \leq 1.$$

Eine kontextfreie Sprache L heißt *deterministisch*, falls es einen deterministischen Kellerautomaten K mit $L = L(K)$ gibt. ☒ 5.5.1

Beispiele:

Die Sprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ ist deterministisch kontextfrei: direkt nach Satz 5.3.6 haben wir einen deterministischen Kellerautomaten K_1 mit $L(K_1) = L_1$ angegeben.

Ebenso ist die Sprache $PAL_c = \{w c w^R \mid w \in \{a, b\}^*\}$ über dem Alphabet $\{a, b, c\}$ deterministisch kontextfrei. Der Grund ist, dass die Mitte des Palindroms eindeutig anhand des dort vorhandenen Buchstabens c erkannt werden kann. Sobald c vom Automaten gelesen wird, kann der vorher aufgebaute Keller wieder abgebaut werden.

Ende der Beispiele

Für deterministische Kellerautomaten gilt der Akzeptanzsatz 5.3.5 nicht, so dass wir $L(K)$ nicht durch $L_\varepsilon(K)$ ersetzen können. Beispielsweise ist die Sprache $\{\varepsilon, a\}$ durch einen deterministischen Kellerautomaten mittels Endzuständen akzeptierbar, aber nicht durch einen deterministischen Kellerautomaten mittels leerem Keller.

Für deterministische Kellerautomaten werden die ε -Übergänge essentiell wichtig. Beispielsweise ist die Sprache

$$\{a^i b^j c a^i \mid i, j \in \mathbb{N}\} \cup \{a^j b^i d a^i \mid i, j \in \mathbb{N}\}$$

über $\{a, b, c, d\}$ deterministisch kontextfrei, mit der intuitiven Begründung, dass nach Erkennen eines Buchstabens c oder d der vorher aufgebaute Keller wieder abgebaut werden kann. Da aber bis dahin nicht klar ist, ob die Anzahl der a s oder der Anzahl der b s im Keller relevant ist, müssen beide gespeichert werden, und um die „falschen“ Buchstaben abzubauen, benötigt man ε -Übergänge.

Wir zeigen jetzt, dass nicht alle kontextfreien Sprachen deterministisch sind. Dazu benutzen wir den folgenden Satz.

Satz 5.5.2 ABSCHLUSS UNTER KOMPLEMENTBILDUNG

Deterministisch kontextfreie Sprachen sind unter Komplementbildung abgeschlossen.

Beweis: (Skizze.)

Man könnte versucht sein, wie bei endlichen Automaten vorzugehen und zu einem deterministischen Kellerautomaten $K = (Q, \Sigma, \Gamma, Z_0, \delta, q_0, F)$ den deterministischen Kellerautomaten K' mit Endzustandsmenge $Q \setminus F$ zu betrachten. Leider gilt im allgemeinen nur $L(K') \subsetneq \Sigma^* \setminus L(K)$. Den Grund dafür, dass nicht alle Wörter aus dem Komplement von $L(K)$ akzeptiert werden, bilden die nichtterminierenden Berechnungen. Wenn z.B. eine Transition

$$(q, A) \xrightarrow{\varepsilon} (q, AA)$$

einmal verwendet wird, dann muss sie im deterministischen Fall immer wieder verwendet werden; der Keller wird dann unendlich lange beschrieben, und K hält nicht an. Falls diese Situation bei einem Eingabewort $w \in \Sigma^*$ eintritt, dann gilt $w \notin L(K)$. Die gleiche Situation tritt aber dann bei K' auf, d.h. es gilt auch $w \notin L(K')$ (sofern w zu diesem Zeitpunkt nicht schon komplett abgearbeitet ist).

Man muss also zunächst K in einen äquivalenten deterministischen Kellerautomaten umwandeln, der für jedes Eingabewort nach endlich vielen Schritten anhält. Eine solche Konstruktion ist bei Kellerautomaten tatsächlich möglich, da man die Menge

$$\{(q, A) \mid \exists \gamma \in \Gamma^* \text{ mit } (q, A) \xrightarrow{\varepsilon} (q, A\gamma)\}$$

effektiv zu K konstruieren und die entsprechenden Transitionen $(q, A) \xrightarrow{\varepsilon} (q', \gamma')$ streichen, bzw. geeignet ersetzen kann. ☒ 5.5.2

Korollar 5.5.3 *Es gibt kontextfreie Sprachen, die nicht deterministisch kontextfrei sind.*

Beweis: Wären alle kontextfreien Sprachen deterministisch kontextfrei, so wären die kontextfreien Sprachen unter Komplementbildung abgeschlossen, im Widerspruch zu Satz 5.4.1. \square 5.5.3

Satz 5.5.4

Deterministische kontextfreie Sprachen sind

- (i) *abgeschlossen gegenüber Komplement und Durchschnitt mit regulären Sprachen,*
- (ii) *nicht abgeschlossen gegenüber Vereinigung, Durchschnitt, Konkatenation und Iteration.*

Beweis:

Teil (i) beweist man mit Hilfe des vorangegangenen Satzes und der selben Konstruktion des nichtdeterministischen Falls (siehe Abschnitt 5.4); der dort aus K_1 und A_2 gebildete Kellerautomat K ist deterministisch, falls K_1 deterministisch ist.

(ii): Beide Sprachen $L_1 = \{a^m b^n c^n \mid m, n \geq 0\}$ und $L_2 = \{a^m b^m c^n \mid m, n \geq 0\}$ sind deterministisch kontextfrei, ihr Durchschnitt ist jedoch noch nicht einmal kontextfrei. Wegen $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ sind die deterministisch kontextfreien Sprachen auch nicht gegen Vereinigung abgeschlossen.

Die Beweise zu Konkatenation und Iteration sind weggelassen (Übungsaufgabe). \square 5.5.4

In Abbildung 5.12 sind die bisher bekannten Abschlusseigenschaften für die drei Klassen der regulären, der kontextfreien und der deterministisch kontextfreien Sprachen tabellarisch zusammengefasst.

Sprachtyp	Schnitt	Vereinigung	Komplement	Konkatenation	Iteration	Schnitt mit regulären Sprachen
Chomsky-3	ja	ja	ja	ja	ja	ja
det. kontextfrei	nein	nein	ja	nein	nein	ja
Chomsky-2	nein	ja	nein	ja	ja	ja

Abbildung 5.12: Abschlusseigenschaften (Zusammenfassung)

Wir geben jetzt ein Beispiel für eine kontextfreie Sprache an, die nicht deterministisch kontextfrei ist:

$$PAL = \{w w^R \mid w \in \{a, b\}^+\},$$

die Sprache aller *nicht leerer Palindrome gerader Länge* über $\{a, b\}$. PAL ist kontextfrei; zur Erzeugung genügen die Produktionen

$$S \rightarrow aa \mid bb \mid aSa \mid bSb.$$

Um zu zeigen, dass PAL nicht deterministisch kontextfrei ist, benutzen wir einen Hilfsoperator Min . Zu einer Sprache $L \subseteq \Sigma^*$ sei

$$Min(L) = \{w \in L \mid \text{es gibt keinen echten Präfix } v \text{ von } w \text{ mit } v \in L\},$$

wobei v *echter Präfix* von w heißt, falls v Präfix von w ist und $v \neq w$ gilt.

Lemma 5.5.5

Wenn L mit $\varepsilon \notin L$ deterministisch kontextfrei ist, dann auch $Min(L)$.

Beweis: Wir betrachten einen deterministischen Kellerautomaten $K = (Q, \Sigma, \Gamma, Z_0, \delta, q_0, F)$ mit $L(K) = L$. Wegen $\varepsilon \notin L$ gilt $q_0 \notin F$. Wir ändern K zu einem Kellerautomaten K_1 ab, der wie K arbeitet, aber in jeder Transitionenfolge *höchstens einmal* in einen Endzustand aus F gerät und dann sofort anhält. Betrachten wir dazu einen neuen Zustand $q_1 \notin Q$ und definieren $K_1 = (Q \cup \{q_1\}, \Sigma, \Gamma, Z_0, \delta_1, q_0, \{q_1\})$ mit

$$\delta_1 = \{(q, Z, \alpha, q', \gamma') \mid q \in Q \setminus F \text{ und } (q, Z, \alpha, q', \gamma') \in \delta\} \\ \cup \{(q, Z, \varepsilon, q_1, Z) \mid q \in F, Z \in \Gamma\}.$$

K_1 ist deterministisch, und es gilt $L(K_1) = Min(L(K))$, weil K deterministisch ist. □ 5.5.5

Satz 5.5.6

Die kontextfreie Sprache PAL ist nicht deterministisch kontextfrei.

Beweis: Annahme: PAL ist deterministisch kontextfrei. Wir konstruieren einen Widerspruch.

Wegen Satz 5.5.4 und Lemma 5.5.5 ist dann auch die Sprache

$$L_0 = Min(PAL) \cap L((ab)^+(ba)^+(ab)^+(ba)^+)$$

deterministisch kontextfrei. Dabei stehen $(ab)^+$ für den regulären Ausdruck $ab(ab)^*$ und $(ba)^+$ für den regulären Ausdruck $ba(ba)^*$. Da alle Wörter in L_0 Palindrome gerader Länge ohne echten Palindrompräfix sind, gilt

$$L_0 = \{(ab)^i(ba)^j(ab)^j(ba)^i \mid i > j > 0\}. \quad (5.6)$$

Insbesondere enthält jedes Wort genau einmal das Teilwort aa (woran man seine Mitte erkennen kann) und genau zweimal das Teilwort bb . Nach dem Pumping-Lemma 5.2.1 gibt es für L_0 eine Zahl $p \in \mathbb{N}$ mit den dort genannten Eigenschaften. Insbesondere lässt sich dann das Wort

$$z = (ab)^{p+1}(ba)^p(ab)^p(ba)^{p+1} \in L_0$$

zerlegen in $z = uvwxy$ mit $|vwx| \leq p$ und $vx \neq \varepsilon$. Weder v noch x können das Teilwort aa enthalten, da dies beim Aufpumpen sofort zu Wörtern führt, die außerhalb von L_0 liegen. Also liegt v links von der Mitte von z , x rechts von der Mitte. Wegen $|vwx| \leq p$ liegen v und x außerdem beide zwischen dem linken bb und dem rechten bb von z . Wegen $vx \neq \varepsilon$ ist entweder $v \neq \varepsilon$ oder $x \neq \varepsilon$ oder beides. Deswegen entstehen beim Aufpumpen entweder zu viele aa s oder zu viele bb s oder aber ein Wort $(ab)^{p+1}(ba)^r(ab)^s(ba)^{p+1}$, für das entweder $r \geq p+1$ oder $s \geq p+1$ gilt (oder beides). Jede dieser Möglichkeiten widerspricht der Darstellung (5.6) von L_0 . Daher kann L_0 noch nicht einmal kontextfrei sein, geschweige denn deterministisch kontextfrei.

Dieser Widerspruch zeigt, dass unsere Annahme falsch war, und daher ist PAL nicht deterministisch kontextfrei. □ 5.5.6

Da L_0 nicht kontextfrei ist, folgt aus diesem Beweis und den Abschlusseigenschaften auch, dass die kontextfreien Sprachen nicht gegenüber dem Operator Min abgeschlossen sind.

Bemerkung:

Deterministisch kontextfreie Sprachen sind von besonderer praktischer Relevanz. Ein Compiler für eine Programmiersprache ist im Wesentlichen ein DPDA mit einigen kontextsensitiven Ergänzungen. In der Praxis beschränkt man sich auf Grammatiken, denen man schon ansehen kann, dass die erzeugte Sprache deterministisch kontextfrei ist. Eine wichtige Klasse derartiger Grammatiken sind die $LR(k)$ -Grammatiken. Diese haben die Eigenschaft, dass man aus k Lookahead-Zeichen (d.h. aus den k Zeichen rechts des gerade gelesenen Eingabezeichens) eindeutig den zur Erzeugung eines aktuellen Teilworts nötigen Rechtsableitungsschritt ersehen kann. Algorithmen zur Syntaxanalyse von $LR(k)$ -Grammatiken werden in den vertiefenden Vorlesungen „Formale Sprachen“ bzw. „Compilerbau“ behandelt.

Ende der Bemerkung

5.6 Entscheidbarkeitsfragen

Wie wir bereits gesehen haben, sind die folgenden Konstruktionen effektiv durchführbar:

- Kontextfreie Grammatik \rightsquigarrow eingeschränkt kontextfreie Grammatik (Abschnitt 3.2).
- Kontextfreie Grammatik \rightsquigarrow Chomsky- oder Greibach-Normalform (Abschnitt 5.1.5).
- Errechnen der Pumping-Lemma-Zahl p aus einer kontextfreien Grammatik (Abschnitt 5.2).
- Kellerautomat akzeptierend mit Endzuständen \rightsquigarrow Kellerautomat akzeptierend mit leerem Keller \rightsquigarrow Kellerautomat akzeptierend mit Endzuständen (Abschnitt 5.3.2).
- Kellerautomat \rightsquigarrow kontextfreie Grammatik \rightsquigarrow Kellerautomat (Abschnitt 5.3.3).

Entscheidbarkeitsfragen über kontextfreie Sprachen können daher nach Belieben über die Darstellung durch kontextfreie Grammatiken (in Normalformen) oder durch Kellerautomaten beantwortet werden. Wir untersuchen die gleichen Probleme wie für reguläre Sprachen (vgl. Abschnitt 4.6).

Satz 5.6.1 ENTSCHEIDBARKEIT

Die Entscheidungsprobleme

- *Wortproblem,*
- *Leerheitsproblem,*
- *Endlichkeitsproblem*

sind für kontextfreie Grammatiken entscheidbar.

Beweis:

Wortproblem: Gegeben sei eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ in Greibach-Normalform und ein Wort $w \in \Sigma^*$. Die Frage lautet: Gilt $w \in L(G)$? Der Fall $w = \varepsilon$ ist sofort zu entscheiden, da G eingeschränkt kontextfrei ist, und daraus folgt:

$$\varepsilon \in L(G) \Leftrightarrow (S \rightarrow \varepsilon) \in P.$$

Sei jetzt $w \neq \varepsilon$. Dann gilt:

$$\begin{aligned} w \in L(G) &\Leftrightarrow \exists n \geq 1: S \underbrace{\rightarrow_G \dots \rightarrow_G}_{n\text{-mal}} w \\ &\Leftrightarrow \left\{ \begin{array}{l} \text{(In Greibach-Normalform produziert jeder Ableitungsschritt} \\ \text{genau einen Buchstaben von } w. \text{)} \end{array} \right. \\ &\quad S \underbrace{\rightarrow_G \dots \rightarrow_G}_{|w|\text{-mal}} w \end{aligned}$$

Um $w \in L(G)$ festzustellen, genügt es also, alle Ableitungsfolgen der Länge $|w|$ in G zu überprüfen.

Es gibt effizientere Verfahren zur Lösung des Wortproblems; ein solches stellen wir nach dem Beweis dieses Satzes vor.

Leerheitsproblem: Gegeben sei eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$. Die Frage lautet: Gilt $L(G) = \emptyset$? Sei p die nach dem Pumping-Lemma zur kontextfreien Sprache $L(G)$ gehörige Zahl. Wie für reguläre Sprachen zeigt man:

$$L(G) = \emptyset \Leftrightarrow \neg \exists w \in L(G): |w| \leq p.$$

Damit lässt sich das Leerheitsproblem entscheiden, indem für alle Wörter $w \in \Sigma^*$ mit $|w| \leq p$ das Wortproblem entschieden wird.

Endlichkeitsproblem: Gegeben sei eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$. Die Frage lautet: Ist $L(G)$ endlich? Sei p wie eben. Dann zeigt man wie für reguläre Sprachen:

$$L(G) \text{ ist endlich} \Leftrightarrow \neg \exists w \in L(G): p \leq |w| \leq 2 \cdot p.$$

Damit kann das Endlichkeitsproblem durch endlichmaliges Lösen des Wortproblems entschieden werden.

□ 5.6.1

Für Grammatiken in Chomsky-Normalform existiert ein effizienter Algorithmus zur Lösung des Wortproblems, bekannt als *CYK-Algorithmus* (nach einem Papier von Cocke, Younger und Kasami). Sucht man eine Ableitung $A \rightarrow^* x = a_1 \dots a_n$ mit $a_i \in \Sigma$, dann ist es günstig, zu wissen, von welchen Variablen aus Teilwörter $x_{i,j}$ abgeleitet werden können (dabei ist $x_{i,j}$ dasjenige Teilwort von $x = a_1 \dots a_n$, das am i ten Index beginnt und die Länge j hat). Denn wenn $a_1 \dots a_n$ länger ist als nur ein Buchstabe, muss zur Ableitung aus A zunächst eine Regel der Form $A \rightarrow BC$ angewendet werden, wobei aus B das Teilwort $x_{1,k}$ und aus C das Teilwort $x_{k+1,i-k}$ abgeleitet wird. Da aber die Trennstelle $k/k+1$ nicht bekannt ist, speichert man am besten *alle* derartigen Informationen. Der CYK-Algorithmus legt eine Tabelle T an, deren Eintrag $T_{i,j}$ nach Beendigung des Algorithmus aus denjenigen Nichtterminalen besteht, aus denen das Teilwort $x_{i,j}$ ableitbar ist (siehe Abbildung 5.13).

In der ersten **for**-Schleife wird die Tabelle T mit direkten Ableitungen initialisiert, in der innersten Schleife wird sie vollständig aufgefüllt. Danach enthält $T_{1,n}$ alle Nichtterminale, von denen aus das Teilwort $x_{1,n}$,

```

input  $G = (N, \Sigma, P, S)$  in Chomsky-Normalform,  $x = a_1 \dots a_n \in \Sigma^+$ ;
var  $i, j, k : \{1, \dots, n\}$ ,  $T : \text{array } \{1, \dots, n\} \times \{1, \dots, n\} \text{ of } 2^N$ ;
for  $i := 1$  to  $n$  do  $T_{i,1} := \{X \mid (X \rightarrow a_i) \in P\}$  endfor;
for  $j := 2$  to  $n$  do
  for  $i := 1$  to  $n + 1 - j$  do
     $T_{i,j} := \emptyset$ ;
    for  $k := 1$  to  $j - 1$  do
       $T_{i,j} := T_{i,j} \cup \{X \mid \exists Y, Z: (X \rightarrow YZ) \in P \wedge Y \in T_{i,k} \wedge Z \in T_{i+k,j-k}\}$ 
    endfor  $k$ 
  endfor  $i$ 
endfor  $j$ ;
output „ $S \in T_{1,n}$ “ % denn es gilt jetzt  $x \in L(G) \Leftrightarrow S \in T_{1,n}$ 

```

Abbildung 5.13: Der CYK-Algorithmus

also das ganze Wort x , ableitbar ist. Der Fall $x = \varepsilon$ kann getrennt behandelt werden: da G in Chomsky-Normalform ist, gilt $\varepsilon \in L(G)$ genau dann, wenn $S \rightarrow \varepsilon$ eine Produktion ist. Die Komplexität des CYK-Algorithmus ist auf Grund der drei geschachtelten Schleifen ungefähr n^3 , wenn n die Länge des Eingabewortes ist.

Beispiel:

Die Sprache $L = \{a^i b^j c^j \mid i, j \geq 1\}$ ist kontextfrei:

$$S \rightarrow AB, \quad A \rightarrow ab \mid aAb, \quad B \rightarrow c \mid cB.$$

Umformen in Chomsky-Normalform ergibt:

$$\begin{aligned}
 G: \quad S &\rightarrow AB \\
 A &\rightarrow CD \mid CF \\
 B &\rightarrow c \mid EB \\
 C &\rightarrow a \\
 D &\rightarrow b \\
 E &\rightarrow c \\
 F &\rightarrow AD.
 \end{aligned}$$

Sei $x = aaabbcc$. Dann erzeugt der Algorithmus die in Abbildung 5.14 abgebildete Tabelle. Da S in der Tabelle ganz unten links auftaucht, ist das Eingabewort $aaabbcc$ in der Sprache enthalten. Mögliche Ableitungen ergeben sich durch „Zurückverfolgen“ nach oben und schräg rechts oben in der Tabelle, z.B. so:

$$\begin{aligned}
 S &\rightarrow_G AB \rightarrow_G CFB \rightarrow_G CADB \rightarrow_G CCFDB \rightarrow_G CCADDB \\
 &\rightarrow_G CCCDDDB \rightarrow_G CCCDDDEB \xrightarrow*_G aaabbcc
 \end{aligned}$$

		$i \rightarrow$							
$x \{$	a	a	a	b	b	b	c	c	
j	C	C	C	D	D	D	E, B	E, B	
↓			A				B		
			F						
		A							
		F							
	A								
	S								
	S								

Abbildung 5.14: CYK-Tabelle für $aaabbbcc$ **Ende des Beispiels.**

Im Gegensatz zum regulären Fall haben wir die folgenden negativen Resultate.

Satz 5.6.2 (UNENTSCHEIDBARKEITSERGEBNISSE FÜR KONTEXTFREIE GRAMMATIKEN)

Die Entscheidungsprobleme

- Schnittproblem,
- Äquivalenzproblem,
- Inklusionsproblem

sind im kontextfreien Fall unentscheidbar.

Satz 5.6.3 UNENTSCHEIDBARKEIT DER MEHRDEUTIGKEIT VON K.F. GRAMMATIKEN

Das Problem: „Gegeben eine kontextfreie Grammatik, ist diese mehrdeutig?“ ist unentscheidbar.

Für den Nachweis der Richtigkeit dieser Sätze steht das Instrumentarium noch nicht zur Verfügung, denn man muss ja eine Aussage über *alle möglichen Algorithmen* treffen („es gibt *keinen* Algorithmus, der die genannten Probleme entscheidet“). Dieses Instrumentarium werden wir erst in den nächsten Kapiteln entwickeln, und deswegen verlagern wir die Beweise der Sätze 5.6.2 und 5.6.3 in einen späteren Teil des Skriptums.

Für die praktische Anwendung von kontextfreien Grammatiken zur Syntaxbeschreibung von Programmiersprachen wäre es angenehm, einen algorithmischen Test auf Mehrdeutigkeit zu haben. Der Satz 5.6.3 zeigt, dass es einen solchen Test nicht gibt. In der Praxis umgeht man das Problem, die Mehrdeutigkeit testen zu müssen, jedoch recht einfach: man beschränkt sich auf $LR(1)$ -Grammatiken bzw. Teilklassen hiervon. Die $LR(1)$ -Eigenschaft kann man algorithmisch entscheiden, und da $LR(k)$ -Grammatiken stets eindeutig sind (weil der letzte Rechtsableitungsschritt immer eindeutig bestimmt sein muss und daher jedes Wort nur eine Rechtsableitung haben kann), entfällt das Problem der Mehrdeutigkeit.

Zum Schluss dieses Abschnitts fassen wir in Abbildung 5.15 die bisher bekannten (Un-)Entscheidbarkeitsresultate zusammen. Diese Probleme wurden und werden im vorliegenden Skript nicht vollständig behandelt. Die Entscheidbarkeit des Äquivalenzproblems für deterministisch kontextfreie Sprachen (genauer: des Problems, ob zwei gegebene DPDAs die gleichen Sprachen akzeptieren) wurde erst vor relativ kurzer Zeit (1997) von Géraud Sénizergues bewiesen.

Sprachtyp	Wortproblem	Leerheitsproblem	Schnittproblem	Inklusionsproblem	Äquivalenzproblem	Mehrdeutigkeitsproblem
Chomsky-3	entsch.	entsch.	entsch.	entsch.	entsch.	–
det. kontextfrei	entsch.	entsch.	unentsch.	unentsch.	entsch.	–
Chomsky-2	entsch.	entsch.	unentsch.	unentsch.	unentsch.	unentsch.

Abbildung 5.15: (Un-)Entscheidbarkeitsresultate / Zusammenfassung

5.7 Übungsaufgaben

1. Man zeige, dass die Sprache $PAL_c = \{wcw^R \mid w \in \{a, b\}^*\}$ über dem Alphabet $\{a, b, c\}$ deterministisch kontextfrei ist.

2. Die Sprache

$$L = \{a^n b^n c^n d^m \mid n \in \mathbb{N} \setminus \{0\} \wedge m \in \mathbb{N}\} \cup a^* b^* c^*$$

ist *nicht* kontextfrei. Zeigen Sie, dass die Bedingungen aus dem Pumping-Lemma für kontextfreie Sprachen dennoch erfüllt sind.

Anmerkung: Diese Aufgabe zeigt, dass die Umkehrung des Pumping-Lemmas nicht gilt. Die Bedingungen aus dem Pumping-Lemma sind zwar notwendige, aber keine hinreichenden Kriterien dafür, dass eine Sprache kontextfrei (oder regulär) ist.

3. Gegeben sei die kontextfreie Grammatik $G = (\{S, X, Y, Z\}, \{a, b, c\}, P, S)$ mit

$$P = \left\{ \begin{array}{l} S \rightarrow aX \mid Ybb \\ X \rightarrow Zb \mid aXb \\ Y \rightarrow aaZ \mid aYb \\ Z \rightarrow c \mid aZb \end{array} \right\}.$$

- a) Zeigen Sie, dass die Grammatik G mehrdeutig ist.
 - b) Bestimmen Sie die von G erzeugte Sprache. Ist $L(G)$ ebenfalls mehrdeutig?
4. Seien L_1 und L_2 zwei kontextfreie Sprachen über einem beliebigen Alphabet Σ . Was ist von der folgenden Behauptung zu halten?

$$(L_1, L_2 \text{ unendlich} \wedge L_1 \cap L_2 \neq \emptyset \wedge L_1 \not\subseteq L_2 \wedge L_2 \not\subseteq L_1) \implies L_1 \cup L_2 \text{ ist inhärent mehrdeutig.}$$

5. Gegeben sei die Sprache $L = \{a^m b^n \mid m \leq n \leq 2m, m \geq 1\}$.

- a) Entwerfen Sie eine kontextfreie Grammatik G mit $L(G) = L$.
- b) *Erkennung durch leeren Keller*: Erzeugen Sie aus Ihrer Grammatik G einen Kellerautomaten K mit $L_\varepsilon(K) = L$.
- c) Zeigen Sie, dass das Wort $aabbb$ von Ihrem Kellerautomaten K erkannt wird, das Wort $aaabb$ jedoch nicht.
- d) *Erkennung durch Endzustand*: Wandeln Sie Ihren Kellerautomaten K in einen Kellerautomaten K' um, so dass $L(K') = L$ gilt.

6. Gegeben sei die Sprache

$$L = \{a^n b^n c^m \mid m, n \in \mathbb{N} \setminus \{0\}\} \cup \{a^n b^m c^m \mid m, n \in \mathbb{N} \setminus \{0\}\}.$$

- a) Konstruieren Sie einen Kellerautomaten K , der die Sprache L akzeptiert.
- b) Zeigen Sie, dass der von Ihnen konstruierte Kellerautomat das Wort $abbcc$ erkennt.
- c) Ist es möglich, einen *deterministischen* Kellerautomaten K' zu konstruieren, der die obige Sprache akzeptiert? Falls ja, geben Sie diesen an. Falls nein, begründen Sie dies einleuchtend (aber nicht notwendigerweise formal).

7. Beweisen oder widerlegen Sie: Wenn L eine kontextfreie Sprache ist, dann ist auch

- a) $L_1 = \{w \mid w^R \in L\}$
- b) $L_2 = \{ww^R \mid w \in L\}$

eine kontextfreie Sprache.

8. Gegeben sei die kontextfreie Grammatik $G = (\{S, A, B\}, \{0, 1\}, P, S)$ mit

$$P = \left\{ \begin{array}{l} S \rightarrow 0B \mid 1A \\ A \rightarrow 0 \mid 0S \mid 1AA \\ B \rightarrow 1 \mid 1S \mid 0BB \end{array} \right\},$$

welche die Sprache $L = \{w \in \{0, 1\}^+ \mid \#_0(w) = \#_1(w)\}$ erzeugt.

- a) Zeigen Sie, dass die Grammatik G mehrdeutig ist.
 - b) Entwerfen Sie einen Kellerautomaten K mit nur einem Zustand, so dass $L = L_\varepsilon(K)$ gilt.
9. Sei $L = \{a^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 0\}$. Existiert ein deterministischer Kellerautomat K , so dass
- a) $L = L(K)$ (Akzeptanz durch Endzustand) ?
 - b) $L = L_\varepsilon(K)$ (Akzeptanz durch leeren Keller) ?

Geben Sie jeweils den gesuchten Kellerautomaten an oder begründen Sie zwingend seine Nichtexistenz.

Kapitel 6

Turingmaschinen

Ende des letzten Kapitels wurde behauptet, dass es *keinen Algorithmus geben kann*, der für zwei gegebene kontextfreie Grammatiken über einem Alphabet Σ entscheidet, ob die beiden erzeugten Sprachen gleich sind oder nicht. Diese Aussage unterscheidet sich ganz erheblich von den positiven Entscheidbarkeitsaussagen, die wir bislang zu untersuchen hatten. Für eine solche positive Aussage genügt es nämlich, einen Algorithmus anzugeben, wobei es nicht sehr wesentlich ist, ob er in **Java**, in Pseudocode oder in einem anderen Formalismus ausgedrückt wird (solange klar ist, dass er funktioniert). Wollen wir aber die *Nichtexistenz* eines Algorithmus zeigen, müssen wir eine Aussage über *alle nur denkbaren* Algorithmen machen! Wir benötigen also ein Modell, das im Prinzip alle Algorithmen ausdrücken kann, sozusagen ein „Urmodell“ für Algorithmen.

Alan M. Turing (1912-1954) bemühte sich ca. 1936, ein Modell zu erfinden, das genau diesem Zweck dienen sollte, und es scheint ihm gut gelungen zu sein. Die heute so bezeichnete *Turingmaschine* modelliert das Rechnen mit Bleistift und Papier und orientiert sich an den Papierstapeln, die zum Rechnen nötig sind. Diese werden durch ein beschriftbares „Arbeitsband“, auf dem man Zeichen eintragen und verändern kann, dargestellt. Das Arbeitsband sollte als potenziell unendlich groß angesehen werden, denn falls sich eine Rechnung in die Länge zieht, sollte man immer neue Papierstapel benutzen dürfen. Die Rechnungen werden durch eine endliche Berechnungsvorschrift (ein „Programm“ bzw. einen „Algorithmus“) gesteuert. Man denke zum Beispiel an die hierzulande übliche Vorschrift zur Addition zweier Zahlen. Hierzu benötigt man die Möglichkeit, zwei Zahlen untereinander zu schreiben, darunter einen Strich zu ziehen, dann stellenweise von rechts nach links zu addieren, sich den Übertrag zu merken (ihn z.B. auf einem unbenutzten Stück Papier aufzuschreiben), usw. Die Definition der Turingmaschine, die wir in den Abschnitten 6.1 bis 6.3 vornehmen werden, geschieht ganz unter dem Eindruck dieser Intuition.

Letzten Endes werden wir Turingmaschinen zur Formalisierung zweier Begriffe heranziehen:

- Im einen Fall wird eine Turingmaschine als „Akzeptor“ benutzt (Abschnitt 6.5). Wir definieren dann, was es für eine Sprache $L \subseteq \Sigma^*$ bedeutet, algorithmisch von einer Turingmaschine *akzeptiert* zu werden. In dieser Funktion ähnelt eine Turingmaschine einem endlichen Automaten bzw. einem PDA (nur „kann sie mehr“, d.h., es gibt Sprachen, die von einer Turingmaschine akzeptiert werden, aber nicht von einem PDA und erst recht nicht von einem NFA).

Entscheidungsprobleme wie z.B. das Äquivalenzproblem für kontextfreie Grammatiken können durch einen glücklichen Zufall *auch* als Sprachen aufgefasst werden. Dann kann die Aussage „ein

Problem ist entscheidbar (oder *unentscheidbar*)“ aufgefasst werden als „es existiert eine Turingmaschine (bzw. es existiert *keine* Turingmaschine), die die dem Problem entsprechende Sprache akzeptiert“. Diese Idee wird uns aber erst im nächsten Kapitel (Kapitel 7) beschäftigen.

- Im anderen Fall wird eine Turingmaschine als „Computer“ benutzt (Abschnitt 6.6). Wir definieren dann, was es für eine (partielle) Funktion von \mathbb{N}^n nach \mathbb{N} bedeutet, *berechenbar* zu sein. Das steht ein wenig im Gegensatz dazu, dass wir bislang Automaten ausschließlich als Sprachakzeptoren benutzt haben.

6.1 Aufbau einer Turingmaschine

Die Abbildung 6.1 zeigt ein *Lese/Schreib-Band* einer Turingmaschine, das nach beiden Seiten hin unendlich ist und aus einer Reihe von *Feldern* besteht. Auf diesen Feldern befinden sich je ein Zeichen, oder sie sind leer; den letzten Fall kann man so interpretieren, dass sie mit einem speziellen Zeichen \sqcup , dem „Blank“ oder „Leerzeichen“, beschriftet sind. Der *Lese/Schreibkopf* (LSK) einer Turingmaschine steht immer auf einem bestimmten Feld des Bandes. Die *Steuereinheit* ist immer in einem bestimmten Zustand (von endlich vielen), anfänglich im *Anfangszustand* q_0 .

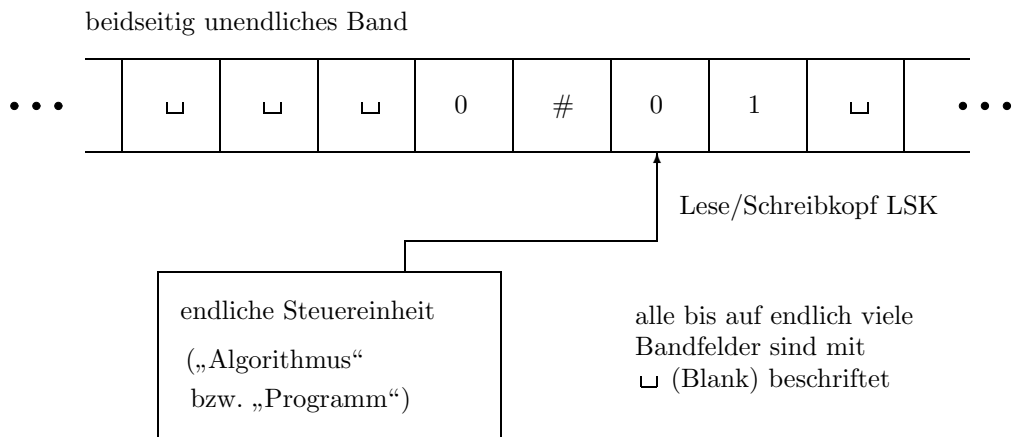


Abbildung 6.1: Schematische Darstellung einer Turingmaschine

Ein *Schritt* der Maschine besteht aus vier Teilen:

1. Der LSK liest das Zeichen im momentanen Feld.
2. Der LSK schreibt ein Zeichen auf das momentane Feld.
3. Der LSK bewegt sich nach links (*L*), rechts (*R*), oder bleibt stehen (*N* wie „nicht bewegen“).
4. Die Steuereinheit geht in einen nächsten Zustand über (der der gleiche wie der alte sein kann).

Welcher neue Zustand eingegangen wird, welches Zeichen geschrieben wird und wie der Kopf sich bewegt, hängt vom aktuellen Zustand und vom gerade gelesenen Zeichen ab.

Das Turingband ist sehr flexibel nutzbar. Es kann sowohl für Eingabewörter, als auch für beliebige Zwischenergebnisse, als auch für Ausgaben verwendet werden. Wir benötigen keine Mengen von Ein- und Ausgabezuständen mehr, sondern kommen ohne Beschränkung der Allgemeinheit mit je einem aus (einem Anfangszustand q_0 und einem Endzustand q_f). Außerdem legen wir – wieder ohne Beschränkung der Allgemeinheit – fest, dass eine Turingmaschine, die den Endzustand erreicht hat, keine weitere Schritte machen darf.

Definition 6.1.1 TURINGMASCHINEN

Eine Turingmaschine (TM oder NTM, für „nichtdeterministische TM“) ist ein 7-Tupel $M = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_f)$ mit folgenden Komponenten:

Q ist eine endliche Menge von *Zuständen* mit $q_0 \in Q$ und $q_f \in Q$.

Σ ist das *Alphabet* der Maschine. Intuitiv sind die Zeichen in Σ die „nach außen sichtbaren“ Zeichen oder „Ein/Ausgabezeichen“ (oft auch nur die „Eingabezeichen“) der Maschine.

$\Gamma \supseteq \Sigma$ ist das *Bandalphabet* von M . Intuitiv enthält Γ alle diejenigen Zeichen, die als Feldbeschriftungen von M vorkommen dürfen. Dazu gehören mindestens alle Zeichen in Σ und das spezielle Blankzeichen \sqcup . Oft möchte man noch andere spezielle Zeichen verwenden, die nur temporär auf dem Band stehen; deswegen heißen die Zeichen in $\Gamma \setminus (\Sigma \cup \{\sqcup\})$ auch „Hilfszeichen“ oder „Sonderzeichen“.

$\sqcup \in \Gamma \setminus \Sigma$ ist das *Leerzeichen* oder „Blankzeichen“ von M .

$\delta \subseteq ((Q \setminus \{q_f\}) \times \Gamma) \times (Q \times \Gamma \times \{L, N, R\})$ ist die *Übergangsrelation*. Der Endzustand q_f ist aus dem Definitionsbereich von δ herausgenommen. Somit kann eine Turingmaschine, die sich im Endzustand befindet, keine Schritte mehr machen.

q_0 ist der *Anfangszustand*.

q_f ist der *Endzustand*.

Eine NTM M heißt *deterministisch* (DTM), wenn δ eine Funktion $\delta: (Q \setminus \{q_f\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, N, R\}$ ist. ☒ 6.1.1

Die Relation δ codiert das „Programm“ bzw. die Steuereinheit der Turingmaschine M . Ein Element von δ ist immer ein (5-) Tupel, z.B. $((q_1, \sqcup), (q_2, a, R))$. Dieses Tupel bedeutet:

ist der momentane Zustand q_1 und ist das momentan unter dem LSK befindliche Zeichen \sqcup , dann kann die Turingmaschine das Zeichen a auf die momentane LSK-Position schreiben (d.h., das \sqcup mit a überschreiben), mit dem LSK um ein Feld weiter nach rechts rücken und in den neuen Zustand q_2 übergehen.

Ist M nicht deterministisch, dann gibt es ein Tupel $((q, a), x)$ und $((q, a), y)$ mit $x \neq y$ in δ . Dies bedeutet, dass bei Vorliegen von q und gelesenen Zeichen a eine Wahl zwischen x und y möglich ist. Ist eine solche Wahlmöglichkeit gegeben, stellt man sich am besten vor, dass die Maschine völlig autonom entscheidet,

welche der möglichen nächsten Schritte eingegangen werden. Bei einer deterministischen Maschine M gibt es solche Wahlmöglichkeiten nicht, weil δ rechtseindeutig ist.

Wenn es für einen Zustand q und ein gerade gelesenes Zeichen a kein Element $((q, a), x)$ in δ gibt, dann gibt es keine weitere Aktionsmöglichkeit für M , und man sagt, dass M *stoppt*. Da es für den Endzustand q_f nach Definition kein Tupel $((q_f, a), x)$ gibt, stoppt die Maschine stets, wenn q_f erreicht wird. Eine DTM stoppt *genau dann*, wenn q_f erreicht ist, da δ linkstotal ist. Eine NTM kann aber auch in anderen Zuständen stoppen.

Eine DTM ist aus diesen Gründen (keine Wahlmöglichkeiten, und Stopp genau dann, wenn Endzustand erreicht) leichter zu handhaben als eine NTM.

Manchmal wird δ als fünfspaltige Tabelle dargestellt, deren Zeilen den Elementen von δ entsprechen. Es gibt immer nur endlich viele solche Zeilen, da alle beteiligten Mengen endlich sind. Diese Darstellung von δ wird *Turingtafel* genannt.

Eine weitere Darstellungsmöglichkeit für Turingmaschinen ist ein *Turingmaschinengraph*. Die Abbildung 6.2 zeigt, wie ein solcher Graph aufgebaut ist. Die Knoten des Graphen entsprechen den Zuständen Q . Die Kanten des Graphen entsprechen den Elementen von δ . Ein Element $((q, a), (q', b, m))$ in δ wird durch eine Kante dargestellt, die von q nach q' führt und mit $a : b$ (Eingabezeichen : Ausgabezeichen) sowie mit m (der Bewegung, die im entsprechenden Schritt vom LSK ausgeführt wird) beschriftet ist. Wie bei endlichen Automaten wird der Anfangszustand mit einem kleinen Eingangspfeil versehen, und der Endzustand wird durch einen Doppelkreis dargestellt.

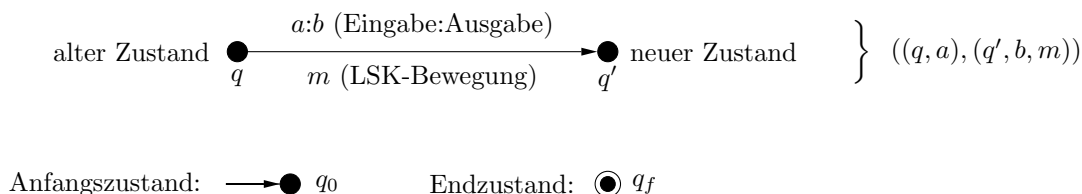


Abbildung 6.2: Darstellung von $((q, a), (q', b, m)) \in \delta$ (oben) und von Anfangs- und Endzustand (unten)

6.2 Beispiele

Beispiel M_1 :

Es sei $M_1 = (Q_1, \Sigma_1, \Gamma_1, \sqcup, \delta_1, q_0, q_1)$ mit

$$\begin{aligned} Q_1 &= \{q_0, q_1\} \\ \Sigma_1 &= \{0, 1\} \\ \Gamma_1 &= \{0, 1, \sqcup\} \\ \delta_1 &= \{((q_0, 0), (q_0, 0, R)), ((q_0, 1), (q_1, 1, N))\}. \end{aligned}$$

Die Turingtafel und der Turing-Graph von M_1 sind in Abbildung 6.3 dargestellt. Man kann aus dem Graphen alle Komponenten der Turingmaschine außer den beiden Alphabeten zurückrechnen. Insbesondere

werden im Graph, im Gegensatz zur Tafel, auch die Anfangs- und Endzustände angezeigt.

δ_1	alter Zustand	gelesenes Zeichen	neuer Zustand	neues Zeichen	Bewegung
Zeile 1	q_0	0	q_0	0	R
Zeile 2	q_0	1	q_1	1	N

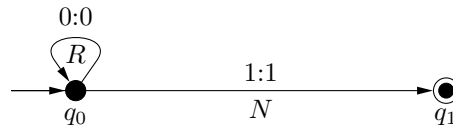


Abbildung 6.3: Die Turingtafel (oben) und der Maschinengraph (unten) von M_1

Wir beschreiben nunmehr die Wirkungsweise der Maschine M_1 . Dabei müssen wir unterschiedliche anfängliche Bandinhalte berücksichtigen. Die Abbildung 6.4 zeigt das leere Band. Der LSK von M_1 steht auf einem beliebigen Bandfeld und der Anfangszustand ist q_0 . Man sagt dazu, dass M_1 auf das leere Band *angesetzt* ist.

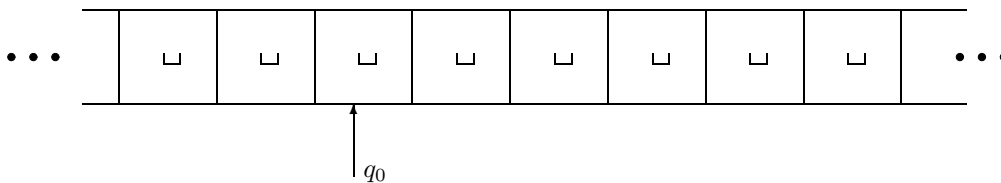
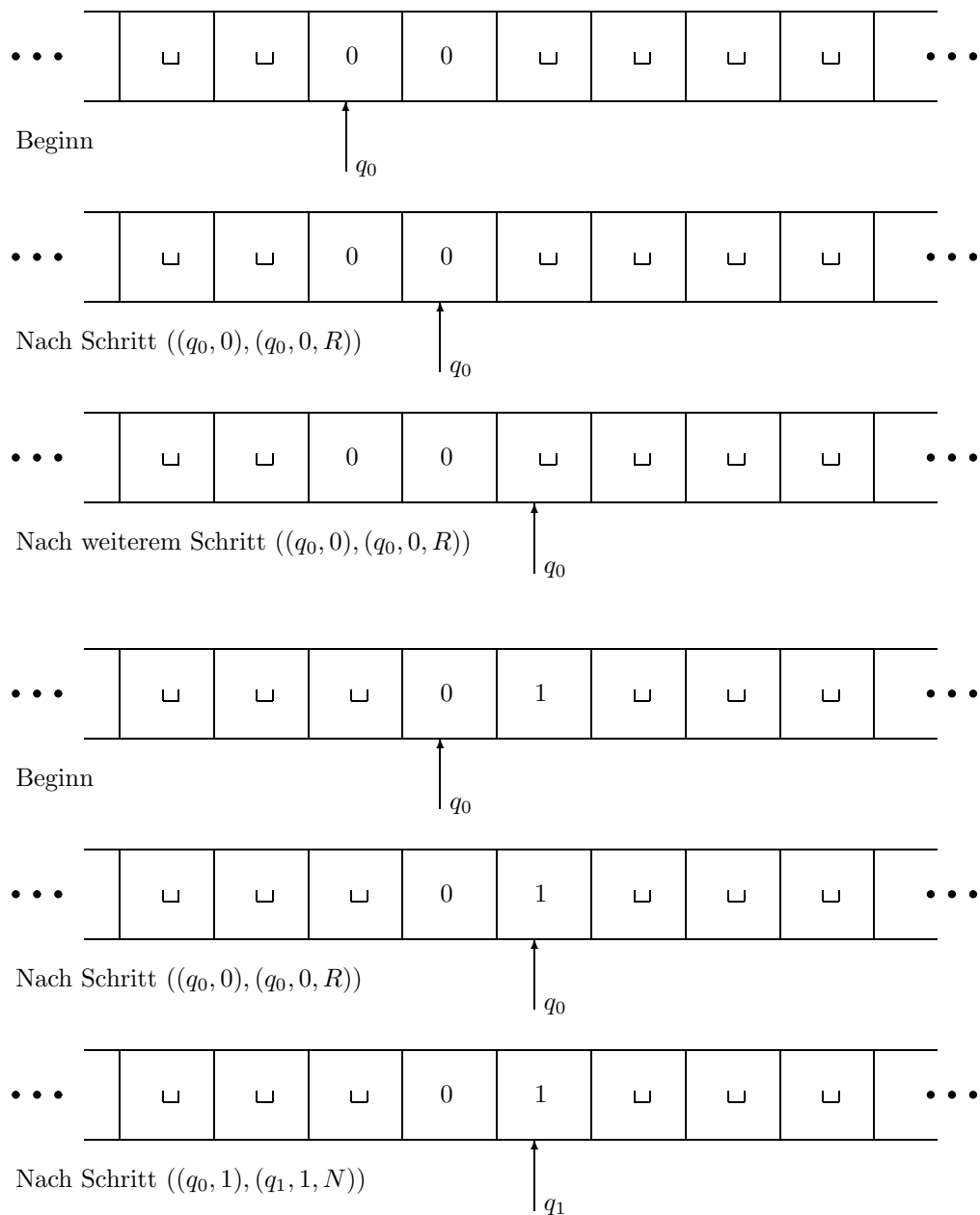


Abbildung 6.4: M_1 angesetzt auf das leere Band

Die Maschine M_1 versucht nun, einen Schritt auszuführen. Dazu werden die Elemente von δ_1 nach einem Element durchsucht, das die Form $((q_0, \sqcup), (q', a, m))$ hat, denn der aktuelle Zustand ist q_0 und das aktuell gesehene Zeichen ist \sqcup . Wird ein solches Element gefunden, ist ein Schritt möglich. Da aber δ_1 kein solches Element enthält, ist kein Schritt möglich und die Maschine stoppt, ohne den Endzustand zu erreichen. Betrachten wir nun eine andere Berechnung, die mit einem anderen Bandinhalt beginnt. In Abbildung 6.5 (oben) ist ein nicht-leeres Band gezeigt, das außer unendlich vielen \sqcup auch noch die Zeichenkette 00 auf zwei nebeneinanderliegenden Feldern enthält. Der LSK von M_1 ist auf die linke der beiden Nullen angesetzt und der Anfangszustand ist wieder q_0 . Nach den beiden in dieser Abbildung gezeigten Schritten liest der LSK ein Zeichen \sqcup im Zustand q_0 . Kein weiterer Schritt ist mehr möglich und die Maschine stoppt im Zustand q_0 , also wieder nicht im Endzustand. Ein anfänglicher Bandinhalt, der ein Stoppen der Maschine im Endzustand q_1 ermöglicht, ist in Abbildung 6.5 (unten) gezeigt. Nach den beiden dort gezeigten Schritten wird eine 1 im Zustand q_1 gelesen und die Maschine stoppt, weil es in δ_1 kein Element gibt, das mit $(q_1, 1)$ beginnt.

Generell ist M_1 eine „1-Suchmaschine“. Der anfängliche Bandinhalt hat dabei die Wirkung eines Parameters der Berechnung: ist er von der Form $\dots 0\dots 01\dots$ und liegt die anfängliche Kopfposition auf einer der Nullen, dann (und nur dann) stoppt die Maschine im Endzustand mit LSK auf der Eins.

Ende des Beispiels M_1 Abbildung 6.5: M_1 angesetzt auf ein nicht-leeres Band mit Zeichenkette 00 (oben) bzw. 01 (unten)

Der anfängliche Bandinhalt hat offenbar eine ähnliche Funktion wie die **input**-Parameter bei einem Pro-

gramm (siehe Abschnitt 2.4.2). Wir lassen zunächst beliebige anfängliche Bandinhalte zu, außer dass wir fordern, dass die Anzahl der nicht- \sqcup -Zeichen endlich ist. Diese Forderung ist sinnvoll, um auszuschließen, dass zum Lesen einer Eingabe schon unendlich viel Zeit verbraucht wird.

Beispiel M_2 :

Es sei $M_2 = (\{q_0, q_1\}, \{0\}, \{0, \sqcup\}, \sqcup, \delta_2, q_0, q_1)$ mit

$$\delta_2 = \{((q_0, \sqcup), (q_0, 0, N)), ((q_0, \sqcup), (q_1, 0, L)), ((q_0, 0), (q_1, 0, L)), ((q_0, 0), (q_1, 0, R))\}.$$

Der Graph von M_2 ist in Abbildung 6.6 gezeigt.

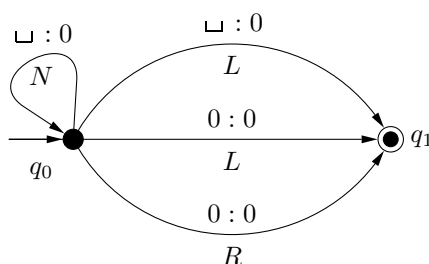


Abbildung 6.6: Der Maschinengraph von M_2

Angesetzt z.B. auf das leere Band gibt es für M_2 zwei Möglichkeiten, einen Schritt zu machen: entweder eine 0 zu schreiben, den LSK nicht zu bewegen und im Anfangszustand zu bleiben (Abbildung 6.7 (oben)), oder eine 0 zu schreiben, den LSK nach links zu bewegen und in den Endzustand überzugehen (Abbildung 6.7 (unten)).

Angesetzt auf ein Zeichen 0 gibt es für M_2 ebenfalls zwei Möglichkeiten, einen Schritt zu machen: entweder durch $((q_0, 0), (q_1, 0, R))$ einen Schritt nach rechts zu machen und in den Endzustand zu gehen, oder durch $((q_0, 0), (q_1, 0, L))$ einen Schritt nach links zu machen und in den Endzustand überzugehen.

Ende des Beispiels M_2

Beispiele $M_{\text{all}}(\Sigma)$ und $M_{\text{loop}}(\Sigma)$:

Sei Σ ein beliebiges Alphabet. Wir betrachten zwei Turingmaschinen:

$$M_{\text{all}}(\Sigma) = (\{q_0, q_f\}, \Sigma, \Sigma \cup \{\sqcup\}, \sqcup, \{((q_0, a), (q_f, a, N)) \mid a \in \Sigma \cup \{\sqcup\}\}, q_0, q_f),$$

$$M_{\text{loop}}(\Sigma) = (\{q_0, q_f\}, \Sigma, \Sigma \cup \{\sqcup\}, \sqcup, \{((q_0, a), (q_0, a, N)) \mid a \in \Sigma \cup \{\sqcup\}\}, q_0, q_f),$$

deren Graphen in Abbildung 6.8 dargestellt sind.

Hier haben wir die Abkürzung $\bullet : \bullet$ verwendet, worin \bullet für ein beliebiges Zeichen aus $\Sigma \cup \{\sqcup\}$ steht (links und rechts des Semikolons stets das gleiche Zeichen).

M_{all} liest ein Zeichen, ändert weder dieses Zeichen noch die LSK-Position, und geht in den Endzustand über. M_{loop} liest ein Zeichen, ändert weder dieses Zeichen noch die LSK-Position, geht aber nicht in den Endzustand, sondern bleibt im Anfangszustand. Danach kann in einer Endlosschleife wieder das gleiche Zeichen gelesen werden. Offensichtlich entsprechen M_{all} und M_{loop} – in gewisser Weise – den beiden Programmen **skip** bzw. **loop** (Abschnitt 2.4.3).

Ende der Beispiele $M_{\text{all}}(\Sigma)$ und $M_{\text{loop}}(\Sigma)$

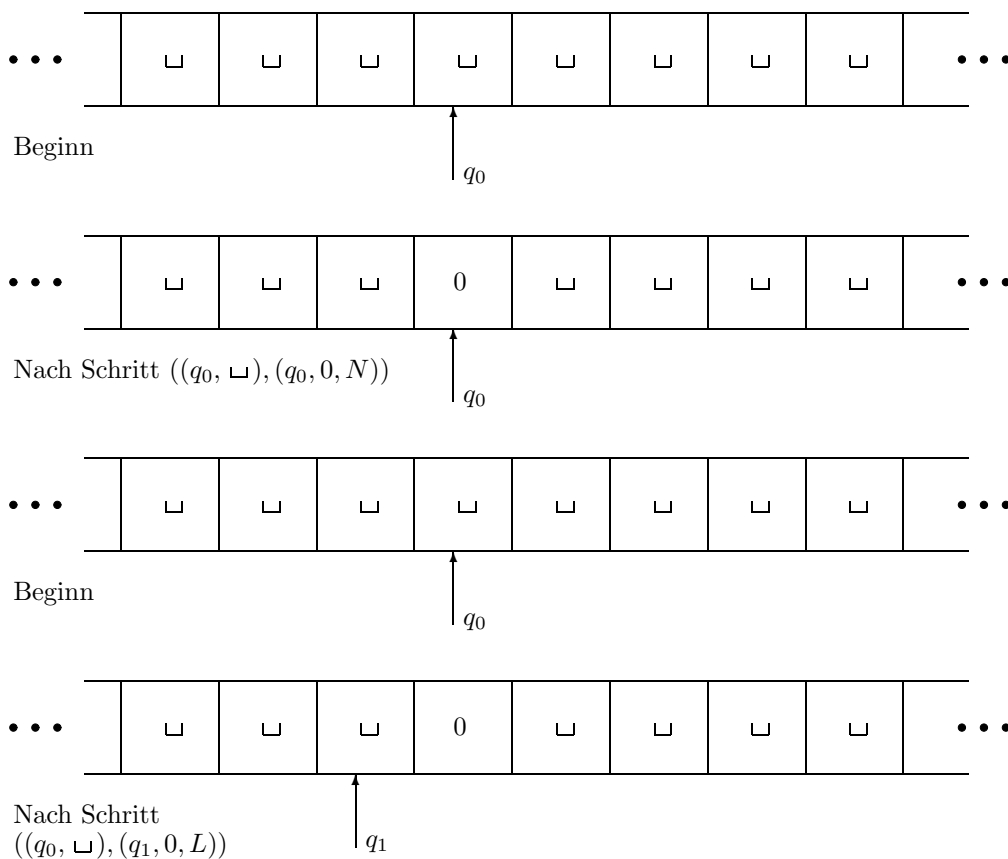


Abbildung 6.7: Zwei Berechnungsmöglichkeiten von M_2 , angesetzt auf das leere Band



Abbildung 6.8: Die Maschinengraphen von $M_{\text{all}}(\Sigma)$ (links) und $M_{\text{loop}}(\Sigma)$ (rechts)

Es handelt sich bei den in diesem Abschnitt betrachteten Turingmaschinen um folgende Typen:

1. M_1 ist eine NTM, aber keine DTM, da δ_1 zwar rechtseindeutig, aber nicht linkstotal ist.
2. M_2 ist eine NTM, aber keine DTM, da δ_2 zwar linkstotal, aber nicht rechtseindeutig ist. Im Gegensatz zu M_1 kann M_2 (wie in Abbildung 6.7 gezeigt) echt nichtdeterministisches Verhalten haben.
3. Beide Maschinen $M_{\text{all}}(\Sigma)$ und $M_{\text{loop}}(\Sigma)$ sind deterministisch, da ihre Übergangsrelationen sowohl linkstotal als auch rechtseindeutig sind.

6.3 Das Verhalten einer Turingmaschine

Offenbar ist der Begriff „Schritt“, den wir in den Beispielen benutzt haben, eine Relation zwischen Konfigurationen, die durch folgende Parameter bestimmt sind:

- den aktuellen Bandinhalt (anfänglich mit einem „Eingabewort“ beschrieben);
- die aktuelle Kopfposition (anfänglich üblicherweise auf dem ersten Zeichen des Eingabewortes, wenn dieses nicht leer ist);
- und den aktuellen Zustand (anfänglich q_0).

Da wir den nicht- \sqcup -Teil des Bandes als endlich annehmen, können wir eine solche Situation durch ein endliches Wort $vgaw$ beschreiben.

Definition 6.3.1 KONFIGURATIONEN EINER TURINGMASCHINE

Sei M eine Turingmaschine. Eine *Konfiguration* von M ist ein Wort $k = vgaw$ mit

- $v \in \{\varepsilon\} \cup ((\Gamma \setminus \{\sqcup\})\Gamma^*)$ ist der Bandinhalt links des LSK;
- $q \in Q$ ist der aktuelle Zustand und $a \in \Gamma$ das gerade unter dem LSK befindliche Zeichen;
- $w \in \{\varepsilon\} \cup (\Gamma^*(\Gamma \setminus \{\sqcup\}))$ ist der Bandinhalt rechts des LSK.

Mit $\mathcal{K}(M)$ bezeichnen wir die Menge der Konfigurationen von M . Eine Konfiguration $k = vgaw$ heißt *Endkonfiguration*, wenn q der Endzustand q_f ist. □ 6.3.1

Es ist nicht ausgeschlossen, dass v oder w (oder beide) das Leerzeichen enthalten können. Die obige Definition macht natürlich nur Sinn, wenn $Q \cap \Gamma^* = \emptyset$ gilt, was wir hiermit o.B.d.A. annehmen wollen.

Definition 6.3.2 SCHRITTE UND FOLGEKONFIGURATIONEN VON TURINGMASCHINEN

Sei $M = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_f)$ eine Turingmaschine. Wir definieren eine Relation

$$\rightarrow_M \subseteq \mathcal{K}(M) \times \mathcal{K}(M),$$

die den Begriff „erreichbar in einem Schritt“ formalisiert. Wir definieren dazu zunächst eine Relation $k \xrightarrow{X} k'$, wobei X ein Element in δ ist, durch die Aufzählung aller solcher Tripel:

$$(1a) \quad qaaw \xrightarrow{((q, a), (q', c, L))} \left\{ \begin{array}{ll} q' \sqcup & \text{falls } c = \sqcup \text{ und } w = \varepsilon \\ q' \sqcup cw & \text{falls } c \neq \sqcup \text{ oder } w \neq \varepsilon \end{array} \right\}$$

$$(1b) \quad v' b q a w \xrightarrow{((q, a), (q', c, L))} \left\{ \begin{array}{ll} v' q' b & \text{falls } c = \sqcup \text{ und } w = \varepsilon \\ v' q' b c w & \text{falls } c \neq \sqcup \text{ oder } w \neq \varepsilon \end{array} \right\}$$

$$(2) \quad v q a w \xrightarrow{((q, a), (q', c, N))} v q' c w$$

$$(3a) \quad v q a \xrightarrow{((q, a), (q', c, R))} \left\{ \begin{array}{ll} q' \sqcup & \text{falls } c = \sqcup \text{ und } v = \varepsilon \\ v c q' \sqcup & \text{falls } c \neq \sqcup \text{ oder } v \neq \varepsilon \end{array} \right\}$$

$$(3b) \quad v q a b w' \xrightarrow{((q, a), (q', c, R))} \left\{ \begin{array}{ll} q' b w' & \text{falls } c = \sqcup \text{ und } v = \varepsilon \\ v c q' b w' & \text{falls } c \neq \sqcup \text{ oder } v \neq \varepsilon \end{array} \right\},$$

mit $v, v' \in \{\varepsilon\} \cup ((\Gamma \setminus \{\sqcup\})\Gamma^*)$, $w, w' \in \{\varepsilon\} \cup (\Gamma^*(\Gamma \setminus \{\sqcup\}))$, $a, b, c \in \Gamma$ und $q, q' \in Q$.

Nun wird ein *Schritt* von einer Konfiguration k zu einer Konfiguration k' folgendermaßen definiert:

$$(k, k') \in \rightarrow_M \text{ oder } k \rightarrow_M k' \text{ gdw. ein } X \in \delta \text{ mit } k \xrightarrow{X} k' \text{ existiert.}$$

Man sagt dann auch, dass k' eine *Folgekonfiguration* von k ist. ☒ 6.3.2

Die Zeile (1a) beschreibt Linksbewegungen, wobei links vom gerade gelesenen Zeichen a nur Blanks stehen. Die Zeile (1b) beschreibt Linksbewegungen, wobei es links vom gerade gelesenen Zeichen noch andere nicht- \sqcup -Zeichen gibt. Die Zeile (2) beschreibt den Fall, dass der LSK nicht bewegt wird. Die Zeilen (3a) und (3b) beschreiben Rechtsbewegungen, analog zu (1a) und (1b).

Beispiel:

Die Konfigurationen und die Schritte der Abbildungen 6.4–6.7 fallen unter die Definitionen 6.3.1 und 6.3.2. Wir zeigen dies für Abbildung 6.7:

$$\begin{array}{l} \text{oben } (q_0 \sqcup \rightarrow_{M_2} q_0 0): \\ \text{unten } (q_0 \sqcup \rightarrow_{M_2} q_1 \sqcup 0): \end{array} \left\{ \begin{array}{ll} \underbrace{q_0}_{v=\varepsilon} \underbrace{\sqcup}_{q=q_0} \underbrace{}_{a=\sqcup} \underbrace{}_{w=\varepsilon} & \xrightarrow{((q_0, \sqcup), (q_0, 0, N))} \underbrace{q_0}_{v=\varepsilon} \underbrace{0}_{q'=q_0} \underbrace{}_{c=0} \underbrace{}_{w=\varepsilon} & \text{(mit Zeile (2))} \\ \underbrace{q_0}_{q=q_0} \underbrace{\sqcup}_{a=\sqcup} \underbrace{}_{w=\varepsilon} & \xrightarrow{((q_0, \sqcup), (q_1, 0, L))} \underbrace{q_1}_{q'=q_1} \underbrace{\sqcup}_{c=0} \underbrace{0}_{w=\varepsilon} & \text{(mit Zeile (1a))} \end{array} \right.$$

Ende des Beispiels

Wir definieren nun Berechnungen einer TM als Folgen einzelner Schritte.

Definition 6.3.3 SCHRITTE UND SCHRITTFOLGEN VON TURINGMASCHINEN

Eine *partielle Berechnung* ist eine endliche oder unendliche Folge von Konfigurationen, bei der jede Konfiguration außer der ersten eine Folgekonfiguration der unmittelbar vorhergehenden ist. Eine Konfiguration k' heißt *erreichbar* aus k , wenn es eine partielle Berechnung gibt, die von k nach k' führt, oder, anders ausgedrückt, wenn gilt: $(k, k') \in \rightarrow_M^*$ (oder suggestiver: $k \rightarrow_M^* k'$).

Eine *maximale Berechnung* (oder nur *Berechnung*) ist eine endliche partielle Berechnung, die mit einer Konfiguration endet, in der keine weiteren Schritte mehr möglich sind. Eine Berechnung heißt *terminal*, wenn sie in einer Endkonfiguration endet. ☒ 6.3.3

Beispiel:

Die Berechnungen in den Abbildungen 6.4 und 6.5 lauten folgendermaßen:

$$\begin{array}{l} \text{Abbildung 6.4 :} \quad q_0 \sqcup \quad (v = \varepsilon \text{ und } w = \varepsilon; \text{ es gibt keine Folgekonfigurationen) } \\ \text{Abbildung 6.5 (oben) :} \quad q_0 00 \xrightarrow{((q_0, 0), (q_0, 0, R))} 0q_0 0 \xrightarrow{((q_0, 0), (q_0, 0, R))} 00q_0 \sqcup \\ \text{Abbildung 6.5 (unten) :} \quad q_0 01 \xrightarrow{((q_0, 0), (q_0, 0, R))} 0q_0 1 \xrightarrow{((q_0, 1), (q_1, 1, N))} 0q_1 1 \end{array}$$

Ende des Beispiels

Terminale Berechnungen sind stets maximal, da δ für q_f nicht definiert ist. Für eine DTM sind die Begriffe maximal und terminal gleichbedeutend.

Gegenbeispiel:

Wird die Maschine M_2 auf das leere Band angesetzt, kann sie eine 0 auf das Feld unter dem LSK schreiben und im Zustand q_0 stoppen (Abbildung 6.7 (oben)). Da diese Berechnung nicht fortgesetzt werden kann, ist sie maximal, da q_0 aber kein Endzustand ist, ist sie nicht terminal.

Ende des Gegenbeispiels**(Weiteres) Beispiel M_5 :**

Sei M_5 die Turingmaschine mit Eingabealphabet $\Sigma = \{0, 1\}$ und mit folgender Turingtafel:

alter Zustand	gelesenes Zeichen	neuer Zustand	neues Zeichen	Bewegung
q_0	0/1	q_0	0/1	R
q_0	\sqcup	q_1	\sqcup	L
q_1	0	q_2	1	L
q_1	1	q_1	0	L
q_1	\sqcup	q_f	1	N
q_2	0/1	q_2	0/1	L
q_2	\sqcup	q_f	\sqcup	R

Diese Maschine ist deterministisch und berechnet zu einer Binärzahl w als Input die nachfolgende Binärzahl, $w + 1$ (Binäraddition). Eine Berechnung ist z.B.:

$$\begin{aligned}
 q_0 1011 &\rightarrow 1q_0 011 &\rightarrow 10q_0 11 &\rightarrow 101q_0 1 &\rightarrow 1011q_0 \sqcup \\
 &\rightarrow 101q_1 1 &\rightarrow 10q_1 10 &\rightarrow 1q_1 000 \\
 &\rightarrow q_2 1100 &\rightarrow q_2 \sqcup 1100 &\rightarrow q_f 1100.
 \end{aligned}$$

Ende des (weiteren) Beispiels M_5

6.4 Umwandlung einer NTM in eine DTM

Wir wollen uns jetzt auf anfängliche Konfigurationen der Form $q_0 w$ mit $w \in \Sigma^*$ beschränken, d.h.:

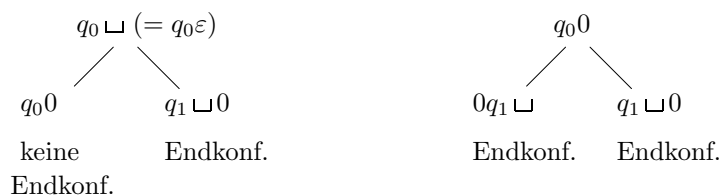
- Falls $w \neq \varepsilon$, steht der LSK auf dem ersten Zeichen des Wortes w , und links von w sowie rechts von w gibt es nur Blankzeichen.
- Falls $w = \varepsilon$, besteht das Band überhaupt nur aus Blankzeichen, und der LSK steht auf irgend einem davon; $q_0 \varepsilon$ ist also eine andere Schreibweise für die Konfiguration $q_0 \sqcup$.

Wir nennen die Konfiguration $q_0 w$ die *Anfangskonfiguration von M , gegeben w* ; man sagt auch: *M ist auf w angesetzt*, und man schreibt dafür in Anlehnung an einen Prozeduraufruf mit einem Parameter: $M(w)$.

Beispiel M_2 (Fortsetzung):

Die Gesamtheit der Berechnungen von $M(w)$ ist im allgemeinen baumförmig strukturiert. Beispielsweise sind in Abbildung 6.9 zwei Konfigurationsbäume der Maschine M_2 gezeigt (mit zwei verschiedenen w). Der erste Baum entspricht den beiden Berechnungen von Abbildung 6.7.

Ende des Beispiels M_2 (Fortsetzung)

Abbildung 6.9: Zwei Konfigurationsbäume von M_2 mit verschiedenen anfänglichen Bandinhalten**Definition 6.4.1** KONFIGURATIONSBAUM EINER TM AUF EINEM ANFÄNGLICHEN BANDINHALT

Gegeben seien eine Turingmaschine M über Σ und eine anfängliche Konfiguration q_0w , wobei $w \in \Sigma^*$, von M . Der *Konfigurationsbaum* von $M(w)$ ist induktiv definiert:

- Die *Wurzel* des Baumes ist die Konfiguration q_0w .
- Die *Kinder* einer Konfiguration k des Baumes sind die Folgekonfigurationen von k . □ 6.4.1

Die Blätter eines Konfigurationsbaumes entsprechen denjenigen Konfigurationen, für die M anhält.

Ein Konfigurationsbaum kann unendlich groß sein, wie z.B. in $\mathbf{M}_{\text{loop}}(\Sigma)$, angesetzt auf ein beliebiges Wort. Er ist aber immer *von endlichem Ausgangsgrad*, d.h., jeder Knoten hat nur endlich viele Kinder, denn wegen der Endlichkeit der Relation δ kann jede Konfiguration höchstens endlich viele Folgekonfigurationen haben. Genauer: Sei

$$r = \max \{ |\delta(q, a)| \mid q \in Q \setminus \{q_f\} \wedge a \in \Gamma \};$$

dann ist r eine wohldefinierte natürliche Zahl, da alle beteiligten Mengen endlich sind, und jeder Knoten in einem beliebigen Konfigurationsbaum von M hat höchstens r Kinder. Deswegen wird r der *Grad des Nichtdeterminismus* von M genannt. Bei einer DTM ist $r = 1$. Nach dem Lemma von König (Lemma 2.2.1) gibt es in jedem unendlichen Konfigurationsbaum auch einen unendlich langen Weg, und die beiden Aussagen: „von q_0w aus gibt es eine unendlich lange partielle Berechnung“ bzw. „der Konfigurationsbaum mit q_0w als Wurzel ist unendlich groß“ sind gleichwertig. Auf Grundlage dieser Bemerkung definieren wir eine Konstruktion, die jeder Turingmaschine M eine deterministische Turingmaschine $\text{det}(M)$ zuordnet. Später zeigen wir, dass M und $\text{det}(M)$ in vielen Fällen „das Gleiche leisten“.

Definition 6.4.2 NTM \rightsquigarrow DTM

Sei $M = (Q, \Sigma, \Gamma, \sqsubset, \delta, q_0, q_f)$ eine beliebige Turingmaschine über Σ . Wir konstruieren $\text{det}(M)$ so, dass $\text{det}(M)$ für jedes Wort $w \in \Sigma^*$ den Konfigurationsbaum von $M(w)$ erzeugt und dabei auch in Breitensuche durchläuft (und so M simuliert). Genauer geht $\text{det}(M)$ folgendermaßen vor:

- Falls $r = 0$, dann ist δ die leere Relation und M kann entweder durch \mathbf{M}_{all} oder durch \mathbf{M}_{loop} simuliert werden, je nachdem, ob $q_0 = q_f$ oder $q_0 \neq q_f$ gilt. Sei im Folgenden ohne Beschränkung der Allgemeinheit $r \geq 1$.
- Das Eingabewort w wird an einer sicheren Stelle gespeichert, z.B. eingeschlossen zwischen Sonderzeichen: $\$_l w \$_r$. Links davon werden Wörter u über dem Alphabet $\{1, \dots, r\}$ in systematischer Reihenfolge gemäß dem Prinzip der Breitensuche, also

„der Länge nach, kürzere zuerst, und bei gleicher Länge in lexikographischer Ordnung“, also $\varepsilon, 1, 2, \dots, r, 11, 12, \dots, 1r, 21, \dots, \dots, rr, 111, \dots$ usw.,

erzeugt. Da das Band von $det(M)$ nach links unendlich ist, steht stets genug Platz zur Erzeugung solcher Wörter u zur Verfügung. Jeder endliche Pfad im Konfigurationsbaum von $M(w)$ entspricht einem solchen u . Z.B. bedeutet $u = 1332$: an der Wurzel die erste Alternative wählen, danach die dritte, dann wieder die dritte, danach die zweite. Es kann sein, dass eines der erzeugten u keine Auswahl darstellt, aber andersherum (und das ist entscheidend) ist jeder Knoten durch eine solche Auswahl erreichbar. Gegeben ein Wort $u \in \{1, \dots, r\}^*$, simuliert $det(M)$ rechts neben $\$lw\$_r$ genau den Pfad des Konfigurationsbaums von $M(w)$ der durch u beschrieben wird (falls ein solcher Pfad vorhanden ist). $det(M)$ hält im Endzustand, wenn der aktuell simulierte Pfad auf ein Blatt des Konfigurationsbaums von M mit q_0w als Wurzel führt, das eine Endkonfiguration darstellt:

```

input  $w$ ;
 $u := \varepsilon$ ;
do   simuliere  $M(w)$  mit Hilfe der Auswahl  $u$ ;
      if   Auswahl führt auf ein Blatt mit Endkonfiguration   $\rightarrow$  halte
       $\square$  else                                            $\rightarrow$  skip
      fi;
      generiere nächstes  $u$ 
od.

```

Der **else**-Fall in der inneren **if**-Anweisung kann eintreten, wenn u zu einer Konfiguration von M führt, die nicht akzeptiert oder wenn u zu keiner Konfiguration von M führt, weil der tatsächliche Grad des Nichtdeterminismus in M kleiner ist als r . $det(M)$ kann in der **do**...**od**-Anweisung in eine unendliche Schleife geraten kann, nicht aber in der Simulation von $M(w)$, da diese Simulation von u kontrolliert wird. \square 6.4.2

Die Breitensuche wird gewählt, weil gewährleistet werden muss, dass *jeder* Knoten des Konfigurationsbaums von $M(w)$ tatsächlich von $det(M)$ erreicht wird; würde stattdessen die Tiefensuche gewählt, könnte es vorkommen, dass ein unendlich langer Ast des Baumes durchlaufen wird und eine daneben liegende akzeptierende Konfiguration „nicht gesehen“ wird.

6.5 Turingmaschinen als Sprachakzeptoren

Wir definieren nun die Verwendung einer Turingmaschine zum Akzeptieren von Sprachen (und kommen damit zur Signifikanz der Parameter Σ und q_f in der Turingmaschinendefinition). Gegeben sei ein festes Alphabet Σ . Wir interessieren uns nun für die Klasse der Turingmaschinen mit dem Eingabealphabet Σ , d.h., für Maschinen der Form $M = (., \Sigma, ., \sqcup, ., q_0, q_f)$.

Definition 6.5.1 AKZEPTANZ DURCH EINE TURINGMASCHINE

M akzeptiert ein Wort $w \in \Sigma^*$, wenn es eine terminale Berechnung von M gibt, die mit der Konfiguration q_0w beginnt. Die von M akzeptierte (erkannte) Sprache, $L(M)$, ist die Menge aller von M akzeptierten Eingabewörter $w \in \Sigma^*$. Zwei Turingmaschinen M und M' über Σ heißen *sprachäquivalent*, wenn $L(M) = L(M')$. \square 6.5.1

Beispiele:

Die Maschine M_1 ist eine Turingmaschine über dem Eingabealphabet $\Sigma_1 = \{0, 1\}$. Sie akzeptiert das Wort 01, nicht aber die Wörter ε und 00. Allgemein ist die von M_1 akzeptierte Sprache diejenige Teilmenge von $\{0, 1\}^*$, die Wörter mit mindestens einer 1 enthält, d.h.

$$L(M_1) = \{0\}^*\{1\}\{0, 1\}^* \subseteq \{0, 1\}^*.$$

Die Maschine M_2 ist eine Turingmaschine über dem Eingabealphabet $\{0\}$. Sie akzeptiert alle Wörter:

$$L(M_2) = \{0\}^*.$$

Das leere Wort ε wird von M_2 beispielsweise durch den Übergang $((q_0, \sqcup), (q_1, 0, L))$ akzeptiert. Es gibt zwar auch den Übergang $((q_0, \sqcup), (q_0, 0, L))$, jedoch genügt zur Akzeptanz die Existenz *einer* terminalen Berechnung, unabhängig davon, wie viele nicht-akzeptierende (oder auch unendliche) es sonst noch geben mag.

$\mathbf{M}_{\text{all}}(\Sigma)$ akzeptiert die volle Sprache Σ^* , während $\mathbf{M}_{\text{loop}}(\Sigma)$ die leere Sprache \emptyset akzeptiert.

Ende der Beispiele**Satz 6.5.2** M UND $\det(M)$ SIND SPRACHÄQUIVALENT

Sei M eine Turingmaschine. Es gilt $L(M) = L(\det(M))$.

Beweis: In Abschnitt 6.4 wurde die Wirkungsweise von $\det(M)$ beschrieben. Dadurch hat man:

$$\begin{aligned} w \in L(M) &\Rightarrow (\text{Definition von } L(M)) \\ &\quad \text{es gibt einen Pfad im Konfigurationsbaum von } M(w), \\ &\quad \text{der auf ein Blatt mit einer Endkonfiguration führt} \\ &\Rightarrow (\text{Definition von } \det(M), \text{ Breitensuche}) \\ &\quad \det(M) \text{ findet diesen Pfad und simuliert ihn} \\ &\Rightarrow (\text{Definition von } \det(M), L(\det(M))) \\ &\quad w \in L(\det(M)). \\ \\ w \notin L(M) &\Rightarrow (\text{Definition von } L(M)) \\ &\quad \text{im } M(w)\text{-Baum gibt es keine akzeptierende Berechnung} \\ &\Rightarrow (\text{Definition von } \det(M)) \\ &\quad \det(M) \text{ läuft unendlich} \\ &\Rightarrow (\text{Definition von } L(\det(M))) \\ &\quad w \notin L(\det(M)). \end{aligned}$$

Also $w \in L(M)$ genau dann, wenn $w \in L(\det(M))$, was $L(M) = L(\det(M))$ bedeutet. ☒ 6.5.2

Man kann also stets zu einer nichtdeterministischen Turingmaschine eine sprachäquivalente deterministische finden:

$$\begin{aligned} &\{ L \subseteq \Sigma^* \mid \text{es gibt eine nichtdeterministische Turingmaschine } M \text{ mit } L = L(M) \} \\ &= \{ K \subseteq \Sigma^* \mid \text{es gibt eine deterministische Turingmaschine } M' \text{ mit } K = L(M') \}. \end{aligned}$$

Definition 6.5.3 DIE NOTATION $M(w) \downarrow$ BZW. $M(w) \uparrow$

Sei M eine DTM. Die Tatsache, dass M mit Eingabe w anhält, wird auch kurz mit $M(w) \downarrow$ bezeichnet (Sprechweise: „ M stoppt, angesetzt auf w “). Die Tatsache, dass M mit Eingabe w nicht stoppt, wird kurz mit $M(w) \uparrow$ bezeichnet (Sprechweise: „ M kreist, angesetzt auf w “). \boxtimes 6.5.3

Wir ändern nunmehr unsere Sichtweise und interessieren uns für die Klasse der Sprachen, die von Turingmaschinen akzeptiert werden.

Definition 6.5.4 TURING-AKZEPTIERBARE SPRACHEN

Sei Σ ein beliebiges Alphabet. Eine Sprache $L \subseteq \Sigma^*$ heißt *Turing-akzeptierbar* (kurz auch T.a.), wenn es eine Turingmaschine M über Σ mit $L = L(M)$ gibt. \boxtimes 6.5.4

Wegen Satz 6.5.2 kann man ohne Beschränkung der Allgemeinheit davon ausgehen, dass diese Maschine M deterministisch ist. Für eine T.a. Sprache L gibt es also eine deterministische Turingmaschine, für die gilt:

$$\begin{aligned} \forall w \in \Sigma^* : \quad & w \in L \Leftrightarrow M(w) \downarrow \Leftrightarrow M(w) \text{ hat genau eine terminierende Berechnung} \\ & \wedge \quad w \notin L \Leftrightarrow M(w) \uparrow \Leftrightarrow M(w) \text{ hat genau eine unendliche Berechnung.} \end{aligned} \quad (6.1)$$

Der nächste Satz stellt eine Verbindung zu WHILE-Programmen (Abschnitt 2.4.4) her. Die Kernaussage dieses Satzes ist analog (6.1): eine Sprache L über Σ ist T.a. genau dann, wenn ein Programm existiert, das Wörter $w \in \Sigma^*$ als Eingabe annimmt und terminiert, wenn $w \in L$, bzw. in eine unendliche Schleife gerät, wenn $w \notin L$.

Satz 6.5.5 WHILE-AKZEPTIERBARE SPRACHEN = TURING-AKZEPTIERBARE SPRACHEN

Sei Σ ein Alphabet und sei $L \subseteq \Sigma^*$. Dann sind die beiden folgenden Eigenschaften äquivalent:

- Es gibt ein WHILE-Programm mit einem Eingabeparameter **input** $x : \Sigma^*$, das stoppt, falls anfänglich x den Wert w hat und $w \in L$ gilt, und loopt, falls anfänglich x den Wert w hat und $w \notin L$ gilt.
- L ist Turing-akzeptierbar.

Beweis: Skizze:

Für die Richtung von oben nach unten nehmen wir die Existenz eines WHILE-Programms an. Dieses Programm hat eine gewisse syntaktische Struktur, und eine simulierende Turingmaschine wird per struktureller Induktion definiert: Für elementare WHILE-Programme (Leerkommando, Zuweisung) kann eine simulierende Turingmaschine direkt angegeben werden. Für zusammengesetzte WHILE-Programme (Hintereinanderausführung, IF-Kommando, WHILE-Kommando) kann eine simulierende Turingmaschine aus den nach Induktionsvoraussetzung existierenden Turingmaschinen für die Teile des Programms zusammengesetzt werden.

Für die Richtung von unten nach oben nehmen wir die Existenz einer Turingmaschine an. Wir konstruieren ein diese Maschine simulierendes WHILE-Programm im Wesentlichen als einen Simulator der Turingtafel der gegebenen Turingmaschine, wobei alle Elemente dieser Tafel so codiert werden, dass sie sich als Werte von Registern des WHILE-Programms auffassen lassen.

Es stellt sich bei der zweiten Konstruktion heraus, dass man im Allgemeinen mit nur **einer** WHILE-Schleife auskommt. \boxtimes 6.5.5

6.6 Turingmaschinen als Berechner von partiellen Funktionen

In diesem Abschnitt betrachten wir Turingmaschinen in einer etwas anderen Rolle, nämlich als „Berechner“ von (partiellen) Funktionen f von \mathbb{N}^n nach \mathbb{N} . Im Unterschied zur vorigen Betrachtungsweise spielt nunmehr der Bandinhalt im akzeptierenden Endzustand eine Rolle. Gerade deswegen – um auszuschließen, dass es zu einem Argumentetupel von f viele verschiedene Ergebnisse gibt – schränken wir die Definition von vornherein auf deterministische Turingmaschinen ein. Die Idee ist, dass eine DTM M die Funktion f berechnet, wenn gilt:

- anfänglich enthält das Band ein Argumente- n -tupel $(x_1, \dots, x_n) \in \mathbb{N}^n$ von f ;
- wenn f auf (x_1, \dots, x_n) definiert ist, gibt es (genau) eine terminierende Berechnung, wobei der Bandinhalt im Endzustand $f(x_1, \dots, x_n)$ enthält;
- und wenn f auf (x_1, \dots, x_n) nicht definiert ist, kreist die Maschine, d.h.: es gibt (genau) eine nicht terminierende Berechnung.

Um diese Idee zu verwirklichen, verwenden wir die in Abschnitt 2.5 beschriebene Idee, Zahlen und Tupel binär zu codieren. Sei bin eine effektive und bijektive Codierung von beliebigen m -Tupeln natürlichen Zahlen in Wörter über dem Binäralphabet, d.h.:

$$bin: \bigcup_{i=0}^{\infty} \mathbb{N}^i \rightarrow \{0,1\}^*, \quad (6.2)$$

wobei zusätzlich das einzige Element von \mathbb{N}^0 , die leere Menge \emptyset , auf das leere Wort ε abgebildet wird. Eine solche Codierung kann man z.B. dadurch bekommen, dass Satz 2.1.6 benutzt wird. Die X_i dieses Satzes sind dann die Mengen \mathbb{N}^i in (6.2), die einzeln wie in Abschnitt 2.5 codiert werden können. Aus einem Codewort $w \in \{0,1\}^*$ kann man also eindeutig sowohl die Stelligkeit m als auch das codierte m -Tupel (x_1, \dots, x_m) mit $bin(x_1, \dots, x_m) = w$ zurückrechnen. So können sowohl die Argumente (ein n -Tupel) als auch der Funktionswert (ein 1-Tupel) einer Funktion $f^{(n)}$ als ein Wort über dem Alphabet $\{0,1\}$ aufgefasst und auf einem Turingband dargestellt werden.

Definition 6.6.1 BERECHENBARKEIT DURCH EINE TURINGMASCHINE

Sei M eine DTM über dem Alphabet $\Sigma = \{0,1\}$. Sei $n \in \mathbb{N}$ beliebig. Die durch M berechnete n -stellige Funktion $f^{(n)}(M)$ ist folgendermaßen definiert: für $(x_1, \dots, x_n) \in \mathbb{N}^n$:

$$f^{(n)}(M)(x_1, \dots, x_n) = \begin{cases} y & \text{falls } M(bin(x_1, \dots, x_n)) \text{ mit der Endkonfiguration } uq_f bin(y) \text{ stoppt} \\ undef & \text{sonst.} \end{cases}$$

Diese Funktion ist wohldefiniert, weil es im ersten Fall keine andere Endkonfiguration geben kann.

Sei $f: \mathbb{N}^n \xrightarrow{p} \mathbb{N}$ eine beliebige Funktion. Dann heißt f *Turing-berechenbar* (kurz auch T.b.), wenn es eine DTM M über $\Sigma = \{0,1\}$ mit der Eigenschaft $f = f^{(n)}(M)$ gibt. □ 6.6.1

Eine DTM *berechnet f normiert*, wenn jede Endkonfiguration von der Art $q_f w$ ist, d.h., wenn links neben dem LSK nur Blankzeichen stehen. Offenbar kann jede DTM, die f berechnet, durch „Bandbereinigung“ leicht in eine andere DTM umgewandelt werden, die f normiert berechnet. Es ist also keine Einschränkung der Allgemeinheit, wenn wir eine f berechnende DTM als normiert annehmen; und das tun wir ab jetzt.

Beispiel $O^{(n)}$:

Die Turingmaschine $M_{\text{loop}}(\{0,1\})$ berechnet die Grundfunktion

$$O^{(n)}: \begin{cases} \mathbb{N}^n & \xrightarrow{p} \mathbb{N} \\ (x_1, \dots, x_n) & \mapsto \text{undef} \end{cases}$$

(siehe Abschnitt 2.1.4) für beliebiges n .

Ende des Beispiels $O^{(n)}$

Auch für alle anderen Grundfunktionen lassen sich unschwer konkrete Turingmaschinen angeben, die sie berechnen.

Beispiel S:

Die Nachfolgefunktion

$$S: \begin{cases} \mathbb{N} & \rightarrow \mathbb{N} \\ x & \mapsto x+1 \end{cases}$$

lässt sich durch die DTM M_S berechnen (siehe Ende des Abschnitts 6.3).

Ende des Beispiels S

Man kann fast wörtlich Definition 6.6.1 benutzen, um die Turing-Berechenbarkeit von partiellen Funktionen $f: (\Sigma_1^* \times \dots \times \Sigma_n^*) \xrightarrow{p} \Sigma^*$ zu definieren, wobei die $\Sigma_1, \dots, \Sigma_n$ und Σ beliebige Alphabete sind. Man kann aber auch die Wörter über diesen Alphabeten zuerst – wie in Abschnitt 2.5 – natürlichzahlig codieren und dann die Turing-Berechenbarkeit solcher Funktionen auf Definition 6.6.1 zurückführen. Beide Methoden laufen offenbar auf den gleichen Begriff von „Turing-berechenbar“ hinaus.

Analog Satz 6.5.5 gilt:

Satz 6.6.2 WHILE-BERECHENBARE FUNKTIONEN = TURING-BERECHENBARE FUNKTIONEN

Sei $f: \mathbb{N}^n \xrightarrow{p} \mathbb{N}$ eine partielle Funktion. Dann sind die beiden folgenden Eigenschaften äquivalent:

- Es gibt ein WHILE-Programm mit n Eingabeparametern **input** $x_1 : \mathbb{N}, \dots, x_n : \mathbb{N}$ und einem Ausgabeparameter **output** $z : \mathbb{N}$, das mit $z = y$ stoppt, falls anfänglich $x_1 = u_1, \dots, x_n = u_n$ und $f(u_1, \dots, u_n) = y$, und loopt, falls anfänglich $x_1 = u_1, \dots, x_n = u_n$ und $f(u_1, \dots, u_n) = \text{undef}$.
- f ist Turing-berechenbar.

Beweis: Analog dem Beweis von Satz 6.5.5. □ 6.6.2

Zum Schluss dieses Abschnitts beschreiben wir zwei Äquivalenzsätze, die zeigen, wie die beiden Sichtweisen von Turingmaschinen als Akzeptoren von Sprachen (Definition 6.5.1), bzw. als Berechner von Funktionen (Definition 6.6.1), miteinander zusammenhängen. Der erste charakterisiert die Turing-Akzeptanz einer beliebigen Sprache durch die Turing-Berechenbarkeit einer gewissen, von der Sprache abgeleiteten, partiellen Funktion. Der zweite charakterisiert die Turing-Berechenbarkeit einer partiellen Funktion durch die Turing-Akzeptanz einer gewissen, von der Funktion abgeleiteten, Sprache. Die Kernaussage der Sätze ist, dass eine partielle Funktion f genau dann Turing-berechenbar ist, wenn die Menge der Paare (*Argument, Funktionswert*), wobei f auf dem *Argument* definiert ist, Turing-akzeptierbar ist.

Satz 6.6.3 CHARAKTERISIERUNG VON TURING-AKZEPTANZ DURCH TURING-BERECHENBARKEIT

Sei Σ ein Alphabet. Eine Sprache $L \subseteq \Sigma^*$ ist T.a. genau dann, wenn die Funktion $\chi_L^+ : \Sigma^* \xrightarrow{p} \{0,1\}$ (siehe Abschnitt 2.3) T.b. ist.

Beweis: Wir beweisen die beiden Teile von Satz 6.6.3 getrennt:

(A): Sei M eine DTM, die L akzeptiert. Wir definieren eine Maschine M' über Σ mit Sonderzeichen $1 \in \Gamma$ folgendermaßen:

$$M' : \begin{cases} \text{input } w \in \Sigma^*; \\ \text{simuliere } M(w); \\ \text{falls } M(w) \downarrow, \text{ dann lösche Band und schreibe } 1; \\ \text{stoppe mit } q_f 1. \end{cases}$$

Dann gilt:

$$\begin{aligned} \chi_L^+(w) = 1 &\Leftrightarrow (\text{Definition von } \chi_L^+) \\ &w \in L \\ &\Leftrightarrow (M \text{ akzeptiert } L) \\ &M(w) \downarrow \\ &\Leftrightarrow (\text{Definition von } M') \\ &M'(w) \text{ stoppt mit } q_f 1. \end{aligned}$$

Aus der Definition von Turing-Berechenbarkeit folgt, dass M' die partielle Funktion χ_L^+ berechnet.

(B): Sei M eine DTM, die χ_L^+ berechnet. Dann akzeptiert M auch L . □ 6.6.3

Satz 6.6.4 CHARAKTERISIERUNG VON TURING-BERECHENBARKEIT DURCH TURING-AKZEPTANZ

Eine partielle Funktion $f: \mathbb{N}^n \xrightarrow{p} \mathbb{N}$ ist T.b. genau dann, wenn die Sprache

$$\text{graph}_f = \{ \text{bin}(x_1, \dots, x_n, y) \mid x_1, \dots, x_n, y \in \mathbb{N} \text{ und } f(x_1, \dots, x_n) = y \}$$

über dem Alphabet $\Sigma = \{0, 1\}$ T.a. ist.

Beweis:

Wir beweisen die beiden Teile von Satz 6.6.4 getrennt:

(A): Sei M eine DTM, die f berechnet. Wir definieren eine Maschine M' über $\{0, 1\}$ folgendermaßen:

```

input  $w \in \{0, 1\}^*$ ;
if  $w$  ist nicht von der Form  $\text{bin}((x_1, \dots, x_n), y)$  mit  $x_1, \dots, x_n, y \in \mathbb{N} \rightarrow$  kreise
□  $w$  ist von der Form  $\text{bin}((x_1, \dots, x_n), y)$  mit  $x_1, \dots, x_n, y \in \mathbb{N} \rightarrow$ 
  speichere  $\text{bin}(y)$ ;
  simuliere  $M(\text{bin}(x_1, \dots, x_n))$ ;
  falls  $M(\text{bin}(x_1, \dots, x_n)) \downarrow$  mit Resultat  $q_f v$ :
if  $v = \text{bin}(y) \rightarrow$  akzeptiere  $w$ 
□  $v \neq \text{bin}(y) \rightarrow$  kreise
fi
fi

```

Dann gilt: $w \in \text{graph}_f \Rightarrow$ (Definition von graph_f)
 $w = \text{bin}((x_1, \dots, x_n), y)$ und $y = f(x_1, \dots, x_n)$
 \Rightarrow (Definition von M')
 $M'(w)$ geht in die zweite Alternative der äußeren **if**-Anweisung
 \Rightarrow (Definition von M und f ist auf x_1, \dots, x_n definiert)
 $M'(w)$ stoppt mit $q_f v$ und erreicht die innere **if**-Anweisung
 \Rightarrow (M berechnet f , also $\text{bin}(y) = \text{bin}(f(x_1, \dots, x_n)) = v$)
 $M'(w)$ führt erste Alternative der inneren **if**-Anweisung aus und stoppt.

$w \notin \text{graph}_f \Rightarrow$ (Definition von graph_f)
 w ist nicht von der Form $\text{bin}((x_1, \dots, x_n), y)$
oder w ist von dieser Form und $f(x_1, \dots, x_n) = \text{undef}$
oder w ist von dieser Form und $y \neq f(x_1, \dots, x_n) \in \mathbb{N}$
 \Rightarrow (Definition von M')
 $M'(w)$ kreist in der ersten Alternative der äußeren **if**-Anweisung
oder $M'(w)$ kreist beim Simulieren von $M(w)$
oder $M'(w)$ kreist in der zweiten Alternative der inneren **if**-Anweisung
 \Rightarrow (Quintessenz der drei Fälle)
 $M(w)$ kreist.

Aus den beiden Implikationen folgt: M' akzeptiert graph_f .

(B): Sei umgekehrt M eine DTM, die graph_f akzeptiert. Wir definieren eine Maschine M' über $\{0, 1\}$ folgendermaßen:

```

input  $x_1, \dots, x_n$ ;
speichere  $\text{bin}(x_1, \dots, x_n)$ ;
installiere Zähler  $v = 0, 1, 10, 11, 100, \dots$  auf nach links unendlichem Band (anfänglich  $v = 0$ );
do true  $\rightarrow v := v+1$  (binäre Addition)
   $\square$  true  $\rightarrow$  simuliere  $M(\text{bin}(x_1, \dots, x_n, v))$ ;
    falls  $M(\text{bin}(x_1, \dots, x_n, v)) \downarrow$ , merke  $v$ , lösche Band, schreibe  $v$ , stoppe mit  $q_f v$ 
od

```

Wegen der Auswahl in der **do**-Anweisung ist M' eine nichtdeterministische Maschine. In der ersten Zeile der Schleife wird v hochgezählt und damit ein Wert „geraten“, der Funktionswert von f an der Stelle (x_1, \dots, x_n) sein könnte. Genauer:

$f(x_1, \dots, x_n) = y \Rightarrow$ (Definition von M')
 $M'(\text{bin}(x_1, \dots, x_n))$ kann y -mal die erste Schleifenalternative wählen (dann $v = \text{bin}(y)$), danach die zweite
 \Rightarrow (M akzeptiert graph_f)
 $M(\text{bin}((x_1, \dots, x_n), y)) \downarrow$
 \Rightarrow (Definition von M' und $v = \text{bin}(y)$)
 $M'(\text{bin}(x_1, \dots, x_n))$ hat eine Berechnung, die mit $q_f v$ stoppt.

$f(n) = \text{undef} \Rightarrow$ (Definition M')
 $M'(\text{bin}(x_1, \dots, x_n))$ kreist entweder dadurch, dass immer nur die erste Alternative gewählt wird, oder dadurch, dass die Simulation von $M(\text{bin}((x_1, \dots, x_n), y))$ kreist.

Offenbar kann man die Maschine $det(M')$ mit Hilfe einer „Bandbereinigung“ so modifizieren, dass sie mit den gleichen Bandinhalten wie M' stoppt. Beide Implikationen zusammen ergeben, dass f durch (die so modifizierte) $det(M')$ berechnet wird. □ 6.6.4

6.7 Turingmaschinen und Grammatiken

Wir charakterisieren in diesem Abschnitt die Chomsky-0- und auch die Chomsky-1-Sprachen durch Turingmaschinen.

6.7.1 Chomsky-0-Sprachen

Für die Turing-akzeptierbaren Sprachen gilt der folgende fundamentale Satz, der auf eine Beziehung zwischen formalen Grammatiken und Turingmaschinen hinausläuft:

Satz 6.7.1 CHOMSKY-0-SPRACHEN = TURING-AKZEPTIERBARE SPRACHEN

Sei Σ ein Alphabet und sei $L \subseteq \Sigma^*$. Dann sind die beiden folgenden Eigenschaften äquivalent:

- L ist Chomsky-0.
- L ist Turing-akzeptierbar.

Wenn L T.a. ist, bedeutet das, dass ein Algorithmus existiert, der Wörter $w \in \Sigma^*$ als Eingabe akzeptiert und anhält, wenn $w \in L$, bzw. kreist, wenn $w \notin L$. Wenn L jedoch von G erzeugt wird, bedeutet das informell, dass ein Algorithmus existiert, der die Wörter von L der Reihe nach „aufschreibt“. Es ist nicht ganz offensichtlich, wie diese beiden verschiedenen Dinge miteinander in Verbindung gebracht werden können.

Beweis:

(\Rightarrow):

Wir zeigen, dass jede Chomsky-0-Sprache L auch Turing-akzeptierbar ist.

L werde von einer Chomsky-0-Grammatik $G = (N, \Sigma, P, S)$ erzeugt: $L(G) = L$. Wir konstruieren eine nichtdeterministische Turingmaschine M , die L akzeptiert. M arbeitet wie folgt:

- (1) M reserviert einen Eingabebereich und belässt ein vorgegebenes Eingabewort $w \in \Sigma^*$ unverändert auf diesem Bereich.
- (2) Auf dem anfangs leeren Restband erzeugt M schrittweise Wörter über $N \cup \Sigma$ gemäß den Regeln aus P , beginnend mit dem Startwort S . In jedem Schritt wählt M nichtdeterministisch ein Teilwort u aus dem zuletzt erzeugten Wort und eine Regel $u \rightarrow v$ aus P und ersetzt dann u durch v . Entsteht in irgendeinem Schritt das Eingabewort w , so wird w akzeptiert.

(\Leftarrow):

Wir zeigen, dass jede Turing-akzeptierbare Sprache L auch eine Chomsky-0-Sprache ist. L werde akzeptiert von einer DTM $M = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_f)$. Wir konstruieren jetzt eine Chomsky-0-Grammatik $G = (N, \Sigma, P, S)$ mit $L(G) = L$ in vier Schritten:

1. Der erste Teil der Grammatik erzeugt ein beliebiges, nicht leeres Wort $w \in \Sigma^+$ doppelt in einer Kette von Nichtterminalzeichen. Die Idee ist: Eine Kopie bleibt erhalten, während auf der zweiten Kopie M simuliert wird. Akzeptiert M , wird alles, was zur Simulation gehört, gelöscht, und übrig bleibt das (dann akzeptierte) Wort.
2. Der zweite Teil der Grammatik simuliert M .
3. Der dritte Teil der Grammatik löscht im Fall der Akzeptanz die Überreste der Simulation, wandelt aber die erste Kopie des Wortes in Terminalzeichen um.
4. Das leere Wort $w = \varepsilon$ wird als ein (kleiner) Sonderfall behandelt.

Um diese Idee zu realisieren, führen wir als Nichtterminale der Grammatik zweidimensionale Vektoren ein. Oben wird ein zu prüfendes Wort geschrieben (und nie gelöscht). Unten wird zuerst ebenfalls das zu prüfende Wort geschrieben, dann aber wird dort auch die Simulation durchgeführt. Dazu gehören zwei spezielle Endzeichen, $\$l$ und $\$r$, die das linke bzw. rechte Ende des bislang inspizierten Turingbandes symbolisieren.

Insgesamt verwenden wir die folgende Nichtterminalmenge:

$$N = \left\{ S \right\} \cup \left\{ \begin{pmatrix} x \\ X \end{pmatrix} \mid x \in \Sigma \cup \{\sqcup\} \text{ und } X \subseteq \Gamma \cup Q \cup \{\$l\} \cup \{\$r\} \right\},$$

wobei die unteren Komponenten X immer nur genau ein Element aus Γ und höchstens je ein Element aus Q , $\{\$l\}$ und $\{\$r\}$ enthalten. Der Klarheit halber verwenden wir im Beweis folgende Buchstaben:

a, b für Elemente aus Σ ,
 x, y für Elemente aus $\Sigma \cup \{\sqcup\}$
 und B, C für Elemente aus Γ (was ja $\Sigma \cup \{\sqcup\}$ einschließt).

Schritt 1: Doppel-Anfangskonfigurationen erzeugen.

Hier wird im Wesentlichen eine Kette von Nichtterminalen der Form $\begin{pmatrix} a \\ \{a\} \end{pmatrix}$ mit $a \in \Sigma$ erzeugt. Diese Kette stellt die beiden Kopien eines Anfangswortes $a_1 \dots a_n \in \Sigma^+$ dar (wir erzeugen also mindestens einen Buchstaben; um das leere Wort kümmern wir uns später). Das erste und das letzte Element des unteren Teils dieser Kette werden passend mit Anfangszustand und Links- bzw. Rechtsmarkierungen versehen:

$$\begin{aligned} 1.1: \quad S &\rightarrow \begin{pmatrix} a \\ \{a, q_0, \$l, \$r\} \end{pmatrix} \mid \begin{pmatrix} b \\ \{b, q_0, \$l\} \end{pmatrix} S' \quad \text{für alle } a, b \in \Sigma \\ 1.2: \quad S' &\rightarrow \begin{pmatrix} a \\ \{a\} \end{pmatrix} S' \mid \begin{pmatrix} b \\ \{b, \$r\} \end{pmatrix} \quad \text{für alle } a, b \in \Sigma \end{aligned}$$

Mit Hilfe dieser Grammatik sind für $n \geq 2$ folgende Ableitungen möglich:

$$S \xrightarrow{1.1, 1.2} \begin{pmatrix} a_1 \\ \{a_1, q_0, \$l\} \end{pmatrix} \begin{pmatrix} a_2 \\ \{a_2\} \end{pmatrix} \cdots \begin{pmatrix} a_n \\ \{a_n, \$r\} \end{pmatrix}.$$

Damit wird auf der oberen Zeile ein Eingabewort $a_1 a_2 \dots a_n \in \Sigma^+$ (fest)geschrieben, während auf der unteren Zeile die Turingmaschine M durch das Vorhandensein des Zustands q_0 quasi darauf wartet,

loszulaufen. $\$l$ markiert den linken Rand, $\$r$ den rechten Rand des „bislang besuchten“ Turingbandes. Zu beachten ist, dass es sich bei dem erzeugten Wort um kein terminales Wort, sondern um eine Kette von n Nichtterminalsymbolen der Grammatik handelt. Die Regeln 1.1, 1.2 sind also nicht rechtslinear! (Aber fast.)

Schritt 2: Transitionsrelation \rightarrow_M von M simulieren.

Hier werden die Bewegungsmöglichkeiten von M aus Definition 6.3.2 in der Grammatik G nachgebildet.

$$\begin{array}{ll}
2.1a: & \begin{pmatrix} x \\ X \end{pmatrix} \rightarrow \begin{pmatrix} \sqcup \\ \{\sqcup, q', \$l\} \end{pmatrix} \begin{pmatrix} x \\ (X \setminus \{B, q, \$l\}) \cup \{C\} \end{pmatrix} \left\{ \begin{array}{l} \text{falls } ((q, B), (q', C, L)) \in \delta \\ \text{und } \{B, q, \$l\} \subseteq X \text{ mit} \\ x \in \Sigma \cup \{\sqcup\}, B, C \in \Gamma, q, q' \in Q \end{array} \right. \\
2.1b: & \begin{pmatrix} y \\ Y \end{pmatrix} \begin{pmatrix} x \\ X \end{pmatrix} \rightarrow \begin{pmatrix} y \\ Y \cup \{q'\} \end{pmatrix} \begin{pmatrix} x \\ (X \setminus \{B, q\}) \cup \{C\} \end{pmatrix} \left\{ \begin{array}{l} \text{falls } ((q, B), (q', C, L)) \in \delta \\ \text{und } \{B, q\} \subseteq X \text{ mit} \\ x, y \in \Sigma \cup \{\sqcup\}, B, C \in \Gamma, q, q' \in Q \end{array} \right. \\
2.2: & \begin{pmatrix} x \\ X \end{pmatrix} \rightarrow \begin{pmatrix} x \\ (X \setminus \{B, q\}) \cup \{C, q'\} \end{pmatrix} \left\{ \begin{array}{l} \text{falls } ((q, B), (q', C, N)) \in \delta \\ \text{und } \{B, q\} \subseteq X \text{ mit} \\ x \in \Sigma \cup \{\sqcup\}, B, C \in \Gamma, q, q' \in Q \end{array} \right. \\
2.3a: & \begin{pmatrix} x \\ X \end{pmatrix} \rightarrow \begin{pmatrix} x \\ (X \setminus \{B, q, \$r\}) \cup \{C\} \end{pmatrix} \begin{pmatrix} \sqcup \\ \{\sqcup, q', \$r\} \end{pmatrix} \left\{ \begin{array}{l} \text{falls } ((q, B), (q', C, R)) \in \delta \\ \text{und } \{B, q, \$r\} \subseteq X \text{ mit} \\ x \in \Sigma \cup \{\sqcup\}, B, C \in \Gamma, q, q' \in Q \end{array} \right. \\
2.3b: & \begin{pmatrix} x \\ X \end{pmatrix} \begin{pmatrix} y \\ Y \end{pmatrix} \rightarrow \begin{pmatrix} x \\ (X \setminus \{B, q\}) \cup \{C\} \end{pmatrix} \begin{pmatrix} y \\ Y \cup \{q'\} \end{pmatrix} \left\{ \begin{array}{l} \text{falls } ((q, B), (q', C, R)) \in \delta \\ \text{und } \{B, q\} \subseteq X \text{ mit} \\ x, y \in \Sigma \cup \{\sqcup\}, B, C \in \Gamma, q, q' \in Q \end{array} \right.
\end{array}$$

Die Klauseln 2.1, 2.2 und 2.3 spiegeln Links- bzw. Nicht- bzw. Rechtsbewegungen des LSK wider. Dabei modellieren die Regeln 2.1a bzw. 2.3a, dass anhand der Randmarkierungen ein linker bzw. rechter Rand erkannt wird. Es wird dann links bzw. rechts ein neues Blankzeichen eingeführt (potenzielle Unendlichkeit!), und die Randzeichen werden weiter nach links bzw. nach rechts (und nie wieder zurück) geschoben. Die Regeln 2.1b und 2.3b sind für Links- bzw. Rechtsbewegungen zuständig, die nicht am Rand stattfinden. Es ist unschwer zu sehen, dass es immer nur *genau ein* Nichtterminalzeichen gibt, das auf der unteren Ebene einen Zustand q enthält. Die Zeichen $\$l$ und $\$r$ befinden sich immer im ganz linken bzw. im ganz rechten Nichtterminal. (Sie sind nötig, weil sonst die Regeln 2.1a und 2.3a auf ein inneres Nichtterminal angewendet werden könnten; dort sind allerdings die Regeln 2.1b bzw. 2.3b zuständig.)

Schritt 3: Endkonfiguration löschen.

Hier werden tatsächliche Terminalzeichen in Σ erzeugt, und zwar aus dem Wort, das in der ersten Zeile steht, vollständig aber nur dann, wenn die Simulation einen Endzustand ergeben hat (3.1). Da die Simulation über den linken bzw. rechten Rand von $a_1 \dots a_n$ hinausgelaufen sein kann, können in der oberen Zeile Blankzeichen entstanden sein. Diese müssen verschwinden, wenn ein Terminalwort entstehen soll, und das geht natürlich nur durch Umwandlung in das leere Wort (3.2).

$$3.1: \begin{pmatrix} a \\ X \end{pmatrix} \rightarrow a \quad \text{für } a \in \Sigma \text{ und } X \subseteq \Gamma \cup \{q_f\} \cup \{\$l\} \cup \{\$r\}$$

$$3.2: \begin{pmatrix} \sqcup \\ X \end{pmatrix} \rightarrow \varepsilon \quad \text{für } X \subseteq \Gamma \cup \{q_f\} \cup \{\$l\} \cup \{\$r\}$$

Schritt 4: Leeres Wort bedenken.

Damit die Grammatik auch das leere Wort erzeugen kann, falls M es akzeptiert, fügen wir noch eine Produktion hinzu:

$$4: S \rightarrow \left(\begin{array}{c} \sqcup \\ \{\sqcup, q_0, \$_l, \$_r\} \end{array} \right)$$

Offenbar entspricht dies der anfänglichen Konfiguration $q_0\varepsilon$ bzw. $q_0\sqcup$. Die Simulation (Schritt 2) kann daraus in der unteren Zeile eine Konfiguration mit Endzustand erzeugen, genau dann, wenn $\varepsilon \in L$ gilt, wonach Schritt 3.2 zur Produktion des leeren Wortes führt.

Insgesamt gilt für die so definierte Grammatik G mit Startsymbol S und für alle $w = a_1 \dots a_n \in \Sigma^+$ ($n \geq 2$) und $v \in \Gamma^*$:

$$\begin{aligned} q_0 w \xrightarrow{*}_M uq_f v &\Leftrightarrow S \vdash_G^* \left(\begin{array}{c} a_1 \\ \{a_1, q_0, \$_l\} \end{array} \right) \left(\begin{array}{c} a_2 \\ \{a_2\} \end{array} \right) \cdots \left(\begin{array}{c} a_n \\ \{a_n, \$_r\} \end{array} \right) && \text{(Regeln 1.1, 1.2)} \\ &\vdash_G^* \left\{ \begin{array}{l} \text{eine Kette von Nichtterminalzeichen, die} \\ \text{in der oberen Zeile } \sqcup^* a_1 \dots a_n \sqcup^* \text{ und} \\ \text{in der unteren Zeile } \sqcup^* uq_f v \sqcup^* \text{ enthält} \end{array} \right\} && \text{(Regeln 2.1a-2.3b)} \\ &\vdash_G^* w (= a_1 \dots a_n) && \text{(Regeln 3.1, 3.2),} \end{aligned}$$

und analog für ein einbuchstabiges Wort $w = a_1 \in \Sigma^+$, sowie für $w = \varepsilon \in \Sigma^*$ und $v \in \Gamma^*$:

$$\begin{aligned} q_0 w \xrightarrow{*}_M uq_f v &\Leftrightarrow S \vdash_G^* \left(\begin{array}{c} \sqcup \\ \{\sqcup, q_0, \$_l, \$_r\} \end{array} \right) && \text{(Regel 4)} \\ &\vdash_G^* \left\{ \begin{array}{l} \text{eine Kette von Nichtterminalzeichen, die} \\ \text{in der oberen Zeile } \sqcup^* \text{ und} \\ \text{in der unteren Zeile } \sqcup^* uq_f v \sqcup^* \text{ enthält} \end{array} \right\} && \text{(Regeln 2.1a-2.3b)} \\ &\vdash_G^* w (= \varepsilon) && \text{(Regeln 3.2).} \end{aligned}$$

Wir erhalten $L(G) = L$, wie gewünscht. □ 6.7.1

6.7.2 Chomsky-1-Sprachen

Wir untersuchen nun, welche Einschränkung von Turingmaschinen – *qua* Satz 6.7.1 – den monotonen Grammatiken entsprechen. Bei einer monotonen Grammatik können die Wörter in einer Ableitung höchstens länger werden (mit Ausnahme der Ableitung $S \rightarrow \varepsilon$). Da die Simulation in Teil (\Rightarrow) des Beweises von Satz 6.7.1, vom Wort w ausgehend, eine Ableitung rückwärts konstruiert, kann sie so von-statten gehen, dass die auf dem Restband erzeugten Wörter höchstens so lang sind wie das Eingabewort. Auch sind rechte Seiten von Produktionen, die länger als das Wort sind, uninteressant, da sie zur Erzeugung nicht nötig sind. Die Ersetzung von v durch u , wenn $u \rightarrow v$ eine monotone Produktion ist, könnte also sogar innerhalb der Wortgrenzen selbst geschehen, weil das Wort dadurch höchstens verkürzt werden kann. Das legt die folgende Definition nahe:

Definition 6.7.2 LINEAR BESCHRÄNKTER AUTOMAT

Eine Turingmaschine $M = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_f)$ heißt *linear beschränkter Automat (LBA)*, wenn für alle partiellen Berechnungen

$$\phi = \underbrace{q_0 w}_{k_0} \rightarrow_M k_1 \rightarrow_M k_2 \rightarrow_M \dots \quad (\text{wobei die } k_i, i = 0, 1, 2, \dots \text{ Konfigurationen sind})$$

gilt: in jedem endlichen Präfix π von ϕ gilt

$$0 \leq \text{anz}(R, \pi) - \text{anz}(L, \pi) \leq \max(0, |w| - 1),$$

wobei $\text{anz}(R, \pi)$ und $\text{anz}(L, \pi)$ die Anzahlen der Rechts- bzw. der Linksbewegungen des LSK in π sind.

☒ 6.7.2

Mit anderen Worten: ein LBA bewegt sich niemals über die linke Grenze des Eingabewortes w nach links, und auch niemals über die rechte Grenze von w nach rechts. Es gilt:

Satz 6.7.3 CHOMSKY-1-SPRACHEN = LBA-AKZEPTIERBARE SPRACHEN

Sei Σ ein Alphabet und sei $L \subseteq \Sigma^*$. Dann sind die beiden folgenden Eigenschaften äquivalent:

- L ist Chomsky-1.
- L ist durch einen LBA akzeptierbar.

Beweis: Der Beweis von Satz 6.7.1 ist fast vollständig wiederverwendbar.

Die Richtung (\Rightarrow) wurde bereits skizziert.

Für die Richtung (\Leftarrow) sei M ein LBA, der L akzeptiert, und sei G die Grammatik, die im Beweis von Satz 6.7.1 (\Leftarrow) konstruiert worden ist. In G sind nur die Produktionen (3.2) nicht monoton. Diese werden jedoch höchstens nötig nach Anwendungen von (2.1a) und (2.3a), oder nach Anwendung von (4). Anwendungen von (2.1a) und (2.3a) kommen allerdings wegen der LBA-Eigenschaft von M nicht vor; diese Produktionen können also sofort gestrichen werden. Die Produktion (4) ist für das leere Wort zuständig. Man kann allerdings leicht durch Simulation von M testen, ob das leere Wort in L liegt, denn M macht mit Eingabe $w = \varepsilon$ laut Definition weder Rechts- noch Linksbewegungen. Deswegen streichen wir auch Regel (4) (und als Konsequenz auch die einzigen nicht-monotonen Regeln (3.2.)) und unterziehen das leere Wort einer Extrabehandlung.

Genauer: sei G' die Grammatik, die aus G durch Streichen von (2.1a), (2.3a), (4), sowie (3.2) erhalten wird. Dann gilt nach dem eben Gesagten (und dem vorigen Beweis) $L(G') = L \setminus \{\varepsilon\}$. Sei

$$G'' = \begin{cases} G' & \text{falls } \varepsilon \notin L \\ G' \text{ plus Produktionen } S'' \rightarrow \varepsilon \mid S \text{ (und neues Startsymbol } S'') & \text{falls } \varepsilon \in L. \end{cases}$$

Es gilt $L = L(G'')$, und G'' ist vom Erweiterungstyp.

☒ 6.7.3

Achtung! In der Literatur wird ein LBA meist so definiert, dass ein Eingabewort zwischen zwei unverrückbare Grenzzeichen gesetzt wird. Das hat den Vorteil, dass man beim Programmieren eines solchen Automaten syntaktisch abfragen kann, ob der linke bzw. der rechte Rand erreicht ist, und dass man einem Automaten direkt ansieht, dass er ein LBA ist (was mit Definition 6.7.2 nicht der Fall ist), aber den Nachteil, dass Akzeptanz extra definiert werden muss und dass die beiden Beweise nicht ganz so direkt ineinander übergehen.

Bemerkungen

- Für praktische Zwecke mag es ungünstig sein, die LSK-Bewegungen einer TM alleine auf den Eingabebereich zu beschränken. Man könnte deshalb versucht sein, die Bedingung der Definition 6.7.2 folgendermaßen abzuschwächen:

$$|\text{anz}(R, \pi) - \text{anz}(L, \pi)| \leq \ell(|w|), \quad (6.3)$$

wobei ℓ irgend eine Funktion von \mathbb{N} nach \mathbb{N} ist. Man kann (durch eine Technik, die der im Beweis ähnelt) zeigen, dass man dadurch keine größere Sprachklasse gewinnt, sofern $\ell(n)$ eine *lineare* Funktion von $n \in \mathbb{N}$ ist. Diese Eigenschaft hat den LBA ihren Namen gegeben.

- Die Maschine M , die in Teil (\Rightarrow) des Beweises aus der gegebenen Grammatik konstruiert wird, ist im Allgemeinen hochgradig nichtdeterministisch. Man kann auf diese Maschine die Konstruktion 6.4.2 anwenden, um eine sprachäquivalente deterministische Maschine $\text{det}(M)$ zu bekommen. Offenbar braucht diese Maschine aber zur Simulation mehr Platz auf dem Band. Ob sie auch *wesentlich* mehr Platz braucht, d.h.: ob es gelingt, die Bedingung (6.3) mit einer deterministischen Maschine und mit einer geeigneten linearen Funktion einzuhalten oder nicht, ist ein offenes Problem – das sogenannte DLBA[?]=LBA-Problem – der theoretischen Informatik.

Ende der Bemerkungen

6.8 Übungsaufgaben

1. Gegeben sei die Sprache

$$L = \{a^n b^m c^n \mid m, n \in \mathbb{N} \wedge m > n\}$$

über $\Sigma = \{a, b, c\}$. Konstruieren Sie eine Turingmaschine TM , welche die Sprache L akzeptiert. Die Darstellung der Übergangsrelation soll anhand einer Turingtafel erfolgen. Beschreiben Sie die Arbeitsweise Ihrer Turingmaschine.

Achtung: Das Arbeitsalphabet Γ soll nur aus den Zeichen a, b, c und \sqcup (Blank) bestehen.

2. Gegeben seien das Alphabet $\Sigma = \{0, 1\}$ und die Palindromsprache (siehe Abschnitt 3.3):

$$L_p = \{w \mid w \text{ ist ein Palindrom über } \Sigma\}.$$

Konstruieren Sie eine Turingmaschine M , welche die Palindromsprache L_p akzeptiert, d.h. $L(M) = L_p$. Die Darstellung der Überföhrungsfunktion soll anhand einer Turingtafel oder eines Turingmaschinengraphen erfolgen. Beschreiben Sie die Arbeitsweise Ihrer Turingmaschine allgemein und durch Angabe der akzeptierenden Berechnung, wenn die Turingmaschine auf das Wort 10001 angesetzt wird.

3. Konstruieren Sie eine Turingmaschine M , welche die folgende Funktion berechnet:

$$f: \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & \begin{cases} 2 * n - 1 & , \text{ falls } n > 0 \\ 0 & , \text{ sonst.} \end{cases} \end{cases}$$

Die Darstellung von n und $f(n)$ soll in binärer Schreibweise erfolgen. Beschreiben Sie die Arbeitsweise Ihrer Turingmaschine informell und anhand einer Turingtafel oder eines Turingmaschinengraphen.

4. Gegeben seien zwei Sprachen L_1 und L_2 über einem Alphabet Σ , sowie zwei Turingmaschinen M_1 und M_2 mit $L(M_1) = L_1$ und $L(M_2) = L_2$. Gesucht sind Turingmaschinen, welche die Sprachen
- a) $L_1 \cap L_2$
 - b) $L_1 \cup L_2$

akzeptieren. Beschreiben Sie ausführlich die Arbeitsweise der zu konstruierenden Turingmaschinen (umgangssprachlich, aber dennoch möglichst präzise).

Kapitel 7

Berechenbarkeit und Entscheidbarkeit

In diesem Kapitel (genauer gesagt, in Abschnitt 7.2) wollen wir erstmals eine konkrete nicht-Turing-berechenbare Funktion und eine konkrete nicht-Turing-akzeptierbare Sprache angeben. Wir verwenden dazu die Methode der „Selbstanwendung“, indem wir die Beschreibungen von Turingmaschinen als Bandinhalte zulassen. Vorher – in Abschnitt 7.1 – untersuchen wir die Begriffe Turing-Berechenbarkeit und Turing-Akzeptierbarkeit noch etwas genauer und setzen sie zu zwei algorithmischen Begriffen, dem der Aufzählbarkeit und dem der Entscheidbarkeit, in Beziehung. In Abschnitt 7.3 wenden wir uns einigen unentscheidbaren Problemen bei formalen Grammatiken zu. Wir beenden dieses Kapitel – Abschnitt 7.4 – mit einer zusammenfassenden Darstellung der Chomsky-Hierarchie.

Wir wollen von der *Churchschen These* ausgehen, die besagt, dass der Begriff „Turing-berechenbar“ eine gute Formalisierung des intuitiven Begriffs „berechenbar“ ist. Diese These wurde von Alonzo Church ca. 1937 zuerst formuliert, als klar wurde, dass sehr viele versuchte Formalisierungen des Begriffs „Berechenbarkeit“ (darunter eben auch die Turing-Berechenbarkeit) auf das Gleiche herausliefen. Die Churchsche These hat intuitiven Charakter und ist im strengen Sinn unbeweisbar. Auch heute noch gibt es Überlegungen, z.B. im Zusammenhang mit neuen Rechenmodellen wie den Quantencomputern oder Rechnern mit massiver Parallelität, diese These eventuell neu zu fassen.

Die Churchsche These erlaubt es uns, manchmal etwas lax im Sprachgebrauch zu sein. Z.B. sagen wir nur „berechenbar“, wo streng genommen eigentlich „Turing-berechenbar“ stehen sollte, oder wir sprechen von einem „Algorithmus“, wo eine Turingmaschine (oder auch ein WHILE-Programm) gemeint ist.

7.1 Aufzählbarkeit und Entscheidbarkeit

7.1.1 Definitionen

Wir knüpfen an Satz 6.7.1 an. Nach diesem Satz kommt es auf das Gleiche heraus, ob eine Sprache von einer Grammatik erzeugt oder von einer Turingmaschine akzeptiert wird. Bei der Erzeugung sind die Wörter der Sprache der *Output* einer Grammatik, bei der Akzeptanz aber der *Input* einer Turingmaschine.

In der nächsten Definition fassen wir eine Sprache nicht als Input, sondern als *Output* einer TM auf. Im Folgenden sei Σ ein festes Alphabet.

Definition 7.1.1 REKURSIVE AUFZÄHLBARKEIT

Eine Sprache $L \subseteq \Sigma^*$ heißt *rekursiv aufzählbar* (kurz: r.a.), wenn sie entweder leer ist oder wenn eine totale Turing-berechenbare Funktion $f: \mathbb{N} \rightarrow \Sigma^*$ mit der Eigenschaft

$$L = \{f(0), f(1), f(2), \dots\}$$

existiert.

☒ 7.1.1

Die Idee ist ähnlich der Erzeugbarkeit durch eine Grammatik: die Wörter in L werden systematisch aufgezählt, indem die Turingmaschine, die f berechnet, die Indices 0, 1, 2 etc. als Eingabe bekommt und der Reihe nach die Wörter $f(0)$, $f(1)$, $f(2)$ etc. ausgibt. Wir erlauben, dass Wörter in dieser Aufzählung sich wiederholen; dies ist nötig, weil die endlichen Sprachen auch als rekursiv aufzählbar definiert sein sollen. Das Wort „rekursiv“, das in dieser Definition neu vorkommt, hat eine historische Bedeutung und kann als Synonym für „algorithmisch“ gelesen werden.

Man kann die Definition der rekursiven Aufzählbarkeit gut mit dem Begriff der Abzählbarkeit vergleichen (siehe Abschnitt 2.1.5). Eine Menge X ist abzählbar, wenn $X = \emptyset$ oder wenn eine Funktion f von \mathbb{N} nach X mit der Eigenschaft

$$X = \{f(0), f(1), f(2), \dots\}$$

existiert. Der einzige und entscheidende Unterschied ist, dass im Fall der rekursiven Aufzählbarkeit die Funktion f als Turing-berechenbar gefordert wird. Intuitiv heißt dies: eine Menge ist abzählbar, wenn man ihre Elemente *im Prinzip* in eine Reihenfolge 0, 1, 2, usw. bringen kann, sie ist rekursiv aufzählbar, wenn es *einen Algorithmus* gibt, der sie in einer solchen Reihenfolge *aufschreibt*. Jede rekursiv aufzählbare Sprache ist damit auch schon *per definitionem* abzählbar (wie auch sowieso schon auf Grund der Tatsache, dass jede Sprache abzählbar ist). Wir werden bald sehen, dass die Umkehrung nicht gilt: es gibt nämlich Sprachen, die nicht rekursiv aufzählbar sind.

Jetzt knüpfen wir an die Charakterisierung von Turing-Berechenbarkeit aus Satz 6.6.3 an, um noch einen weiteren Begriff zu definieren:

Definition 7.1.2 SEMI-ENTSCHEIDBARKEIT

Eine Sprache $L \subseteq \Sigma^*$ heißt *semi-entscheidbar*, wenn die Funktion χ_L^+ Turing-berechenbar ist. ☒ 7.1.2

Das bedeutet, dass zu einer semi-entscheidbaren Sprache ein Algorithmus existiert, der für ein beliebiges vorgegebenes Wort anhält, wenn das Wort in der Sprache ist, und andernfalls kreist.

Korollar 7.1.3 UMFORMULIERUNG VON SATZ 6.6.3

Eine Sprache ist semi-entscheidbar genau dann, wenn sie Turing-akzeptierbar ist. ☒ 7.1.3

Es liegt nahe, eine entsprechende Eigenschaft unter Benutzung der Funktion χ_L (an Stelle von χ_L^+) zu definieren:

Definition 7.1.4 ENTSCHEIDBARKEIT

Eine Sprache $L \subseteq \Sigma^*$ heißt *entscheidbar*, wenn die Funktion χ_L Turing-berechenbar ist. □ 7.1.4

Das bedeutet, dass es für eine entscheidbare Sprache einen Algorithmus gibt, der für ein beliebiges vorgegebenes Wort stets mit dem Ergebnis 1 anhält, wenn das Wort in der Sprache ist, und mit dem Ergebnis 0, wenn das Wort nicht in der Sprache ist.

Der entscheidende Unterschied zwischen Semi-Entscheidbarkeit und Entscheidbarkeit ist der Hochindex ⁺. Eine TM, die eine Sprache entscheidet, stoppt für *alle* Inputs, während eine TM, die eine Sprache L semi-entscheidet, nur für Inputs aus L stoppt.

7.1.2 Codierung von Entscheidungsproblemen als Sprachen

Dieser Abschnitt dient dazu, verschiedene Verwendungen des Wortes „entscheidbar“ miteinander in Bezug zu bringen (siehe die Abschnitte 4.6, 5.6 und 7.1.1). Er dient auch dazu, die Bedingungen zu erläutern, die an eine gute Codierung gestellt werden (siehe Abschnitt 2.5). Leser/innen, denen diese Beziehung klar ist, können ohne Weiteres direkt zu Abschnitt 7.1.3 springen.

Den Begriff „entscheidbar“ haben wir bereits im Zusammenhang mit Entscheidungsproblemen bei Grammatiken und Automaten kennen gelernt, zum Beispiel in Abschnitt 4.6 und in Abschnitt 5.6. Dort war der Begriff allerdings nur informell verstanden worden. Er kann jetzt durch den scharf gefassten Begriff der Definition 7.1.4 ersetzt werden,

indem ein Entscheidungsproblem als eine Sprache aufgefasst wird.

Wie das geschehen kann, wird in diesem Abschnitt erklärt. Wir übersetzen einfach ein Entscheidungsproblem in eine Sprache,

wobei die Ja-Instanzen genau den Wörtern der Sprache entsprechen.

Unter einem *Entscheidungsproblem* verstanden wir informell eine Spezifikation mit n Eingabeparametern und einem Booleschen Ausgabeparameter. Für die n Eingabeparameter I_1, \dots, I_n soll eine Eigenschaft $F(I_1, \dots, I_n)$ untersucht werden, und es soll ein „wahr“ oder ein „falsch“ – bzw. ein „ja“ oder ein „nein“ bzw. eine „1“ oder eine „0“ – als Ausgabe geliefert werden, je nachdem, ob die Eigenschaft gilt oder nicht (siehe Abbildung 7.1 links). Beispielsweise hat das *Wortproblem für allgemeine Grammatiken* zwei Eingabeparameter: eine Chomsky-0-Grammatik G und ein Wort $w \in \Sigma^*$ und die Frage lautet, ob $w \in L(G)$ gilt oder nicht (Abbildung 7.1 rechts). Je nachdem, welche Klasse von Grammatiken für G zugelassen ist, unterscheidet man weitere Wortprobleme. In der Abbildung 7.2 sind das *Wortproblem für monotone Grammatiken* und das *Wortproblem für kontextfreie Grammatiken* dargestellt.

Die *Semi-Entscheidbarkeit* eines Entscheidungsproblems mit den Eingabeparametern I_1, \dots, I_n und der Frage $F(I_1, \dots, I_n)$ bedeutet die Existenz eines Algorithmus (z.B. einer Turingmaschine) der folgenden Art:

```
input  $I_1, \dots, I_n$ ;
if  $F(I_1, \dots, I_n) \rightarrow$  stoppe □  $\neg F(I_1, \dots, I_n) \rightarrow$  kreise fi
```

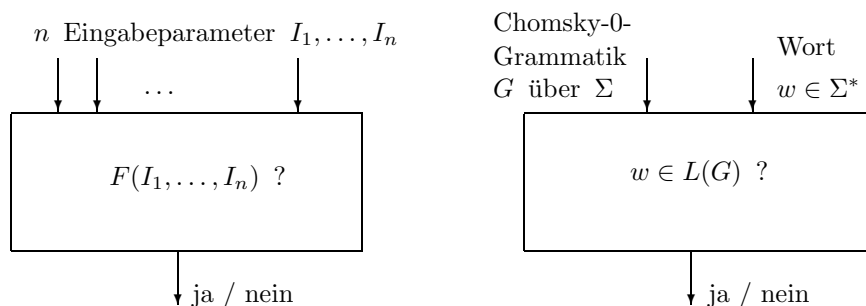


Abbildung 7.1: Entscheidungsproblem: schematische Darstellung (links); Wortproblem (rechts)

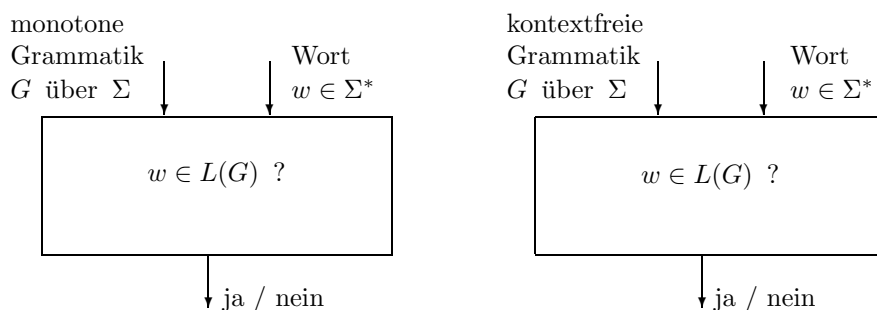


Abbildung 7.2: Wortproblem für monotone Grammatiken (links) und für k.f. Grammatiken (rechts)

Die *Entscheidbarkeit* bedeutet hingegen die Existenz eines Algorithmus der Art:

```

input  $I_1, \dots, I_n$ ;
if  $F(I_1, \dots, I_n) \rightarrow$  stoppe mit „ja“  $\square$   $\neg F(I_1, \dots, I_n) \rightarrow$  stoppe mit „nein“ fi

```

Der einzige – aber entscheidende – Unterschied liegt in der zweiten Alternative der **if**-Anweisung: hier kreist ein Semi-Entscheidungsalgorithmus, während ein Entscheidungsalgorithmus mit „nein“ stoppt.

Beispiele:

In Abschnitt 5.6 wurde bereits gezeigt, dass das Wortproblem für kontextfreie Grammatiken entscheidbar ist. Auch das Wortproblem für monotone Grammatiken ist entscheidbar. Dies weisen wir durch die Angabe des folgenden Algorithmus nach:

```

input monotone Grammatik  $G = (N, \Sigma, P, S)$ , Wort  $w \in \Sigma^*$ ;
if  $w = \varepsilon \rightarrow$  stoppe mit „ja“, falls  $(S \rightarrow \varepsilon) \in P$ , und mit „nein“ andernfalls
 $\square$   $w \neq \varepsilon \rightarrow$  konstruiere folgenden Graph:
    Knoten  $\{v \in (N \cup \Sigma)^* \mid |v| \leq |w|\}$ , Kanten  $\{(v', v) \mid v' \vdash_G v\}$ ;
    { Dieser Graph ist endlich und enthält sowohl  $S$  als auch  $w$  }
    stoppe mit „ja“, falls ein Pfad von  $S$  nach  $w$  in diesem Graphen existiert,
    und mit „nein“ andernfalls
fi

```

Dieser Algorithmus ist von exponentieller Laufzeit, gemessen an der Länge $|w|$ des Eingabewortes, weil der konstruierte Graph genau $(|N|+|\Sigma|)^{|w|}$ Knoten hat. Seine Korrektheit hängt von der Eigenschaft der Monotonie ab. Wenn ein Pfad von S nach w in dem angegebenen Graphen existiert, kann, ohne die Monotonie zu bemühen, gesagt werden, dass w in $L(G)$ liegt. Wenn jedoch kein solcher Pfad existiert, lässt dies nur im Falle der Monotonie auch den Schluss zu, dass w nicht in $L(G)$ liegt. Wäre G nicht monoton, könnte es eine Ableitung geben, die aus dem Graphen heraus zu einem Wort x der Länge $|x| > |w|$ führt, und danach erst (verkürzend) zu w . Der angegebene Algorithmus ist also kein Entscheidungsalgorithmus für das Wortproblem für allgemeine Grammatiken, sondern nur für das Wortproblem für monotone Grammatiken.

Das Wortproblem für allgemeine Grammatiken ist semi-entscheidbar:

```

input Chomsky-0-Grammatik  $G = (N, \Sigma, P, S)$ , Wort  $w \in \Sigma^*$ ;
durchsuche den Ableitungsbaum mit Wurzel  $S$  in Breitensuche;
if  $w$  wird gefunden  $\rightarrow$  stoppe mit „ $w \in L(G)$ “
   $\square$  der Baum ist endlich und enthält  $w$  nicht  $\rightarrow$  kreise
   $\square$  andernfalls  $\rightarrow$  fahre mit der Suche fort
fi

```

Dieser Algorithmus zeigt aber weder, dass das allgemeine Wortproblem entscheidbar ist, noch, dass es *nicht* entscheidbar ist! Es könnte ja noch einen „clevereren“ Algorithmus geben. Die Unentscheidbarkeit des allgemeinen Wortproblems folgt erst aus einigen Unmöglichkeitensargumenten, auf die wir bald eingehen werden.

Ende der Beispiele.

Um die Verbindung zwischen einem Entscheidungsproblem und einer formalen Sprache herzustellen und um den Input eines Entscheidungsproblems auf ein Turingband schreiben zu können, muss ein solcher Input als ein Wort einer Sprache codiert werden. Wir erklären das Prinzip anhand des Wortproblems. Eingaben sind also eine Grammatik G über Σ und ein Wort $w \in \Sigma^*$. Die Bestandteile N, Σ, P, S von G und das Wort w können als Tupel aufgefasst und mit den Methoden des Abschnitts 2.5 z.B. binär codiert werden. Üblicherweise werden, wie gesagt, diejenigen Codewörter in die Sprache aufgenommen, die zu einer „Ja“-Antwort gehören, in diesem Fall (Wortproblem) also genau die Codewörter, die zu G und w mit $w \in L(G)$ gehören. Für Wörter über dem Codealphabet gibt es im Allgemeinen drei Möglichkeiten:

- es handelt sich um ein Codewort in der Sprache (dann gibt ein eventuell existierender Entscheidungsalgorithmus eine „Ja“-Antwort aus);
- oder es handelt sich um ein Codewort, das nicht in der Sprache liegt (dann gibt ein eventuell existierender Entscheidungsalgorithmus eine „Nein“-Antwort aus, und ein eventuell existierender Semi-Entscheidungsalgorithmus kreist);
- oder es ist ein Wort, das gar kein Codewort ist, das also nicht zu einem Paar G, w gehört (dann braucht ein eventuell existierender Entscheidungsalgorithmus oder Semi-Entscheidungsalgorithmus nichts weiter zu tun, weil solche Eingaben uninteressant sind).

Die dritte dieser Möglichkeiten ist nur ein kleines formales Ärgernis, denn mit einer *surjektiven* Codierung kann man stets erreichen, dass *jedes* Codewort einem Paar G, w entspricht, und dann gibt es nur die beiden anderen Möglichkeiten. Ist die Codierung nicht surjektiv, gibt es wegen der vernünftigen (effektiven) Codierungsmethode zumindest immer einen Algorithmus, der für ein gegebenes Wort bestimmt, ob es

ein Codewort für eine Entscheidungsproblemeingabe ist oder nicht. Es gibt jedoch nicht immer einen Algorithmus, der für ein gegebenes Codewort entscheidet, ob dieses zur Sprache gehört oder nicht – das ist eben der Unterschied zwischen Entscheidbarkeit und Unentscheidbarkeit.

Ein Entscheidungsproblem kann also von jetzt an formal als eine Sprache über einem geeigneten Alphabet, z.B. über $\{0,1\}$, aufgefasst werden. Diese Sprache enthält genau die (codierten) „Ja“-Fälle des Entscheidungsproblems. Zum Beispiel entspricht dem Wortproblem für allgemeine Grammatiken die folgende Sprache:

$$\boxed{WP = \{bin(G, w) \mid G \text{ ist Chomsky-0-Grammatik über } \Sigma \text{ und } w \in L(G)\}} \subseteq \{0,1\}^*.$$

Dem Wortproblem für monotone Grammatiken entspricht die folgende Sprache:

$$\boxed{WP_{mon} = \{bin(G, w) \mid G \text{ ist monotone Grammatik über } \Sigma \text{ und } w \in L(G)\}} \subseteq \{0,1\}^*,$$

und dem Wortproblem für kontextfreie Grammatiken entspricht die Sprache

$$\boxed{WP_{kf} = \{bin(G, w) \mid G \text{ ist kontextfreie Grammatik über } \Sigma \text{ und } w \in L(G)\}} \subseteq \{0,1\}^*.$$

Diese Sprachen sind alle verschieden! (Es gilt allerdings $WP_{kf} \subseteq WP_{mon} \subseteq WP$.)

Früher (Abschnitte 4.6 und 5.6) haben wir die Entscheidbarkeit eines Entscheidungsproblems durch die Angabe eines Algorithmus gezeigt, der die Eingaben des Problems als **input**-Parameter interpretiert. Diese Betrachtungsweise ist gleichbedeutend mit der Angabe eines Algorithmus für die Entscheidbarkeit der entsprechenden Sprache über $\{0,1\}$, und zwar wieder wegen der (umkehrbaren) Effektivität der Codierung. Wir zeigen dies anhand des *Äquivalenzproblems für Grammatiken*: „Gegeben zwei Chomsky-0-Grammatiken G_1 und G_2 , gilt $L(G_1) = L(G_2)$?“ Das Problem hat zwei Inputparameter, G_1 und G_2 , und wir betrachten deswegen die – effektiv konstruierbare – Codierung dieses Problems als Sprache:

$$\boxed{\ddot{A}Q = \{bin(G_1, G_2) \mid G_1, G_2 \text{ sind Chomsky-0-Grammatiken über } \Sigma \text{ und } L(G_1) = L(G_2)\}}.$$

Behauptung: $\ddot{A}Q$ ist (als Sprache über $\{0,1\}$) entscheidbar im Sinne von Definition 7.1.4 genau dann, wenn das Äquivalenzproblem für Grammatiken entscheidbar ist.

Beweis: Sei $\ddot{A}Q$ entscheidbar. Ein Algorithmus zur Entscheidung des Äquivalenzproblems lautet:

```

input  $G_1, G_2$ ;
 $w := bin(G_1, G_2)$ ; { hier benutzen wir die Effektivität der Codierung }
stoppe mit „ja“, falls  $w \in \ddot{A}Q$ , und sonst mit „nein“ { durch Entscheidungsalgorithmus für  $\ddot{A}Q$  }

```

Sei umgekehrt das Äquivalenzproblem entscheidbar. Ein Algorithmus zur Entscheidung von $\ddot{A}Q$ lautet:

```

input  $w \in \{0,1\}^*$ ;
decodiere  $w$ ; { hier benutzen wir die effektive Umkehrbarkeit der Codierung }
if  $w$  nicht von der Form  $bin(G_1, G_2) \rightarrow$  kreise
   $\square$  else  $\rightarrow$  errechne  $G_1, G_2$  aus  $bin(G_1, G_2)$ 
    { auch hier benutzen wir die effektive Umkehrbarkeit der Codierung }
    stoppe mit „ja“, falls  $L(G_1) = L(G_2)$ , und sonst mit „nein“
    { durch Entscheidungsalgorithmus für  $L(G_1) \stackrel{?}{=} L(G_2)$  }
fi

```

Der Leser möge beachten, dass diese Beziehung ganz unabhängig von der Frage gilt, ob $\ddot{A}Q$ tatsächlich entscheidbar ist oder nicht! (Um ganz genau zu sein, ist $\ddot{A}Q$ *nicht* entscheidbar; siehe Abschnitt 7.3.3.)

7.1.3 Beziehungen zwischen Aufzählbarkeit und Entscheidbarkeit

Wir haben bereits gesehen, dass Semi-Entscheidbarkeit mit Turing-Akzeptierbarkeit zusammenfällt. Die beiden nächsten Sätze liefern weitere Charakterisierungen. Sei Σ ein Alphabet.

Satz 7.1.5 ZUSAMMENHANG ZWISCHEN SEMI-ENTSCHEIDBARKEIT UND ENTSCHEIDBARKEIT

$L \subseteq \Sigma^*$ ist genau dann entscheidbar, wenn sowohl L als auch $\bar{L} = \Sigma^* \setminus L$ semi-entscheidbar sind.

Beweis: Ein Entscheidungsalgorithmus für L kann so umgebaut werden, dass im Nein-Fall eine unendliche Schleife eingegangen wird, aber auch so, dass im Ja-Fall eine unendliche Schleife eingegangen wird. Im ersten Fall entsteht ein Semi-Entscheidungsalgorithmus für L , im zweiten Fall ein Semi-Entscheidungsalgorithmus für \bar{L} .

Seien umgekehrt M ein deterministischer Semi-Entscheidungsalgorithmus (eine DTM) für L und \bar{M} ein deterministischer Semi-Entscheidungsalgorithmus für \bar{L} . Wir konstruieren einen Entscheidungsalgorithmus für L :

```

input  $w \in \Sigma^*$ ;
for  $i = 0, 1, \dots$  do
  if  $M(w)$  stoppt nach  $i$  Schritten  $\rightarrow$  stoppe mit „ja“ { es gilt  $w \in L$  }
   $\square$   $\bar{M}(w)$  stoppt nach  $i$  Schritten  $\rightarrow$  stoppe mit „nein“ { es gilt  $w \notin L$  }
fi
end for

```

Wenn $w \in L$, wird die erste Alternative irgendwann ausgeführt, wenn $w \notin L$, die zweite. Also terminiert dieser Algorithmus stets. ☒ 7.1.5

Ähnlich wie der erste Teil oben kann der folgende Satz bewiesen werden:

Satz 7.1.6 EINE ABSCHLUSSEIGENSCHAFT ENTSCHEIDBARER SPRACHEN

$L \subseteq \Sigma^*$ ist entscheidbar genau dann, wenn \bar{L} entscheidbar ist.

Beweis: Ein Entscheidungsalgorithmus für L kann zu einem Entscheidungsalgorithmus für \bar{L} umgebaut werden, indem der Ja-Fall mit dem Nein-Fall vertauscht wird. ☒ 7.1.6

Schwerer zu beweisen ist der folgende Satz.

Satz 7.1.7 ZUSAMMENHANG ZWISCHEN SEMI-ENTSCHEIDBARKEIT UND AUFZÄHLBARKEIT

$L \subseteq \Sigma^*$ ist semi-entscheidbar genau dann, wenn L rekursiv aufzählbar ist.

Beweis: Den Fall $L = \emptyset$ kann man vorweg behandeln. L ist dann semi-entscheidbar durch $\mathbf{M}_{\text{loop}}(\{0, 1\})$ und rekursiv aufzählbar *per definitionem*.

(\Rightarrow): Sei $L \neq \emptyset$ semi-entscheidbar mittels (deterministischem) Algorithmus M . Sei $z \in L$ beliebig, aber fest gewählt. Wir benutzen eine beliebige effektiv umkehrbare und bijektive Codierung $g: \mathbb{N} \rightarrow (\Sigma^* \times \mathbb{N})$ und definieren eine Funktion f durch folgenden Algorithmus:

```

input  $n \in \mathbb{N}$ ;
 $(w, m) := g(n)$ ;
if  $M(w)$  stoppt in  $m$  Schritten  $\rightarrow$  stoppe mit Ausgabe  $w$  {  $w \in L$ , da  $M$  nur dann stoppt }
   $\square$   $M(w)$  stoppt in  $m$  Schritten nicht  $\rightarrow$  stoppe mit Ausgabe  $z$  {  $z \in L$  }
fi

```

Dann liefert f zu jedem $n \in \mathbb{N}$ ein Wort in Σ^* , ist also eine totale und berechenbare (da durch einen stets terminierenden Algorithmus definierte) Funktion von \mathbb{N} nach Σ^* . Außerdem liefert f nur Wörter aus L . Es gilt also $\text{cod}(f) \subseteq L$.

Beweis von $L \subseteq \text{cod}(f)$: sei $x \in L$. Dann stoppt $M(x)$ nach i Schritten. Sei $n = g^{-1}(x, i)$. Nach Definition von f gilt $f(n) = x$, also $x \in \text{cod}(f)$.

Es gilt also: f ist eine Funktion von \mathbb{N} nach Σ^* ; f ist berechenbar; und $L = \text{cod}(f)$. Der Beweis, dass L rekursiv aufzählbar ist, ist damit erbracht.

(\Leftarrow): Sei umgekehrt $L \neq \emptyset$ rekursiv aufzählbar mittels einer totalen und berechenbaren Funktion f von \mathbb{N} nach Σ^* . Ein Semi-Entscheidungsverfahren für L kann so definiert werden:

```

input  $w \in \Sigma^*$ ;
for  $i = 0, 1, 2, \dots$  do
  if  $f(i) = w \rightarrow$  stoppe mit „ja“ { es gilt  $w \in L$  }
   $\square$   $f(i) \neq w \rightarrow$  skip
fi
end for

```

Der Algorithmus hält genau dann, wenn $w \in L$.

☒ 7.1.7

Mit dem ersten Teil dieses Beweises kann der Beweis von Lemma 2.1.2(a) \Rightarrow (b) (der besagt: aus der Abzählbarkeit einer nicht leeren Menge folgt die Existenz einer surjektiven Funktion von \mathbb{N} auf diese Menge) verglichen werden.

Wir haben bisher die Äquivalenz folgender Begriffe (bezogen auf eine Sprache $L \subseteq \Sigma^*$) gezeigt:

- L ist *Turing-akzeptierbar*;
- L ist *akzeptierbar durch ein WHILE-Programm*;
- L ist *Chomsky-0*, d.h. durch eine Grammatik *erzeugbar*;
- L ist *semi-entscheidbar*, d.h., L hat eine berechenbare halbe charakteristische Funktion;
- L ist *rekursiv aufzählbar*, d.h., L ist entweder leer oder der Wertebereich einer totalen berechenbaren Funktion.

Dagegen sind die beiden folgenden Begriffe davon zu unterscheiden:

- *Entscheidbar*, weil Entscheidbarkeit mindestens so stark ist wie Semi-Entscheidbarkeit. Sobald wir einmal eine Sprache definiert haben, die semi-entscheidbar, aber nicht entscheidbar ist (was gleich im nächsten Abschnitt der Fall sein wird), sehen wir, dass Entscheidbarkeit *echt* stärker als Semi-Entscheidbarkeit ist. Wir haben außerdem die Sätze 7.1.5 und 7.1.6, wodurch die genaue Beziehung zwischen diesen Eigenschaften erklärt wird.
- *Turing-berechenbar*, weil sich dieser Begriff nicht auf Sprachen, sondern auf Funktionen bezieht (aber wir haben die Sätze 6.6.3 und 6.6.4, die die genaue Beziehung zwischen Turing-Berechenbarkeit und Turing-Akzeptierbarkeit angeben).

7.2 Unentscheidbarkeit und Unberechenbarkeit

In diesem Abschnitt diskutieren wir, ob die im letzten Abschnitt eingeführten Begriffe echt einschränkend sind. D.h., wir interessieren uns für folgende Fragen:

- (A) Gibt es Sprachen, die nicht semi-entscheidbar (Turing-akzeptierbar, rekursiv aufzählbar, ...) sind?
- (B) Gibt es Funktionen, die nicht Turing-berechenbar sind?
- (C) Gibt es außerdem Sprachen, die semi-entscheidbar, aber nicht entscheidbar sind?

Wir geben auf diese Fragen zwei Arten von Antworten: indirekte (durch ein Abzählbarkeitsargument) und direkte (indem für jede nachgefragte Kategorie eine Sprache bzw. Funktion tatsächlich konkret angegeben werden).

7.2.1 Abzählbarkeitsargumente und DTM-Codierungen

Die Fragen (A) und (B) sind leicht mit Hilfe der folgenden Sätze mit „Ja“ zu beantworten.

Satz 7.2.1 EXISTENZ NICHT-TURING-AKZEPTIERBARER SPRACHEN

Sei Σ ein beliebiges Alphabet. Es gibt eine Sprache $L \subseteq \Sigma^*$, die nicht Turing-akzeptierbar ist.

Beweis: Die Menge aller Sprachen über Σ ist nicht abzählbar, denn sie ist gleichmächtig mit der Menge aller Teilmengen von Σ^* , d.h., 2^{Σ^*} , und Lemma 2.1.7 ist (analog) anwendbar, da Σ^* eine abzählbar unendliche Menge ist.

Wir zeigen nun, dass die Menge der Turing-akzeptierbaren Sprachen über Σ abzählbar ist. Dazu genügt es, zu zeigen, dass die Menge der Turingmaschinen über Σ abzählbar ist, denn die Menge der von solchen Maschinen akzeptierten Sprachen kann keine größere Mächtigkeit haben als die Menge der Maschinen selbst.

Hier benötigen wir – zum ersten Mal – die Abzählbarkeit der *Anzahl* der Alphabete (Abschnitt 2.3). Dadurch ist die Anzahl der möglichen Bandalphabete Γ von Maschinen mit festem Eingabealphabet Σ abzählbar. Das Gleiche dürfen wir auch von der Anzahl der möglichen Zustandsmengen Q annehmen, während alle vorkommenden Mengen selbst endlich sind. Damit ist auch die Anzahl der Turingmaschinen mit Eingabealphabet Σ abzählbar. □ 7.2.1

Also ist die Menge aller T.a. Sprachen über dem Alphabet Σ sogar sehr klein, verglichen mit der Menge aller Sprachen über Σ . Es gilt genauso:

Satz 7.2.2 EXISTENZ NICHT-TURING-BERECHENBARER FUNKTIONEN

Es gibt eine partielle Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$, die nicht Turing-berechenbar ist.

Beweis: Analog unter Ausnutzung von Lemma 2.1.8.

☒ 7.2.2

Die Existenz einer Sprache über Σ , die semi-entscheidbar, aber nicht entscheidbar ist (Frage (C)), lässt sich nicht mit einem solchen Mächtigkeitsargument nachweisen, denn beide Mengen von Sprachen (die entscheidbaren wie auch die semi-entscheidbaren) sind abzählbar.

Dagegen lässt sich das Mächtigkeitsargument auf die Menge *aller* Turingmaschinen (nicht nur derjenigen über einem festen Alphabet Σ) ausdehnen, denn laut Festsetzung zu Beginn des Abschnitts 2.3 gilt, dass auch die Anzahl der in Frage kommenden Σ abzählbar ist.

Jetzt interessieren wir uns deterministische Turingmaschinen, deren Eingabealphabet die 0 und die 1 enthalten. Mit der genannten Abzählbarkeit und der systematischen Definition von Turingmaschinen existiert auch eine effektive und umkehrbare Codierung dieser Maschinen in die Menge der Wörter über dem Alphabet $\{0, 1\}$. Für eine Turingmaschine M bezeichnen wir mit $\text{bin}(M)$ ihren Code, und falls $w \in \{0, 1\}^*$ ein Codewort einer Maschine ist, bezeichnen wir mit M^w diese (wegen der Injektivität eindeutig bestimmte) Maschine. Die 0/1-Folge w heißt der *Index* von M^w .

Für Wörter $w \in \{0, 1\}^*$, die in dieser Codierung keine zugeordneten Maschinen haben, setzen wir fest, dass M^w die Maschine $\mathbf{M}_{\text{loop}}(\{0, 1\})$ ist (diese Maschine terminiert nie). Somit ist die Maschine M^w für *alle* Wörter $w \in \{0, 1\}^*$ eindeutig bestimmt, und darüber hinaus gilt Folgendes:

- Es existiert ein Algorithmus, der aus der Siebentupeldarstellung einer Turingmaschine M den Index $w \in \{0, 1\}^*$ mit $M = M^w$ berechnet.
- Es existiert ein Algorithmus, der aus einem beliebigen Wort $w \in \{0, 1\}^*$ die Siebentupeldarstellung von M mit $M^w = M$ errechnet.

Einer DTM M mit Eingabealphabet $\{0, 1\}$ kann kanonisch (und konsistent mit den Definitionen in den Abschnitten 6.5 und 6.6) sowohl eine akzeptierte Sprache als auch eine Schar berechneter Funktionen zugeordnet werden: $L(M)$, die von M akzeptierte Sprache, ist die Menge von Eingabewörtern über $\{0, 1\}$, für die M anhält, und $f^{(n)}(M): \mathbb{N}^n \xrightarrow{p} \mathbb{N}$, die von M berechnete n -stellige Funktion, ist folgendermaßen definiert:

$$(f(M))(x_1, \dots, x_n) = \begin{cases} y & \text{falls } M, \text{ angesetzt auf } \text{bin}(x_1, \dots, x_n), \text{ mit } uq_f \text{ bin}(y) \text{ anhält} \\ \text{undef} & \text{sonst.} \end{cases} \quad (7.1)$$

Wie vorher dürfen wir ohne Einschränkung normierte Berechenbarkeit annehmen; dann ist in (7.1) stets $u = \varepsilon$.

7.2.2 Das spezielle Halteproblem

Wir geben eine Sprache an, die nicht entscheidbar ist. Das *spezielle Halteproblem* oder *Selbstanwendungsproblem für Turingmaschinen* ist definiert als die Sprache

$$\boxed{sHP = \{w \in \{0, 1\}^* \mid M^w(w) \downarrow\}.}$$

$M^w(w) \downarrow$ bedeutet: „die w zugeordnete TM, angesetzt auf ihren eigenen Index, terminiert“. Das entsprechende Entscheidungsproblem ist:

Eingabe: die w zugeordnete Turingmaschine M ; Frage: hält M , angesetzt auf w ?

Das Problem heißt „speziell“, weil der Input von M die spezielle Form des Index dieser TM hat. Es heißt „Selbstanwendungsproblem“, weil die Maschine M auf sich selbst (d.h., auf ihren eigenen Index) angesetzt wird.

Satz 7.2.3 UNENTSCHEIDBARKEIT VON sHP / DIAGONALISIERUNGSARGUMENT
 sHP ist semi-entscheidbar, aber nicht entscheidbar.

Beweis: sHP ist durch den folgenden Algorithmus semi-entscheidbar:

input $w \in \{0, 1\}^*$;
 decodiere w und konstruiere $M = M^w$;
 simuliere $M(w)$;
 falls $M(w) \downarrow$, stoppe und akzeptiere.

Die Unentscheidbarkeit von sHP beweisen wir durch einen Widerspruch. Wir nehmen an, dass der Entscheidungsalgorithmus M_{sHP} die Sprache sHP entscheidet. Aus M_{sHP} konstruieren wir eine neue Turingmaschine M'_{sHP} wie folgt:

M'_{sHP} : **input** $w \in \{0, 1\}^*$;
 simuliere $M_{sHP}(w)$;
if $M_{sHP}(w)$ hält mit „ $w \in sHP$ “ \rightarrow kreise
 \square $M_{sHP}(w)$ hält mit „ $w \notin sHP$ “ \rightarrow stoppe
fi

Sei u der Index von M'_{sHP} , d.h. $M'_{sHP} = M^u$. Dann gilt:

$$\begin{aligned} M'_{sHP}(u) \downarrow &\Leftrightarrow (\text{Definition von } M'_{sHP}) \\ &\quad M_{sHP}(u) \text{ hält mit „} u \notin sHP\text{“} \\ &\Leftrightarrow (M_{sHP} \text{ ist Entscheidungsalgorithmus für } sHP) \\ &\quad u \notin sHP \\ &\Leftrightarrow (\text{Definition von } sHP) \\ &\quad \neg(M^u(u) \downarrow) \\ &\Leftrightarrow (M'_{sHP} = M^u) \\ &\quad \neg(M'_{sHP}(u) \downarrow). \end{aligned}$$

Dieser Widerspruch zeigt, dass eine solche Maschine M_{sHP} nicht existieren kann, dass also sHP unentscheidbar ist. □ 7.2.3

Es gibt in diesem Beweis eine kaum zu übersehende Analogie zu den Beweisen der Lemmata 2.1.7 und 2.1.8. Die Sprache sHP spielt eine ähnliche Rolle wie die Menge A im Beweis von Lemma 2.1.7. Die Maschine M'_{sHP} simuliert M_{sHP} , dreht aber deren Akzeptanzverhalten um (das kann sie sich leisten, weil M_{sHP} laut Annahme *stets* terminiert). Der Selbstanwendungswiderspruch ergibt sich hieraus, wenn die TM M'_{sHP} auf ihren eigenen Index angesetzt wird.

7.2.3 Algorithmische Reduktion

In diesem Abschnitt definieren wir eine Methode, um aus der bereits bekannten Unentscheidbarkeit eines Problems die Unentscheidbarkeit eines anderen Problems zu folgern. Beispielsweise können wir

die Unentscheidbarkeit von *sHP* benutzen, um die Unentscheidbarkeit eines anderen Halteproblems zu beweisen, und zwar ohne ein Diagonalisierungsargument benutzen zu müssen. Die Methode beruht auf der in der nächsten Definition eingeführten algorithmischen Reduktion.

Definition 7.2.4 REDUKTION

Es seien Σ_1 und Σ_2 zwei Alphabete und $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Sprachen. Dann heißt L_1 auf L_2 *reduzierbar* (in Zeichen: $L_1 \leq L_2$), wenn es eine Turing-berechenbare (totale) Funktion $f: \Sigma_1^* \rightarrow \Sigma_2^*$ gibt, so dass für alle $w \in \Sigma_1^*$ gilt:

$$w \in L_1 \Leftrightarrow f(w) \in L_2.$$

Wir sagen auch: $L_1 \leq L_2$ *mittels* f oder kürzer: $L_1 \leq_f L_2$. ☒ 7.2.4

Da f nicht surjektiv oder injektiv zu sein braucht, ist diese Definition nicht symmetrisch, obwohl die Äquivalenz \Leftrightarrow in ihr vorkommt: wenn L_1 auf L_2 reduzierbar ist, muss nicht notwendiger Weise auch L_2 auf L_1 reduzierbar sein. Jedoch ist \leq reflexiv und transitiv (leichte Übungsaufgabe).

Falls $L_1 \leq L_2$ mittels f gilt, ist dies eine Indikation dafür, dass L_2 „allgemeiner“ oder „algorithmisch schwerer zu lösen“ als L_1 ist. D.h.: hat man einen Algorithmus für L_2 , dann auch einen für L_1 . Das nächste Lemma formalisiert dies.

Lemma 7.2.5 REDUKTIONSLemma

- (a) Falls $L_1 \leq L_2$ gilt und L_2 semi-entscheidbar ist, dann ist auch L_1 semi-entscheidbar.
- (b) Falls $L_1 \leq L_2$ gilt und L_2 entscheidbar ist, dann ist auch L_1 entscheidbar.

Beweis:

- (a) Sei $L_1 \leq L_2$ mittels f und sei L_2 semi-entscheidbar, d.h. $\chi_{L_2}^+$ Turing-berechenbar. Es gilt für $x \in \Sigma_1^*$:

$$\begin{aligned} \chi_{L_1}^+(x) &= (\text{Definition von } \chi_{L_1}^+) \\ &\quad \mathbf{if } x \in L_1 \rightarrow 1 \quad \square \quad x \notin L_1 \rightarrow \mathbf{undef} \mathbf{fi} \\ &= (\text{Definition von } \leq, x \in L_1 \Leftrightarrow f(x) \in L_2) \\ &\quad \mathbf{if } f(x) \in L_2 \rightarrow 1 \quad \square \quad f(x) \notin L_2 \rightarrow \mathbf{undef} \mathbf{fi} \\ &= (\text{Definition von } \chi_{L_2}^+) \\ &\quad \chi_{L_2}^+(f(x)) \\ &= (f \circ \chi_{L_2}^+)(x). \end{aligned}$$

Mit f und $\chi_{L_2}^+$ ist aber auch die Komposition $f \circ \chi_{L_2}^+$ Turing-berechenbar (durch einen Algorithmus, der, gegeben $x \in \Sigma_1^*$, zuerst $f(x)$ berechnet und danach den Algorithmus für $\chi_{L_2}^+$ simuliert). Also folgt aus der Semi-Entscheidbarkeit von L_2 die Semi-Entscheidbarkeit von L_1 .

- (b) Analog unter Benutzung von χ_{L_1} und χ_{L_2} . ☒ 7.2.5

In der zweiten Zeile der obigen Argumentation wurden beide Richtungen von $x \in L_1 \Leftrightarrow f(x) \in L_2$ benutzt, die erste für die Alternative mit 1 und die zweite für die Alternative mit *undef*.

7.2.4 Weitere Halteprobleme und universelle Turingmaschinen

Die logische Kontraposition des Reduktionslemmas lautet:

Gilt $L_1 \leq L_2$ und ist L_1 nicht semi-entscheidbar (bzw. nicht entscheidbar), dann ist auch L_2 nicht semi-entscheidbar (bzw. nicht entscheidbar).

Wir benutzen das Reduktionslemma in dieser kontraponierten Form, um die Unentscheidbarkeit weiterer Halteprobleme nachzuweisen. Zunächst verwenden wir dazu $L_1 = sHP$ und $L_2 =$ das neue Problem.

Das spezielle Halteproblem ist ein Entscheidungsproblem mit nur einem Inputparameter, dem Index einer TM. Das *allgemeine Halteproblem* dehnt dies aus, indem nicht ein spezieller, sondern ein beliebiger anfänglicher Bandinhalt angenommen wird. Es ist definiert als die folgende Sprache über $\{0, 1\}$:

$$HP = \{bin(w, x) \mid w, x \in \{0, 1\}^* \text{ und } M^w(x) \downarrow\}.$$

Das allgemeine Halteproblem besagt intuitiv: Gegeben seien der Index einer Prozedur (Turingmaschine) w und ein Input x ; hält die Prozedur mit diesem Input an oder nicht? Dies ist „allgemeiner“ als das spezielle Halteproblem: angenommen, es gäbe einen Algorithmus, der das allgemeine Halteproblem entscheidet, dann ließe sich auch ein Algorithmus ableiten, der das spezielle Halteproblem löst, indem der Input speziell als Index der Prozedur (Turingmaschine) gewählt wird. Formal weisen wir die Unentscheidbarkeit von HP nach, indem wir eine Reduktion $sHP \leq HP$ angeben und Lemma 7.2.5 anwenden.

Satz 7.2.6 UNENTSCHEIDBARKEIT VON HP

HP ist semi-entscheidbar, aber nicht entscheidbar.

Beweis: HP ist durch den folgenden Algorithmus semi-entscheidbar:

input $w, x \in \{0, 1\}^*$;
 decodiere w und konstruiere $M = M^w$;
 simuliere $M(x)$;
 falls $M(x) \downarrow$, stoppe und akzeptiere.

Um die Unentscheidbarkeit von HP zu zeigen, reduzieren wir von sHP . Wir definieren eine Funktion $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ durch

$$f(w) = bin(w, w).$$

Dieses f ist total und Turing-berechenbar. Direkt nach den Definitionen von sHP , HP und f gilt:

$$w \in sHP \Leftrightarrow (\text{Def. } sHP) \ M^w(w) \downarrow \Leftrightarrow (\text{Def. } HP) \ bin(w, w) \in HP \Leftrightarrow (\text{Def. } f) \ f(w) \in HP,$$

insgesamt also $w \in sHP \Leftrightarrow f(w) \in HP$. Damit vermittelt f eine Reduktion von sHP nach HP (d.h., es gilt $sHP \leq HP$ mittels f), und HP ist unentscheidbar wegen Satz 7.2.3 und (der Kontraposition von) Lemma 7.2.5(b). □ 7.2.6

In diesem Beweis formalisiert f genau die vorher skizzierte Idee, den Parameter x als w zu wählen. Die „Richtung“ der Reduktion ist wichtig: eine Reduktion $HP \leq sHP$ würde nicht zum Ziel führen!

Wir wenden die Technik noch einmal an.

Wir fragen nun, ob es für eine fest vorgegebene Turingmaschine ein Entscheidungsverfahren gibt, welches ihr Halten entscheidet. Gegeben sei eine Turingmaschine M . Das *Halteproblem für M* ist die Sprache

$$\boxed{HP(M) = \{x \in \{0,1\}^* \mid M(x) \downarrow\}}.$$

Es gibt Turingmaschinen, deren zugeordnetes Halteproblem entscheidbar ist; zum Beispiel gilt das für eine Maschine, die die partielle Funktion $x - 2$ ($x \in \mathbb{N}$), die nur für $x \geq 2$ definiert ist, berechnet. Der Entscheidungsalgorithmus ist:

gegeben x , teste ob $x = 0$ oder $x = 1$; wenn ja, hält die Maschine nicht, andernfalls hält sie.

Es gibt jedoch andere Turingmaschinen, deren Halteproblem nicht entscheidbar ist. Betrachten wir zum Beispiel die Maschine M_{uni} über $\{0,1\}$, die folgendermaßen definiert ist:

```

 $M_{uni}$  : input  $u \in \{0,1\}^*$ ;
           if  $u = bin(w,x)$  mit  $w, x \in \{0,1\}^* \rightarrow$  simuliere  $M^w(x)$ 
            $\square$   $u \neq bin(w,x)$  mit  $w, x \in \{0,1\}^* \rightarrow$  kreise
           fi

```

M_{uni} wird *universelle Turingmaschine* oder *programmierbare Turingmaschine* genannt, weil sie die Indices w anderer Turingmaschinen auf ihrem Band interpretieren und diese anderen Turingmaschinen dann mit Input x simulieren kann.

Satz 7.2.8 UNENTSCHEIDBARKEIT DES HALTEPROBLEMS FÜR UNIVERSELLE TURINGMASCHINEN

$HP(M_{uni})$ ist semi-entscheidbar, aber nicht entscheidbar.

Beweis: Die Semi-Entscheidbarkeit wird analog wie oben gezeigt.

Zur Unentscheidbarkeit betrachten wir die Funktion $f = id_{\{0,1\}^*}$ und reduzieren von (nicht „auf“!) HP . Es gilt:

$$\begin{aligned} u \in HP &\Leftrightarrow (\text{Definition von } HP) \\ &u = bin(w, x) \wedge M^w(x) \downarrow \\ &\Leftrightarrow (\text{Definition von } M_{uni}) \\ &M_{uni}(u) \downarrow \\ &\Leftrightarrow (\text{Definitionen von } f \text{ und von } HP(M_{uni})) \\ &u = f(u) \in HP(M_{uni}). \end{aligned}$$

Also vermittelt f eine Reduktion $HP \leq HP(M_{uni})$. □ 7.2.8

Nun kommen wir auf die Fragen zurück, die zu Beginn des Abschnitts 7.2 gestellt worden sind. Die Frage (C), die ja nicht durch ein Mächtigkeitsargument beantwortbar war, ist durch die Angabe einer Reihe konkreter Sprachen beantwortet worden:

$sHP \subseteq \{0,1\}^*$	das spezielle Halteproblem
$HP \subseteq \{0,1\}^*$	das allgemeine Halteproblem
$HP_0 \subseteq \{0,1\}^*$	das Leerband-Halteproblem
$HP(M_{uni}) \subseteq \{0,1\}^*$	das Halteproblem für M_{uni} .

Alle diese Probleme sind semi-entscheidbar, aber nicht entscheidbar.

Die Frage (A) ist mit diesen Semientscheidbarkeitsresultaten und mit Hilfe von Satz 7.1.5 leicht konkret zu beantworten, denn jedes der genannten Probleme bringt auch ein nicht-semi-entscheidbares mit sich: das entsprechende komplementäre Problem. Also sind die Probleme

$$\begin{aligned}\overline{sHP} &= \{0, 1\}^* \setminus sHP \\ \overline{HP} &= \{0, 1\}^* \setminus HP \\ \overline{HP_0} &= \{0, 1\}^* \setminus HP_0 \\ \overline{HP(M_{uni})} &= \{0, 1\}^* \setminus HP(M_{uni})\end{aligned}$$

nicht semi-entscheidbar und daher auch weder Turing-akzeptierbar (nach Korollar 7.1.3) noch rekursiv aufzählbar (nach Satz 7.1.7).

Es verbleibt noch die Aufgabe (B), eine nicht-berechenbare Funktion zu finden. Dies ist jetzt sehr einfach. Zum Beispiel ist keine der Funktionen χ_{sHP} , χ_{HP} , χ_{HP_0} und $\chi_{HP(M_{uni})}$ Turing-berechenbar.

7.2.5 Der Satz von Rice

Bisher sind wir so vorgegangen: erst wurde ein Problem (nämlich sHP) als unentscheidbar erkannt. Dann wurden durch die Technik der Reduktion einige andere Probleme als unentscheidbar nachgewiesen. Das geschah immer einzeln; für jedes dieser Probleme musste ein passendes f gefunden werden. Der Satz von Rice, der in diesem Abschnitt bewiesen werden soll, hebt diese Vorgehensweise auf eine andere Stufe. Es wird nämlich mit einem Schlag eine unendlich große Klasse von nicht-entscheidbaren Problemen charakterisiert. Gehört ein Problem zu dieser Klasse, muss zu seiner Unentscheidbarkeit nicht noch extra eine passende Reduktion gefunden werden; diese ist sozusagen schon in den Beweis eingebaut. Es sei dazu

$$\mathcal{F}_{\{0,1\}} = \{g: \{0, 1\}^* \xrightarrow{p} \{0, 1\}^* \mid g \text{ ist Turing-berechenbar}\}$$

die Menge der partiellen, Turing-berechenbaren Funktionen von $\{0, 1\}^*$ nach $\{0, 1\}^*$.

Satz 7.2.9 SATZ VON RICE (FUNKTIONALE VERSION)

Sei \mathcal{S} eine beliebige Teilmenge von $\mathcal{F}_{\{0,1\}}$ mit $\emptyset \neq \mathcal{S} \neq \mathcal{F}_{\{0,1\}}$. Dann ist die Sprache

$$L_{\mathcal{S}} = \{w \in \{0, 1\}^* \mid M^w \text{ berechnet eine Funktion, die in } \mathcal{S} \text{ liegt}\}$$

unentscheidbar.

Dabei soll die Kurzaussage „ M^w berechnet eine Funktion, die in \mathcal{S} liegt“ so verstanden werden:

Es gibt eine Funktion $g \in \mathcal{S}$, so dass $M^w(x)$ mit dem Ergebnis $g(x)$ anhält, wenn $g(x)$ definiert ist, und kreist, wenn $g(x)$ nicht definiert ist.

Intuitiv besagt dieser Satz folgendes: interpretiert man \mathcal{S} als eine funktionale Spezifikation (d.h., wir suchen Turingmaschinen, deren berechnete Funktionen innerhalb von \mathcal{S} liegen), dann ist es – außer für die Spezialfälle $\emptyset = \mathcal{S}$ und $\mathcal{S} = \mathcal{F}_{\{0,1\}}$ – nicht möglich, einen Algorithmus anzugeben, der nachprüft, ob eine beliebige Turingmaschine diese Spezifikation erfüllt. Gilt $\emptyset = \mathcal{S}$, dann ist $L_{\mathcal{S}} = \emptyset$, und ein Entscheidungsalgorithmus, der stets „Nein“ liefert, prüft nach, ob $f(M^w)$ in \mathcal{S} liegt. Gilt andererseits $\mathcal{S} = \mathcal{F}_{\{0,1\}}$, ist $L_{\mathcal{S}} = \{0, 1\}^*$, und ein Algorithmus, der immer „Ja“ liefert, entscheidet die Zugehörigkeit von $f(M^w)$ zu \mathcal{S} .

Beweis: Sei U die auf $\{0,1\}^*$ überall undefinierte Funktion, d.h. $U(w) = \text{undef}$ für alle $w \in \{0,1\}^*$. Die Funktion U ist (zum Beispiel durch $\mathbf{M}_{\text{loop}}(\{0,1\})$) Turing-berechenbar und liegt somit in $\mathcal{F}_{\{0,1\}}$.

Sei nun $\mathcal{S} \subseteq \mathcal{F}_{\{0,1\}}$ mit $\emptyset \neq \mathcal{S} \neq \mathcal{F}_{\{0,1\}}$. Unser Ziel ist es, $sHP \leq L_{\mathcal{S}}$ zu zeigen.

Wir betrachten zunächst den Fall, dass $U \notin \mathcal{S}$.

Wegen $\emptyset \neq \mathcal{S}$ gibt es eine Funktion $g \in \mathcal{S}$, und wegen $\mathcal{S} \subseteq \mathcal{F}_{\{0,1\}}$ gibt es eine Turingmaschine $M(g)$, die g berechnet; wegen $U \notin \mathcal{S}$ gilt $g \neq U$. Wir halten eine beliebige solche Funktion g fest und betrachten ein beliebiges Wort $w \in \{0,1\}^*$. Wir ordnen (g und) w eine Maschine $M_g(w)$ über $\{0,1\}$ mit folgendem Algorithmus zu:

$M_g(w)$: **input** $y \in \{0,1\}^*$;
 simuliere $M^w(w)$;
 falls $M^w(w) \downarrow$, simuliere $(M(g))(y)$.

Dann gilt: Falls $M^w(w) \uparrow$, berechnet die Maschine $M_g(w)$ die Funktion $U \notin \mathcal{S}$ (da sie immer, d.h. für jede Eingabe y , kreist), und falls $M^w(w) \downarrow$, berechnet $M_g(w)$ die Funktion $g \in \mathcal{S}$ (da sie sich verhält wie $M(g)$). Betrachten wir jetzt $f_g: \{0,1\}^* \rightarrow \{0,1\}^*$ mit $f_g(w) = \text{Index von } M_g(w)$. Es gilt:

$w \in sHP \iff$ (Definition von sHP)
 $M^w(w) \downarrow$
 \iff (da $M^w(w) \uparrow \Rightarrow M_g(w)$ berechnet U und $M^w(w) \downarrow \Rightarrow M_g(w)$ berechnet g)
 $M_g(w)$ berechnet die Funktion g und nicht die Funktion U
 \iff (Definition von f_g , sowie $g \in \mathcal{S}$ (für \Rightarrow) und $U \notin \mathcal{S}$ (für \Leftarrow))
 $M^{f_g(w)}$ berechnet eine Funktion, die in \mathcal{S} liegt
 \iff (Definition von $L_{\mathcal{S}}$)
 $f_g(w) \in L_{\mathcal{S}}$

Damit vermittelt f_g eine Reduktion von sHP nach $L_{\mathcal{S}}$, und da sHP unentscheidbar ist, ist auch $L_{\mathcal{S}}$ unentscheidbar.

Sei andererseits $U \in \mathcal{S}$.

Wir betrachten $\overline{\mathcal{S}} = (\mathcal{F}_{\{0,1\}} \setminus \mathcal{S})$. Die Menge $\overline{\mathcal{S}}$ erbt die Eigenschaft $\emptyset \neq \overline{\mathcal{S}} \neq \mathcal{F}_{\{0,1\}}$ von \mathcal{S} : $\emptyset \neq \overline{\mathcal{S}}$ folgt aus $\mathcal{S} \neq \mathcal{F}_{\{0,1\}}$ und $\overline{\mathcal{S}} \neq \mathcal{F}_{\{0,1\}}$ folgt aus $\emptyset \neq \mathcal{S}$. Mit dem gleichen Argument wie oben zeigt man $sHP \leq \overline{\mathcal{S}}$, und daraus folgt, dass, $\overline{\mathcal{S}}$ unentscheidbar ist. Da eine Menge genau dann entscheidbar ist, wenn ihr Komplement entscheidbar ist (siehe Satz 7.1.6), ist auch \mathcal{S} unentscheidbar. □ 7.2.9

Die Formulierung dieses Satzes mag es etwas undurchsichtig erscheinen lassen, welches nun genau die Sprachen $L_{\mathcal{S}}$ sind, die durch ihn erfasst werden. Um dies zu verdeutlichen, geben wir ein paar Beispiele und Gegenbeispiele.

Drei Beispiele für Anwendungen des Satzes von Rice lauten: das Problem

Gegeben : Ein Wort $w \in \{0,1\}^*$;
 Frage : Berechnet M^w eine totale Funktion?

ist unentscheidbar; Begründung: es gibt sowohl eine totale als auch eine nicht-totale Funktion. Desgleichen sind die Probleme

Gegeben : Ein Wort $w \in \{0,1\}^*$;
 Frage : Berechnet M^w eine konstante Funktion?

und

Gegeben : Ein Wort $w \in \{0, 1\}^*$;

Frage : Berechnet M^w eine Funktion $h: \{0, 1\}^* \rightarrow \{0, 1\}^*$ mit $\text{bin}(42) = 101010 \in \text{cod}(h)$?

unentscheidbar.

Ende der drei Beispiele

Gegenbeispiele:

Dagegen sagt der Satz von Rice nichts über Probleme wie etwa das Folgende aus. Sei $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ eine feste berechenbare Funktion.

Gegeben : Ein Wort $w \in \{0, 1\}^*$;

Frage : Berechnen M^w und $M^{f(w)}$ die gleiche Funktion?

Es lässt sich bei diesem letzten Entscheidungsproblem keine geeignete Menge \mathcal{S} angeben, die den Satz von Rice anwendbar macht, denn die Frage des Problems bezieht sich auf eine Indexberechnung, und nicht nur auf das funktionale Verhalten der (codierten) Inputmaschine M^w .

Der Satz von Rice sollte auch nicht so missverstanden werden, als sei „fast jede“ Eigenschaft von Turingmaschinen unentscheidbar. Beispielsweise ist das Problem

Gegeben : Ein Wort $w \in \{0, 1\}^*$;

Frage : Macht M^w auf Leerband mehr als 42 Schritte?

entscheidbar; man braucht M^w dazu nur bis maximal 43 Schritte lang zu simulieren. Der Satz von Rice ist hier aus dem gleichen Grund wie oben nicht anwendbar: da es sich um eine Eigenschaft der Maschine handelt, nicht aber der von ihr berechneten Funktion, kann keine geeignete Menge \mathcal{S} gefunden werden.

Ende der Gegenbeispiele

Der Satz von Rice kann mit Hilfe Turing-akzeptierbarer Sprachen statt mit Turing-berechenbaren Funktionen umformuliert werden (mit dem gleichen Beweis). Dazu betrachtet man anstelle der Menge $\mathcal{F}_{\{0,1\}}$ die Menge $\mathcal{L}_{\{0,1\}}$ der T.a. Sprachen über $\{0, 1\}$. Eine Eigenschaft solcher Sprachen ist dann eine nicht-triviale Teilmenge $\mathcal{S} \subseteq \mathcal{L}_{\{0,1\}}$, und der Satz lautet folgendermaßen:

Satz 7.2.10 SATZ VON RICE (SPRACHLICHE VERSION)

Sei \mathcal{S} eine beliebige Teilmenge von $\mathcal{L}_{\{0,1\}}$ mit $\emptyset \neq \mathcal{S} \neq \mathcal{L}_{\{0,1\}}$. Dann ist die Sprache

$$L_{\mathcal{S}} = \{w \in \{0, 1\}^* \mid M^w \text{ akzeptiert eine Sprache, die in } \mathcal{S} \text{ liegt}\}$$

unentscheidbar.

Beispiele:

Aus Satz 7.2.10 folgt sofort, dass das Leerheitsproblem

Gegeben : Ein Wort $w \in \{0, 1\}^*$;

Frage : Gilt $L(M^w) = \emptyset$?

(7.2)

und das Nicht-Leerheitsproblem

Gegeben : Ein Wort $w \in \{0, 1\}^*$;

Frage : Gilt $L(M^w) \neq \emptyset$?

unentscheidbar sind. Zur Unentscheidbarkeit des Leerheitsproblems betrachtet man die Eigenschaft $\mathcal{S} = \{\emptyset\}$ (*nicht* $\mathcal{S} = \emptyset!$) und die Unentscheidbarkeit des Nicht-Leerheitsproblems folgt hieraus mit Hilfe von Satz 7.1.6.

Ende der Beispiele

Aus Satz 7.1.5 folgt, dass entweder das Leerheitsproblem oder das Nicht-Leerheitsproblem *nicht* semi-entscheidbar ist. Welches der beiden nicht semi-entscheidbar ist, macht man sich leicht durch die Angabe eines Semi-Entscheidungsalgorithmus klar. Man kann die anfänglichen Bandinhalte x systematisch (nicht-deterministisch) generieren und M^w damit simulieren. Wird bei der Simulation von $M^w(x)$ ein akzeptiertes Wort x entdeckt, kann das Nicht-Leerheitsproblem mit „ja“ beantwortet werden. Das Leerheitsproblem kann hingegen bei keiner endlichen Simulation mit „ja“ beantwortet werden, wenn bis dahin kein akzeptiertes Wort entdeckt wurde, weil es ja später noch eins geben kann. Somit ist das Nicht-Leerheitsproblem semi-entscheidbar, das Leerheitsproblem ist dagegen nicht semi-entscheidbar.

7.3 Unentscheidbarkeitsresultate bei Grammatiken

Zuerst zeigen wir in diesem Abschnitt die Unentscheidbarkeit einiger in Abschnitt 7.1.2 definierter Entscheidungsprobleme. Danach wenden wir uns speziell kontextfreien Grammatiken zu und schließen die Lücken, die am Ende des Kapitels 5 offen gelassen worden waren.

7.3.1 Unentscheidbarkeitsresultate bei allgemeinen Grammatiken

Zuerst definieren wir ein weiteres Problem, das *allgemeine Ableitungsproblem für Grammatiken*:

Gegeben : Eine Chomsky-0-Grammatik $G = (N, \Sigma, P, S)$ über Σ und zwei Wörter $u, v \in (N \cup \Sigma)^*$;
Frage : Gilt $u \vdash_G^* v$?

Die zugehörige Sprache ist:

$$AP = \{bin(G, u, v) \in \{0, 1\}^* \mid G \text{ ist Grammatik über } \Sigma, u, v \in (N \cup \Sigma)^*, \text{ und } u \vdash_G^* v\}.$$

Satz 7.3.1 *WP, ÄQ UND AP SIND UNENTSCHEIDBAR*

Das allgemeine Wortproblem, das allgemeine Äquivalenzproblem und das allgemeine Ableitungsproblem für Grammatiken sind unentscheidbar.

Beweis: Für das Wortproblem verwenden wir eine Reduktion vom allgemeinen Halteproblem *HP*. Zu $w \in \{0, 1\}^*$ kann (effektiv) eine Grammatik G_w mit $L(G_w) = L(M^w)$ konstruiert werden. Eine solche Konstruktion wird im Beweis von Satz 6.7.1 angegeben. Sei f die Funktion, die dem Wort $bin(w, x) \in$

$\{0, 1\}^*$ mit $w, x \in \{0, 1\}^*$ das Wort $\text{bin}(G_w, x) \in \{0, 1\}^*$ zuordnet. Dann gilt:

$$\begin{aligned} \text{bin}(w, x) \in HP &\Leftrightarrow (\text{Definition von } HP) \\ &M^w(x) \downarrow \\ &\Leftrightarrow (\text{Definition von } L(M^w)) \\ &x \in L(M^w) \\ &\Leftrightarrow (L(G_w) = L(M^w)) \\ &x \in L(G_w) \\ &\Leftrightarrow (\text{Definition von } f, \text{ Definition von } WP) \\ &f(\text{bin}(w, x)) \in WP. \end{aligned}$$

Damit vermittelt f eine Reduktion von HP nach WP .

Für das Äquivalenzproblem verwenden wir eine Zwischenstufe, das *Enthaltenseinsproblem*:

Gegeben: G_1 und G_2 . Frage: Gilt $L(G_1) \subseteq L(G_2)$?

Für zwei beliebige Mengen X und Y gilt: $X \subseteq Y$ genau dann, wenn $X \cup Y = Y$.

Wenn das Äquivalenzproblem entscheidbar wäre, dann wäre auch das Enthaltenseinsproblem auf folgende Weise entscheidbar. Gegeben zwei Grammatiken G_1 und G_2 über Σ , konstruiere zuerst eine Grammatik G mit $L(G) = L(G_1) \cup L(G_2)$ (durch die Konstruktion in Satz 5.4.1(i), die allgemein gilt); teste dann, ob $L(G) = L(G_2)$; wenn ja, gilt $L(G_1) \subseteq L(G_2)$, andernfalls gilt $L(G_1) \not\subseteq L(G_2)$.

Wenn andererseits das Enthaltenseinsproblem entscheidbar wäre, dann wäre auch das Wortproblem folgendermaßen entscheidbar. Gegeben eine Grammatik G über Σ und ein Wort $w \in \Sigma^*$, konstruiere eine Grammatik $G(w)$, die die Sprache $\{w\}$ erzeugt (z.B. mit der einzigen Produktion $S \rightarrow w$); teste dann, ob $L(G(w)) \subseteq L(G)$; falls ja, gilt $w \in L(G)$, falls nein, gilt $w \notin L(G)$.

Diese Reduktionskette führt erst das Enthaltenseinsproblem auf das Äquivalenzproblem zurück, und dann das Wortproblem auf das Enthaltenseinsproblem. Wegen der Transitivität der Reduktionsrelation ist damit auch das Wortproblem auf das Äquivalenzproblem zurückgeführt, und wegen der Unentscheidbarkeit des ersteren ist auch das Äquivalenzproblem unentscheidbar, was zu zeigen war.

Die Unentscheidbarkeit des Ableitungsproblems zeigen wir schließlich dadurch, dass wir das Wortproblem darauf zurückführen. Wenn das Ableitungsproblem entscheidbar wäre, dann auch das Wortproblem auf folgende Weise. Gegeben eine Grammatik $G = (N, \Sigma, P, S)$ und ein Wort $w \in \Sigma^*$, teste durch den fiktiven Algorithmus für das Ableitungsproblem, ob $S \vdash_G^* w$; falls ja, $w \in L(G)$, falls nein, $w \notin L(G)$. \square 7.3.1

7.3.2 Das Postsche Korrespondenzproblem

Bevor wir uns speziell kontextfreien Grammatiken zuwenden, betrachten wir ein wichtiges (unentscheidbares) Problem, das sich im kontextfreien Zusammenhang besonders gut benutzen lässt. Das *Postsche Korrespondenzproblem* wurde von Emil Post ca. 1935 gestellt, der Unentscheidbarkeitsbeweis wurde allerdings erst ca. 1957 gefunden. In diesem Problem geht es darum, ein Wort auf zwei verschiedene Weisen zu konstruieren. Eine Instanz des Postschen Korrespondenzproblems ist folgendermaßen charakterisiert:

Gegeben : Zwei nicht leere Listen gleicher Länge von nicht leeren Wörtern :
 $A = u_1, \dots, u_n$ und $B = v_1, \dots, v_n$ mit $n \neq 0$ und $u_i, v_i \neq \varepsilon$ für alle $1 \leq i \leq n$.
 Frage : Gibt es eine Indexfolge i_1, i_2, \dots, i_m mit $u_{i_1}u_{i_2} \dots u_{i_m} = v_{i_1}v_{i_2} \dots v_{i_m}$?

Die Indexfolge i_1, i_2, \dots, i_m (falls sie existiert) wird auch eine *Lösung* der Probleminstance – oder eine *Korrespondenz* – genannt. Als Sprache ist das Postsche Korrespondenzproblem folgendermaßen definiert:

$$PCP = \{((u_1, v_1), \dots, (u_n, v_n)) \mid \exists \text{ Alphabet } \Sigma \text{ mit } u_i, v_i \in \Sigma^+, n \in \mathbb{N} \setminus \{0\}, \\ \exists m \in \mathbb{N} \setminus \{0\} \exists i_1, \dots, i_m \in \mathbb{N} \setminus \{0\}: u_{i_1} u_{i_2} \dots u_{i_m} = v_{i_1} v_{i_2} \dots v_{i_m}\}$$

(dabei steht *PCP* für „Post’s Correspondence Problem“). Das *Postsche Korrespondenzproblem über festgehaltenem Alphabet* Σ , kurz *PCP* $_{\Sigma}$, ist die Sprache

$$PCP_{\Sigma} = \{((u_1, v_1), \dots, (u_n, v_n)) \mid u_i, v_i \in \Sigma^+, n \in \mathbb{N} \setminus \{0\}, \\ \exists m \in \mathbb{N} \setminus \{0\} \exists i_1, \dots, i_m \in \mathbb{N} \setminus \{0\}: u_{i_1} u_{i_2} \dots u_{i_m} = v_{i_1} v_{i_2} \dots v_{i_m}\}.$$

Beispiele (vier Korrespondenzprobleminstance K1, K2, K3 und K4):

K1	A	B
i	u_i	v_i
1	1	111
2	10111	10
3	10	0

K2	A	B
i	u_i	v_i
1	00	10
2	1	0
3	101	0

K3	A	B
i	u_i	v_i
1	011	0
2	01	011
3	01	101
4	10	001

K4	A	B
i	u_i	v_i
1	10	101
2	1011	11
3	101	011

Für K1 gibt es die folgende Lösung: 2, 1, 1, 3; denn es gilt $u_2 u_1 u_1 u_3 = v_2 v_1 v_1 v_3$; man sagt auch: K1 ist eine *Ja-Instanz* des Postschen Korrespondenzproblems. Das Problem K2 ist eine *Nein-Instanz*, denn es hat keine Lösung. Das ist folgendermaßen zu sehen: angenommen, $u_{i_1} u_{i_2} \dots u_{i_m} = v_{i_1} v_{i_2} \dots v_{i_m}$ mit $1 \leq i_j \leq 3$, dann müsste entweder u_{i_1} ein Präfix von v_{i_1} sein oder umgekehrt (oder beides). Da in K2 jedoch alle Paare (u_i, v_i) unvergleichbar im Sinne der Präfixordnung sind, kann es keine Lösung geben. Die Probleme K3 und K4 deuten an, dass das *PCP* eine hohe algorithmische Komplexität haben könnte. Das Problem K3 ist eine Ja-Instanz, aber eine kleinste Lösung hat die Länge 66. Das Problem K4 ist eine Nein-Instanz, aber die Begründung dafür ist nicht so offensichtlich wie bei Problem K2. In der Tat gilt allgemein:

Satz 7.3.2 *PCP* IST UNENTSCHEIDBAR

PCP ist semi-entscheidbar, aber nicht entscheidbar.

Beweis:

Die Semi-Entscheidbarkeit zeigt man durch einen naheliegenden „brute-force“-Algorithmus: erzeuge die Indexfolgen $i_1 \dots i_m$ in irgendeiner systematischen Reihenfolge und prüfe jedes Mal, ob die *PCP*-Bedingung erfüllt werden kann. Dieses Verfahren bricht genau dann ab, wenn es eine Lösung gibt.

Die Unentscheidbarkeit des *PCP* zeigen wir in zwei Reduktionsschritten:

$$AP \leq MPCP \quad \text{und} \quad MPCP \leq PCP,$$

wobei *MPCP* das folgendermaßen definierte *modifizierte Postsche Korrespondenzproblem* ist:

- Gegeben : Zwei nicht leere Listen gleicher Länge von nicht leeren Wörtern:
 $A = u_1, \dots, u_n$ und $B = v_1, \dots, v_n$ mit $n \neq 0$ und $u_i, v_i \neq \varepsilon$ für alle $1 \leq i \leq n$.
 Frage : Gibt es eine Indexfolge i_1, i_2, \dots, i_m mit $u_{i_1} u_{i_2} \dots u_{i_m} = v_{i_1} v_{i_2} \dots v_{i_m}$ und $i_1 = 1$?

Beim modifizierten Korrespondenzproblem wird also gefragt, ob eine Lösung existiert, die mit dem ersten Wortpaar beginnt.

Um $AP \leq MPCP$ zu zeigen, seien $G = (N, \Sigma, P, S)$ eine Chomsky-0-Grammatik und $u, v \in (N \cup \Sigma)^*$ zwei Wörter. Sei $\#$ ein Symbol, das weder in N noch in Σ vorkommen möge. Wir konstruieren folgendermaßen zwei Listen A und B als Eingabe für eine $MPCP$ -Instanz:

	A	B	
Erstes Wortpaar	$\#$	$\#u\#$	
Produktionen	p	q	für alle $(p \rightarrow q) \in P$
Kopieren	a	a	für alle $a \in N \cup \Sigma \cup \{\#\}$
Abschluss	$\#v\#$	$\#$	

Wir zeigen jetzt, dass $u \vdash_G^* v$ genau dann, wenn diese A, B eine Korrespondenz haben.

(\Rightarrow): Es gelte $u \vdash_G^* v$. Dann gibt es eine Ableitung von u nach v der Form

$$\begin{aligned}
 u &= u_0 p_0 v_0 \vdash_G u_0 q_0 v_0 &= u_1 p_1 v_1 & \text{(mit } (p_0 \rightarrow q_0) \in P \text{)} \\
 &\vdash_G u_1 q_1 v_1 &= u_2 p_2 v_2 & \text{(mit } (p_1 \rightarrow q_1) \in P \text{)} \\
 &\vdots && \\
 &\vdash_G u_{k-1} q_{k-1} v_{k-1} &= u_k p_k v_k & \text{(mit } (p_{k-1} \rightarrow q_{k-1}) \in P \text{)} \\
 &\vdash_G u_k q_k v_k &= v & \text{(mit } (p_k \rightarrow q_k) \in P \text{)}.
 \end{aligned}
 \tag{7.3}$$

Zu dieser Ableitung gibt es die in Abbildung 7.3 gezeigte Korrespondenz von A, B , wobei die erste Wortreihe in der oberen Zeile und die zweite Wortreihe in der unteren Zeile stehen. Die beiden Wortreihen (obere und untere Zeile in der Abbildung) sind wegen der Gleichungen in (7.3) genau gleich.

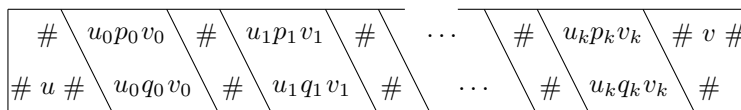


Abbildung 7.3: Korrespondenz bei gegebener Ableitung

Die „Endstücke“ mit u und v stellen das erste Wortpaar bzw. das Abschlusspaar dar. Die Teilwörter $u_i, v_i, \#$ werden symbolweise durch die Kopier-Paare in A, B gebildet. Die Teilwörter p_i, q_i werden durch das Produktionspaar (p_i, q_i) gelegt.

(\Leftarrow): Gegeben sei jetzt eine Korrespondenz von A, B , die mit dem ersten Wortpaar beginnt. Aus der Gestalt der Wortpaare in A, B folgt, dass die Korrespondenz so aufgebaut sein muss, wie es in Abbildung 7.4 gezeigt ist. Es gibt also Wörter w_0, \dots, w_k mit $u = w_0, w_k = v, w_i \in (N \cup \Sigma)^*$ und

$$\# w_0 \# w_1 \# w_2 \dots \# v \# = \# u \# w_1 \# w_2 \# \dots w_k \#.$$

Es gilt $w_i \vdash_G^* w_{i+1}$, für alle i mit $0 \leq i < k$. Denn da die w_i kein Zeichen $\#$ enthalten, können sie höchstens durch die Korrespondenzen der Zeilen 2 (Produktionen) oder 2 (Kopieren) gelegt worden sein, also durch null- oder mehrmalige Anwendung einer Produktion. Es folgt also

$$u = w_0 \vdash_G^* w_1 \vdash_G^* w_2 \vdash_G^* \dots \vdash_G^* w_k = v,$$

und damit $u \vdash_G^* v$, wie behauptet.

(\Leftarrow) gilt nur, weil wir im Sinne der MPCP mit dem ersten Wortpaar beginnen. Ohne diese Einschränkung würde z.B. das Kopierpaar $(\#, \#)$ eine triviale Korrespondenz liefern, aus der nicht $u \vdash_G^* v$ folgt.

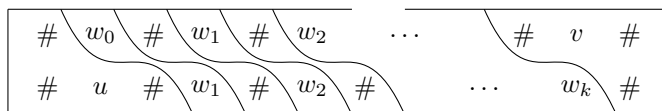


Abbildung 7.4: Ableitung bei gegebener Korrespondenz

Um die zweite Reduktion, $MPCP \leq PCP$, zu zeigen, seien $A = u_1, \dots, u_n$ und $B = v_1, \dots, v_n$ eine Eingabe von MPCP. Aus A und B konstruieren wir zwei neue, um je zwei Wörter längere, Listen A' und B' , so dass A und B eine Lösung im Sinne des MPCP haben, genau dann, wenn A' und B' eine Lösung im Sinne des PCP haben. Seien zunächst \star und $\$$ zwei neue Zeichen, die nicht in dem Alphabet Σ des MPCP A, B vorkommen. Für $a_1 \dots a_k \in \Sigma^+$ sei

$$\text{links}(a_1 \dots a_k) = \star a_1 \star a_2 \dots \star a_k \quad \text{und} \quad \text{rechts}(a_1 \dots a_k) = a_1 \star a_2 \star \dots a_k \star.$$

Die Konstruktion der Listen A' und B' aus A und B ist in Abbildung 7.5 gezeigt.

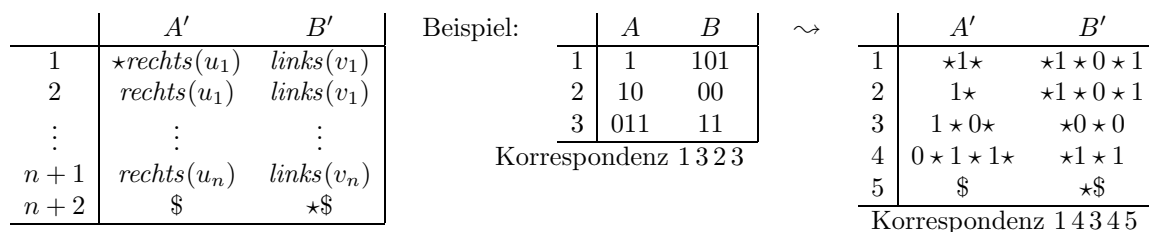


Abbildung 7.5: Zum Beweis von $MPCP \leq PCP$

Die beiden Funktionen links und rechts erfüllen die Worthomorphieeigenschaften $\text{links}(w_1 w_2) = \text{links}(w_1) \text{links}(w_2)$ und $\text{rechts}(w_1 w_2) = \text{rechts}(w_1) \text{rechts}(w_2)$, und außerdem gilt

$$w_1 = w_2 \quad \text{genau dann, wenn} \quad \star \text{rechts}(w_1) \$ = \text{links}(w_2) \star \$$$

direkt nach der Definition der beiden Funktionen. Sei nun die Indexfolge $1i_2i_3 \dots i_m$ eine Korrespondenz von A, B , d.h. es gilt $u_1 u_{i_2} \dots u_{i_m} = v_1 v_{i_2} \dots v_{i_m}$. Dann gilt nach dem eben gesagten auch

$$\star \text{rechts}(u_1 u_{i_2} \dots u_{i_m}) \$ = \text{links}(v_1 v_{i_2} \dots v_{i_m}) \star \$,$$

und durch mehrfaches Anwenden der Homorphieeigenschaft erhalten wir:

$$\begin{aligned} & \star \text{rechts}(u_1) \quad \text{rechts}(u_{i_2}) \quad \dots \quad \text{rechts}(u_{i_m}) \quad \$ \\ = & \text{links}(v_1) \quad \text{links}(v_{i_2}) \quad \dots \quad \text{links}(v_{i_m}) \quad \star \$, \end{aligned}$$

also eine Korrespondenz $1, i_2+1, \dots, i_m+1, n+2$ von A', B' . Sei umgekehrt i_1, \dots, i_m eine Korrespondenz von A', B' . Dann kann wegen der Gestalt der Worte (und der Tatsache, dass \star und $\$$ neue Zeichen sind) nur $i_1 = 1$ und $i_2, \dots, i_{m-1} \in \{2, \dots, n+1\}$ und $i_m = n+2$ gelten. Also ist die Indexfolge $1, i_2-1, \dots, i_{m-1}-1$ eine Lösung für A, B . Damit ist $MPCP \leq PCP$ gezeigt. \square 7.3.2

Es gilt sogar, dass $PCP_{\{0,1\}}$ (und damit – als leichtes Korollar – auch jedes andere Problem PCP_Σ mit $|\Sigma| \geq 2$) unentscheidbar ist:

Satz 7.3.3 $PCP_{\{0,1\}}$ IST UNENTSCHEIDBAR

$PCP \leq PCP_{\{0,1\}}$ (und als Folge davon: $PCP_{\{0,1\}}$ ist unentscheidbar).

Beweis: Sei durch A und B eine beliebige Eingabe einer PCP -Instanz gegeben und sei Σ ein Alphabet, so dass alle Wörter aus A und B in Σ liegen. Sei weiterhin $f: \Sigma^* \rightarrow \{0,1\}^*$ eine beliebige, (umkehrbar) effektive Codierung der Wörter von Σ^* in Wörter aus $\{0,1\}^*$, mit der zusätzlichen Eigenschaft, dass sich Wortketten eindeutig reproduzieren lassen. Ein Beispiel für eine solche Codierung ist folgende: sei $\Sigma = \{a_1, \dots, a_n\}$; dann setzen wir $f(a_j) = 01^j$ und $f(a_1 \dots a_m) = f(a_1) \dots f(a_m)$. An der Position der Nullen kann man eindeutig erkennen, dass dort die Codierung eines neuen Symbols beginnt, und an der Anzahl der Einsen nach einer solchen Null kann man erkennen, welches Symbol dort codiert ist. Es gilt offensichtlich, dass, wenn alle Wörter in A und B durch ihre Codierungen ersetzt werden, die Ursprungsinstanz eine Korrespondenz hat, genau dann, wenn die codierte Instanz eine Korrespondenz hat. \square 7.3.3

Für eine Codierung, wie sie im letzten Beweis verwendet wurde, benötigt man notwendigerweise ein Alphabet mit zwei oder mehr Zeichen. Eine solche Codierung kann nicht in ein Einalphabet angegeben werden. Zwar kann Σ^* immer umkehrbar effektiv auf $\{\}\^*$ abgebildet werden, aber Wortketten über Σ^* (d.h., Elemente von $(\Sigma^*)^*$) sind dann nicht mehr eindeutig Wortketten über $\{\}\^*$ zugeordnet. Der vorige Beweis lässt sich also auf einelementige Alphabete nicht übertragen, und in der Tat gilt stattdessen, dass $PCP_{\{\}}$ (und damit auch jedes andere PCP_Σ mit $|\Sigma| = 1$) entscheidbar ist:

Satz 7.3.4 $PCP_{\{\}}$ IST ENTSCHEIDBAR

Es gibt einen Entscheidungsalgorithmus für $PCP_{\{\}}$.

Beweis: Jedes Wort $|^n$ über $\{\}$ kann mit der natürlichen Zahl n identifiziert werden. Jede Eingabe A, B von $PCP_{\{\}}$ kann daher als Paar gleich langer Folgen von natürlichen Zahlen aufgefasst werden:

$$A = u_1, \dots, u_n \quad \text{und} \quad B = v_1, \dots, v_n$$

mit $n \in \mathbb{N}$ und $u_i, v_i \in \mathbb{N} \setminus \{0\}$ für $i = 1, \dots, n$. Die Instanz A, B ist lösbar, genau dann, wenn es eine Folge von Indizes (i_1, \dots, i_m) mit $i_j \in \{1, \dots, n\}$ für $j = 1, \dots, m$ gibt, so dass

$$\sum_{j=1}^m u_{i_j} = \sum_{j=1}^m v_{i_j}$$

gilt. Wir zeigen, dass es eine solche Korrespondenz gibt, genau dann, wenn

$$\underbrace{\exists j \in \{1, \dots, n\}: u_j = v_j}_{(a)} \quad \text{oder} \quad \underbrace{\exists k, l \in \{1, \dots, n\}: (u_k < v_k) \wedge (u_l > v_l)}_{(b)}.$$

(\Rightarrow): Falls weder (a) noch (b) gelten, haben entweder alle Paare (u_i, v_i) die Eigenschaft $u_i < v_i$, oder alle Paare (u_i, v_i) haben die Eigenschaft $u_i > v_i$. Durch u -Teile zusammengesetzte Wörter sind also stets kürzer oder stets länger als durch v -Teile zusammengesetzte Wörter. Damit ist A, B nicht lösbar.

(\Leftarrow): Falls (a), d.h. $u_j = v_j$, gilt, ist die triviale Indexfolge j eine Korrespondenz. Falls (b), d.h. $u_k < v_k$ und $u_l > v_l$, gilt, ist die Folge

$$\underbrace{k, \dots, k}_{(u_l - v_l)\text{-mal}}, \quad \underbrace{l, \dots, l}_{(v_k - u_k)\text{-mal}}$$

eine Korrespondenz, denn es gilt $u_k(u_l - v_l) + u_l(v_k - u_k) = v_k(u_l - v_l) + v_l(v_k - u_k)$.

Da die Eigenschaften (a) und (b) algorithmisch nachprüfbar sind, ist $PCP_{\{\}} \in \text{P}$ entscheidbar. \square 7.3.4

7.3.3 Unentscheidbarkeitsresultate bei kontextfreien Grammatiken

Wir beweisen die Unentscheidbarkeit des Schnittproblems, des Äquivalenzproblems und des Inklusionsproblems für kontextfreie Grammatiken (siehe Satz 5.6.2 und Satz 5.6.3) durch Reduktion vom Postschen Korrespondenzproblem.

Schnittproblem:

Gegeben seien zwei kontextfreie Grammatiken G_1 und G_2 . Die Frage lautet: Gilt $L(G_1) \cap L(G_2) = \emptyset$? Wir zeigen, dass das Postsche Korrespondenzproblem auf das Schnittproblem reduzierbar ist:

$$PCP \leq \text{Schnittproblem.}$$

Daraus folgt die Unentscheidbarkeit des Schnittproblems.

Gegeben sei eine beliebige Eingabe $A = u_1, \dots, u_n$ und $B = v_1, \dots, v_n$ des PCP mit $u_i, v_i \in \Sigma^*$ für ein Alphabet Σ . Wir geben einen Algorithmus an, der für eine solche Eingabe zwei kontextfreie Grammatiken G_1 und G_2 konstruiert, so dass folgendes gilt:

$$A, B \text{ besitzen eine Korrespondenz} \quad \Leftrightarrow \quad L(G_1) \cap L(G_2) \neq \emptyset. \quad (7.4)$$

Die Idee der Konstruktion ist, dass G_1 alle Wörter erzeugt, die durch Aneinanderlegen der u_i entstehen können, G_2 dagegen alle Wörter, die durch Aneinanderlegen der v_i entstehen können. Damit die gewünschte Beziehung (7.4) gilt, müssen G_1 und G_2 auch die Indizes i der „gelegten“ u_i und v_i festhalten. Dazu benutzen wir n neue Symbole $a_1, \dots, a_n \notin \Sigma$ und wählen für G_1 und G_2 als Menge der Terminalsymbole

$$\Sigma' = \Sigma \cup \{a_1, \dots, a_n\}.$$

Wir setzen dann $G_i = (\{S_i\}, \Sigma', P_i, S_i)$ (für $i = 1, 2$). Dabei ist P_1 durch folgende Produktionen gegeben:

$$S_1 \rightarrow a_1 u_1 \mid a_1 S_1 u_1 \mid \dots \mid a_n u_n \mid a_n S_1 u_n,$$

während P_2 durch folgende Produktionen gegeben ist:

$$S_2 \rightarrow a_1 v_1 \mid a_1 S_2 v_1 \mid \dots \mid a_n v_n \mid a_n S_2 v_n.$$

Offenbar gilt

$$L(G_1) = \{a_{i_m} \dots a_{i_1} u_{i_1} \dots u_{i_m} \mid m \geq 1 \text{ und } i_1, \dots, i_m \in \{1, \dots, n\}\}$$

und $L(G_2) = \{a_{i_m} \dots a_{i_1} v_{i_1} \dots v_{i_m} \mid m \geq 1 \text{ und } i_1, \dots, i_m \in \{1, \dots, n\}\}.$

Daraus folgt:

$$A, B \text{ besitzen die Korrespondenz } (i_1, \dots, i_m)$$

$$\Leftrightarrow$$

$$a_{i_m} \dots a_{i_1} u_{i_1} \dots u_{i_m} = a_{i_m} \dots a_{i_1} v_{i_1} \dots v_{i_m} \in L(G_1) \cap L(G_2).$$

Also gilt (7.4) und damit die Unentscheidbarkeit des Schnittproblems. Wir haben sogar ein stärkeres Resultat bewiesen: die Unentscheidbarkeit des Schnittproblems für *deterministisch* kontextfreie Sprachen. Wie man leicht nachprüft, sind nämlich die Sprachen $L(G_1)$ und $L(G_2)$ deterministisch.

Äquivalenzproblem:

Gegeben seien zwei kontextfreie Grammatiken G_1 und G_2 . Die Frage lautet: Gilt $L(G_1) = L(G_2)$? Wir zeigen die Unentscheidbarkeit dieses Problems durch folgende Reduktion:

Schnittproblem für deterministisch kontextfreie Sprachen \leq Äquivalenzproblem.

Seien also zwei deterministische Kellerautomaten K_1 und K_2 gegeben. Wir zeigen, dass man daraus algorithmisch zwei (nicht notwendig deterministisch) kontextfreie Grammatiken G_1 und G_2 konstruieren kann, so dass folgendes gilt:

$$L(K_1) \cap L(K_2) = \emptyset \quad \Leftrightarrow \quad L(G_1) = L(G_2).$$

Wir nutzen dabei aus, dass zu K_2 effektiv der Komplement-Kellerautomat $\overline{K_2}$ mit $L(\overline{K_2}) = \overline{L(K_2)}$ konstruiert werden kann (vgl. Abschnitt 5.5). Aus K_1 und $\overline{K_2}$ können dann effektiv kontextfreie Grammatiken G_1 und G_2 mit

$$L(G_1) = L(K_1) \cup L(\overline{K_2}) \quad \text{und} \quad L(G_2) = L(\overline{K_2})$$

konstruiert werden. Damit gilt

$$L(K_1) \cap L(K_2) = \emptyset \quad \Leftrightarrow \quad L(K_1) \subseteq \overline{L(K_2)}$$

$$\Leftrightarrow \quad L(K_1) \subseteq L(\overline{K_2})$$

$$\Leftrightarrow \quad L(K_1) \cup L(\overline{K_2}) = L(\overline{K_2})$$

$$\Leftrightarrow \quad L(G_1) = L(G_2),$$

wie gewünscht.

Inklusionsproblem:

Gegeben seien zwei kontextfreie Grammatiken G_1 und G_2 . Die Frage lautet: Gilt $L(G_1) \subseteq L(G_2)$? Offensichtlich kann das Äquivalenzproblem auf das Inklusionsproblem reduziert werden. Daher ist auch das Inklusionsproblem unentscheidbar. □ 5.6.2

Ebenfalls durch Reduktion vom Postschen Korrespondenzproblem zeigt man, dass das Mehrdeutigkeitsproblem für kontextfreie Grammatiken unentscheidbar ist:

$$PCP \leq \text{Mehrdeutigkeitsproblem.} \tag{7.5}$$

Gegeben sei eine Eingabe $A = u_1, \dots, u_n$ und $B = v_1, \dots, v_n$ des PCP mit $u_i, v_i \in \Sigma^*$ für ein Alphabet Σ . Wir konstruieren zunächst die kontextfreien Grammatiken $G_i = (\{S_i\}, \Sigma', P_i, S_i)$ ($i = 1, 2$), wie für die Reduktion des PCP auf das Schnittproblem. Anschließend konstruieren wir daraus die kontextfreie Grammatik $G = (\{S, S_1, S_2\}, \Sigma', P, S)$ mit

$$P = P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}.$$

Da G_1 und G_2 eindeutig sind, liegt die einzig mögliche Mehrdeutigkeit von G bei der Herstellung eines Ableitungsbaums von G zu einem Wort $w \in \Sigma'^*$ in der Benutzung der Produktionen $S \rightarrow S_1$ bzw. $S \rightarrow S_2$. Daher ergibt sich folgendes:

$$\begin{aligned} G \text{ ist mehrdeutig} &\Leftrightarrow L(G_1) \cap L(G_2) \neq \emptyset \\ &\Leftrightarrow A, B \text{ besitzen eine Korrespondenz.} \end{aligned}$$

Damit ist die Reduktion (7.5) gezeigt, und es folgt die Unentscheidbarkeit des Mehrdeutigkeitsproblems.

□ 5.6.3

7.4 Zusammenfassende Bemerkungen zur Chomsky-Hierarchie

In diesem Abschnitt geben wir einen Überblick über die Chomsky-Hierarchie. Fast alle Aspekte haben wir schon früher einzeln betrachtet, allerdings verstreut über das ganze Skriptum. Wir gehen in drei Schritten vor. Ein erster Satz beschreibt eine Reihe von äquivalenten Begriffen. Dadurch werden die Stufen der Chomsky-Hierarchie separat erläutert und charakterisiert. Ein zweiter Satz beschreibt die Chomsky-Hierarchie mengentheoretisch. Damit wird die Hierarchie aufgebaut. Ein dritter Satz hat schließlich die Striktheit der Chomsky-Hierarchie zum Inhalt. Dadurch wird klar, dass diese Hierarchie relativ groß und nicht-trivial ist. Im Folgenden sei $L \subseteq \Sigma^*$ eine Sprache.

Satz 7.4.1 DIE CHOMSKY-HIERARCHIE (TEIL 1: ÄQUIVALENTE EIGENSCHAFTEN)

Die folgenden Aussagen sind jeweils äquivalent:

- (Chomsky-3:) L ist regulär / von einem NFA akzeptierbar / von einem DFA akzeptierbar / rechts-linear / linkslinear.
- (Chomsky-2:) L ist kontextfrei / von einem PDA akzeptierbar.
- (Chomsky-1:) L ist kontextsensitiv / monoton / von einem LBA akzeptierbar.
- L ist entscheidbar / \bar{L} ist entscheidbar / L und \bar{L} sind Chomsky-0.
- (Chomsky-0:) L ist von einer Grammatik erzeugbar / von einer NTM akzeptierbar (d.h., T.a.) / von einer DTM akzeptierbar / semi-entscheidbar / rekursiv aufzählbar / durch ein WHILE-Programm akzeptierbar.

Beweis: Alle Aussagen wurden bereits im Detail untersucht, außer dass eine monotone Sprache auch kontextsensitiv ist. Sei also G eine monotone Grammatik, die L erzeugt. Wir konstruieren eine kontextsensitive Grammatik, die L erzeugt, indem jede monotone Produktion durch eine Reihe kontextsensitiver Produktionen ersetzt wird. Beispielsweise wird

$$B_1B_2B_3 \rightarrow C_1C_2C_3C_4$$

ersetzt durch:

$$\begin{aligned} B_1B_2B_3 &\rightarrow D_1B_2B_3 \\ D_1B_2B_3 &\rightarrow D_1D_2B_3 \\ C_1D_2B_3 &\rightarrow C_1D_2D_3 \\ D_1D_2B_3 &\rightarrow C_1D_2B_3 \\ C_1D_2D_3 &\rightarrow C_1C_2D_3 \\ C_1C_2D_3 &\rightarrow C_1C_2C_3C_4, \end{aligned}$$

wobei die D_1 , D_2 und D_3 neue Nichtterminalsymbole sind. Die monotone Ersetzung wird *quasi* stückweise von links nach rechts vorgenommen. ☒ 7.4.1

Satz 7.4.2 DIE CHOMSKY-HIERARCHIE (TEIL 2: ENTHALTENSEINBEZIEHUNGEN)

- Wenn L regulär (Chomsky-3) ist, dann ist L auch deterministisch kontextfrei, d.h., von einem DPDA akzeptierbar.
- Wenn L deterministisch kontextfrei ist, dann ist L auch kontextfrei (Chomsky-2).
- Wenn L kontextfrei (Chomsky-2) ist, dann ist L auch kontextsensitiv (Chomsky-1).
- Wenn L kontextsensitiv (Chomsky-1) ist, dann ist L auch entscheidbar.
- Wenn L entscheidbar ist, dann ist L auch Turing-akzeptierbar (Chomsky-0).

Beweis: Auch hier wurde bereits alles explizit untersucht, außer dass aus der Kontextsensitivität die Entscheidbarkeit folgt. Dies geht allerdings sofort unter Zuhilfenahme des Algorithmus für das (im kontextsensitiven Fall entscheidbare) Wortproblem. ☒ 7.4.2

Satz 7.4.3 DIE CHOMSKY-HIERARCHIE (TEIL 3: STRIKTHEIT DER HIERARCHIE)

- Es gibt eine Sprache, die deterministisch kontextfrei, aber nicht Chomsky-3 ist.
- Es gibt eine Sprache, die Chomsky-2, aber nicht deterministisch kontextfrei ist.
- Es gibt eine Sprache, die Chomsky-1, aber nicht kontextfrei ist.
- Es gibt eine Sprache, die entscheidbar, aber nicht Chomsky-1 ist.
- Es gibt eine Sprache, die Turing-akzeptierbar, aber nicht entscheidbar ist.
- Es gibt eine Sprache, die nicht Turing-akzeptierbar ist.

Beweis: $\{a^n b^n \mid n \in \mathbb{N}\}$ ist deterministisch kontextfrei, aber nicht Chomsky-3.

$\{ww^R \mid w \in \{a, b\}^+\}$ ist Chomsky-2, aber nicht deterministisch kontextfrei.

$\{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist Chomsky-1, aber nicht kontextfrei.

sHP ist Turing-akzeptierbar, aber nicht entscheidbar.

\overline{sHP} ist nicht Turing-akzeptierbar. Auch das Leerheitsproblem für Turingmaschinen (7.2) – siehe Abschnitt 7.2.5 – ist nicht semi-entscheidbar und damit auch nicht Turing-akzeptierbar.

Zur Vervollständigung des Beweises fehlt nur noch eine Sprache, die entscheidbar, aber nicht kontextsensitiv ist. Eine solche explizit anzugeben ist schwer! Einfacher ist ein indirektes Argument (durch eine Diagonalisierung), mit der Konsequenz, dass eine solche Sprache existieren muss.

Sei G eine kontextsensitive Grammatik und sei $bin(G)$ deren binäre Codierung. Wir dürfen annehmen, dass bin die kontextsensitiven Grammatiken injektiv codiert. Sei M_G eine Turingmaschine, die das Wortproblem für G löst. D.h., mit $bin(w)$ als Eingabe liefert die Maschine M_G ein „ja“, wenn $w \in L(G)$ und ein „nein“, wenn $w \notin L(G)$. Jetzt definieren wir eine Diagonalsprache:

$$L_0 = \{w \in \{0, 1\}^* \mid \exists \text{ kontextsensitive Grammatik } G \text{ mit } w = bin(G) \wedge M_G(w) \text{ hält mit „nein“}\}.$$

L_0 ist sicherlich entscheidbar, denn man kann w algorithmisch daraufhin untersuchen, ob es eine kontextsensitive Grammatik codiert, und wenn ja, die Maschine M_G mit Eingabe w simulieren und mit „ja“ bzw. „nein“ anhalten, wenn die Simulation mit „nein“ bzw. „ja“ endet. Wäre L kontextsensitiv, dann gäbe es eine kontextsensitive Grammatik G_0 mit $L_0 = L(G_0)$. Sei w_0 der Code $w_0 = bin(G_0)$. Es gilt dann

$$\begin{aligned} w_0 \in L_0 &\Rightarrow (\text{Definition von } L_0, bin \text{ ist injektiv}) \\ &\quad M_{G_0}(w_0) \text{ hält mit „nein“} \\ &\Rightarrow (M_{G_0} \text{ löst das Wortproblem für } G_0) \\ &\quad w_0 \notin L(G_0), \end{aligned}$$

im Widerspruch zu $L_0 = L(G_0)$. Damit ist L_0 eine Sprache, die entscheidbar, aber nicht Chomsky-1 ist. □ 7.4.3

Letzten Endes haben wir eine Einteilung in sechs Sprachklassen: Chomsky-0 bis Chomsky-3, und die deterministisch kontextfreien Sprachen, die echt zwischen den Chomsky-3 und den Chomsky-2-Sprachen liegen, sowie die entscheidbaren Sprachen, die echt zwischen den Chomsky-1 und den Chomsky-0-Sprachen liegen. Schematisch ist dies in Abbildung 7.6 dargestellt.

Die Chomsky-Hierarchie dient in vielen Fällen hauptsächlich als Anhaltspunkt für weitere Untersuchungen. Man interessiert sich, je nach Interessenlage, für weitere Sprachklassen innerhalb (oder auch quer zu) dieser Hierarchie. Als Beispiele wären zu nennen:

- Verfeinerungen der Chomsky-3-Sprachklasse, d.h., eine Hierarchie innerhalb des oberen linken Kastens in Abbildung 7.6: siehe hierzu auch die weiterführenden Module *Automatentheorie und Logik* und *Model-Checking*, die regelmäßig an der Carl von Ossietzky Universität Oldenburg angeboten werden.
- Klassen kontextfreier Sprachen; siehe auch die weiterführenden Module *Formale Sprachen* und *Compilerbau*.
- Klassen entscheidbarer Sprachen; siehe auch das weiterführende Modul *Komplexitätstheorie*, aber auch die Module *Kryptologie* und *Effiziente Algorithmen*.

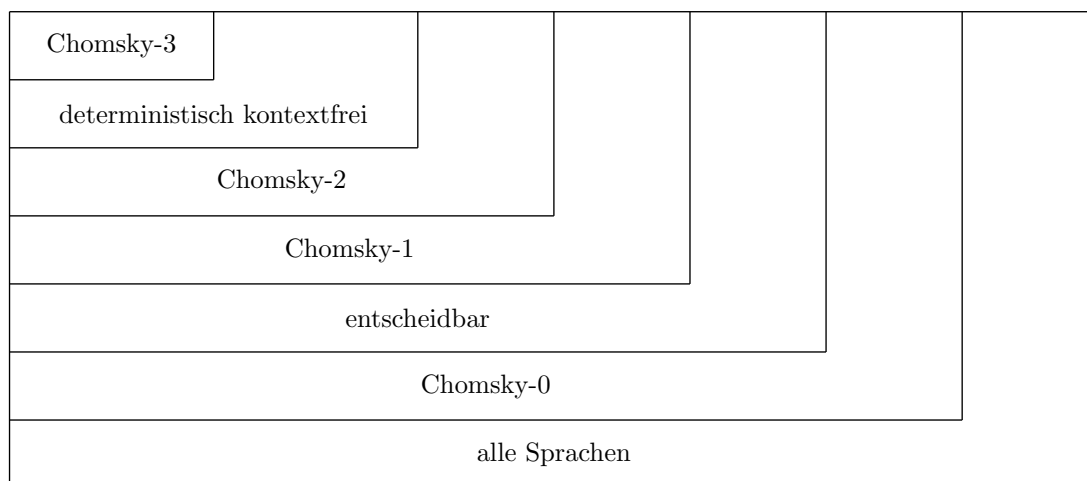


Abbildung 7.6: Die Chomsky-Hierarchie

- Untersuchungen zu „degrees of unsolvability“, d.h., Unterscheidungen innerhalb der Unentscheidbarkeit (das ist Hardcore-Theorie; hierzu wird in Oldenburg kein Modul angeboten).
- Und schließlich Sprachklassen, die sich nicht vollständig in die Chomsky-Hierarchie einfügen lassen; siehe hierzu auch das weiterführende Modul *Petrinetze*, das an der C.v.O. Universität regelmäßig angeboten wird.

7.5 Übungsaufgaben

1. Es seien $L_1, L_2 \subseteq \Sigma^*$. Beweisen oder widerlegen Sie die folgenden Aussagen:

- Wenn L_1 und L_2 entscheidbar sind, dann ist auch $L_1 \cup L_2$ entscheidbar.
- Wenn L_1 und L_2 entscheidbar sind, dann ist auch $L_1 \setminus L_2$ entscheidbar.
- Wenn L_1 endlich ist, dann ist L_1 entscheidbar.
- Wenn $\Sigma^* \setminus L_1$ endlich ist, dann ist L_1 entscheidbar.

Diese Aufgabe erweitert Satz 7.1.6.

- In welchem mathematischen Zusammenhang stehen die Begriffe *semi-entscheidbar* und *abzählbar*?
- Sei $\Sigma = \{0, 1\}$. In Abschnitt 7.2.4 wurde bewiesen, dass das Halteproblem für Turingmaschinen auf dem leeren Band HP_0 semi-entscheidbar und demnach auch rekursiv aufzählbar ist. Geben Sie also einen Algorithmus an, der die Elemente von HP_0 explizit aufzählt. Eine Angabe des Semientscheidungsalgorithmus reicht dafür nicht aus. Erinnern Sie sich an den Beweis des Satzes, welcher den Zusammenhang zwischen Semi-Entscheidbarkeit und Aufzählbarkeit beschreibt.

4. Gegeben sei ein Alphabet Σ .

a) Beweisen Sie ausführlich die Transitivität der \leq -Relation, d.h.

$$L_1 \leq L_2 \wedge L_2 \leq L_3 \Rightarrow L_1 \leq L_3 \quad (L_1, L_2, L_3 \subseteq \Sigma^*).$$

b) Gegeben seien eine entscheidbare Sprache $L_1 \subseteq \Sigma^*$ und eine endliche Sprache $L_2 \subseteq \Sigma^*$ mit $\emptyset \neq L_2 \neq \Sigma^*$. Zeigen Sie, dass dann immer $L_1 \leq L_2$ gilt.

5. Ist das folgende Problem entscheidbar?

Gegeben: Eine Turingmaschine M und eine Eingabe x .

Frage: Gibt es Zustände in der Zustandsmenge von M , die bei der Abarbeitung von x nicht benutzt werden?

Ein Hinweis: Für eventuelle Reduktionsbeweise sollten Sie „passende Varianten“ des Halteproblems verwenden.

6. Beweisen oder widerlegen Sie die Aussage

Wenn L_1 entscheidbar und L_2 semi-entscheidbar ist, dann ist $L_1 \setminus L_2$ semi-entscheidbar.

Ein Hinweis: Betrachten Sie das semi-entscheidbare *Selbstanwendungsproblem für Turingmaschinen* (spezielle Halteproblem) *sHP*.

7. In Abschnitt 6.8 wurden Sie aufgefordert, eine Turingmaschine zu konstruieren, welche die folgende Funktion berechnet:

$$f: \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & \begin{cases} 2 * n - 1 & , \text{ falls } n > 0 \\ 0 & , \text{ sonst.} \end{cases} \end{cases}$$

Ist für eine beliebige Turingmaschine entscheidbar, ob sie die obige Funktion berechnet, d.h., ist das folgende Problem entscheidbar?

Gegeben: Eine Turingmaschine M . Frage: Berechnet M die Funktion f ?

8. Finden Sie für die folgenden Korrespondenzprobleme über $\{a, b\}$ entweder eine Lösung oder beweisen Sie, dass es keine Lösung gibt.

a) $K_1 = ((ab, abb), (b, ba), (b, bb))$

b) $K_2 = ((ab, bb), (a, ab), (b, ba), (bbaaa, aaa))$.

c) $K_3 = ((ab, aba), (baa, aa), (aba, baa))$.

Kapitel 8

Komplexität

In Kapitel 7 haben wir uns mit der *prinzipiellen Effektivität* oder der *Berechenbarkeit* von Problemen beschäftigt, d.h., mit der Frage, ob vorgegebene Probleme überhaupt algorithmisch lösbar sind. Wenn eine algorithmische Lösung existierte, hat uns bislang die Frage nach ihrer *Effizienz* (d.h., ihrer Laufzeit bzw. ihrem Speicherplatzbedarf) nicht interessiert. Insbesondere hat uns *strukturelle Komplexität* interessiert, d.h., die Unterscheidung, welcher Typ von Maschine zur algorithmischen Lösung von Problemen benötigt wird.

Jetzt wollen wir uns mit der *Effizienz* oder *Berechnungskomplexität* befassen, d.h., mit der Frage, wieviel *Rechenzeit* und wieviel *Speicherplatz* man benötigt, um ein Problem algorithmisch zu lösen. Genauer werden Zeit und Platz in Abhängigkeit von der *Größe der Eingabe*, im Folgenden meist mit n bezeichnet, studiert. Man unterscheidet zwei Arbeitsrichtungen:

- a) Für konkrete Probleme möchte man *möglichst effiziente Algorithmen* angeben:
 - dies ist wichtig für praktische Problemlösungen
 - und theoretisch interessant als Nachweis von *oberen Schranken* für ein Problem: z.B. besagt ein existierender n^3 -Algorithmus, dass das Problem höchstens n^3 „schwer“ ist.
- b) Für ein Problem möchte man auch, sofern möglich, eine *untere Schranke* nachweisen, so dass *jeder* Algorithmus mindestens die Komplexität der unteren Schranke annimmt. Eine triviale untere Schranke für die Zeitkomplexität ist n , die Größe der Eingabe, da diese (außer in uninteressanten Trivialfällen) zumindest ganz gelesen werden muss. Oft ist es jedoch sehr schwer, nicht-triviale untere Schranken zu finden.

Aussagen über die Komplexität hängen vom benutzten algorithmischen Modell ab. Man kann beispielsweise Turingmaschinen, aber auch WHILE-Programme benutzen. In der Theorie (so auch in diesem Kapitel) betrachtet man meistens deterministische oder auch nichtdeterministische Turingmaschinen, manchmal auch Maschinen mit mehreren Bändern. Das ist keine Einschränkung, weil wir nur an der Gesamtklasse polynomiell berechenbarer Funktionen interessiert sein werden und – wie es sich herausstellt, ohne dass wir in diesem Kapitel genauer darauf eingehen – alle „vernünftigen“ Berechnungsmodelle bis auf einen polynomiellen Faktor ineinander umrechenbar sind.

8.1 Modifizierte Turingmaschinenmodelle

8.1.1 Mehrspurmaschinen

Eine Turingmaschine M über dem Alphabet Σ mit k -Spur-Band ist dadurch gekennzeichnet, dass ihr Bandalphabet ein k -Tupel von Elementen aus Γ ist. D.h., M ist ein Siebentupel

$$M = (Q, \Sigma, \Gamma^k, \sqcup, \delta, q_0, q_f)$$

mit einer Übergangsrelation

$$\delta \subseteq ((Q \setminus \{q_f\}) \times \Gamma^k) \times (Q \times \Gamma^k \times \{L, N, R\}).$$

Bei einer solchen Maschine steht der Lese/Schreibkopf stets auf einer „Spalte“ (einem Vektor der Größe k mit Einträgen aus Γ) der durch die k Spuren definierten, nach links und rechts unendlichen „Matrix“. In einem Schritt kann sich der LSK auf den Vektor links davon oder den Vektor rechts davon bewegen, oder er kann auf dem gerade gelesenen Vektor stehen bleiben. Der Begriff $L(M) \subseteq \Sigma^*$ ist ebenso direkt übertragbar wie der Begriff, dass M eine (partielle) Funktion in $\mathbb{N}^m \xrightarrow{p} \mathbb{N}$ berechnet. Bildlich stellt man sich vor, dass das „Ansetzen von M auf ein Wort w “ bedeutet, dass w auf der ersten Spur geschrieben wird, alle anderen Felder (insbesondere alle Felder auf allen anderen Spuren) das Blankzeichen tragen und dass die anfängliche Kopfposition der durch das Anfangszeichen von w bestimmte Vektor (oder irgend einer, wenn $w = \varepsilon$) ist.

Da Mehrspurmaschinen sich nur durch die Tatsache, dass Γ strukturiert ist, von normalen Turingmaschinen unterscheiden, unterscheidet sich die Klasse der von Mehrspurmaschinen akzeptierten Sprachen nicht von der Klasse der T.a. Sprachen. Genauer:

- Jede 1-Band-Turingmaschine ist auch eine k -Spurmaschine mit $k = 1$.
- Umgekehrt ist jede k -Spur-TM auch eine 1-Band-TM, indem Elemente $a \in \Sigma$ mit Vektoren $(a, \sqcup, \dots, \sqcup)$ der Länge k identifiziert werden.

8.1.2 Mehrbandmaschinen

Wir verallgemeinern die Klasse der betrachteten Maschinen noch weiter, indem wir zulassen, dass es k Bänder und für jedes Band einen eigenen Lese/Schreibkopf gibt. Es sei wieder Γ das Bandalphabet für alle k Bänder. Die Übergangsrelation hat in diesem Fall die Form

$$\delta \subseteq ((Q \setminus \{q_f\}) \times \Gamma^k) \times (Q \times \Gamma^k \times \{L, N, R\}^k).$$

Der wesentliche Unterschied zu Mehrspurmaschinen besteht darin, dass sich die k Lese/Schreibköpfe nunmehr in verschiedenen Richtungen bewegen können. Formal ist dies durch den Faktor $\{L, N, R\}^k$ (und nicht nur $\{L, N, R\}$ wie bei Mehrspurmaschinen) in der Definition von δ erfasst.

Eine Mehrbandmaschine wird auf ein Wort $w \in \Sigma^*$ genau wie eine Mehrspurmaschine angesetzt:

- Das Wort w steht auf dem ersten Band und der LSK des ersten Bandes steht auf dem Feld, das durch den linken Buchstaben von w definiert wird (bzw. auf einem beliebigen Feld, wenn $w = \varepsilon$). Alle anderen Felder des ersten Bandes tragen das Blankzeichen.

- Alle Bänder von 2 bis k tragen nur Blankzeichen und ihre Lese/Schreibköpfe stehen auf einem beliebigen Feld.

Die Begriffe der akzeptierten Sprache und der berechneten (partiellen) Funktion lassen sich entsprechend übertragen. Einen Übergang in einer Mehrbandmaschine stellen wir grafisch wie in Abbildung 8.1 dar. Hierbei sind a_1, \dots, a_k die gerade gelesenen Zeichen der k Bänder, b_1, \dots, b_k die neu geschriebenen Zeichen und m_1, \dots, m_k die Bewegungen der k Lese/Schreibköpfe.

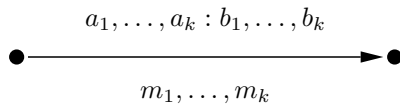


Abbildung 8.1: Übergang bei einer Mehrbandmaschine

Konfigurationen von Mehrbandmaschinen bestehen aus:

- dem gerade aktuellen Zustand q (anfänglich q_0 , in einer Endkonfiguration q_f);
- den k Bandinhalten;
- und den k Kopfpositionen.

8.1.3 Offline-Maschinen

Eine Mehrband-Turingmaschine heißt *Offline*, wenn das erste Band nur in Richtung von links nach rechts gelesen wird. Das erste Band heißt hier „Eingabeband“, alle anderen Bänder heißen „Arbeitsbänder“. In diesem Fall hat die Übergangsrelation die Form

$$\delta \subseteq ((Q \setminus \{q_f\}) \times \Gamma^k) \times (Q \times \Gamma^{k-1} \times (\{R\} \times \{L, N, R\}^{k-1})).$$

Wenn eine Offline-TM nur $k = 1$ Bänder hat, ist ihre Mächtigkeit auf die eines endlichen Automaten reduziert. Deshalb nimmt man bei Offline-TMs standardmäßig die Existenz mindestens eines Arbeitsbandes, d.h. $k \geq 2$, an.

Satz 8.1.1 BANDREDUKTION

Sei M eine k -Band-Turingmaschine ($k \geq 1$) oder eine Offline-TM mit k ($k \geq 2$) Bändern. Dann gibt es eine 1-Band-Turingmaschine M' mit $L(M') = L(M)$.

Beweis: (Skizze.)

Sei M eine Mehrbandmaschine mit k Bändern. Wir simulieren M durch eine Mehrspurmaschine mit $2 \cdot k$ Spuren. Dabei simulieren die Spuren mit ungeradem Index (1, 3, ..., $2k - 1$) genau die k Bänder von M . Die Spuren mit geradem Index (2, 4, ..., $2k$) enthalten genau ein nicht-Blankzeichen, z.B. *, das angibt, wo sich der Lese/Schreibkopf des „darüber liegenden“ Bandes (d.h., der Spur mit um 1 kleinerem Index) befindet.

Anfänglich (und immer nach einem vollendeten Simulationsschritt) befindet sich die Kopfposition der simulierenden Maschine auf dem Vektor, der durch ein erstes *-Zeichen (von links) bestimmt ist. Um einen Schritt von M zu simulieren,

- bewegt die simulierende Maschine ihren LSK solange nach rechts und speichert die Kopfpositionen und die dort gelesenen Zeichen, bis genau k Zeichen $*$ gelesen worden sind (hierdurch gewinnt sie die nötige Information, um M simulieren zu können),
- führt dann die k Teilschritte von M aus (Schreiben neuer Zeichen und evtl. Versetzen der $*$ -Zeichen), bis alle k Teilschritte ausgeführt sind,
- und sucht schließlich das erste (von links) mit $*$ markierte Feld auf und ist bereit zum nächsten Schritt.

Der Fall, dass M eine Offline-TM ist, kann analog behandelt werden.

☒ 8.1.1

8.2 Zeit- und Platzkomplexitätsbegriffe

Sei M eine DTM mit akzeptierter Sprache $L(M)$. Wir interessieren uns für die Längen der Berechnungen von M (Zeitkomplexität) und für den Bandverbrauch (Platz- oder Speicherkomplexität). Dazu zählen wir einfach die Anzahl der Schritte bzw. die Anzahl der besuchten Bandfelder während einer Rechnung, die ja eindeutig ist.

Für Wörter nicht in $L(M)$ ist die Anzahl der Schritte unendlich groß. Die Anzahl der besuchten Bandfelder mag endlich sein oder auch nicht. Für Wörter, die in $L(M)$ liegen, sind jedoch beide Größen wohldefinierte natürliche Zahlen. Diese Zahlen interessieren uns in erster Linie.

Außerdem machen wir eine (realistische) *worst-case*-Abstraktion. Es kann beobachtet werden, dass die Lauf- und Platzbedarfszeiten mit der Länge (Größe) n der Eingabe $w = a_1 \dots a_n$ im Wesentlichen wachsen. Wir interessieren uns daher nur für die Abhängigkeit der Komplexitätsgrößen von der *Länge* der Eingabe, nicht von der Eingabe selbst. Konkret soll der schlechteste Fall einer Eingabe der festen Länge n betrachtet werden, d.h., diejenige Laufzeit, die maximal mit gegebener Eingabelänge n erreichbar ist. Das ergibt eine obere Schranke für die Laufzeit und motiviert insgesamt die folgende Definition.

Definition 8.2.1 ZEIT- UND PLATZKOMPLEXITÄT EINER DTM

Seien $f: \mathbb{N} \rightarrow \mathbb{N}$ eine (totale) Funktion und sei M eine deterministische Mehrband-TM mit Eingabealphabet Σ .

- M heißt *f*-zeitbeschränkt (oder *f*(*n*)-zeitbeschränkt), falls für jedes Wort $w \in L(M)$ der Länge $|w| = n$ gilt: wenn M auf die Eingabe w angesetzt wird, hält die – eindeutige und wegen $w \in L(M)$ endliche – Berechnung in höchstens $f(n)$ Schritten an.
- M heißt *f*-platzbeschränkt (oder *f*(*n*)-platzbeschränkt), falls für jedes Wort $w \in L(M)$ der Länge $|w| = n$ gilt: wenn M auf die Eingabe w angesetzt wird, benutzt die Berechnung auf jedem Band höchstens $f(n)$ Felder. ☒ 8.2.1

Es gilt:

Lemma 8.2.2 DETERMINISTISCH ZEITBESCHRÄNKT \Rightarrow ENTSCHEIDBAR

Sei L eine Sprache, die von einer mit Hilfe einer Turing-berechenbaren Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$ zeitbeschränkten DTM akzeptierbar ist. Dann ist L entscheidbar.

Beweis: Wir konstruieren eine DTM mit folgender Wirkungsweise:

- a) Gegeben w , wird $n = |w|$ berechnet.
- b) Durch Benutzung einer DTM, die f berechnet, wird $f(n)$ auf ein Zusatzband geschrieben.
- c) Jetzt wird die Maschine M simuliert, die L akzeptiert, wobei die Zahl auf dem Zusatzband bei jedem Schritt um 1 vermindert wird.
- d) Ist die Zahl auf dem Zusatzband 0, wird die Rechnung abgebrochen. Hat M bis dahin einen Endzustand erreicht, wird w akzeptiert ($w \in L$), ansonsten verworfen ($w \notin L$).

Offensichtlich entscheidet diese Maschine L , denn wenn nach $f(n)$ Schritten in M kein akzeptierender (terminaler) Zustand erreicht wird, kann wegen der Zeitschranke auch später keiner mehr kommen.

□ 8.2.2

Im Verbund mit Satz 7.1.5 folgt auch, dass unter den Voraussetzungen des Lemmas die Sprache \bar{L} Turingakzeptierbar ist. Die Funktion f ist aber nicht notwendigerweise eine Zeitschranke für eine \bar{L} akzeptierende Maschine, weil im Schritt b) des Beweises, wo $f(n)$ ausgerechnet wird, viel mehr Zeit vergehen kann, als durch die Schranke $f(n)$ bestimmt ist.

Die Definition 8.2.1 ist leicht auf DTMs zu übertragen, die partielle Funktionen berechnen, anstatt Sprachen zu akzeptieren (Abschnitt 6.6). Zum Beispiel heißt eine Maschine M , die eine partielle, m -stellige Funktion g berechnet, *f-zeitbeschränkt* (mit $f: \mathbb{N} \rightarrow \mathbb{N}$), wenn für die Binärdarstellung $w = \text{bin}(x_1, \dots, x_m)$ eines m -Tupels als Eingabe gilt: falls $|w| = n$ und g ist auf (x_1, \dots, x_m) definiert, dann liefert M in höchstens $f(n)$ Schritten das Ergebnis $g(x_1, \dots, x_m)$.

Nun verallgemeinern wir diese Definition auf nichtdeterministische Turingmaschinen. Wenn L durch eine NTM M akzeptiert wird, dann sind die Rechnungen nicht eindeutig. Trotzdem lassen sich die Zeit- und Bandverbrauchsbegriffe auf Grundlage der folgenden Idee übertragen. Bei möglichen Verzweigungspunkten einer NTM, d.h., an den Stellen, an denen die Maschine eine Auswahl aus k Möglichkeiten hat, stellen wir uns vor, dass k gleiche (disjunkte) Kopien der Maschine hergestellt werden und dass jede dieser Kopien mit einer der k Möglichkeiten weiter rechnet. Das geht eigentlich nur mit einer gedanklichen Abstraktion: dass es so viele Prozessoren gibt, wie man gerade braucht (also potenziell unendlich viele). Sobald eine dieser möglicherweise sehr vielen Maschinen einen Endzustand erreicht, lautet die Antwort „die Eingabe ist akzeptiert“, und die anderen Zweige könnten im Prinzip ihre Rechnungen abbrechen. Deshalb wird der Zeitverbrauch als das *Minimum* aller möglichen Schrittlängen definiert, nach dem Motto: „die schnellste Maschine (-nkopie) gewinnt“.

Definition 8.2.3 ZEIT- UND PLATZKOMPLEXITÄT EINER NTM

Seien $f: \mathbb{N} \rightarrow \mathbb{N}$ eine (totale) Funktion und sei M eine (nicht notwendigerweise deterministische) Mehrband-TM mit Eingabealphabet Σ .

- (i) M heißt *f-zeitbeschränkt* (oder *f(n)-zeitbeschränkt*), falls für jedes Wort $w \in L(M)$ der Länge n gilt: wenn M auf die Eingabe w angesetzt wird, hält die *kürzestmögliche akzeptierende* Berechnung – und es gibt mindestens eine, wegen $w \in L(M)$ – in höchstens $f(n)$ Schritten an.
- (ii) M heißt *f-platzbeschränkt* (oder *f(n)-platzbeschränkt*), falls für jedes Wort $w \in L(M)$ der Länge n gilt: wenn M auf die Eingabe w angesetzt wird, benutzt die *kürzestmögliche akzeptierende* Berechnung auf jedem Band höchstens $f(n)$ Felder.

□ 8.2.3

Auch jetzt gilt:

Lemma 8.2.4 NICHTDETERMINISTISCH ZEITBESCHRÄNKT \Rightarrow ENTSCHIEDBAR

Sei L eine Sprache, die von einer mit Hilfe einer Turing-berechenbaren Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$ zeitbeschränkten NTM akzeptierbar ist. Dann ist L entscheidbar.

Beweis: Der Beweis von Lemma 8.2.2 kann im Wesentlichen übernommen werden. Die simulierende Maschine muss jetzt im Schritt c) nicht nur eine einzige Rechnung simulieren, sondern einen Berechnungsbaum, allerdings nur bis zur Tiefe $f(n)$. Es handelt sich also um einen endlichen Baum, und die Simulation bricht deswegen ab. Günstigerweise kann man in Schritt b) des Beweises gleich zwei neue Arbeitsbänder einrichten, eins zum Merken der Zahl $f(n)$ und eins zum Steuern der Simulation. Dazu können (wie in Abschnitt 6.4, wo eine Konstruktion $\text{NTM} \rightsquigarrow \text{DTM}$ beschrieben wurde) Wörter über $\{1, \dots, r\}$ verwendet werden, wobei r der Grad des Nichtdeterminismus ist. Hier brauchen wir allerdings nur Wörter der Länge $\leq f(n)$ zu betrachten, also eine endliche Anzahl. Alternativ kann man sich vorstellen, dass man die gegebene NTM parallel simuliert und jede der simulierenden Maschinen höchstens $f(n)$ Schritte machen lässt. □ 8.2.4

Es ist im Allgemeinen sehr schwierig, die tatsächliche Zeitkomplexität einer Maschine herauszufinden. Deshalb ist man an Abstraktionen und an asymptotischem Verhalten interessiert. Die Definition ist zu diesem Zweck flexibel gefasst. Zum Beispiel ist mit f auch jede „größere“ Funktion (etwa die Funktion g mit $g(n) = f(n) + 1$) eine Zeitschranke für M .

Auf einer Mehrbandmaschine wird für den Platzverbrauch nur die maximale Ausdehnung der Bänder nach links und rechts gezählt. Die Bandfelder der k Bänder werden nicht addiert, weil das nur einen konstanten (und asymptotisch vernachlässigbaren) Faktor beitragen würde.

Wir fassen jetzt Probleme, also Sprachen, die mit gleicher Komplexität akzeptiert werden können, zu *Komplexitätsklassen* zusammen.

Definition 8.2.5 KOMPLEXITÄTSKLASSEN (ALLGEMEIN)

Sei $f: \mathbb{N} \rightarrow \mathbb{N}$.

$\text{DTIME}(f) = \{L \mid \text{es gibt eine deterministische Mehrband-TM der Zeitkomplexität } f, \text{ die } L \text{ akzeptiert}\}$

$\text{NTIME}(f) = \{L \mid \text{es gibt eine nichtdeterministische Mehrband-TM der Zeitkomplexität } f, \text{ die } L \text{ akzeptiert}\}$

$\text{DSPACE}(f) = \{L \mid \text{es gibt eine deterministische Mehrband-TM der Platzkomplexität } f, \text{ die } L \text{ akzeptiert}\}$

$\text{NSPACE}(f) = \{L \mid \text{es gibt eine nichtdeterministische Mehrband-TM der Platzkomplexität } f, \text{ die } L \text{ akzeptiert}\}$ □ 8.2.5

Da eine TM in jedem Rechenschritt höchstens ein neues Feld auf ihren Bändern besuchen kann, folgt:

$$\begin{array}{ccccc} \text{DTIME}(f) & \subseteq & \text{DSPACE}(f) & \subseteq & \text{NSPACE}(f) \\ & \subseteq & & \subseteq & \\ & & \text{NTIME}(f) & & \end{array}$$

Beispiel:

Wir betrachten $L_{\text{doppelt}} = \{w \mid w = uu \wedge u \in \{a, b\}^*\}$ und wollen eine möglichst effiziente Mehrband-TM konstruieren, die L_{doppelt} akzeptiert. Zunächst gehen wir deterministisch vor.

Lösungsidee: Zu einem gegebenem Wort $w \in \{a, b\}^*$ stellen wir zuerst die Mitte von w fest und vergleichen dann die beiden Hälften. Zum Feststellen der Mitte wird ein 2. Band benutzt. Wir fassen also eine deterministische 2-Band TM ins Auge und unterteilen die Arbeit der Maschine in mehrere Phasen:

- **Phase 1:** Hier gehen wir das Eingabewort auf dem oberen Band von links nach rechts einmal ab und bewegen den unteren Kopf genau halb so schnell. Ein Test, ob n ungerade ist, kann eingebaut werden; falls ja, wird w verworfen. Falls n gerade ist, steht der obere Kopf auf dem ersten Blank rechts von w und der zweite Kopf auf dem 1. Buchstaben der zweiten Hälfte von w .
Der Zeitverbrauch dieser Phase ist n , der Platzverbrauch $n + 1$. (Das gilt auch im Spezialfall $n = 0$.)
- **Phase 2:** Jetzt wissen wir schon, dass n gerade ist. Der obere Kopf bewegt sich ein Feld nach links, desgleichen der untere Kopf. Jetzt steht der obere Kopf auf dem letzten Buchstaben von w (wenn $w \neq \varepsilon$), und der untere Kopf steht auf dem letzten Buchstaben der ersten Hälfte von w .
Der Zeitverbrauch dieser Phase ist 1. Der Platzverbrauch der Phasen 1 und 2 ist $n + 1$, falls $w \neq \varepsilon$, und $n + 2$, also 2, falls $w = \varepsilon$.
- **Phase 3:** Jetzt wird die zweite Hälfte von w rückwärts vom 1. Band auf das zweite Band kopiert. Danach steht der obere Kopf auf dem letzten Zeichen der 1. Hälfte von w , der zweite Kopf auf dem Feld direkt links von w .
Der Zeitverbrauch dieser Phase ist $\frac{n}{2}$. Der Platzverbrauch ist bislang insgesamt $n + 2$.
- **Phase 4:** Jetzt bewegen wir den Kopf des oberen Bandes auf das Feld links neben den Anfang von w . Danach stehen beide Köpfe direkt links neben dem Anfangsfeld.
Der Zeitverbrauch dieser Phase ist $\frac{n}{2}$. Der Platzverbrauch ist bleibt $n + 2$.
- **Phase 5:** Jetzt vergleichen wir die $\frac{n}{2}$ Buchstaben rechts neben den beiden Köpfen auf Gleichheit. Falls ja, wird das Wort akzeptiert, falls nein, wird das Wort verworfen.
Der Zeitverbrauch dieser Phase ist $\frac{n}{2} + 1$. Der Platzverbrauch bleibt insgesamt $n + 2$.

Insgesamt ergibt sich die Zeitkomplexität aus einer Aufsummierung der Verbräuche der 5 Phasen:

$$(n + 1) + (1) + \left(\frac{n}{2}\right) + \left(\frac{n}{2}\right) + \left(\frac{n}{2} + 1\right) = \frac{5n}{2} + 3.$$

Die Platzkomplexität ist $n + 2$. Also gilt:

$$\begin{aligned} L_{\text{doppelt}} &\in \text{DTIME}\left(\frac{5n}{2} + 3\right), \\ L_{\text{doppelt}} &\in \text{DSPACE}(n + 2). \end{aligned}$$

Nichtdeterministisch kann man folgendermaßen vorgehen:

- **Phase 1:** Solange kein Blank gelesen wird, wird zeichenweise vom Band 1 auf das Band 2 kopiert. Dieser Prozess wird nichtdeterministisch abgebrochen.
- **Phase 2:** Auf Band 2 wird an den Anfang zurückgekehrt.
- **Phase 3:** Nun wird zeichenweise verglichen, ob ab der in Phase 1 erreichten Position auf dem ersten Band das Gleiche steht wie auf dem zweiten Band. Falls dies zutrifft und beide Lese/Schreibköpfe gleichzeitig am Ende auf ein Blank stoßen, dann wird akzeptiert.

Dieses Vorgehen benötigt im Erfolgsfall

$$\frac{n}{2} + \left(\frac{n}{2} + 1\right) + \left(\frac{n}{2} + 1\right) = \frac{3n}{2} + 2 \text{ Schritte,}$$

also gilt: $L_{\text{doppelt}} \in \text{NTIME}\left(\frac{3n}{2} + 2\right)$.

Ende des Beispiels.

8.3 O-Notationen

Der genaue Zeit- und Bandverbrauch ist in der Regel schwer zu bestimmen und nicht sehr aussagekräftig. Wichtiger ist es, zu wissen, wie stark diese Verbräuche mit der Eingabelänge wachsen. Um solche asymptotischen Abschätzungen vorzunehmen, definieren wir:

Definition 8.3.1 O-NOTATIONEN

Sei $f: \mathbb{N} \rightarrow \mathbb{N}$.

$O(f)$ ist die Menge der Funktionen g , für die ein $r > 0$, $r \in \mathbb{Q}$ so existiert, dass für fast alle (d.h. alle bis auf endlich viele) $n \in \mathbb{N}$ die Ungleichung $g(n) < r \cdot f(n)$ gilt.

$o(f)$ ist die Menge der Funktionen g , für die für alle $r > 0$ und für fast alle $n \in \mathbb{N}$ gilt: $g(n) < r \cdot f(n)$.

☐ 8.3.1

Beispiel: Die Funktionen f_1 und f_2 von \mathbb{N} nach \mathbb{N} mit $f_1(n) = \frac{5n}{2} + 3$ und $f_2(n) = n + 2$ sind beide in $O(n)$. Wir sagen deshalb auch: die Sprache L_{doppelt} aus Abschnitt 8.2 kann mit Zeit- und Platzkomplexität $O(n)$ akzeptiert werden, bzw. L „kann mit linearer Zeit- und Platzkomplexität“ akzeptiert werden.

Beispiel: „Liegen die angegebenen Funktionen (Spalten) in den angegebenen Klassen (Zeilen)?“

	n^3	$n^2 + 1$	$n^2 - 1$	n
$O(n^2)$	Nein	Ja ($r = 3$)	Ja ($r = 1$)	Ja ($r = 1$)
$o(n^2)$	Nein ($r = 1$)	Nein ($r = 1$)	Nein ($r = 0.5$)	Ja

Etwas salopp kann man sich die beiden Symbole etwa so veranschaulichen:

$$\begin{aligned} g \in O(f) & \text{ „wächst maximal so schnell wie“ } f \\ g \in o(f) & \text{ „wächst langsamer als“ } f. \end{aligned}$$

8.4 Die Klassen P, NP und PSPACE

Für praktisch brauchbare Algorithmen sollte die Komplexitätsfunktion $f(n)$ ein *Polynom* $p(n)$ k -ten Grades sein, also von der Form

$$p(n) = a_k n^k + \dots + a_1 n + a_0$$

mit $a_i \in \mathbb{N}$ für $i = 0, 1, \dots, k$, $k \in \mathbb{N}$ und $a_k \neq 0$.

Definition 8.4.1 (COBHAM, 1964)

$$\begin{aligned}
 P &= \bigcup_{p \text{ Polynom}} \text{DTIME}(p) \\
 NP &= \bigcup_{p \text{ Polynom}} \text{NTIME}(p) \\
 PSPACE &= \bigcup_{p \text{ Polynom}} \text{DSPACE}(p) \\
 NPSpace &= \bigcup_{p \text{ Polynom}} \text{NSPACE}(p)
 \end{aligned}$$

☒ 8.4.1

Satz 8.4.2 (SAVITCH, 1970; OHNE BEWEIS)

Für alle Polynome p in n gilt $\text{NSPACE}(p(n)) = \text{DSPACE}(p^2(n))$.

Damit gilt auch $\text{NPSpace} = \text{PSPACE}$. Ferner gilt:

$$P \subseteq NP \subseteq \text{PSPACE}.$$

Offenes Problem der Informatik: Sind diese Inklusionen echt oder gilt die Gleichheit?

Die Klassen P und NP sind von großer Bedeutung, denn sie markieren den Übergang von praktisch durchführbarer Berechenbarkeit bzw. Entscheidbarkeit zu mehr oder weniger nur theoretisch interessanter Berechenbarkeit bzw. Entscheidbarkeit. Dieser Übergang sei durch das Diagramm in Abbildung 8.2 veranschaulicht.

alle Probleme bzw. Sprachen

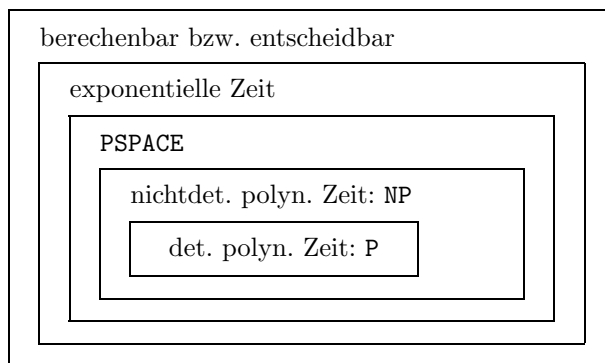


Abbildung 8.2: Klasseneinteilung der entscheidbaren Sprachen

Praktisch lösbar, d.h. praktisch berechenbar bzw. entscheidbar sind die Probleme in der Klasse P , deren Lösungsprinzip durch folgenden Merksatz charakterisiert werden kann:

P : *Konstruiere* in deterministischer Weise und polynomieller Zeit eine richtige Lösung.

Dabei sollten die zeitbeschränkenden Polynome einen möglichst kleinen Grad wie n , n^2 oder n^3 haben. Praktisch unlösbar sind dagegen alle Probleme, für die der Zeitaufwand nachweislich exponentiell mit der Größe n der Eingabe wächst, jedenfalls wenn beliebige n zugelassen werden. Zwischen diesen beiden Extremen liegt eine große Klasse von praktisch wichtigen Problemen, für die im Augenblick nur exponentielle deterministische Algorithmen bekannt sind, die aber durch *nichtdeterministische* Algorithmen in polynomieller Zeit gelöst werden können. Dieses ist die Klasse NP, deren Lösungsprinzip im Vergleich zu P wie folgt formuliert werden kann:

NP: *Rate nichtdeterministisch* einen Lösungsvorschlag und *verifiziere* bzw. *prüfe* dann in deterministischer Weise und polynomieller Zeit, ob dieser Vorschlag eine richtige Lösung ist.

Auch diese Probleme sind bis heute *in voller Allgemeinheit* praktisch unlösbar. In der Praxis behilft man sich mit sogenannten *Heuristiken*, die den nichtdeterministischen Suchraum der möglichen Lösungsvorschläge stark einschränken. Durch diese Heuristiken versucht man, eine „Ideallösung“ zu approximieren.

8.5 Beispiele für Probleme aus der Klasse NP

(1) Problem des Hamiltonschen Pfades:

Gegeben: Ein endlicher Graph mit n Knoten.

Frage: Gibt es einen Hamiltonschen Pfad in dem Graphen, d.h. einen Kantenzug, der jeden Knoten genau einmal trifft?

Betrachten wir z.B. den Graphen in Abbildung 8.3. Dann ist der Kantenzug $K1-K2-K3-K4$ ein Hamiltonscher Pfad. Es ist leicht zu sehen, dass das Problem des Hamiltonschen Pfades in NP liegt: Man rät zunächst einen Pfad und prüft dann, ob jeder Knoten genau einmal getroffen wird. Da es $O(n!)$ in Frage kommende Pfade im Graphen gibt, wäre dieses Verfahren nur mit exponentiellem Aufwand in deterministischer Weise anwendbar.

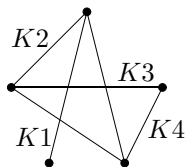


Abbildung 8.3: Ein Graph

Das umgekehrte Problem:

Gegeben: Ein endlicher Graph mit n Knoten.

Frage: Gibt es *keinen* Hamiltonschen Pfad in dem Graphen, d.h. einen Kantenzug, der jeden Knoten genau einmal trifft?

ist übrigens ein Beispiel eines Problems, das sich mit der „Rate-und prüfe“-Methode nicht (direkt) lösen lässt. Das bedeutet, dass die Mitgliedschaft dieses Problems in NP nicht (direkt) zu sehen ist; tatsächlich ist unbekannt, ob dieses umgekehrte Problem in NP liegt oder nicht.

(2) Problem des Handlungsreisenden

Gegeben: Ein endlicher Graph mit n Knoten und natürlichzahligen Längenangaben für jede Kante, sowie eine Zahl $k \in \mathbb{N}$.

Frage: Gibt es für den Handlungsreisenden eine Rundreise der Länge $\leq k$ oder mathematischer: gibt es einen Kreis, d.h. einen geschlossenen Kantenzug der Länge $\leq k$, der jeden Knoten mindestens einmal trifft?

Auch dieses Problem liegt in NP: Man rät zunächst einen Kreis und berechnet dann dessen Länge. Das Problem des Handlungsreisenden ist von praktischer Bedeutung z.B. beim Entwurf von Telefonnetzwerken oder integrierten Schaltungen.

(3) Erfüllbarkeitsproblem für Boolesche Ausdrücke (abgekürzt SAT für „satisfiability“):

Gegeben: Ein Boolescher Ausdruck B , also ein Ausdruck, der aus Booleschen Variablen x_1, x_2, \dots, x_m besteht, die durch Operatoren \neg (*not*), \wedge (*and*) und \vee (*or*), sowie durch Klammern miteinander verknüpft sind.

Frage: Ist B erfüllbar, d.h., gibt es eine Belegung der Variablen x_1, x_2, \dots, x_m in B mit 0 und 1, so dass B insgesamt den Wert 1 liefert?

Zum Beispiel ist $B = (x_1 \wedge x_2) \vee \neg x_3$ erfüllbar durch die Werte $x_1 = x_2 = \mathbf{true}$ oder $x_3 = \mathbf{false}$.

Dieses Problem definieren und untersuchen wir in Abschnitt 8.7 noch etwas genauer.

8.6 Die Klasse NPC und polynomielle Reduzierbarkeit

Bei den Untersuchungen zur bislang offenen Frage, ob $P=NP$ gilt, stößt man auf eine erstaunliche Teilklasse von NP, die Klasse NPC der *NP-vollständigen* (NP-complete) Probleme. Wir werden sehen, dass gilt:

Wenn *ein* Problem aus NPC in P liegt, so liegen bereits *alle* Probleme aus NP in P, d.h., dann gilt $P=NP$.

Um die offene Frage $P \stackrel{?}{=} NP$ positiv zu lösen, würde es also schon genügen, für ein einziges NP-vollständiges Problem einen deterministisch-polynomiellen Algorithmus anzugeben. Es wird allerdings als sehr unwahrscheinlich erachtet, dass ein solches Problem mit einem derartigen Lösungsalgorithmus existiert.

Die Klasse NPC wurde 1971 von STEPHEN A. COOK eingeführt. Das eben vorgestellte Problem SAT wurde von COOK als Erstes als NP-vollständig bestätigt. Ab 1972 hat R. KARP viele weitere Probleme als NP-vollständig nachgewiesen. Heute kennt man weit über 1000 Beispiele aus der Klasse NPC; siehe auch <http://www.nada.kth.se/~viggo/wwwcompendium/>, ein Kompendium solcher Probleme.

Im folgenden wollen wir den Begriff NP-vollständig definieren. Dazu benötigen wir den Begriff der *polynomiellen Reduktion*, die KARP 1972 als Beweistechnik zum Nachweis von NP-Vollständigkeit einführte. Es handelt sich dabei um eine Verschärfung des Begriffs der Reduktion $L_1 \leq L_2$ aus Abschnitt 7.2 (Definition 7.2.4).

Definition 8.6.1 POLYNOMIELLE REDUZIERBARKEIT

Seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ Sprachen. Dann heißt L_1 auf L_2 *polynomiell reduzierbar*, abgekürzt

$$L_1 \leq_p L_2,$$

falls es eine totale und mit der Zeitkomplexität eines Polynoms berechenbare Funktion $f: \Sigma_1^* \rightarrow \Sigma_2^*$ gibt, so dass für alle $w \in \Sigma_1^*$ gilt: $w \in L_1 \Leftrightarrow f(w) \in L_2$. Wir sagen auch: $L_1 \leq_p L_2$ *mittels* f . \square 8.6.1

Anschaulich besagt $L_1 \leq_p L_2$, dass L_1 noch nicht einmal im Rahmen polynomieller Berechenbarkeit aufwändiger oder „schwerer“ als L_2 ist. Man erkennt leicht, dass \leq_p eine reflexive und transitive Relation auf Sprachen ist, da mit zwei Polynomen $p_1(n)$ und $p_2(n)$ auch $p_1(p_2(n))$ ein Polynom ist.

Definition 8.6.2 NP-VOLLSTÄNDIGKEIT (COOK, 1971)

Eine Sprache K heißt *NP-vollständig*, falls $K \in \text{NP}$ gilt und $\forall L \in \text{NP}: L \leq_p K$. \square 8.6.2

Lemma 8.6.3 POLYNOMIELLE REDUKTION

Sei $L_1 \leq_p L_2$. Dann gilt:

- (i) Falls $L_2 \in \text{P}$ gilt, so auch $L_1 \in \text{P}$.
- (ii) Falls $L_2 \in \text{NP}$ gilt, so auch $L_1 \in \text{NP}$.
- (iii) Falls L_1 NP-vollständig ist und $L_2 \in \text{NP}$ gilt, so ist auch L_2 NP-vollständig.

Beweis: zu (i) und (ii): Es gelte $L_1 \leq_p L_2$ mittels einer Funktion f , die durch eine Turingmaschine M_1 berechnet wird. Das Polynom p_1 begrenze die Rechenzeit von M_1 . Da $L_2 \in \text{P}$ (bzw. $L_2 \in \text{NP}$) ist, gibt es eine durch ein Polynom p_2 zeitbeschränkte (nichtdeterministische) Turingmaschine M_2 , die die charakteristische Funktion χ_{L_2} berechnet.

Wie bei der allgemeinen Reduktion ist dann die charakteristische Funktion χ_{L_1} für alle $w \in \Sigma_1^*$ wie folgt berechenbar: $\chi_{L_1}(w) = \chi_{L_2}(f(w))$. Dieses geschieht durch Hintereinanderschalten der Turingmaschinen M_1 und M_2 . Sei jetzt $|w| = n$. Dann berechnet M_1 das Wort $f(w)$ in $p_1(n)$ Schritten. Diese Zeitschranke beschränkt zugleich die Länge von $f(w)$, d.h. $|f(w)| \leq p_1(n)$. Damit wird die Berechnung von $\chi_{L_2}(f(w))$ in

$$p_1(n) + p_2(p_1(n))$$

Schritten durchgeführt. Also ist auch $L_1 \in \text{P}$ (bzw. $L_1 \in \text{NP}$).

zu (iii): Sei $L \in \text{NP}$. Da L_1 NP-vollständig ist, gilt $L \leq_p L_1$. Ferner gilt $L_1 \leq_p L_2$. Aus der Transitivität von \leq_p folgt $L \leq_p L_2$, was zu zeigen war. \square 8.6.3

Korollar 8.6.4 (KARP)

Sei L_0 eine NP-vollständige Sprache. Dann gilt: $L_0 \in \text{P} \Leftrightarrow \text{P} = \text{NP}$.

Beweis: „ \Rightarrow “: Aussage (i) des Lemmas. „ \Leftarrow “: klar.

☒ 8.6.4

Beispiel:

Wir zeigen folgende polynomielle Reduktion:

Hamiltonscher Pfad \leq_p Handlungsreisender.

Gegeben sei ein Graph G mit n Knoten. Als Reduktionsfunktion betrachten wir folgende Konstruktion $f : G \mapsto G'$ eines neuen Graphen G' mit Kantenlängen:

- G' übernimmt die n Knoten von G .
- Jede Kante von G wird Kante von G' der Länge 1.
- Jede „Nichtkante“ von G wird Kante von G' der Länge 2.

Damit ist G' ein vollständiger Graph, d.h. je zwei Knoten sind durch eine Kante verbunden. Ein Beispiel ist in Abbildung 8.4 angegeben.

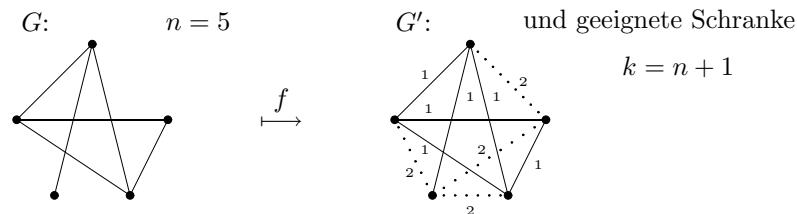


Abbildung 8.4: Transformation von G (Hamiltonkreisproblem) in G' (Handlungsreisendenproblem)

Die Konstruktion f erfolgt in polynomieller Zeit. (Für das Finden der Nichtkanten: man kann eine Inzidenzmatrix betrachten, die Konstruktion ist also $O(n^2)$.)

Wir zeigen jetzt die Reduktionseigenschaft von f , d.h.

G hat Hamiltonschen Pfad $\Leftrightarrow G'$ hat Rundreise der Länge $\leq n + 1$.

Beweis von „ \Rightarrow “: Man braucht $n - 1$ Kanten der Länge 1 (d.h. „aus G “), um n verschiedene Knoten zu verbinden. Um diesen Hamiltonschen Pfad zu schließen, braucht man eine weitere Kante der Länge ≤ 2 . Insgesamt hat die so konstruierte Rundreise die Länge $\leq n + 1$.

Beweis von „ \Leftarrow “: In der Rundreise gibt es mindestens n Kanten (um n Knoten auf einem Kreis zu verbinden) und höchstens $n + 1$ Kanten (wegen der Kantenlängen ≥ 1).

Auf der Rundreise wird mindestens 1 Knoten zweimal erreicht (Start = Ende). Man muss also auf jeden Fall eine Kante entfernen, um einen Pfad mit lauter verschiedenen Knoten zu erhalten. Wir untersuchen, ob das reicht.

Fall 1: Nach dem Entfernen der einen Kante hat der verbleibende Kantenzug die Länge $n - 1$.

Die damit erreichbaren Knoten sind alle verschieden. Man hat also einen Hamiltonschen Pfad gefunden.

Beispiel zu Fall 1: s.o.

Fall 2: Sonst hat der verbleibende Kantenzug die Länge n .

Dann gibt es in diesem Kantenzug n Kanten und es wird ein Knoten auf einem verbleibenden Kreis zweimal angetroffen. Dann erhält man einen Hamiltonschen Pfad nach dem Entfernen einer weiteren Kante.

Die Abbildung 8.5 zeigt ein Beispiel zu Fall 2.



Abbildung 8.5: Hier ist $n = 3$. Zwei Kanten müssen weggenommen werden

8.7 Das Erfüllbarkeitsproblem für Boolesche Ausdrücke

Definition 8.7.1 SYNTAX BOOLESCHER AUSDRÜCKE

Ein *Boolescher Ausdruck* ist ein Term über $\{\mathbf{false}, \mathbf{true}\}$, Variablen aus $V = \{x_1, x_2, \dots\}$, den Operationen \neg , \wedge und \vee und Klammern $(,)$ nach folgendem Bildungsgesetz:

- **Anfang:** \mathbf{false} und \mathbf{true} sind Boolesche Ausdrücke; jede Variable $x_i \in V$ ist ein Boolescher Ausdruck.
- **Schritt:** Wenn E ein Boolescher Ausdruck ist, dann auch $\neg E$; wenn E_1, E_2 Boolesche Ausdrücke sind, dann auch $(E_1 \wedge E_2)$ und $(E_1 \vee E_2)$.
- **Abschluss:** Nichts sonst ist ein Boolescher Ausdruck.

Wenn $x \in V$, dann schreibt man statt $\neg x$ oft \bar{x} , und man nennt x und \bar{x} *Literale*. Operatorprioritäten sind wie üblich (\neg vor \wedge vor \vee) definiert. ☒ 8.7.1

Definition 8.7.2 KLAUSELN UND KONJUNKTIVE FORM

Wenn y_1, y_2, \dots, y_k Literale sind, dann heißt $(y_1 \vee y_2 \vee \dots \vee y_k)$ eine *Klausel* der Ordnung k .

Wenn c_1, c_2, \dots, c_r Klauseln der Ordnung $\leq k$ sind, dann heißt $E = c_1 \wedge c_2 \wedge \dots \wedge c_r$ Boolescher Ausdruck in *konjunktiver Form* oder auch in *konjunktiver Normalform* (CNF) der Ordnung $\leq k$. Wenn mindestens eine Klausel k verschiedene Literale enthält, dann heißt E eine CNF der Ordnung k . ☒ 8.7.2

Definition 8.7.3 SEMANTIK BOOLESCHER AUSDRÜCKE, ERFÜLLBARKEIT

Eine *Belegung* β ist eine Abbildung $\beta: V \rightarrow \{\mathbf{false}, \mathbf{true}\}$, die jeder Variablen einen Wahrheitswert zuordnet. Jede Belegung β kann kanonisch auf Boolesche Ausdrücke fortgesetzt werden:

$$\begin{aligned} \beta(\mathbf{false}) &= \mathbf{false}, \\ \beta(\mathbf{true}) &= \mathbf{true}, \\ \beta(\neg E) &= \neg \beta(E), \\ \beta(E_1 \wedge E_2) &= \beta(E_1) \wedge \beta(E_2), \\ \beta(E_1 \vee E_2) &= \beta(E_1) \vee \beta(E_2), \end{aligned}$$

wobei auf den rechten Seiten die bekannten Booleschen Operatoren auf der Menge $\{\mathbf{false}, \mathbf{true}\}$ zu verstehen sind.

Ein Boolescher Ausdruck E heißt *erfüllbar* („satisfiable“), wenn es eine Belegung β mit $\beta(E) = \mathbf{true}$ gibt. ☒ 8.7.3

Das Erfüllbarkeitsproblem wollen wir als Sprache

$$\{E \mid E \text{ ist Boolescher Ausdruck und } E \text{ ist erfüllbar}\}$$

darstellen. Um ganz genau zu sein, codiert man noch V folgendermaßen:

$$x_i \mapsto x_j, \text{ wobei } j \text{ die Dualdarstellung der Zahl } i \text{ ist,}$$

also $x_1 \mapsto x1, x_2 \mapsto x10, x_3 \mapsto x11, x_4 \mapsto x100$ usw. (Sonst hätte man ein unendlich großes Alphabet zu betrachten, da V eine Teilmenge dieses Alphabets ist.) Insgesamt definiert man:

Definition 8.7.4 ERFÜLLBARKEITSPROBLEM(E) FÜR BOOLESCHE AUSDRÜCKE

- SAT = $\{E \mid E \text{ ist Boolescher Ausdruck über } \{x1, x10, x11, \dots\} \text{ und } E \text{ ist erfüllbar}\}$.
- CNF-SAT = $\{E \mid E \text{ ist Boolescher Ausdruck in SAT und } E \text{ ist in konjunktiver Form}\}$.
- SAT(k) = $\{E \mid E \text{ ist Boolescher Ausdruck in CNF-SAT und } E \text{ ist von der Ordnung } k\}$.

☒ 8.7.4

Ohne Einschränkung der Allgemeinheit kann stets angenommen werden, dass in E genau die ersten m Variablen enthalten sind (für ein geeignetes $m \in \mathbb{N}$). Wenn nötig, können auch die Konstanten **false** und **true** durch geeignete äquivalente Umformungen entfernt werden: **false** durch $(x1 \wedge \overline{x1})$ und **true** durch $(x1 \vee \overline{x1})$.

Satz 8.7.5 COOK 1971, NP-VOLLSTÄNDIGKEIT VON CNF-SAT

CNF-SAT ist NP-vollständig.

Beweis:

Zunächst erkennt man leicht, dass CNF-SAT in NP liegt: Zu einem Booleschen Ausdruck E in konjunktiver Form, der genau m Variablen enthält, rät man eine Belegung

$$\beta: \{x_1, \dots, x_m\} \rightarrow \{\mathbf{false}, \mathbf{true}\}.$$

Dann setzt man für jede Variable x_i den Wert $\beta(x_i)$ ein und rechnet $\beta(E)$ nach den üblichen Rechenregeln (Definition 8.7.3) aus. Wenn $|E| = n$ galt, dann besitzt E nicht mehr als n Variablen, d.h. $m \leq n$. Das Raten einer Belegung β erfolgt in linearer Zeit (Tabelle anlegen, m Schritte), die Ersetzung in E dauert $|E| \cdot \text{const}$ Schritte und ebenso die Auswertung; d.h. CNF-SAT \in NTIME($c_1 \cdot n + c_2$) für geeignete Konstanten c_1 und c_2 . Speziell: CNF-SAT \in NP.

Der schwierigere Teil besteht darin zu zeigen, dass für jede Sprache $L \in$ NP gilt: $L \leq$ CNF-SAT. Betrachten wir hierzu eine beliebige Sprache $L \in$ NP. Dann gibt es eine nichtdeterministische Turingmaschine $M = (Q, X, \Gamma, \sqsubset, \delta, q_i, q_f)$ mit Zustandsmenge Q , Eingabealphabet X , Bandalphabet Γ , das X umfaßt,

Anfangszustand $q_i \in Q$ und Endzustand q_f . Da M L in nichtdeterministisch polynomieller Zeit akzeptiert, gibt es ein Polynom p , so dass M für jedes $w \in X^*$ und jede Berechnungsfolge nach höchstens $p(n)$ (mit $n = |w|$) Schritten anhält, und $w \in L$ gilt genau dann, wenn es eine Berechnungsfolge von τ gibt, die die Anfangskonfiguration $q_i w$ in eine Endkonfiguration $u_1 q_f u_2$ überführt. Wir nehmen o.B.d.A. an, dass M nur ein Band besitzt.

Zu jedem $w \in X^*$ konstruieren wir nun einen Booleschen Ausdruck $g(w) \in \Sigma^*$, mit

$$\Sigma = \{x, 0, 1, \mathbf{false}, \mathbf{true}, \neg, \wedge, \vee, (,)\},$$

in konjunktiver Form, so dass gilt:

$$w \in L \Leftrightarrow g(w) \in \text{CNF-SAT}.$$

Wir werden dann noch sehen, dass diese Abbildung $g: X^* \rightarrow \Sigma^*$ eine polynomielle Reduktion ist. Dann folgt, da L aus NP beliebig war, dass CNF-SAT NP-vollständig ist.

Konstruktion von $g(w)$ aus w :

Sei $Q = \{q_1, \dots, q_s\}$, $q_1 = q_i$, $q_s = q_f$. Sei $\Gamma = \{c_1, \dots, c_\gamma\}$ mit $c_1 = \sqcup$. O.B.d.A. sei

$$\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{-1, 0, +1\})$$

mit $\delta(q_f, c) = \emptyset$ für alle $c \in \Gamma$, wobei $-1, 0, +1$ für L, N, R (Kopfbewegungen) stehen.

Wir vervollständigen δ , indem wir jedes $\delta(q_f, c) = \emptyset$ in $\delta(q_f, c) = \{(q_f, c, 0)\}$ abwandeln. Auf diese Weise tritt die leere Menge \emptyset nie als Bild von δ auf. Da das Ende einer Berechnung durch $\delta(q_f, c) = \emptyset$ bestimmt wurde, wird durch die Änderung eine niemals haltende Turingmaschine M' definiert, die jedoch die Arbeitsweise von M genau widerspiegelt. Wenn M , angesetzt auf w , im Zustand q_f anhielt, so erreicht M' nach $p(n)$ (mit $n = |w|$) Schritten eine Konfiguration $u_1 q_f u_2$ und M' behält diese Konfiguration unendlich lange bei; das Umgekehrte gilt auch. Also:

$$\begin{aligned} w \in L &\Leftrightarrow M \text{ ist angesetzt auf } w \text{ nach } p(|w|) \text{ Schritten in einer Konfiguration } u_1 q_f u_2 \\ &\Leftrightarrow M' \text{ durchläuft eine Folge } k_1, k_2, \dots, k_{p(n)} \text{ von Konfigurationen mit} \\ &\quad \text{(i) } k_1 = q_1 w \text{ ist Anfangskonfiguration;} \\ &\quad \text{(ii) } k_{i+1} \text{ ist Folgekonfiguration von } k_i \text{ für alle } i \geq 1; \\ &\quad \text{(iii) } n = |w| \text{ und } k_{p(n)} \text{ enthält den Endzustand } q_f. \end{aligned}$$

Wir erreichen also durch diese künstliche Vervollständigung, dass alle Konfigurationenfolgen o.B.d.A. gleich lang (Länge $p(n)$ für ein Eingabewort der Länge n) sind.

Die Übergangsrelation δ habe m Elemente und es sei $\delta = \{tupel_1, tupel_2, \dots, tupel_m\}$ irgend eine feste Durchnummerierung von δ .

Sei $w \in X^*$ mit $|w| = n$ gegeben, $w = c_{j_1} c_{j_2} \dots c_{j_n}$. Die Formel $g(w)$ wird mit folgenden Booleschen Variablen aufgebaut:

- $z_{t,k}$, $1 \leq t \leq p(n)$, $1 \leq k \leq s$.
 $z_{t,k} = \mathbf{true}$ bedeutet: M' ist zum Zeitpunkt t im Zustand q_k .
- $a_{t,i,j}$, $1 \leq t \leq p(n)$, $-p(n) \leq i \leq p(n)$, $1 \leq j \leq \gamma$.
 $a_{t,i,j} = \mathbf{true}$ bedeutet: Zum Zeitpunkt t trägt das Feld i den Inhalt c_j .

- $s_{t,i}$, $1 \leq t \leq p(n)$, $-p(n) \leq i \leq p(n)$.
 $s_{t,i} = \mathbf{true}$ bedeutet: zum Zeitpunkt t befindet sich der Lese/Schreibkopf von M' auf dem Feld i .
- $b_{t,l}$, $1 \leq t \leq p(n)-1$, $1 \leq l \leq m$.
 $b_{t,l} = \mathbf{true}$ bedeutet: Für die Überführung vom Zeitpunkt t zum Zeitpunkt $t+1$ wird das l te Tupel von δ benutzt.

Da τ höchstens $p(n)$ Schritte macht, kann man stets $|i| \leq p(n)$ und $t \leq p(n)$ annehmen.

Der Boolesche Ausdruck $g(w)$ soll nun genau die obige Konfigurationenfolge $k_1, k_2, \dots, k_{p(n)}$ beschreiben (bzw. genau die zulässigen Konfigurationenfolgen dieser Art). Hierzu müssen folgende Bedingungen erfüllt werden:

- (1) Anfangskonfigurierung: M' ist im Zustand q_1 ; der Zeitpunkt ist $t = 1$; $\sqcup^{p(n)+1} w \sqcup^{p(n)-n}$ steht auf dem Band; der Lese/Schreibkopf ist auf Feld 1 (definiert als Feld des ersten Buchstabens von w , falls $w \neq \varepsilon$, beliebig sonst).
- (2) Endkonfigurierung, sofern w akzeptiert wird: der Zeitpunkt ist $t = p(n)$; M' ist im Zustand q_f .
- (3) Übergangsbedingung: M' ist zu jedem Zeitpunkt $1 \leq t \leq p(n)$ in genau einem Zustand; jedes Feld von $-p(n)$ bis $+p(n)$ enthält genau ein Symbol aus Γ ; der Lese/Schreibkopf befindet sich auf genau einem dieser Felder; und genau eins der Tupel von δ wird für die Überführung ausgewählt.
- (4) Folgekonfigurierung: die nächste Konfiguration ergibt sich aus der vorhergehenden Konfiguration auf Grund der Überführung, die durch das unter (3) beschriebene Tupel von δ bestimmt ist.

Insgesamt sei $g(w) = A_1 \wedge A_2 \wedge A_3 \wedge A_4$ mit:

- Ad (1):

$$\begin{aligned}
 A_1 &= a_{1,-p(n),1} \wedge a_{1,-p(n)+1,1} \wedge \dots \wedge a_{1,0,1} && \left(\sqcup^{p(n)+1} \right. \\
 & && \left. \text{steht links} \right) \\
 &\wedge a_{1,1,j_1} \wedge a_{1,2,j_2} \wedge \dots \wedge a_{1,n,j_n} && (w = c_{j_1} \dots c_{j_n}) \\
 &\wedge a_{1,n+1,1} \wedge a_{1,n+2,1} \wedge \dots \wedge a_{1,p(n),1} && \left(\sqcup^{p(n)-n} \right. \\
 & && \left. \text{steht rechts von } w \right) \\
 &\wedge z_{1,1} \wedge s_{1,1} . && (\text{anfängl. Zustand und Kopfposition})
 \end{aligned}$$

Diese Formel beschreibt die Anfangskonfigurierung mit $2 \cdot p(n) + 3$ Variablen.

- Ad (2):

$$A_2 = z_{p(n),s} \quad (\text{eine Variable; } q_s = q_f).$$

- Ad (3):

Wir beschreiben zunächst einen Hilfsausdruck: Für Variablen x_1, \dots, x_k sei

$$\begin{aligned} \text{genauein}(x_1, \dots, x_k) &= \text{mindestensein}(x_1, \dots, x_k) \wedge \text{höchstensein}(x_1, \dots, x_k) \\ \text{mit } \text{mindestensein}(x_1, \dots, x_k) &= (x_1 \vee x_2 \vee \dots \vee x_k) \\ \text{und } \text{höchstensein}(x_1, \dots, x_k) &= \bigwedge_{1 \leq i < j \leq k} (\bar{x}_i \vee \bar{x}_j). \end{aligned}$$

Der Ausdruck $\text{genauein}(x_1, \dots, x_k)$ ist in CNF und besitzt $k + \frac{1}{2} \cdot k \cdot (k-1) \cdot 2 = k^2$ Variablen. Er wird **true**, wenn genau ein x_i den Wert 1 hat.

Setze nun

$$\begin{aligned} A_3 &= \bigwedge_{1 \leq t \leq p(n)} \left(A_3^{\text{Zustand}}(t) \wedge A_3^{\text{Stelle}}(t) \wedge A_3^{\text{Feld}}(t) \wedge A_3^{\ddot{U}}(t) \right) \\ \text{mit: } A_3^{\text{Zustand}}(t) &= \text{genauein}(z_{t,1}, \dots, z_{t,s}) \\ A_3^{\text{Stelle}}(t) &= \text{genauein}(s_{t,-p(n)}, s_{t,-p(n)+1}, \dots, s_{t,p(n)}) \\ A_3^{\text{Feld}}(t) &= \bigwedge_{-p(n) \leq i \leq p(n)} \text{genauein}(a_{t,i,1}, \dots, a_{t,i,\gamma}) \\ A_3^{\ddot{U}}(t) &= \text{genauein}(b_{t,1}, \dots, b_{t,m}). \end{aligned}$$

Diese Formel ist ebenfalls in CNF. A_3 beschreibt die Übergangsbedingung (3) und besitzt

$$p(n) \cdot (s^2 + (2 \cdot p(n) + 1)^2 + (2 \cdot p(n) + 1)^2 \cdot \gamma^2 + m^2)$$

Variablen.

- Ad (4):

$$A_4 = \bigwedge_{1 \leq t < p(n)} A_4(t).$$

Zur Definition der Terme $A_4(t)$ benennen wir die (Elemente der) Tupel von δ .

Für $l = 1, 2, \dots, m$ sei das l te Tupel von δ gegeben durch

$$\begin{aligned} \text{tupel}_l &= (q_{k_l}, c_{j_l}, q_{\bar{k}_l}, c_{\bar{j}_l}, d_l) \\ &\quad \in \quad \in \quad \in \quad \in \quad \in \\ &\quad Q \quad \Gamma \quad Q \quad \Gamma \quad \{+1, 0, -1\} \end{aligned}$$

Mit diesen Indexbezeichnungen setzen wir:

$$\begin{aligned} A_4(t) &= \bigwedge_{-p(n) \leq i \leq p(n)} \left[\left(\bigwedge_{1 \leq j \leq \gamma} (s_{t,i} \vee \bar{a}_{t,i,j} \vee a_{t+1,i,j}) \right) \wedge \right. \\ &\quad \bigwedge_{1 \leq l \leq m} \left((\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee z_{t,k_l}) \wedge (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee a_{t,i,j_l}) \wedge \right. \\ &\quad (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee z_{t+1,\bar{k}_l}) \wedge (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee a_{t+1,i,\bar{j}_l}) \wedge \\ &\quad \left. \left. (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee s_{t+1,i+d_l}) \right) \right]. \end{aligned}$$

Erläuterung der Klausel in der ersten Zeile:

$(s_{t,i} \vee \bar{a}_{t,i,j} \vee a_{t+1,i,j})$ ist **true** genau dann, wenn

aus $s_{t,i} = \mathbf{false}$ und $a_{t,i,j} = \mathbf{true}$ folgt $a_{t+1,i,j} = \mathbf{true}$.

Das bedeutet: nicht betrachtete Felder werden von M' nicht verändert.

Erläuterung der ersten Klausel in der zweiten Zeile:

$(\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee z_{t,k_l})$ wird **true** genau unter folgenden Umständen: wenn M' zum Zeitpunkt t das Feld i betrachtet und als Überführung das l te Tupel gewählt wird, dann befand M' sich zum Zeitpunkt t im Zustand q_{k_l} .

Die erste Klausel in der dritten Zeile besagt, dass unter den gleichen Bedingungen M' sich zum Zeitpunkt $t + 1$ im Zustand $q_{\bar{k}_l}$ befindet.

Ähnlich sind die restlichen Klauseln zu lesen.

A_4 ist in konjunktiver Normalform und enthält

$$p(n) \cdot (2 \cdot p(n) + 1) \cdot (3 \cdot \gamma + 15 \cdot m)$$

Variablen.

Man zeigt nun leicht, dass $g(w)$ in CNF ist. Des Weiteren:

Behauptung 1: g ist polynomiell.

Die obige Konstruktion liefert eine Formel mit

$$\begin{aligned} p'(n) = & (2 \cdot p(n) + 3) + (s - r + 1) + p(n) \cdot (s^2 + (2p(n) + 1)^2 \cdot (\gamma^2 + 1) + m^2) \\ & + p(n) \cdot (2 \cdot p(n) + 1) \cdot (3\gamma + 15m) \end{aligned}$$

Variablen. Die Erstellung von $g(w)$ aus w ist offensichtlich proportional zu $p'(n)$, also auf einer deterministischen 1-Band-Turingmaschine in einer Zeit von höchstens $\text{const} \cdot (p'(n))^2$ Schritten zu berechnen, d.h. polynomiell.

Behauptung 2: g ist eine Reduktion, d.h., $\forall w \in X^* : (w \in L \Leftrightarrow g(w) \in \text{CNF-SAT})$. Diese Aussage folgt aus der Konstruktion, wobei nur „ \Leftarrow “ etwas aufwändiger zu beweisen ist.

Also definiert g eine polynomielle Reduktion $L \leq_p \text{CNF-SAT}$. Da L beliebig war, folgt insgesamt, dass CNF-SAT NP-vollständig ist. □ 8.7.5

Man kann auch zeigen, dass SAT NP-vollständig ist. Zum Beweis kann man jede Formel in SAT in eine *erfüllbarkeitsäquivalente* Formel in CNF-SAT polynomiell transformieren. (Die stärkere und bekanntere Transformation in eine *äquivalente* Formel in CNF-SAT ist hier nicht allgemein anwendbar, da sie nicht notwendigerweise polynomiell ist.)

Es gilt sogar:

Satz 8.7.6 NP-VOLLSTÄNDIGKEIT VON SAT(3)

SAT(3) ist NP-vollständig.

Beweis: Wegen $\text{CNF-SAT} \in \text{NP}$ gilt auch $\text{SAT}(3) \in \text{NP}$. Man kann CNF-SAT auf SAT(3) reduzieren, indem jede Klausel $(x_1 \vee x_2 \vee \dots \vee x_r)$ durch

$$(x_1 \vee \overline{y_1}) \wedge (y_1 \vee x_2 \vee \overline{y_2}) \wedge (y_2 \vee x_3 \vee \overline{y_3}) \wedge \dots \wedge (y_{r-1} \vee x_r \vee \overline{y_r}) \wedge y_r$$

mit neuen Variablen y_1, y_2, \dots, y_r ersetzt wird. Man erkennt leicht, dass $(x_1 \vee \dots \vee x_r)$ genau dann erfüllbar ist, wenn es die längere Formel (der Ordnung 3) ist. Die Überführung ist von polynomieller Größenordnung. Die resultierende Formel hat noch Klauseln mit weniger als drei Literalen. Diese lassen sich in einem letzten Schritt jedoch sehr einfach in Klauseln mit genau drei Literalen umformen. Also lässt sich CNF-SAT polynomiell auf SAT(3) reduzieren. \square 8.7.6

8.8 Einige weitere NP-vollständige Probleme

In der Literatur werden Tausende von NP-vollständigen Problemen aus verschiedenen (Anwendungs-) Bereichen betrachtet, die fast alle auch praktische Bedeutung haben. Wir geben hier eine kleine Auswahl an.

KÜ (Knotenüberdeckung)

Gegeben: Ein ungerichteter endlicher Graph $G = (V, E)$ und eine Zahl $K \leq |V|$.

Frage: Gibt es eine Knotenüberdeckung von G mit der Größe höchstens K ?

Dabei heißt $V' \subseteq V$ eine Knotenüberdeckung von G genau dann, wenn für alle $\{x, y\} \in E$ gilt:

$$x \in V' \vee y \in V'.$$

Die Abbildung 8.6 zeigt ein Beispiel.

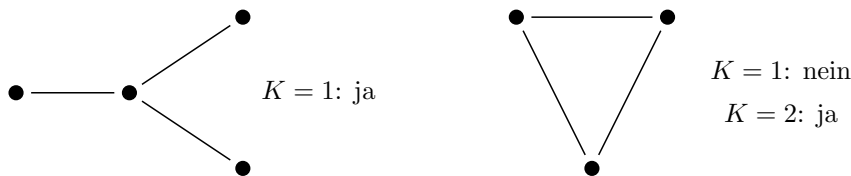


Abbildung 8.6: Eine Ja-Instanz (links) und eine Ja- und eine Nein-Instanz (rechts) von KÜ

Algorithmus:

- Rate V' mit $|V'| \leq K$.
- Prüfe, ob V' die Bedingungen erfüllt.

Also liegt KÜ in NP.

CLIQUE

Gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $J \leq |V|$.

Frage: Gibt es eine Clique in G mit der Größe mindestens J ?

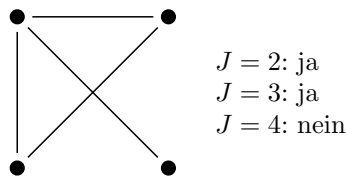


Abbildung 8.7: Zwei Ja-Instanzen und eine Nein-Instanz von CLIQUE

Dabei heißt $V' \subseteq V$ eine Clique von G genau dann, wenn für alle $x, y \in V'$ gilt: $\{x, y\} \in E$. Die Abbildung 8.7 zeigt ein Beispiel.

3DM (Dreidimensionales Matching)

Gegeben: Paarweise disjunkte Mengen W, X, Y mit $|W| = |X| = |Y| = q$, sowie eine Menge $M \subseteq W \times X \times Y$.

Frage: Enthält M ein Matching, d.h., eine Teilmenge $M' \subseteq M$ mit $|M'| = q$ und

$$\forall (w, x, y), (w', x', y') \in M': (w = w' \vee x = x' \vee y = y') \Rightarrow (w, x, y) = (w', x', y')$$

Dieses Problem liegt in NP: Wir raten M' und prüfen in Polynomzeit.

Beispiel mit $q = 2$: $W = \{1, 2\}, X = \{3, 4\}, Y = \{5, 6\}$.

Eine Ja-Instanz: $M = \{(1, 3, 5), (1, 4, 5), (2, 3, 6), (2, 4, 6)\}$
 M' z. B. $= \{(1, 3, 5), (2, 4, 6)\}$ oder $= \{(1, 4, 5), (2, 3, 6)\}$

Eine Nein-Instanz: $M = \{(1, 3, 5), (1, 4, 5), (1, 3, 6), (1, 4, 6)\}$
 $M' = ??$

PARTITION

Gegeben: Eine endliche Menge A und ein Gewicht $s(a) \in \mathbb{N}^+$ für jedes $a \in A$.

Frage: Gibt es eine Teilmenge $A' \subseteq A$ mit

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a) ?$$

Das Problem liegt in NP: Wir raten A' und prüfen nach. (Größe des Problems: $O(\sum_{a \in A} \log(s(a)))!$)

Beispiel: $A = \{a_1, a_2, \dots, a_7\}$

$$s(a_1) = 1, s(a_2) = 5, s(a_3) = 2, s(a_4) = 2,$$

$$s(a_5) = 10, s(a_6) = 3, s(a_7) = 3$$

$$A' = \underbrace{\{a_1, a_3, a_5\}}_{\sum_{a \in A'} s(a) = 1 + 2 + 10 = 13} \quad A \setminus A' = \underbrace{\{a_2, a_4, a_6, a_7\}}_{\sum_{a \in A \setminus A'} s(a) = 5 + 2 + 3 + 3 = 13}$$

Dies ist also eine Ja-Instanz.

RUCK (Rucksackproblem)

Gegeben: Eine endliche Menge U , eine Größe $s(u) \in \mathbb{N}$ und ein Wert $v(u) \in \mathbb{N}$ für jedes $u \in U$, eine obere Schranke $B \in \mathbb{N}$ für die Größe und eine untere Schranke $K \in \mathbb{N}$ für den Wert.

Frage: Gibt es eine Teilmenge $U' \subseteq U$ mit

$$\sum_{u \in U'} s(u) \leq B \text{ und } \sum_{u \in U'} v(u) \geq K ?$$

Ein Beispiel ist in Abbildung 8.8 gezeigt.

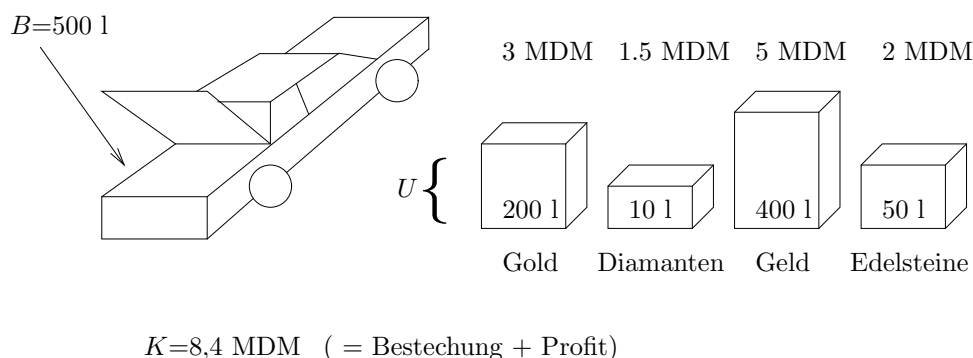


Abbildung 8.8: Beispiel zum Rucksackproblem

Einige dieser Probleme sind relativ leicht ineinander überführbar (bzw. aufeinander reduzierbar). Zum Beispiel lassen sich KÜ, CLIQUE und das folgende Problem gut ineinander überführen.

UM (Unabhängige Mengen)

Gegeben: Ein Graph $G = (V, E)$, eine Zahl $J \in \mathbb{N}^+$ mit $J \leq |V|$.

Frage: Gibt es eine Unabhängigkeitsmenge $V' \subseteq V$ mit $|V'| \geq J$?

Dabei heißt V' Unabhängigkeitsmenge, falls $\forall u, v \in V': \{u, v\} \notin E$.

Lemma 8.8.1 POLYNOMIELLE REDUKTION VON KÜ, CLIQUE, UM (OHNE BEWEIS)

Sei $G = (V, E)$ ein Graph und sei $V' \subseteq V$. Dann sind äquivalent:

- (i) V' ist Knotenüberdeckung.
- (ii) $V \setminus V'$ ist Unabhängigkeitsmenge.
- (iii) $V \setminus V'$ ist Clique im Komplementgraph $G^C = (V, (V \times V) \setminus E)$ von G .

Also zieht jeder Algorithmus zur Lösung eines der Probleme KÜ, CLIQUE oder UM auch direkt einen Algorithmus zur Lösung der beiden anderen nach sich; die Probleme sind polynomiell aufeinander reduzierbar.

Wir zeigen im Folgenden die NP-Vollständigkeit des Problems Knotenüberdeckung. Auch die anderen genannten Probleme sind NP-vollständig, für Reduktionsbeweise muss allerdings auf die Literatur verwiesen werden.

Satz 8.8.2 NP-VOLLSTÄNDIGKEIT VON KÜ

KÜ ist NP-vollständig.

Beweis: Dass KÜ in NP liegt, ist einfach zu zeigen: wir raten V' und prüfen die gewünschte Eigenschaft. Um die NP-Härte zu zeigen, reduzieren wir von SAT(3), d.h., wir zeigen $\text{SAT}(3) \leq_p \text{KÜ}$.

Sei $F = c_1 \wedge \dots \wedge c_m$ eine Instanz von SAT(3) mit Variablen $U = \{u_1, \dots, u_n\}$. Wir konstruieren einen Graphen $G = (V, E)$ und eine Zahl $K \leq |V|$. Für jede Variable $u_i \in U$ führen wir zwei Knoten $V_i = \{u_i, \bar{u}_i\}$ und eine Kante $E_i = \{\{u_i, \bar{u}_i\}\}$ ein. (Jede Knotenüberdeckung muss mindestens u_i oder \bar{u}_i enthalten.) Für jede Klausel c_j in F ($|c_j| = 3!$) führen wir einen Teilgraphen (V'_j, E'_j) ein:

$$\begin{aligned} V'_j &= \{a_1[j], a_2[j], a_3[j]\} \\ E'_j &= \{\{a_1[j], a_2[j]\}, \{a_1[j], a_3[j]\}, \{a_2[j], a_3[j]\}\} \end{aligned}$$

(Jede Überdeckung muss mindestens 2 Knoten aus V'_j enthalten.)

Sei $c_j = \{x_j, y_j, z_j\}$ mit den Literalen x_j, y_j, z_j .

Wir führen neue Kanten E''_j ein:

$$E''_j = \{\{a_1[j], x_j\}, \{a_2[j], y_j\}, \{a_3[j], z_j\}\}.$$

Dann setzen wir $K = n + 2m$ und $G = (V, E)$ mit

$$\begin{aligned} V &= \bigcup_{i=1}^n V_i \cup \bigcup_{j=1}^m V'_j \\ E &= \bigcup_{i=1}^n E_i \cup \bigcup_{j=1}^m (E'_j \cup E''_j). \end{aligned}$$

Beispiel: (siehe Abbildung 8.9)

$$\begin{aligned} U &= \{u_1, u_2, u_3, u_4\} \\ E &= (u_1 \vee \bar{u}_3 \vee \bar{u}_4) \wedge (\bar{u}_1 \vee u_2 \vee \bar{u}_4). \end{aligned}$$

(G, V) hat eine minimale Überdeckung der Größe K (ist minimale Größe für Überdeckung) gdw. F ist erfüllbar. ☒ 8.8.2

Aus Lemma 8.8.1 folgt direkt:

Korollar 8.8.3 NP-VOLLSTÄNDIGKEIT VON CLIQUE UND UM

CLIQUE und UM sind NP-vollständig.

☒ 8.8.3

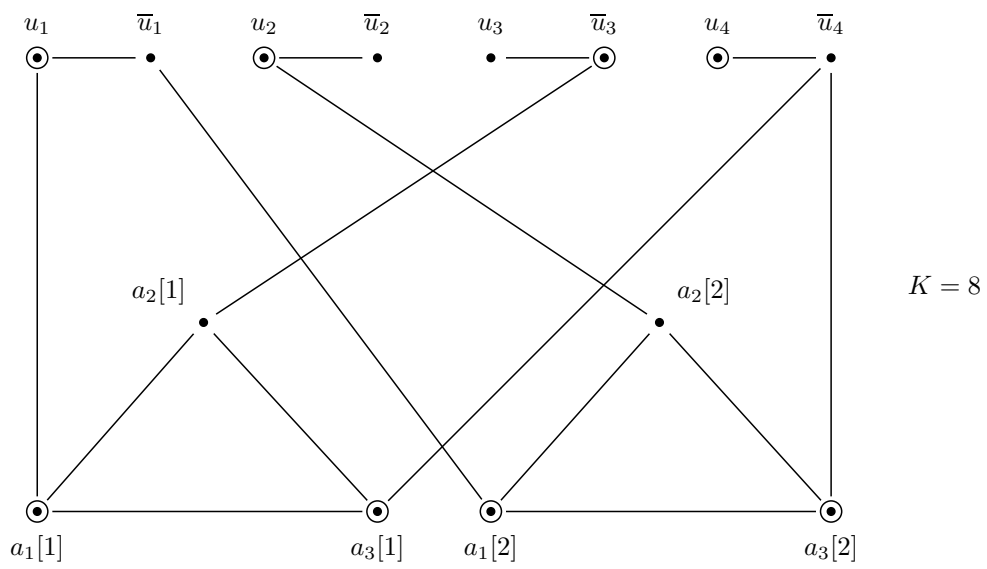


Abbildung 8.9: Beispiel zur Reduktion von SAT(3) auf KÜ

Index

- M_{uni} , 161
- O -Notationen, 186
- HP , 159
- $HP(M)$, 161
- HP_0 , 160
- LOOP-Programm, 31
- LOOP-berechenbar, 31
- WHILE-Programm, 31
- WHILE-berechenbar, 31
- loop**, 28
- skip**, 28
- stop**, 28
- ε , 23
- ε -Produktion, 46
- sHP , 156
- n -stellig, 14
- AP , 165
- $\dot{A}Q$, 152
- $MPCP$, 167
- PCP , 167
- PCP_Σ , 167
- $WPkf$, 152
- $WPmon$, 152
- WP , 152
- 3DM, 199

- Abgeschlossenheit einer Sprachklasse, 26
- Ableitungsproblem
 - allgemeine Grammatiken, 165
- Äquivalenzklassen, 10
- Äquivalenzproblem
 - allgemeine Grammatiken, 152
- Äquivalenzrelation, 10
 - von einer Relation erzeugt, 10
- Alphabet, 22
- Anzahl der Elemente einer Menge, 16
- Ausdruck, 28

- Auswahlaxiom, 16
- Auswahlfunktion, 14

- Baum, 21
 - Blatt, 21
 - Kind, 21
 - Nachfolger, 21
 - Pfad, 89
 - Vater, 21
 - Verzweigungsgrad, 89
 - Vorgänger, 21
 - Wiederholungsbaum, 90
 - Wurzel, 21
- Baum der Ableitungen, 38
- Berechnungskomplexität, 179
- Bierdeckelalphabet, 33
- Binärcodierung
 - natürlicher Zahlen, 34
- Boolescher Ausdruck, 29, 192
- Buchstaben, 22

- charakteristische Funktion einer Sprache, 24
 - halbe, 24
 - partielle, 24
- Chomsky-Grammatik, 44
- Chomsky-Hierarchie, 46
- Clique, 198
- Codierung, 32
 - umkehrbar effektiv, 35
- Codierung von Wort-Tupeln, 35
- Compiler, 93

- Deklaration, 26
- DFA-Äquivalenzproblem, 77
- DFA-Endlichkeitsproblem, 77
- DFA-Inklusionsproblem, 77
- DFA-Leerheitsproblem, 77

- DFA-Schnittproblem, 77
- DFA-Wortproblem, 77
- Diagonalisierungsargument, 20
- Diagonalschema, 18
- Differenzmenge, 6
- DPDA, 110
- DSPACE, 184
- DTIME, 184
- Durchschnitt, 6

- Effizienz, 179
- Einermenge, 6
- Einschränkung einer Relation, 9
- Element, 6
- endlicher Automat, 53
 - Abschlusseigenschaften, 62
 - akzeptierte Sprache, 55
 - Alphabet, 54
 - Anfangszustand, 54
 - deterministischer, 54
 - endliche Kontrolle, 53
 - Endzustände, 54
 - Isomorphie, 73
 - nichtdeterministischer, 53
 - Potenzmengen-Konstruktion, 58
 - Sprachäquivalenz, 55
 - Transition, 54
 - Transitionenfolge, 55
 - Übergangsrelation, 54
 - Zustände, 53
 - Zustandsdiagramm, 54
- endlicher Automat (deterministisch)
 - Übergangsfunktion, 54
- Enthaltenseinsproblem
 - allgemeine Grammatiken, 166
- entscheidbar, 77, 149
- Entscheidungsproblem, 149
- Entscheidungsprobleme
 - DFA-, 77
 - für kontextfreie Grammatiken, 114, 117
- Erfüllbarkeitsproblem für Boolesche Ausdrücke, 189
- Ersetzungssystem, 37
 - Ableitung, 37
 - aus Wort abgeleitete Sprache, 38
 - Church-Rosser-Eigenschaft, 40
 - deterministisch, 40
 - einer Grammatik zugeordnet, 44
 - konfluent, 40
 - Noethersch, 40
 - Normalform, 38
 - Produktionen, 37
 - stark konfluent, 40, 41
 - terminal, 38
 - terminierend, 40

- Funktion, 11
 - Argument, 12
 - bijektiv, 13
 - Erweiterung, 13
 - injektiv, 13
 - monoton, 17
 - partiell, 11
 - surjektiv, 13
 - T.b., 136
 - total, 11
 - zahlentheoretisch, 14
- Funktionswert, 12

- Grammatik
 - erzeugte Sprache, 44
 - formale, 44
 - kontextfrei, 46
 - kontextfreie Produktion, 46
 - kontextsensitiv, 46
 - kontextsensitive Produktion, 46
 - linkslinear, 46
 - linkslineare Produktion, 46
 - monoton, 46
 - monotone Produktion, 46
 - Nichtterminalsymbole, 44
 - Produktionen, 44
 - rechtslinear, 46
 - rechtslineare Produktion, 46
 - Regeln, 44
 - Startsymbol, 44
 - Terminalsymbole, 44
 - vom Erweiterungstyp, 46
 - zugeordnetes Ersetzungssystem, 44

- Graph, 20
 - Ausgangsgrad, 21
 - Ausgangsknoten, 21

- azyklisch, 21
- Eingangsgrad, 21
- Eingangsknoten, 21
- endlich, 20
- endlicher Weg, 21
- gerichtet, 20
- Kanten, 20
- Knoten, 20
- unendlich, 20
- unendlicher Weg, 21
- ungerichtet, 20
- Zyklus, 21
- Grundfunktion
 - konstante Funktion, 14
 - Nachfolgefunktion, 14
 - Projektionsfunktion, 14
 - Selektionsfunktion, 14
 - überall undefinierte Funktion, 14
- Halbordnungsrelation, 11
- Halteproblem
 - allgemeines, 159
 - auf leerem Band, 160
 - für eine einzelne Turingmaschine, 161
 - spezielles, 156
- Hamiltonscher Pfad, 188
- Handlungsreisendenproblem, 189
- Identitätsrelation, 9
- Index einer Äquivalenzrelation, 10
- Kartesisches Produkt, 6
 - X^n , 6
- Kellerautomat, 93
 - Akzeptanz, 96
 - Anfangszustand, 94
 - deterministisch, 110
 - Eingabealphabet, 93
 - Endzustände, 94
 - Kelleralphabet, 93
 - Kellerstartsymbol, 93
 - Konfiguration, 95
 - nichtdeterministisch, 93
 - Pop, 95
 - Push, 95
 - Transitionen, 94
 - Transitionsrelation, 93
 - Übergangsrelation, 93
 - Zustände, 93
- Kette, 22
- Kettenproduktion, 57
- Klausel, 192
- Knotenüberdeckung, 198
- Kommando
 - IF, 29
 - LOOP, 30
 - WHILE, 30
 - allgemeine Alternative, 29
 - allgemeine Schleife, 30
 - Hintereinanderausführung, 28
 - leer, 28
 - Nichtterminierung, 28
 - Terminierung, 28
 - Zuweisung, 28
- Komplementmenge, 6
- Komplexitätsklassen, 184
- Komposition
 - relationale, 7
- Konkatenation, 23
- kontextfreie Grammatik
 - ε -frei, 47
 - Ableitung, 82
 - Ableitungsbaum, 84
 - Chomsky-Normalform, 88
 - CYK-Algorithmus, 115
 - eindeutig, 86
 - eingeschränkt, 47
 - Greibach-Normalform, 88
 - Linksableitung, 82
 - mehrdeutig, 86
 - Normalformen, 87
 - Parsebaum, 83
 - Rechtsableitung, 82
- kontextfreie Sprache
 - deterministisch, 110
 - eindeutig, 86
 - mehrdeutig, 86
 - Pumping-Lemma, 89
- kontextfreie Sprachen
 - Entscheidbarkeit, 114
 - Satz über Abschlusseigenschaften, 108
 - Unentscheidbarkeit, 117

- Lemma von König, 132
- Literale, 192
- Mehrbandmaschinen, 180
- Mehrspurmaschinen, 180
- Menge
 - abzählbar, 15
 - abzählbar unendlich, 18
 - endlich, 15
 - leer, 6
 - Mächtigkeit, 15
 - Satz von Bernstein-Cantor-Dedekind-Schröder, 16
 - überabzählbar, 15
 - unendlich, 15
- Mengenkompression, 6
- Nerode-Rechtskongruenz, 70
- NP, 187
- NP-vollständig, 189
- NPC, 189
- NPSPACE, 187
- NSPACE, 184
- NTIME, 184
- Offline-TM, 181
- Operation, 14
 - assoziative, 14
 - kommutative, 14
- Opponent, 92
- P, 187
- Palindrom, 51
- Palindromsprache
 - deterministisch, 111
 - nicht deterministisch, 112
- partielle Ordnung, 11
- Partition, 199
- PDA, 93
- Platzkomplexität
 - DTM, 182
 - NTM, 183
- Postisches Korrespondenzproblem, 166
 - modifiziert, 167
- Potenzierte einer Relation, 9
- Potenzmenge, 6
- Präfix, 24
- Problem
 - entscheidbar, 77
- Problem des Hamiltonschen Pfades, 188
- Proponent, 92
- PSPACE, 187
- Pushdown-Automat, 93
- Rechenzeit, 179
- Registerprogramme, 31
- Reguläre Sprachen
 - Äquivalenzklassen-Automat, 72
- Regulärer Ausdruck, 65
- Relation, 7
 - antisymmetrisch, 9
 - Inverse, 7
 - irreflexiv, 9
 - Komplement, 7
 - linkseindeutig, 7
 - linkstotal, 7
 - Nachbereich, 7
 - rechtseindeutig, 7
 - reflexiv, 9
 - reflexive und transitive Hülle, 10
 - symmetrisch, 9
 - total, 7
 - transitiv, 9
 - transitive Hülle, 10
 - Vorbereich, 7
- Repräsentant, 10
- Satz
 - Myhill und Nerode, 71
 - Rabin und Scott, 58
- Selbstanwendungsargument, 20
- Selbstanwendungsproblem, 156
- semi-entscheidbar, 148
- Semi-Thue-System, 37
- Speicherplatz, 179
- Spiegelsprache, 25
- Spiegelwort, 24
- spontaner Übergang, 94
- Sprache, 24
 - n*te Iterierte, 25
 - Chomsky-0, 49
 - Chomsky-1, 49

- Chomsky-2, 49
- Chomsky-3, 49
- deterministisch endlich akzeptierbar, 55
- Einheitssprache, 24
- endlich akzeptierbar, 55
- entscheidbar, 149
- Komplementbildung von, 25
- kontextfrei, 49
- kontextsensitiv, 49
- leere, 24
- linkslinear, 49
- Nullte Iterierte, 25
- Plusabschluss, 25
- r.a., 148
- rechtslinear, 49
- regulär, 66
- rekursiv aufzählbar, 148
- reverse, 25
- semi-entscheidbar, 148
- Sternabschluss, 25
- T.a., 135
- Turing-akzeptierbar, 135
- volle, 24
- vom Erweiterungstyp, 49
- Sprachen
 - Differenzbildung von, 25
 - Durchschnitt von, 25
 - Konkatenation von, 25
 - Vereinigung von, 25
- Stelligkeit, 14
 - Hochindex, 14
- Strichalphabet, 33
- String, 22
- Suffix, 24
- Symbole, 22
- Table-Filling-Algorithmus, 76
- Teilmenge, 6
 - echt, 6
- Teilwort, 24
- Thue-System, 37
- Tupel, 6
 - n -, 6
- Turingmaschine, 121, 123
 - Akzeptieren eines Wortes, 133
 - akzeptierte Sprache, 133
 - Alphabet, 123
 - Anfangskonfiguration, 131
 - Anfangszustand, 122, 123
 - Ansetzen auf ein Band, 125
 - Ansetzen auf ein Wort, 131
 - Band, 122
 - Bandalphabet, 123
 - Berechnung, 130
 - deterministische, 123
 - Endkonfiguration, 129
 - Endzustand, 123
 - erreichbare Konfiguration, 130
 - Feld, 122
 - Folgekonfiguration, 130
 - Grad des Nichtdeterminismus, 132
 - Index, 156
 - Konfiguration, 129
 - Konfigurationsbaum, 132
 - LBA, 144
 - Leerzeichen, 123
 - Lese/Schreibkopf (LSK), 122
 - maximale Berechnung, 130
 - nichtdeterministische, 123
 - normierte Berechnung, 136
 - partielle Berechnung, 130
 - programmierbare, 161
 - Schritt, 122, 130
 - Sprachäquivalenz, 133
 - Steuereinheit, 122
 - Stopp, 124, 125
 - Tafel, 124
 - terminale Berechnung, 130
 - Turing-akzeptierbare Sprache, 135
 - Turing-berechenbare Funktion, 136
 - Übergangsrelation, 123
 - universelle, 161
 - Zustände, 123
- Typ einer Variablen, 26
- Übersetzer, 93
- UM, 200
- Unärcodierung, 34
- Vereinigung, 6
- Verfeinerung, 10
- Verkettung, 23

Verknüpfung, 14

Wertebereich einer Variablen, 26

Wiederholungsbaum, 90

Wort, 22

 Länge, 22

 leer, 23

 reverses, 24

Wortproblem

 allgemeine Grammatiken, 149, 152

 kontextfreie Grammatiken, 149, 152

 monotone Grammatiken, 149, 152

Zahlen

 ganze, 6

 natürliche, 6

Zeichen, 22

Zeitkomplexität

 DTM, 182

 NTM, 183

Zweipersonenspiel, 92