Theoretische Grundlagen der Informatik 1
TheGI 1 WS 00/01

Skript von Hans-Jörg Burtschick und Dirk Siefkes
unter Mitarbeit von
Dieter Hofbauer, Arfst Nickelsen, Lars With,
Birgit Schelm und Till Tantau

Revidierte Fassung – Oktober 2000

Inhaltsverzeichnis

1 Zur Vorbereitung

Wir empfehlen, die Einleitung zu Beginn gründlich zu lesen, danach gelegentlich und am Ende noch einmal gründlich, die folgenden Abschnitte dagegen bei Bedarf zu Rate zu ziehen.

1.1 Einleitung

Die ersten Computer wurden mit der Hand programmiert: Frauen stöpselten die Verbindungen in den Schränken voller Röhren und Relais der ENIAC, des ersten US-amerikanischen Rechners. Die Männer hatten anderes zu tun; es war Krieg, zweiter Weltkrieg, man wollte mit den Computern Geschoßbahnen berechnen. Frauen führten damals solche und andere (z.B. kaufmännische) Massenrechnungen in Großbüros nach vorgedruckten Rechenplänen durch, die Berufsbezeichnung „Computer“ ging von ihnen auf die Maschinen über, die ihre Arbeit übernahmen. So beschreibt es Ute Hoffmann in dem Büchlein „Computerfrauen“. Erst nach dem Krieg fing man an, Schaltungen als „Logik“ zu sehen und mit Hilfe von „wahr“, „falsch“ oder 0,1 zu codieren; erst dann konnte man Programme elektronisch speichern und vom Rechner ändern lassen. John von Neumann, der daran maßgeblich beteiligt war, hat der bis heute gängigen „Architektur“ von Rechnern seinen Namen gegeben. Erst damit wurden Programme „Texte“, Zeichenfolgen, die nach festen Regeln konstruiert und modifiziert wurden; erst damit wurden Computer „logische Maschinen“, die mathematischer Behandlung zugänglich waren. In den 50er Jahren entstand die Informatik („Computer Science“), die als wissenschaftliche Disziplin stark von der Mathematik geprägt ist. – In Deutschland verlief die Geschichte etwas anders: Konrad Zuse wollte baustatische Berechnungen auf Maschinen bringen. Er entwickelte elektromechanische Rechner – Modelle sind im Deutschen Technikmuseum Berlin zu sehen – und einen Berechnungsformalismus, den er in Anlehnung an die Schemablätter der Rechnersäle „Plankalkül“ nannte. Als er das Militär um Unterstützung anging, wurde er angestellt, Tragflügelprofile für Bomben zu berechnen; bei seinen Maschinen wurde er, anders als seine amerikanischen Kollegen, nicht unterstützt. Daher wurde er lange Zeit kaum beachtet und hat die frühe Entwicklung nicht beeinflußt.

„Mathematischer Behandlung zugänglich“ war zunächst nicht viel. Der norwegische Mathematiker Axel Thue hatte schon zu Anfang des Jahrhunderts über „Veränderungen von Zeichenreihen nach gegebenen Regeln“ geschrieben. Ohne das zu kennen, brachte der amerikanische Linguist Noam Chomsky solche Ideen in den 50er Jahren wieder auf. Er wollte die Sätze der natürlichen Sprache mit formalen Regeln erzeugen und analysieren; daher nannte er die Regelsysteme „Grammatiken“, die Zeichen „Buchstaben“, die Zeichenreihen „strings“ (übersetzt als „Wörter“), die erzeugbaren strings „Sätze“, die Wortmengen „Sprachen“. Er sah später ein, daß in den natürlichen Sprachen das Aufbauen und Verändern von Sätzen vom Verstehen, also von der Bedeutung, nicht zu trennen ist, eine „formale“ Behandlung, bei der es nur um die Zeichenfolge, also die „Form“ der Sätze geht, nicht viel

bringt. Seine Kollegen aus der Computer Science, mit denen er am M.I.T. (Massachusetts Institute of Technology) zusammenarbeitete, wendeten seine Theorie auf die Zeichenreihen ihrer Programme an, die ja nach formalen Regeln konstruiert und modifiziert werden. So entstand die „Theorie der Formalen Sprachen“; die linguistischen Namen blieben, weil sie so schön natürlich klingen. Die Regelsysteme heißen jetzt „Chomsky-Grammatiken“, gelegentlich „Semi-Thue-Systeme“ (semi – halb, weil Thues Regeln beidseits anwendbar waren).

Zur gleichen Zeit wurden sogenannte „endliche Automaten“ als einfachste mathematische Computermodelle verwendet. Chomsky führte sie zur Analyse von „Sprachen“ ein. Sie stammen aber als Schaltkreismodelle aus der Elektrotechnik und wurden schon in den 40er Jahren zur Darstellung von Nervennetzen verwendet. Die zugkräftige Metapher „Elektronengehirn“ legte nahe, Computer mathematisch als endliche Automaten anzusehen. Als zweite mathematische Grundlage der Informatik entstand so die „Automatentheorie“ (die besser „Theorie der endlichen Automaten“ heißen würde). „Automatentheorie und Formale Sprachen“ war viele Jahre lang der offizielle Name der Fachgebiete der beiden Professoren für Theoretische Informatik an der TU Berlin, als beide längst andere Dinge betrieben.

Die Soziologin Bettina Heintz beschreibt in ihrem Buch „Die Herrschaft der Regel“, wie in der ersten Hälfte des 20. Jahrhunderts eine zunehmende Begeisterung für Formalisierung und Rationalisierung alle Bereiche der westlichen Zivilisation durchzog, insbesondere die Wirtschaft und die Mathematik selber. Alle Arbeit sollte bis ins kleinste vorweg geplant, geregelt und berechnet, alles Rechnen formal, nach Regeln und ohne Anschauung, durchgeführt werden. Im Kampf um Sicherheit und Effizienz wollte man menschliche Unvollkommenheit eliminieren und so menschliche Tätigkeiten (und das heißt natürlich: die tätigen Menschen) besser beherrschen. In der Mathematik kulminierte die Entwicklung 1936, als eine ganze Reihe von Mathematikern und Logikern an verschiedenen Orten sogenannte „Berechnungsformalismen“ veröffentlichten, die sich alle als gleichwertig erwiesen. Man nannte sie deswegen „universell“: Kalküle zur Berechnung von allem, was sich überhaupt berechnen läßt. (1936 drehte Charly Chaplin den Film „Modern Times“, in dem er die lächerliche Trostlosigkeit automatisierter Arbeit zeigt. 1936 fand die Olympiade in Berlin statt, in der man eine vollkommene Beherrschung des Körpers und damit eine „unmenschliche Vollkommenheit“ der Bewegungsabläufe anstrebte, wie Hilmar Hoffmann in „Mythos Olympia“ schreibt.) Zwei der Berechnungsformalismen fallen aus dem mathematischen Rahmen: Der engliche Mathematiker Alan Turing nennt seinen „Computer“ eine Maschine, der amerikanische Logiker Emil Post vergleicht seinen mit einem Fließbandarbeiter; trotzdem sind beide praktisch gleich, man kann mit ihnen auf einem endlosen Band durch Lesen, Löschen und Schreiben von Zeichen beliebige Berechnungen simulieren. Obwohl Turing während des zweiten Weltkrieges für den britischen Geheimdienst Maschinen zur Code-Entschlüsselung baut und verwendet, bleibt die „Theorie der Berechenbarkeit“ zunächst mathematisch und dringt erst in den 60er Jahren mit der sich entwickelnden „Komplexitätstheorie“ in die Informatik ein. Für die Bestimmung von Zeit- und Platzaufwand für Berechnungen und damit der „Komplexität“ von Problemen hat sich die „Turingmaschine“ als Modell durchgesetzt. Maschinen zu beherrschen ist eben netter als Fließbandarbeiter. Wissenschaftler sind so ins Beherrschen verstrickt, daß sie Frauen dabei nicht brauchen können; daher ist Wissenschaft gleichzeitig „männlich“ und geschlechtslos. Es lohnt sich, das Buch „Liebe, Macht und Erkenntnis“ der Biophysikerin Evelyn Fox Keller zu lesen.

Mit formalen Grammatiken, endlichen Automaten, Turingmaschinen und Komplexität hat also die Theoretische Informatik als mathematische Theorie der Informatik angefangen. Damit beschäftigen wir uns in TheGI 1. Man kann sich streiten – und streitet sich –, welche Rolle diese Gegenstände heute in der Praxis der Informatik spielen. Aber das ist nicht so wichtig: Theorien sollte man nicht einfach daraus rechtfertigen, daß sie irgendwelchen Praktiken entsprechen; wesentlich ist, daß sie verstehen helfen, was wir tun, und uns so aufmerksamer und sicherer machen. Formale Grammatiken, endliche Automaten und Turingmaschinen sind einerseits recht konkrete Gebilde, andererseits einfache mathematische Systeme, mit denen man das formale Arbeiten mit Zeichenketten anschaulich und präzise fassen kann. Das ist nur ein Aspekt im Verstehen von Programmieren, der aber vielfach im Vordergrund steht; über andere Aspekte schreibe ich am Ende der Einleitung. – In TheGI 2 wird es dann abstrakter: Maschinen sollen funktionieren, also bestimmte Funktionen erfüllen – genau wie vorher die Menschen, deren Arbeit sie übernehmen. Um Funktionen als Ausgangspunkt und Ergebnis von Berechnungen geht es in TheGI 2. In TheGI 3 schließlich wird mathematische Logik als ein besonders reichhaltiger Formalismus zum Beschreiben von Problemen, Berechnungen und Beweisen eingeführt.

Mit Grammatiken und Automaten bzw. Maschinen kann man Zeichenreihen nach formalen Regeln erzeugen, erkennen oder verändern. Ich stelle jetzt etwas genauer dar, um was es dabei geht. An einem Beispiel, das von Chomsky stammt, will ich zuerst das Arbeiten mit Regeln verdeutlichen.

Chomsky folgend nennen wir jede endliche nichtleere Menge (Ansammlung) von Zeichen *Alphabet*; die Zeichen heißen auch *Buchstaben* oder *Symbole*. Aus Buchstaben bilden wir *Wörter* durch Hintereinanderschreiben (Aufreihen). Eine *Wortregel*, geschrieben als $u \to v$, erlaubt uns, in beliebigen Wörtern das Teilwort u, wo wir es finden, durch v zu ersetzen. Betrachten wir beispielsweise das folgende System von sechs Regeln, wie Chomsky sie benutzt hat:

$$\begin{array}{ll} S \to N\,V & V \to \text{machen}\,A \\ N \to \text{Kleider} & N \to \text{Kinder} \\ A \to \text{Leute} & A \to \text{Unsinn} \end{array}$$

Dabei stehen N, V, A für *noun phrase*, *verb phrase*, *attribute phrase*. Ein solches System heißt auch *Grammatik*, eine Folge von Regelanwendungen *Ableitung*. Mit den Regeln kann man aus dem „Startsymbol“ S (für „Sentence“) die Sätze

Kleider machen Leute   Kinder machen Unsinn

erzeugen, allerdings auch

Kleider machen Unsinn   Kinder machen Leute

Drehen wir die Pfeile in Regeln um, so kommen wir von den erzeugten Sätzen, aber von keinen anderen, zurück zum Startsymbol, wir *erkennen* also bestimmte Zeichenfolgen als erzeugbar. Man nennt das auch das *Wortproblem*: Bei vorgelegten Wörtern zu erkennen, ob sie von einer gegebenen Grammatik erzeugbar sind. Manche werden solche Spielchen aus dem Sprachunterricht kennen, vielleicht mit farbigem Unterstreichen zum Analysieren statt abstrakter Regeln.

Auf dieselbe Weise kann man die Form von gewissen Konstrukten in Programmiersprachen durch Regeln festlegen: Nur die erzeugbaren Ausdrücke sind als Zahlen, Namen und so weiter zulässig, andere sind „syntaktisch inkorrekt“. Umgekehrt kann man Programmtexte einer formalen „Syntaxanalyse“ unterziehen: Nur die mit den umgekehrten Regeln erkennbaren Wörter sind syntaktisch korrekt. So erzeugt man mit den Regeln

$$\underline{N} \to \mathrm{A}\underline{K}\,,\ \dots\,,\ \underline{N} \to \mathrm{Z}\underline{K}\,,\ \underline{K} \to \mathrm{a}\underline{K}\,,\ \dots\,,\ \underline{K} \to \mathrm{z}\underline{K}\,,\ \underline{K} \to \mathrm{a}\,,\ \dots\,,\ \underline{K} \to \mathrm{z}$$

„Namen“, Wörter mit großen Anfangsbuchstaben, indem man sie von links nach rechts schreibt; beim Erkennen mit den umgedrehten Regeln liest man von rechts nach links und radiert dabei. $\underline{N}$ steht dabei für „Name“ – unterstrichen, um es von dem Großbuchstaben N zu unterscheiden, $\underline{K}$ steht für „Kleinbuchstaben“, ebenfalls unterstrichen.

Alle erzeugbaren Wortmengen sind dabei erkennbar und umgekehrt; man muß ja nur die Regeln umdrehen. Schränkt man sich auf bestimmte Typen von Regelsystemen ein, geht das im allgemeinen nicht mehr, und es wird schwierig oder unmöglich, zu erzeugenden Regeln erkennende desselben Typs zu finden.

Deswegen geht man in der Praxis (und daher in der Theorie) meist anders vor: Man sucht nach Verfahren (Algorithmen), die das Wortproblem lösen, indem sie Ableitungen der vorliegenden Grammatik konstruieren, die das gegebene Wort erzeugen, oder feststellen, daß das Wort nicht erzeugbar ist. Diese Verfahren arbeiten i.a. mit den ursprünglichen (nicht den umgedrehten) Regeln. In der Theorie heißen sie *Automaten* oder *Maschinen*. Allerdings sind sie teilweise nicht *deterministisch*, d.h. der jeweils nächste Schritt ist nicht eindeutig bestimmt, wie man das von ordentlichen Maschinen erwartet. Auch müssen sie nicht immer *terminieren*, d.h. das Verfahren beenden.

Die Aufgabe ist dann, zu einer gegebenen Grammatik einen *äquivalenten* Automaten zu konstruieren, d.h. einen, der gerade die Wörter erkennt, die man mit der Grammatik erzeugen kann. Natürlich wird die Frage auch umgedreht: Wie sieht eine Grammatik aus, mit der man die Sprache erzeugt, die ein gegebener Automat erkennt. Im Lauf der Zeit hat man eine ganze Reihe von *Grammatiktypen* aufgestellt, die aus unterschiedlichen (historischen, theoretischen und praktischen) Gründen von Interesse sind. Zu jedem Grammatiktyp gibt es einen (oder mehrere) passenden *Automatentyp*. Diese Typen von Grammatiken und Automaten und dadurch gegebenen Sprachen bilden eine ganze *Hierarchie* von immer mächtigeren Typen, die allgemein *Chomsky-Hierarchie* heißt. Mit ihr werden wir uns im wesentlichen beschäftigen. Am Schluß analysieren wir den *Zeit- und Platzaufwand* von Automatenberechnungen und führen damit in die *Komplexitätstheorie* ein.

Das Ziel der LV ist aber nicht, die Chomsky-Hierarchie und das P-NP-Problem darzustellen. Es geht darum, mit den ungewohnten mathematischen Objekten der Theoretischen Informatik wie Automaten und formalen Grammatiken umgehen zu lernen und dabei mit Begriffspaaren wie Erzeugen/Akzeptieren, Determinismus/Nichtdeterminismus, Akzeptieren/Entscheiden vertraut zu werden. Dazu sind die mathematischen Inhalte ein Hilfsmittel.

Gibt es denn keine anderen Grundlagen der Informatik als die mathematischen, mit denen wir uns in TheGI befassen? Sicher, aber sie spielen im Studium, insbesondere im Grundstudium, kaum eine Rolle. In TechGI geht es um physikalische, insbesondere elektronische Grundlagen, in MafI um klassische Mathematik; ansonsten steht der Rechner und seine Programmierung im Mittelpunkt. Informatik = Computer Science!? In dieser Einleitung habe ich andere Themen anklingen lassen: Als Informatiker ersetzen wir menschliche Tätigkeiten durch Maschinenvorgänge. Dabei verändern sich nicht nur die ersetzten, sondern alle drumherum; in der Regel wird durch Computerisierung der ganze Betrieb umgestellt. Wie wir durch Entwicklung und Einsatz von Softwaresystemen die Realität umkonstruieren, ist das Thema der Beiträge in dem Band „Software Development and Reality Construction“ von Christiane Floyd. Wir müßten also nicht nur Rechner und Programme konstruieren, sondern Arbeitszusammenhänge verstehen und gestalten können. „Analyse und (Re-) Organisation der Arbeit mit Hilfe informationstechnischer Mittel“ nennt Wolfgang Coy in dem Sammelband „Sichtweisen der Informatik“ als ersten Gegenstand der Informatik. Arno Rolf vergleicht in demselben Band die Informatiker mit Architekten, weil sie wie diese verantwortlich sind, daß Menschen sich in der von ihnen gestalteten Umgebung wohlfühlen. Arbeitssoziologie, -psychologie, -pädagogik sollte daher im Informatikstudium einen breiten Raum einnehmen; der ist aber höchstens im Nebenfach vorhanden. An anderer Stelle in dem Band schreibt Frieder Nake (das sind alles Informatiker), „daß Arbeit (oder allgemeiner: Tätigkeit), soll sie auf Maschinen übertragen werden, von Menschen bereits maschinenähnlich durchgeführt werden muß.“ In den vergangenen 200 Jahren hat sich das menschliche Leben durch die Maschinisierung völlig gewandelt. Als Informatiker bringen wir geistige Tätigkeiten auf die Maschine. Wie wird diese „Maschinisierung von Kopfarbeit“ (Nake) Menschen erst verändern? Damit beschäftige ich mich in dem Band „Formale Methoden und Kleine Systeme – Leben, lernen und arbeiten in formaler Umgebung“. Sollten Informatiker nicht zuerst Psychologie und Philosophie studieren, bevor sie auf das Umgestalten geistiger Prozesse losgelassen werden? Geist kommt nur in sprachlicher Form auf die Maschine, in der Informatik wimmelt es daher von formalen Sprachen, die bis in die Mathematik hineingeraten, wie wir oben sahen. Also gehören Linguistik und Literaturwissenschaft in den Studienplan.

Natürlich kann niemand so viel studieren. Als Opa wäre ich noch nicht fertig, und hätte vor allem Ballast eingesammelt: Verbindungen zwischen verschiedenen Studiengängen sind nicht so einfach herzustellen. Aber die Augen und Ohren offenhalten, ist schon viel wert;

verschiedene Sichtweisen in, auf und aus Informatik wahr- und einnehmen können; Dozenten fragen, wenn die Lehrveranstaltungen eng bleiben; mit anderen Studenten reden; Bücher lesen; Anwendungsprobleme nicht einfach „lösen“ (wohin fallen sie dann?), sondern als Probleme sehen. Wer nur auf dem technisch/formalen Auge sieht, ist so gut wie blind. Literatur dazu findet sich in Kapitel 13.

1.2 Mathematische Texte: Geheimnisse und Wunder?

Bevor wir mit dem (tonnenschweren und knochentrockenen) „Stoff“ loslegen, möchten wir etwas auf die Art und Weise eingehen, wie MathematikerInnen Texte schreiben. Wer bereits etwas Übung darin hat, mathematische Texte zu lesen, der/die kann diesen Abschnitt überspringen.

Mathematische Texte sind auf lustige Art strukturiert:

Das Schlüsselwort „Definition“ zeigt an, daß dort Dinge benannt werden, im folgenden Text wird dann die eingeführte Benennung verwendet.

Das Wort „Lemma“ ist keine orthographisch mißglückte biologische Bezeichnung, sondern zeigt an, daß hier ein „Hilfssatz“ dargestellt (und meist bewiesen) wird. Das sind Aussagen, die entweder zu einfach sind, als daß sie die Ehre und den Rang eines Satzes genössen, oder sie werden in einem Beweis verwendet und sind so kompliziert, daß sie eines eigenen Beweises bedürfen.

Mit „Beweis“ wird eine Art Überredungsversuch angekündigt. Beweise dienen dazu, die LeserInnen von der Richtigkeit der zu beweisenden Aussage zu überzeugen. Sie haben aber auch die Funktion, daß in ihnen klar wird, wie die einzelnen Begriffe und Aussagen zusammenwirken und meist wird erst in Beweisen (und vor allem während des Beweisens) klar, warum diese Begriffe und Aussagen so sind, wie sie sind. Für mich ist „Beweisen“ ein Teil meiner Arbeit, bei der ich

- meine Gedanken ordne,
- Fehler finde,
- auf neue Ideen komme und
- mit anderen darüber rede.

Definition 1.1
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

Definition 1.2
bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

Lemma 1.3

bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

Beweis
Offenbar gilt bla bla bla. Trivialerweise bla bla bla. Klar ist auch bla bla. O.b.d.A. bla bla bla bla. Selbstverständlich bla bla bla. □

Satz 1.4

bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

Beweis
Wie man leicht sieht bla bla. Offensichtlich gilt bla bla. Analog folgt ähnlich bla bla. Einfache Rechnung zeigt bla bla. Sicherlich gilt bla bla bla. Rest durch Induktion. □

Korollar 1.5

bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

Damit die LeserInnen wieder aufwachen bzw. wissen, wo sie weiterlesen können (wenn sie den Beweis überlesen...) wird das Ende eines Beweises meist durch ein □ am rechten Rand der Seite angezeigt. Statt „Satz“ wird oft auch „Theorem“ verwendet. In jedem Fall sind damit aber Aussagen gemeint, die der Autor oder die Autorin für wichtig hält und die eines

Beweises bedürfen. „Korollare“ sind Sätze, die so nebenbei aus einem schon bewiesenen Satz folgen. Ein Korollar ist ein Korollar, wenn es keinen eigenen Beweis braucht.

Im Laufe dieses Semesters gibt es noch viele Gelegenheiten, das Lesen und Schreiben mathematischer Texte zu üben. Die mathematische Art zu schreiben ist sehr speziell und in keinem anderen Kontext sinnvoll. Mehr zum Schreiben und Lesen mathematischer Texte findet Ihr in dem Buch „Das ist o.B.d.A. trivial!“ von Albrecht Beutelspacher. Auf jeden Fall solltet Ihr Euch von dieser merkwürdigen Art, Gedanken aufzuschreiben, nicht abschrecken lassen.

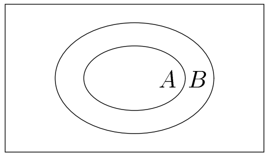
1.3 Menge, Funktion, Relation

Die „richtigen“ MathematikerInnen finden es meistens „schön“ oder „elegant“, wenn etwas kurz und kryptisch dasteht. Dazu verwenden sie u.a. Mengenschreibweisen.

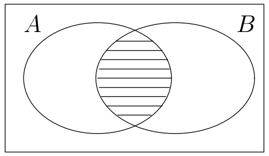
Menge: Wir verwenden in diesem Skript Mengen in ihrem intuitiven Sinn: als Zusammenfassung von Objekten aus einem vorgegebenen Bereich („Universum“), der selber eine Menge ist. Die Reihenfolge der Objekte ist dabei egal und Objekte können immer nur einmal vorkommen. Man kann eine Menge auf zwei Arten aufschreiben: Am einfachsten schreibt man alle Elemente der Menge in geschweifte Klammern, zum Beispiel: $\{1, 2, 3\}$. Da die Reihenfolge egal ist, könnte man auch schreiben $\{2, 1, 3\}$. Bei der zweiten Art, Mengen aufzuschreiben, schreibt man $\{x \mid \text{bla bla}\}$ und meint damit die Menge aller Dinge x für die „bla bla“ zutrifft. So ist $\{x \mid x \text{ ist schön}\}$ die Menge aller schönen Dinge.

Die einfachste unendliche Menge besteht aus den *natürlichen Zahlen* $0, 1, 2, \ldots$; wir bezeichnen sie mit $\mathbb{N}$. Mit $\in$ drücken wir aus, daß etwas in einer Menge enthalten ist: $0 \in \mathbb{N}$. Die 0 gehört bei uns mit dazu, sonst schreiben wir $\mathbb{N}^+$. Mit $\notin$ wird dargestellt, daß etwas nicht in der Menge enthalten ist: $3{,}1415 \notin \mathbb{N}$, $0 \notin \mathbb{N}^+$. Merkwürdig, aber nützlich (wie die 0 beim Rechnen) ist die *leere Menge* $\emptyset$, die keine Elemente enthält. Beim Umgang mit Mengen benutzen wir folgende Bezeichnungen; dabei seien A und B irgendwelche Mengen:

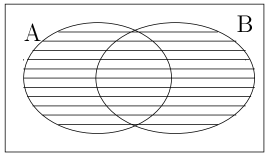
Teilmenge: $A \subseteq B$: wenn $x \in A$, dann $x \in B$.



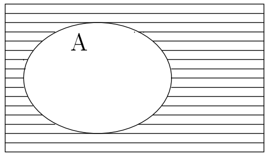
Durchschnitt: $A \cap B = \{x \mid x \in A \text{ und } x \in B\}$



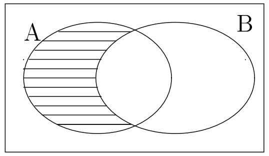
Vereinigung: $A \cup B = \{x \mid x \in A \text{ oder } x \in B\}$



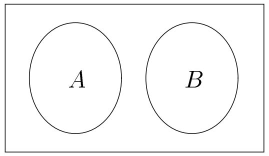
Komplement[1]: $\overline{A} = \{x \mid x \notin A\}$



Differenz: $A \setminus B = A \cap \overline{B} = \{x \in A \mid x \notin B\}$



Disjunkt: A und B sind disjunkt, wenn $A \cap B = \emptyset$



In den Bildchen für Durchschnitt, Vereinigung und Differenz ist ein besonderer Fall dargestellt: A und B überlappen sich. Wie sehen die Bilder jeweils aus, wenn A und B disjunkt sind, oder eine Menge Teilmenge der anderen ist?

Gleichheit: Zwei Mengen A und B sind genau dann gleich, wenn $A \subseteq B$ *und* $B \subseteq A$. Wenn die Gleichheit für zwei Mengen gezeigt werden soll, so müssen immer beide Inklusionen verifiziert werden!

Quantoren: Die Zeichen $\forall$ (Allquantor) und $\exists$ (Existenzquantor) werden häufig als Abkürzung verwendet. Der Allquantor wird für Aussagen der Form „*alle* Elemente einer Menge besitzen eine bestimmte Eigenschaft“ verwendet, der Existenzquantor für Aussagen wie „*es existiert* ein Element (in einer Menge) mit einer bestimmten Eigenschaft“. Beispiele dazu findet Ihr in Aufgabe 2 im Anschluß an dieses Kapitel.

[1]Das Komplement einer Menge wird immer bezogen auf das jeweilige Universum gebildet (im Bild ist das der eckige Kasten), d.h. zu $\overline{A}$ gehören keine beliebigen Objekte, sondern nur Objekte aus dem Universum, die nicht zu A gehören.

Kardinalität: Bei endlichen Mengen A bezeichnet $|A|$ die Anzahl der Elemente in A. Dies nennt man auch die Kardinalität der Menge.

Produkt: $A \times B = \{(a, b) \mid a \in A,\ b \in B\}$ ist die Menge der Paare mit erster Komponente aus A, zweiter Komponente aus B.

n-faches Produkt für $n \in \mathbb{N}^+$: $A^n = \underbrace{A \times \cdots \times A}_{n\text{-mal}}$ ist die Menge der n-Tupel über A. Das kann durch $A^1 = A$ und $A^{n+1} = A^n \times A$ rekursiv definiert werden; manchmal wird $A^0 = \{()\}$ benutzt, diese Menge enthält nur das 0-Tupel. Im Gegensatz zu Mengen ist bei Tupeln die Reihenfolge der Elemente festgelegt.

Potenzmenge: $\mathcal{P}(A) = \{B \mid B \subseteq A\}$ ist die Menge aller Teilmengen von A.

Funktion: Eine *Funktion f von A nach B* für Mengen A, B ist durch eine Vorschrift gegeben, bei der *Argumenten* aus A *Werte* aus B zugeordnet werden; wir schreiben

$$f\colon A \to B \quad \text{und} \quad f(a) = b \text{ für } a \in A, b \in B$$

und nennen A den *Argument-*, B den *Wertebereich*. Ist $A = A_1 \times \cdots \times A_n$ ein n-faches Produkt, so ist f *n-stellig* und wir schreiben

$$f(a_1, \ldots, a_n) = b \text{ für } a_1 \in A_1, \ldots, a_n \in A_n,\ b \in B.$$

Ist f nicht für alle Argumente *definiert* (d.h. ein Wert festgelegt), nennen wir f *partiell*, sonst *total*. Zum Beispiel sind Rechenvorschriften oft nicht überall definiert, wie die Division. (Warnung: Nicht alle Vorschriften sind Rechenvorschriften, nicht alle Funktionen sind daher berechenbar. Das sehen wir in Kap. 11.)

Relation: Eine *Relation* R zwischen zwei Mengen A und B ist durch eine Beziehung zwischen Elementen aus A und aus B gegeben; wir schreiben

$$aRb$$

für $a \in A$, $b \in B$. Eine Relation ist also eine Teilmenge $R \subseteq A \times B$, wir schreiben daher auch

$$(a, b) \in R \quad \text{oder} \quad R(a, b)$$

statt aRb. Ist $A = B$, nennen wir R eine Relation *auf* A. Eine Relation auf A ist

- *reflexiv*, falls aRa für alle $a \in A$ gilt.
- *transitiv*, falls für alle $a, b, c \in A$ gilt: wenn aRb und bRc, dann aRc.

Allgemeiner ist eine *n-stellige Relation* für $n \in \mathbb{N}$ eine Teilmenge eines n-fachen Produkts $A_1 \times \cdots \times A_n$. Eine einstellige Relation ist also einfach eine Teilmenge. Eine Relation $R \subseteq A \times B$ ist *rechtseindeutig*, falls für alle $a \in A$, $b, c \in B$ gilt:

$$\text{wenn } aRb \text{ und } aRc, \text{ dann } b = c.$$

Dann ist durch R eine (partielle) Funktion $f\colon A \to B$ gegeben mit

$$f(a) = b, \text{ falls } aRb.$$

Manche fassen daher Funktionen als spezielle Relationen und Relationen als Mengen auf und führen alles auf Mengen zurück; andere fangen umgekehrt mit den Funktionen an. Für uns spielt das keine Rolle.

1.4 Alphabet, Wort, Sprache

Nachdem im vorigen Abschnitt schon die Mengenschreibweise eingeführt wurde, werden jetzt weitere Sprech- und Schreibweisen vorgestellt, wie sie in diesem Skript verwendet werden. Wer diesen Abschnitt beim ersten Mal schwierig zu lesen findet (vielleicht weil er in so einem „mathematischen Stil" geschrieben ist), sollte sich nicht abschrecken lassen. Im Laufe des Semesters wird durch den Gebrauch der Begriffe ihre Bedeutung klarer werden. Dieser Abschnitt ist also vor allem zum Nachschlagen gedacht.

Alphabet: Jede nichtleere endliche Menge von Zeichen heißt *Alphabet*. Die Zeichen heißen auch *Buchstaben* oder *Symbole*. Meist benutzen wir lateinische Buchstaben oder Ziffern. Alphabete werden oft mit großen griechischen Buchstaben bezeichnet; am häufigsten mit Σ (Sigma) und Γ (Gamma).

Wort: Durch Hintereinanderschreiben (Aufreihen) von Buchstaben bilden wir *Wörter*, Wörter sind also endliche Folgen von Symbolen. Wörter, deren Buchstaben aus Σ sind, nennen wir Wörter *über* Σ. Für $\Sigma = \{0,1\}$ etwa ist $\Sigma^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, \dots\}$ die Menge aller Wörter über Σ. Die Reihenfolge, in der die Wörter von Σ^* hier aufgeschrieben sind, nennt man die *Standardreihenfolge*. Die Wörter sind zunächst der Länge nach geordnet, gleichlange Wörter kommen in der Reihenfolge, in der sie im Telefonbuch stehen würden. Die Länge eines Wortes w (also die Länge der Zeichenfolge) bezeichnen wir mit $|w|$. Es gibt auch ein Wort der Länge 0; es heißt *leeres Wort* und wird mit λ (kleines Lambda) bezeichnet. $|w|_a$ gibt die Häufigkeit des Zeichens a im Wort w an. Σ^* ist immer unendlich, obwohl nur endlich viele Buchstaben zur Verfügung stehen und jedes Wort endliche Länge hat. Mit vw bezeichnen wir die *Verkettung* (oder: *Konkatenation*) der Wörter v und w. Für ein Wort v und eine natürliche Zahl n definieren wir das Wort v^n rekursiv durch

$$v^0 = \lambda, \quad v^{n+1} = v^n v \text{ für } n \geq 0.$$

Wörter der Länge n können wir formal als n-Tupel über Σ definieren; jedes Wort ist damit eindeutig in Buchstaben zerlegbar. Mit Σ^* bezeichnen wir die Menge aller Wörter über Σ, mit Σ^+ bzw. Σ^n bzw. $\Sigma^{\leq n}$ die Menge aller Wörter über Σ, die mindestens ein bzw. genau n bzw. höchstens n Zeichen enthalten:

$$\Sigma^* = \bigcup_{0\leq i} \Sigma^i, \quad \Sigma^+ = \bigcup_{1\leq i} \Sigma^i, \quad \Sigma^{\leq n} = \bigcup_{0\leq i \leq n} \Sigma^i;$$

insbesondere gilt $\Sigma^* = \Sigma^+ \cup \{\lambda\}$.

Das Wort w in umgekehrter Reihenfolge geschrieben ist w^{-1}, z.B. $(abb)^{-1} = bba$. Also gilt

$$\lambda^{-1} = \lambda, \quad (uv)^{-1} = v^{-1}u^{-1}.$$

Sprache, Kleene-Operationen: Sei Σ ein Alphabet. Jede (endliche oder unendliche) Teilmenge von Σ^* heißt *Sprache*. Für zwei Sprachen V und W ist

$$V \circ W = \{vw \mid v \in V, w \in W\}$$

die *Verkettung* von V und W. Das n-fache *Produkt* V^n von V ist rekursiv durch

$$V^0 = \{\lambda\}, \quad V^{n+1} = V^n \circ V \text{ für } n \geq 0$$

definiert; insbesondere ist $V^1 = V$. Das ist nicht mit dem „kartesischen" Produkt aus 1.3 zu verwechseln! Was ist der Unterschied? Schließlich ist der *(Kleene-)Stern* von V die Menge

$$V^* = \bigcup_{0\leq n} V^n.$$

1.5 Aufgaben

Aufgabe 1.1: (Mathematische Schreibweisen, speziell Mengenschreibweisen) In a) bis c) sind Mengen in „geschlossener Mengendarstellung" aufgeschrieben. Beschreibt diese Mengen sprachlich, wie z.B. in d) bis f).

a) $\{2n \mid n \in \mathbb{N}\}$ b) $\{2^k \mid k \in \mathbb{N}\}$ c) $\{n^k \mid n, k \in \mathbb{N} \text{ und } 1 < k \leq 3\}$

In d) bis f) sind Mengen sprachlich dargestellt; beschreibt sie in geschlossener Darstellung.

d) Die Menge aller natürlichen Zahlen, die größer als 4 sind.

e) Die Menge der Quadratwurzeln von ungeraden natürlichen Zahlen.

f) Die Menge der Potenzen mit gerader Basis und ungeradem Exponenten.

In einigen Fällen ist es einfacher, die Mengen in „Pünktchenschreibweise" darzustellen. Diese Schreibweise sollte nur dann verwendet werden, wenn die Darstellung nicht mißverständlich ist. Beschreibt die folgenden Mengen in geschlossener Darstellung.

g) $\{0, 1, 2, 3, 4, \dots, 10\}$ h) $\{0, 2, 4, 6, 8, 10, \dots\}$

Die folgenden Darstellungen sind mehrdeutig oder unverständlich. Beschreibt zu j) und k) jeweils mindestens zwei Mengen, die gemeint sein könnten und gebt jeweils für eine dieser

Mengen eine geschlossene Darstellung oder eine weniger mißverständliche Darstellung mit Pünktchen an. Bei l) ist es unwahrscheinlich, daß Ihr eine geschlossene Darstellung findet.

j) $\{3, 5, 7, \dots\}$ k) $\{0, 2, 4, 6, \dots, 99\}$ l) $\{10, 13, 60, 1039, 26254, 805265, 4826819, \dots\}$

Aufgabe 1.2: (Quantoren) Gelten die folgenden Aussagen für alle natürliche Zahlen?

a) $\forall x\colon x > 0$ b) $\exists x\colon x > 0$ c) $\forall x\colon x \geq 0$
d) $\exists x \forall y\colon x + y = x$ e) $\forall x \exists y\colon x + y = x$

Manchmal werden Quantoren auch in Mengendarstellungen verwendet. Beschreibt die folgenden Mengen sprachlich oder in geschlossener Darstellung ohne Quantoren.

f) $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N}\colon n = 2k\}$

g) $\{\{n, k\} \mid n, k \in \mathbb{N} \text{ und } \exists m \in \mathbb{N}\colon n = k \cdot m\}$

h) $\{\{n, k\} \mid n, k \in \mathbb{N} \text{ und } 1 < k < n \text{ und } \exists m \in \mathbb{N}\colon n = k \cdot m\}$

Aufgabe 1.3: (Komplement, Durchschnitt, Vereinigung und Differenz) Beschreibt die folgenden Mengen sprachlich oder in geschlossener Darstellung.

a) $\{2n \mid n \in \mathbb{N}\} \cap \{3n \mid n \in \mathbb{N}\}$
b) $\{2n \mid n \in \mathbb{N}\} \cup \{2n + 1 \mid n \in \mathbb{N}\}$
c) $\{2n \mid n \in \mathbb{N}\} \cap \{4n \mid n \in \mathbb{N}\}$
d) $\{2n \mid n \in \mathbb{N}\} \cup \{4n \mid n \in \mathbb{N}\}$
e) $\overline{\{2n \mid n \in \mathbb{N}\}}$
f) $\{2n \mid n \in \mathbb{N}\} \setminus \{4n \mid n \in \mathbb{N}\}$
g) $\{2n \mid n \in \mathbb{N} \setminus \{0, 1\}\}$

Aufgabe 1.4: (Tupel, kartesisches Produkt, Relation, Potenzmenge, leere Menge) Welche Gleichheiten gelten?

a) $\{1, 2, 3\} = \{3, 1, 2\}$ b) $(1, 2, 3) = (3, 1, 2)$

Schreibt alle Elemente der folgenden Mengen auf.

c) $R = \{0, 1\} \times \{0, 2, 3\}$
d) $\{n \in \mathbb{N} \mid (n, n) \in R\}$
e) $\{x \in \mathbb{N} \mid \exists y\colon (x, y) \in R\}$
f) $\{y \in \mathbb{N} \mid \exists x\colon (x, y) \in R\}$
g) $\{(0, y) \mid y \in \mathbb{N} \text{ und } (0, y) \in R\}$
i) $\mathcal{P}(\{\emptyset\})$
j) $\mathcal{P}(\emptyset)$
h) $\mathcal{P}(\{0, 1, 2\})$

Aufgabe 1.5: (Kardinalität von Mengen) Berechnet die folgenden Anzahlen für $A = \{0, 1, 2\}$.

a) $|\mathcal{P}(A)|$ b) $|A \times A|$ c) $|\{T \mid T \subseteq A \text{ und } |T| = 2\}|$

Es seien $n, k \in \mathbb{N}$ und $A_n = \{0, 1, 2, \dots, n-1\}$, also gilt $|A_n| = n$. Berechnet:

d) $|\mathcal{P}(A_n)|$ e) $|A_n^k|$ f) $|\{T \mid T \subseteq A_n \text{ und } |T| = k\}|$

Aufgabe 1.6: (Relation vs. Funktion) Welche Relationen sind Funktionen von $\mathbb{N}$ nach $\mathbb{N}$?

a) $\mathbb{N} \times \{1\}$
b) $\mathbb{N} \times \{0, 1\}$
c) $\{(n, k) \mid n, k \in \mathbb{Z} \text{ und } n = k^2\}$
d) $\{(k, n) \mid n, k \in \mathbb{N} \text{ und } n = k^2\}$
e) $\{(n, k) \mid n, k \in \mathbb{N} \text{ und } n = k^2\}$

Aufgabe 1.7: (a^n-Schreibweise) Beschreibt die folgenden formalen Sprachen über $\Sigma = \{a, b\}$ (natürlich-)sprachlich.

a) $\{a^n b^n \mid n \in \mathbb{N}\}$ b) $\{a^{2n} \mid n \in \mathbb{N}\}$ c) $\{aw \mid w \in \{b\}^*\}$

Beschreibt die folgenden Mengen in geschlossener Darstellung.

d) Die Menge aller Wörter über Σ, in denen niemals ein a nach einem b kommt und in denen die Zahl der a's genau um eins größer als die Zahl der $b's$ ist.

e) Die Menge aller Wörter über Σ, die mit a beginnen und mit b enden.

Aufgabe 1.8: (Formale Sprachen und Mengenoperationen) Beschreibt die folgenden Mengen sprachlich oder in geschlossener Darstellung für $\Sigma = \{a, b\}$.

a) $\{awb \mid w \in \Sigma^*\} \cup \{bwa \mid w \in \Sigma^*\}$

b) $\Sigma^* \setminus \{awa \mid w \in \Sigma^*\}$

c) $\{a^n b^n \mid n \in \mathbb{N}\} \cap \{x_1 x_2 \dots x_n x_n \dots x_2 x_1 \mid n \in \mathbb{N}^+ \text{ und } x_i \in \Sigma \text{ für } 1 \leq i \leq n\}$

Aufgabe 1.9: (Kleene-Operationen) Welche Aussagen gelten für $\Sigma = \{a, b\}$ und $\Gamma = \{0, 1\}$?

a) $\Sigma^* \cup \Gamma^* = (\Sigma \cup \Gamma)^*$ b) $\Sigma^* \cap \Gamma^* = \emptyset$ c) $\Sigma^+ \cap \Gamma^+ = \emptyset$

Aufgabe 1.10: (Mengenoperationen und Konkatenation) Es sei Σ ein endliches Alphabet. Welche der folgenden Aussagen gelten für alle Sprachen $L_1, L_2, L_3 \subseteq \Sigma^*$? (Hinweis: Zwei Aussagen sind falsch!)

a) $L_1 \cap L_2 = L_2 \cap L_1$
b) $L_1 \circ L_2 = L_2 \circ L_1$
c) $L_1 \circ (L_2 \cup L_3) = (L_1 \circ L_2) \cup (L_1 \circ L_3)$
d) $L_1 \circ (L_2 \cap L_3) = (L_1 \circ L_2) \cap (L_1 \circ L_3)$

2 Reguläre Grammatiken und Sprachen

Wir betrachten in dieser Lehrveranstaltung Grammatiken zur Erzeugung von Wörtern formaler Sprachen. Wir müssen in einer Grammatik festlegen, aus welchen Symbolen die Wörter der erzeugten Sprache bestehen sollen. Außerdem muß die Grammatik die Produktionen (Regeln) enthalten, nach denen die Wörter der Sprache (durch Ersetzen) gebildet werden sollen.

Die von uns betrachteten Grammatiken sind um zusätzliche Bestandteile erweitert, die es gestatten, den Ableitungsprozeß zu steuern. So wollen wir in diesem Kapitel nur Grammatiken betrachten, in deren Ableitungen jeweils nur das letzte Zeichen eines Wortes ersetzt werden soll. Die Grammatiken enthalten deshalb eine zweite Sorte von Symbolen („nonterminale Symbole“), und die Produktionen werden so definiert, daß nur die nonterminalen Symbole ersetzt werden. (Ein Symbol darf nicht gleichzeitig Terminalsymbol und Nonterminalsymbol sein, d.h. die entsprechenden Alphabete müssen disjunkt sein.) Außerdem wird ein nonterminales Symbol als Startsymbol festgelegt.

Wir bezeichnen das Alphabet, das die terminalen Symbole enthält, mit T. Das Alphabet, das die nonterminalen Symbole enthält, nennen wir N. Für das Startsymbol verwenden wir S. Die Menge der Produktionen nennen wir P. Formal betrachtet ist P eine Relation auf $(T \cup N)^*$. Für eine Produktion $(u, v) \in P$ verwenden wir die Schreibweise $u \to v$ und nennen u die linke Seite der Regel und v die rechte Seite der Regel.

Damit nur nonterminale Zeichen ersetzt werden, sind alle Produktionen von der Form

$$X \to v \quad \text{mit } X \in N \text{ und } v \in (N \cup T)^*.$$

Um zu erreichen, daß in den Ableitungen jeweils nur das letzte Zeichen ersetzt wird, soll (wenn überhaupt) nur das letzte Zeichen der rechten Regelseiten ein nonterminales Symbol sein. Also sollen die Produktionen die folgende Form haben:

$$X \to vY \text{ oder } X \to v \quad \text{mit } X, Y \in N \text{ und } v \in T^*.$$

Üblicherweise wird in Lehrbüchern $v \in T$ verlangt, d.h. es darf in jedem Schritt nur ein terminales Symbol geschrieben werden. Das ist keine wirkliche Einschränkung, wie wir in Abschnitt 2.4 sehen werden.

2.1 Definition: reguläre Grammatik

Eine reguläre Grammatik G wird durch drei endliche Mengen N, T, P und das Symbol S spezifiziert; wir schreiben auch $G = (N, T, S, P)$. Dabei ist

- N ein Alphabet mit *nonterminalen Symbolen*,
- T ein Alphabet mit *terminalen Symbolen*,
- $S \in N$ das *Startsymbol* und
- P eine Menge von *Produktionen* der Form $X \to v$ oder $X \to vY$ mit $X, Y \in N$ und $v \in T^*$.

Die Alphabete N und T müssen disjunkt sein.

2.2 Schreibweisen

Unsere Beispielgrammatik G_1 soll die Sprache $L_1 = \{a^n b^m \mid n, m \in \mathbb{N}^+\}$ generieren. Wir notieren sie zunächst in der Mengenschreibweise der Definition gemäß als $G_1 = (N_1, T_1, A, P_1)$ mit $N_1 = \{A, B\}$, $T_1 = \{a, b\}$ und

$$P_1 = \{A \to aA, \;\; A \to aB, \;\; B \to bB, \;\; B \to b\}.$$

Übersichtlicher schreiben wir eine solche Grammatik folgendermaßen auf:

$$\begin{aligned} A &\to aA \mid aB \\ B &\to bB \mid \;\; b \end{aligned}$$

Wir verwenden dabei die folgenden Konventionen: Terminale Symbole bezeichnen wir mit Ziffern oder kleinen Buchstaben, nonterminale Symbole mit Großbuchstaben. Dadurch sind die Alphabete implizit festgelegt. Regeln mit gleicher linker Seite schreiben wir in eine Zeile, die verschiedenen rechten Seiten trennen wir dabei durch senkrechte Striche.

2.3 Ableiten, erzeugte Sprache

$$\begin{array}{ll} A & \\ \Big\Downarrow & A \to aA \\ aA & \\ \Big\Downarrow & A \to aA \\ aaA & \\ \Big\Downarrow & A \to aB \\ aaaB & \\ \Big\Downarrow & B \to bB \\ aaabB & \\ \Big\Downarrow & B \to bB \\ aaabbB & \\ \Big\Downarrow & B \to bB \\ aaabbbB & \\ \Big\Downarrow & B \to b \\ aaabbbb & \end{array}$$

Durch eine Produktion $u \to v$ einer regulären Grammatik G kann das Wort wu in das Wort wv transformiert werden. Wir schreiben dafür

$$wu \Rightarrow_G wv.$$

Falls klar ist, welche Grammatik gemeint ist, lassen wir das $_G$ weg (also: $wu \Rightarrow wv$).

Wenn das Wort w in mehreren aufeinanderfolgenden Schritten durch Produktionen aus G zu w' umgeformt werden kann, dann schreiben wir

$$w \Rightarrow_G^+ w'.$$

Wenn $w \Rightarrow_G^+ w'$ oder $w = w'$ ist, dann schreiben wir

$$w \Rightarrow_G^* w'.$$

$\Rightarrow_G$ ist eine Infixnotation für die oben beschriebene Ableitungsrelation. $\Rightarrow_G^+$ ist der *transitive Abschluß* von $\Rightarrow_G$, und $\Rightarrow_G^*$ ist der *reflexive und transitive Abschluß* von $\Rightarrow_G$.

Oft schreiben wir statt G die Nummer der verwendeten Regel unten an den Pfeil; ähnlich oben statt $*$ oder $+$ die Zahl der Schritte.

Eine solche Folge von Produktionen nennen wir eine *Ableitung* von G. Formal können wir Ableitungen „rekursiv" definieren: $w \Rightarrow^* w'$ gdw (kurz für: genau dann wenn) $w = w'$ (kein Schritt) oder es gibt ein Wort v mit $w \Rightarrow v \Rightarrow^* w'$ (mindestens ein Schritt). Die Relation $\Rightarrow^+$ brauchen wir nicht mehr. Wie sähe eine rekursive Definition dafür aus?

Vom Startsymbol ausgehend kann durch eine Grammatik G die Wortmenge

$$SF(G) = \{w \in (T \cup N)^* \mid S \Rightarrow_G^* w\}$$

erzeugt werden. Diese Wörter, die auch nonterminale Zeichen enthalten können, werden oft *Satzformen* genannt. Für unsere Beispielgrammatik wäre beispielsweise $aaabbbb \in SF(G)$, aber auch $aaaB \in SF(G)$. Meist interessieren uns aber nur die Wörter, die ausschließlich terminale Zeichen enthalten. Deshalb definieren wir die von G *erzeugte Sprache* als

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

Eine Sprache $L \subseteq \Sigma^*$ nennen wir *regulär*, wenn es eine reguläre Grammatik G mit $T = \Sigma$ gibt, die L erzeugt, wenn also $L = L(G)$ ist. Wir bezeichnen die Klasse aller regulären Sprachen mit REG.

Wir beweisen jetzt, daß mit unserer Beispielgrammatik G_1 wirklich die Menge

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

erzeugt wird, d.h. daß $L_1 = L(G_1)$ ist. Dazu beweisen wir zuerst $L_1 \subseteq L(G_1)$ (das ist einfach), dann $L(G_1) \subseteq L_1$ (da müssen wir uns mehr Mühe geben). Um Ableitungen aufschreiben zu können, numerieren wir die Regeln von (1) bis (4) durch.

(I) $L_1 \subseteq L(G_1)$, d.h. wenn $w \in L_1$, dann $w \in L(G_1)$: Dazu geben wir zu jedem Wort $w \in L_1$ eine Ableitung $A \Rightarrow^* w$ an. Das ist leicht, denn $w = a^n b^m \in L_1$ mit $n, m > 0$ können wir ableiten, indem wir $(n-1)$-mal die Regel (1), dann einmal die (2), $(m-1)$-mal die (3), und schließlich einmal die (4) anwenden, in Symbolen

$$A \Rightarrow_{(1)}^{n-1} a^{n-1}A \Rightarrow_{(2)} a^n B \Rightarrow_{(3)}^{m-1} a^n b^{m-1} B \Rightarrow_{(4)} a^n b^m.$$

(Dabei heißt „0-mal" bei $n = 1$ oder $m = 1$ „keinmal".) Wenn Ihr genauer sein wollt, könnt Ihr diese Aussagen durch Induktion beweisen, etwa $vA \Rightarrow_{(1)}^k va^k A$ für $v \in T^*$ und $k \geq 0$ durch Induktion nach k. Wir können also jedes Wort in L_1 aus dem Startsymbol ableiten. Und brauchen dazu auch wirklich alle Regeln.

(II) $L(G_1) \subseteq L_1$, d.h. wenn $w \in L(G_1)$, dann $w \in L_1$; oder noch anders formuliert: Jedes ableitbare Wort liegt in L_1. Dazu müssen wir uns eine Übersicht über alle möglichen Ableitungen verschaffen. Das ist bei unserem Beispiel auch leicht: Jede Ableitung beginnt mit Regel (1) oder (2): $A \Rightarrow aA$ oder $A \Rightarrow aB$. Auf ein Wort vA läßt sich nur k-mal $(k \geq 0)$ die (1) und dann (bzw. gleich) einmal die (2) anwenden:

$$vA \Rightarrow_{(1)}^k va^k A \Rightarrow_{(2)} va^{k+1} B.$$

(Das beweist man wieder, wenn gewünscht, mit Induktion nach k.) Analog für B:

$$vB \Rightarrow_{(3)}^k vb^k B \Rightarrow_{(4)} vb^{k+1}.$$

Jede Ableitung, die mit dem Startsymbol beginnt und mit einem Terminalwort endet, hat also die Form $A \Rightarrow^n a^n B \Rightarrow^m a^n b^m$ mit $n, m > 0$. Anders geht's nicht, wir können also nur Wörter aus L_1 ableiten. □

Im allgemeinen wird der Beweis komplizierter, wir müssen mit verschiedenen Satzformen und Teilstücken von Ableitungen arbeiten. Oft brauchen wir dann Induktion, um wirklich sicher zu sein. Aber wir gehen immer auf diese Weise vor.[2]

[2] Für den zweiten Teil hätten wir auch umgekehrt zeigen können: Wenn $w \notin L_1$, dann $w \notin L(G_1)$. Dazu unterscheiden wir für $w \notin L_1$ vier Fälle: $w = \lambda$, $w = a^n$ mit $n > 0$, w beginnt mit b, $w = a^n b^m av$ mit $n, m > 0$ und $v \in \{a, b\}^*$. Bei jeder Ableitung wird zu Beginn ein a und zum Schluß ein b geschrieben (warum?); also ist in den ersten drei Fällen w nicht ableitbar. Der vierte Fall erledigt sich so: Wenn einmal ein a und dann ein b geschrieben wurde, kann nicht wieder ein a kommen (warum nicht?).

Wenn wir das leere Wort zu L_1 dazunehmen, sowie Wörter, die nur aus a's bzw. nur aus b's bestehen (d.h. auch den Fall $n = 0$ oder $m = 0$ zulassen), also zu $\{a^n b^m \mid n, m \in \mathbb{N}\}$ übergehen, dann kommen wir mit „einfacheren“ Grammatiken aus:

$$\begin{array}{lcl} A \to aA \mid bB \mid \lambda & \quad\text{oder}\quad & A \to aA \mid bB \mid a \mid b \mid \lambda \\ B \to bB \mid \lambda & & B \to bB \mid b \end{array}$$

Warum ist das einfacher? Beweist wie oben, daß beide Grammatiken korrekt sind, d.h. die richtige Sprache liefern. Worin unterscheiden sie sich? (Seht Euch die Ableitungslängen an, d.h. die Zahl der Schritte beim Ableiten verschiedener Wörter.) Geht es noch einfacher? Welche Wörter sind nicht mehr ableitbar, wenn Ihr irgendwelche Regeln weglaßt?

2.4 Normalform

Eine reguläre Grammatik ist *in Normalform*, wenn in jedem Schritt nur *ein* terminales Symbol geschrieben werden darf; zugelassen sind außerdem λ-Regeln. Es sind also drei Formen von Regeln erlaubt:

$$X \to yZ, \quad X \to y, \quad X \to \lambda$$

mit $X, Z \in N$, $y \in T$. Verboten sind die „trivialen“ Regeln $X \to Z$ und die „großen“ Regeln $X \to vZ$ mit $|v| > 1$, bei denen nichts bzw. viel auf einmal geschrieben wird. Die sind zwar praktisch nützlich, wie Ihr inzwischen hoffentlich festgestellt habt, aber entbehrlich: Jede große Regel

$$X \to a_1 \ldots a_n Z$$

mit $a_i \in T$ und $n > 1$ läßt sich ersetzen durch n Produktionen

$$X \to a_1 A_1, \quad A_1 \to a_2 A_2, \; \ldots, A_{n-2} \to a_{n-1} A_{n-1}, \quad A_{n-1} \to a_n Z$$

mit neuen nichtterminalen Symbolen $A_1, \ldots, A_{n-1}$.

Interessanterweise machen die trivialen Regeln mehr Mühe. Die ganz trivialen $X \to X$ läßt man einfach weg. Für die anderen, $X \to Z$ mit $X \neq Z$, fügt man zu jeder Regel $Z \to v$ mit $v \in (N \cup T)^*$ (also auch $v \in N!$) die Regel $X \to v$ zur Grammatik hinzu. Das wiederholt man, bis man keine neuen Regeln mehr erhält. Dann kann man die trivialen Regeln weglassen, weil sie überflüssig geworden sind. Der ganze Beweis bleibt als Übungsaufgabe.

Jede reguläre Grammatik in unserem Sinn läßt sich also in diese Normalform bringen. Oft wird für reguläre Grammatiken außerdem gefordert, daß die Produktionen die Länge des abgeleiteten Wortes nicht verkürzen. Dann ist allerdings das leere Wort nicht erzeugbar, und es müssen dafür Sonderregelungen erlassen werden.

2.5 Das Wortproblem für reguläre Sprachen

Wenn man mit einer formalen Sprache L arbeitet, ist es oft wichtig, ein Verfahren (einen Algorithmus) zu haben, mit dem man entscheiden kann, ob irgendwelche Wörter zu der Sprache gehören. Ist L explizit gegeben, kann man die Frage „$w \in L$?“ meist anhand der Form von w beantworten. Zum Beispiel „Ist w von der Form $a^n b^m$?“, oder „Ist w eine Dezimalzahl?“. Ist aber L durch eine Grammatik G mit Startsymbol S erzeugt, also $L = L(G)$, dann wird die Frage „$w \in L$?“ zu „$S \Rightarrow_G^* w$?“. Das ist nicht so schnell zu entscheiden, man müßte ja alle möglichen Ableitungen durchmustern.

Das ist das *Wortproblem für reguläre Sprachen*: Gibt es ein Verfahren, um für eine reguläre Grammatik G und ein Wort w zu entscheiden, ob $w \in L(G)$ ist? Für reguläre Sprachen ist das Wortproblem lösbar, sogar sehr leicht: Wir werden im nächsten Kapitel einen einfachen Mechanismus einführen, sog. *endliche Automaten*, der Wörter „akzeptiert“ oder „verwirft“. Und im Kapitel 5 werden wir zu jeder regulären Grammatik G einen endlichen Automaten konstruieren, der genau die Wörter aus $L(G)$ akzeptiert, also die Frage „$w \in L(G)$?“ korrekt beantwortet. Später werden wir sehen, daß für kompliziertere Typen von Grammatiken das analoge Wortproblem komplizierter oder gar nicht lösbar ist.

Mit regulären Grammatiken kann man Wörter von links nach rechts schreiben, ohne dabei das schon Geschriebene noch einmal ansehen zu müssen: Das einzige Hilfsmittel sind die nonterminalen Zeichen am Ende („cursors“), mit denen man einen „Schreibmodus“ vorgeben oder „modulo“ zählen kann. Typisch regulär sind „Namen“ (ein Groß-, dann Kleinbuchstaben), „Dezimalzahlen“, „identifiers“ wie sie zur Bezeichnung von Variablen, Funktionen oder Prozeduren in Progammiersprachen auftauchen; ebenso Wörter mit fester Länge oder Anzahl bestimmter Zeichen wie „passwords“ oder „Wörter mit einer geraden Anzahl von Einsen“. Es ist wichtig, solche Ausdrucksmittel mit einfachen Verfahren auf Korrektheit überprüfen zu können. Dafür dienen die endlichen Automaten.

2.6 Aufgaben

Von hier ab sind grundsätzlich alle formalen Konstruktionen zu „dokumentieren“ (d.h. Formeln und Diagramme in natürlicher Sprache verständlich zu machen) und alle Aussagen zu beweisen bzw. zu begründen.

Aufgabe 2.1: Welche Sprache erzeugt diese reguläre Grammatik aus S?

$$\begin{array}{l} S \to aA \\ A \to aS \mid a \end{array}$$

Aufgabe 2.2: Es sei $\Sigma = \{a, b\}$ ein Alphabet und $L = \{aa, bab, aabab, bb\}$. Gebt reguläre Grammatiken für die folgenden Sprachen an.

a) L b) Σ^* c) $\Sigma^* \setminus L$

Aufgabe 2.3: Gebt eine reguläre Grammatik für die Menge der Dezimaldarstellungen der geraden natürlichen Zahlen an; benutzt das Alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Die Zahlen sollen ohne führende Nullen dargestellt werden.

Aufgabe 2.4: Gebt reguläre Grammatiken für die folgenden Sprachen über $\Sigma = \{0, 1\}$ an.

$$\begin{aligned}
A &= \{w \in \Sigma^* \mid w \text{ enthält 00 nicht als Teilwort}\}\\
B &= \{w \in \Sigma^* \mid w \text{ enthält 00 mindestens einmal als Teilwort}\}\\
C &= \{w \in \Sigma^* \mid w \text{ enthält 00 höchstens einmal als Teilwort}\}\\
D &= \{w \in \Sigma^* \mid w \text{ enthält 00 genau einmal als Teilwort}\}\\
E &= \{0^n 1 \mid n \in \mathbb{N}\}\\
F &= \{0^n 1^k \mid n, k \in \mathbb{N}\}\\
G &= \{w \in \Sigma^* \mid |w| \text{ ist ungerade}\}
\end{aligned}$$

Aufgabe 2.5: Welche Sprache erzeugt diese reguläre Grammatik aus S?

$$\begin{aligned}
S &\to a \mid aA\\
A &\to bB \mid b\\
B &\to bB \mid a
\end{aligned}$$

Aufgabe 2.6: Welche Sprache erzeugt diese reguläre Grammatik aus S?

$$\begin{aligned}
S &\to b \mid aA\\
A &\to aB \mid a\\
B &\to aS \mid a
\end{aligned}$$

Aufgabe 2.7: Es sei $L \subseteq \Sigma^*$ eine Sprache und $w \in \Sigma^*$ ein beliebiges Wort. Dann ist die *Präfixerweiterung* L_w wie folgt definiert: Sie enthält alle Wörter aus L, nur daß vor diese immer das Wort w geschrieben wird. Es gilt also $L_w := \{wv \mid v \in L\}$.

Zeigt, daß die Klasse der regulären Sprachen unter Präfixerweiterung abgeschlossen ist, d.h. falls L regulär ist und w ein beliebiges Wort, so ist L_w ebenfalls regulär.

3 Endliche deterministische Automaten

Sicher wolltet Ihr schon immer wissen, was Automaten sind. Die folgende Erklärung steht in Dennerts Konversations-Lexikon (Berlin, 3. Auflage 1910), S. 293:

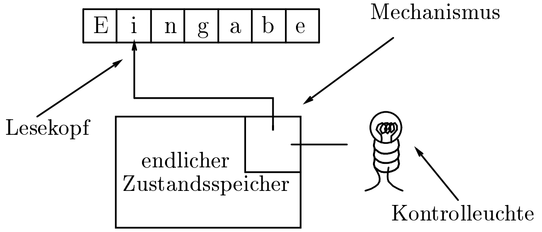
> ***Automat*** *griech., „Selbstbeweger", Lebewesen nachahmende Maschinen; solche sind schon im Altertum und Mittelalter gemacht, heute nennt man alle selbsttätigen Maschinen A., namentlich solche, die zum Verkauf von irgend welchen Waren (z.B. Stollwercks Schokolade) eingerichtet sind, wobei ein eingeworfenes Geldstück von bestimmtem Gewicht durch seine Schwere einen Mechanismus in Bewegung setzt; dabei entsteht eine Öffnung, durch welche jene Ware herausfällt. Auch amtlich werden Automaten jetzt benutzt, z.B. bei der Telephonie (s.d.); s. auch Telegraphie. — Auch mechanisch, puppenhaft handelnde Menschen nennt man A.; —* ***automatisch****, mechanisch, ohne Nachdenken handelnd.*

3.1 Nichtformale Beschreibung von endlichen Automaten

Ich stelle mir also ein (mechanisches oder elektronisches) Gerät vor, das ein Wort als Eingabe erhält, es Buchstabe für Buchstabe liest und am Ende (durch eine Kontrolleuchte) anzeigt, ob es das Wort akzeptiert:

- Das Wort ist auf einem Papierband aufgeschrieben, und der Automat liest es Zeichen für Zeichen von links nach rechts mittels eines *Lesekopfes.*
- Der Automat arbeitet taktweise und liest in jedem Takt ein Zeichen.
- In dem Takt, nachdem das letzte Zeichen gelesen wurde, gibt der Automat das Ergebnis aus.
- In jedem Takt befindet sich der Automat in einem von endlich vielen möglichen *Zuständen.* Seinen momentanen Zustand merkt er sich in seinem Zustandsspeicher.

Skizze:



Der endliche Zustandsspeicher sei mit einem Anfangswert q_A „initialisiert", dem *Anfangszustand.* Im folgenden versuchen wir die „Arbeitsweise" des „Mechanismus" schrittweise zu formalisieren.

Der Mechanismus liest ein Zeichen (mit dem Lesekopf) und den Inhalt des Zustandsspeichers und „berechnet" mit diesen Informationen den neuen Inhalt des Zustandsspeichers („wählt" einen der endlich vielen Zustände aus). Dieser neue Zustand wird dann gespeichert und der Lesekopf ein Zeichen weiterbewegt. Wir können dies durch eine *Zustandsüberführungsfunktion* δ formalisieren:

$$\delta : \text{Zustandsmenge} \times \text{Eingabealphabet} \to \text{Zustandsmenge}.$$

Das Eingabealphabet bezeichnen wir mit Σ und die Zustandsmenge mit Q. Der Mechanismus kann nur aufgrund der gespeicherten Zustände entscheiden, ob das Wort akzeptiert wird. Dies formalisieren wir dadurch, daß wir die Zustände, bei denen der Mechanismus akzeptiert, in einer Menge E zusammenfassen.

3.2 Definition: (deterministischer) endlicher Automat

Ein endlicher Automat M wird durch die fünf Angaben Q, Σ, q_A, δ und E spezifiziert; wir schreiben auch $M = (Q, \Sigma, q_A, \delta, E)$. Dabei ist

- Q eine nichtleere endliche Menge von *Zuständen*,
- Σ ein Alphabet von *Eingabesymbolen*,
- $q_A \in Q$ der *Anfangszustand*,
- $\delta : Q \times \Sigma \to Q$ die (totale) *Zustandsüberführungsfunktion*,
- $E \subseteq Q$ eine Menge von *akzeptierenden Zuständen*.

Die Forderung, daß δ total ist, werden wir gleich wieder aufgeben (am Ende von 3.3).

3.3 Zustandsgraphen endlicher Automaten

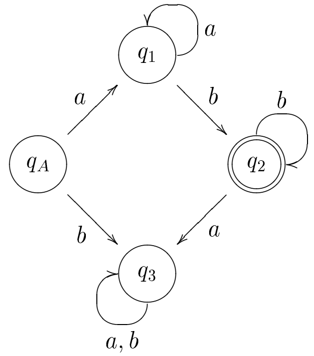
Unser Beispielautomat M_1 soll die Sprache $L_1 = \{a^n b^m \mid n, m \in \mathbb{N}^+\}$ akzeptieren. Wie bei den regulären Grammatiken verwenden wir auch hier erstmal die Mengenschreibweise und führen dann eine übersichtliche Darstellung von Automaten ein. Es sei $M_1 = (Q_1, \Sigma_1, q_A, \delta_1, E_1)$ mit $Q_1 = \{q_A, q_1, q_2, q_3\}$, $\Sigma_1 = \{a, b\}$, $E_1 = \{q_2\}$ und

$$\begin{array}{llll} \delta_1(q_A, a) = q_1, & \delta_1(q_1, a) = q_1, & \delta_1(q_2, a) = q_3, & \delta_1(q_3, a) = q_3, \\ \delta_1(q_A, b) = q_3, & \delta_1(q_1, b) = q_2, & \delta_1(q_2, b) = q_2, & \delta_1(q_3, b) = q_3. \end{array}$$

Die Funktion δ läßt sich übersichtlicher in einer Tabelle darstellen:

δ	a	b
q_A	q_1	q_3
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_3	q_3

Der *Zustandsgraph* ist allerdings noch anschaulicher: Die Knoten werden mit Zuständen, die Kanten mit den gelesenen Zeichen markiert. Der Teilgraph $p \xrightarrow{x} q$ bedeutet dabei $\delta(p, x) = q$. Wenn der Automat also im Zustand p ein x liest, dann geht er in den Zustand q. Die akzeptierenden Zustände werden durch q dargestellt.
Durch den Zustandsgraphen sind implizit alle Bestandteile eines Automaten gegeben. Wir werden deshalb im folgenden alle Automaten mit Zustandsgraphen darstellen.



Betrachten wir, wie unser Beispielautomat auf einer Eingabe arbeitet. Bei jedem Schritt liest er ein weiteres Zeichen der Eingabe, löscht es und geht in den Zustand über, den δ vorschreibt. Solche Übergänge stellen wir mit dem Symbol $\vdash_M$ dar:

$$(p, xv) \vdash_M (q, v) \quad \text{falls} \quad \delta(p, x) = q$$

für $p, q \in Q$, $x \in \Sigma$ und $v \in \Sigma^*$.

Bei Eingabe von aab erreicht M_1 den Zustand $q_2 \in E_1$, die Eingabe wird akzeptiert (ich schreibe $\vdash$ für $\vdash_{M_1}$):

$$(q_A, aab) \vdash (q_1, ab) \vdash (q_1, b) \vdash (q_2, \lambda).$$

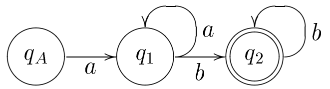
Bei der Eingabe von aaa erreicht M_1 den Zustand $q_1 \notin E_1$, die Eingabe wird nicht akzeptiert:

$$(q_A, aaa) \vdash (q_1, aa) \vdash (q_1, a) \vdash (q_1, \lambda).$$

Bei Eingabe von b oder aba erreicht M_1 den Zustand $q_3 \notin E_1$, diese Wörter werden nicht akzeptiert:

$$(q_A, b) \vdash (q_3, \lambda), \quad (q_A, aba) \vdash (q_1, ba) \vdash (q_2, a) \vdash (q_3, \lambda).$$

Wir haben den Zustand q_3 eingeführt, damit δ für jede Kombination aus Zustand und Eingabesymbol definiert ist. Das ist aber in anderen Fällen meist sehr mühselig. Außerdem werden die Zustandsgraphen dadurch unübersichtlich. Deshalb verzichten wir meist darauf, alle Eingabemöglichkeiten zu berücksichtigen, d.h. δ darf eine partielle Funktion sein. Der Zustandsgraph sieht dann etwas übersichtlicher aus:



Geht es noch einfacher? Was ändert sich, wenn wir wie in 2.3 den Fall $n = 0$ oder $m = 0$ zulassen, also $L = \{a^n b^m \mid n, m \in \mathbb{N}\}$ betrachten?

3.4 Konfiguration, Berechnung, akzeptierte Sprache

Der Linguist Chomsky hat reguläre Grammatiken und „Transformations-Grammatiken“, die wie endliche Automaten arbeiten, zur Analyse natürlicher Sprachen eingeführt; siehe Einleitung. Die Informatiker haben die Grammatiken für ihre Programmiersprachen übernommen und gleichzeitig endliche Automaten als Modelle von Computern (Rechnern) betrachtet. „Automatentheorie und Formale Sprachen“ war daher für lange Zeit die Basistheorie der Informatik.

Dementsprechend nennt man die Zustandsübergänge endlicher Automaten *Berechnungsschritte*, Folgen davon *Berechnungen.* Die Paare (q, w) von Zustand und Wort, die die jeweilige Situation der Berechnung zeigen, heißen *Konfigurationen.* Die Berechnungsschritte eines endlichen Automaten M sind also eine Relation $\vdash_M$ auf Konfigurationen. Analog zu Abschnitt 2.3 bezeichnen wir Berechnungen, die ja Folgen von Schritten sind, mit $\vdash_M^*$; sie sind ebenfalls Relationen auf Konfigurationen. Den Index M lassen wir weg, wenn er sich aus dem Zusammenhang ergibt. Formal können wir Berechnungen wieder rekursiv definieren: Für Zustände $p, q \in Q$ und Wörter $v, w \in \Sigma^*$ gilt

$$
\begin{array}{lll}
(p,v) \vdash^* (q,w) & \text{gdw} & p = q \text{ und } v = w \text{ (kein Schritt)} \\
 & & \text{oder } v = xu \text{ für } x \in \Sigma, u \in \Sigma^*,\ \delta(p,x) = s \in Q \text{ und} \\
 & & (p,v) \vdash (s,u) \vdash^* (q,w) \text{ (mindestens ein Schritt).}
\end{array}
$$

In der zweiten Zeile haben wir $(p, v) \vdash (s, u)$ nur der Anschaulichkeit halber hinzugefügt; laut Definition in 3.3 wiederholt das nur $v = xu$, $\delta(p, x) = s$.

Eine Konfiguration (q, w) heißt

- *Anfangskonfiguration* zur Eingabe w, wenn $q = q_A$;
- *Folgekonfiguration von* (p, v), wenn $(p, v) \vdash (q, w)$;
- *Endkonfiguration*, wenn es keine Folgekonfiguration gibt; insbesondere ist jede Konfiguration (q, λ) eine Endkonfiguration (warum?);
- *akzeptierende Konfiguration*, wenn $w = \lambda$ und $q \in E$.

Der Automat M *akzeptiert* eine Eingabe w, wenn die Berechnung von der Anfangskonfiguration zu w zu einer akzeptierenden Konfiguration führt:

$$(q_A, w) \vdash_M^* (q, \lambda) \text{ mit } q \in E.$$

Die *von M akzeptierte Sprache* $L(M)$ besteht aus allen akzeptierten Wörtern:

$$L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}.$$

Endliche Automaten sind *deterministisch*: δ ist eine Funktion, d.h. für jeden Zustand ist für jedes gelesene Symbol der Folgezustand (falls es einen gibt) eindeutig bestimmt. Daher sind auch Berechnungen durch die Konfiguration, von der sie ausgehen, eindeutig bestimmt; man kann nur vor Erreichen einer Endkonfiguration aufhören. In der Literatur wird daher häufig mit Funktionen statt mit Relationen gearbeitet: Man erweitert δ (statt $\vdash$) rekursiv von Buchstaben auf Wörter, d.h. man definiert die (partielle!) Funktion $\hat{\delta} : Q \times \Sigma^* \to Q$ durch

$$
\begin{aligned}
\hat{\delta}(q, \lambda) &= q, \\
\hat{\delta}(q, xu) &= \hat{\delta}(\delta(q,x), u), \text{ für } q \in Q,\ x \in \Sigma,\ u \in \Sigma^*.
\end{aligned}
$$

Durch Induktion nach der Länge von w beweist man leicht den Zusammenhang

$$\hat{\delta}(p, w) = q \quad \text{gdw} \quad (p, w) \vdash^* (q, \lambda).$$

Wir können daher einfach zwischen der relationalen und der funktionalen Schreibweise wechseln und werden das je nach Bedarf tun. Zum Beispiel ist die Gleichung

$$\hat{\delta}(p, uv) = \hat{\delta}(\hat{\delta}(p, u), v)$$

für $p \in Q$, $u, v \in \Sigma^*$ übersichtlicher als die entsprechende Aussage für Berechnungen (wie lautet die?); der Beweis (durch Induktion nach der Länge von u) ist in beiden Fällen gleich, wir empfehlen ihn als Übung.

Jetzt beweisen wir wie in Abschnitt 2.3 in zwei Teilen, daß der Automat M_1 aus 3.3 die Sprache $L_1 = \{a^n b^m \mid n, m \in \mathbb{N}^+\}$ akzeptiert, also $L_1 = L(M_1)$.

(I) $L_1 \subseteq L(M_1)$: Zu jedem Wort $w = a^n b^m \in L_1$ läßt sich sofort eine akzeptierende Berechnung hinschreiben:

$$(q_A, a^n b^m) \vdash (q_1, a^{n-1} b^m) \vdash^{n-1} (q_1, b^m) \vdash (q_2, b^{m-1}) \vdash^{m-1} (q_2, \lambda).$$

(II) $L(M_1) \subseteq L_1$: Wir beweisen stattdessen $\overline{L_1} \subseteq \overline{L(M_1)}$, d.h. wenn $w \notin L_1$, dann $w \notin L(M_1)$. Dazu unterscheiden wir für $w \notin L_1$ vier Fälle:

1. Fall: $w = \lambda$. Die Berechnung (q_A, λ) mit 0 Schritten endet in q_A, akzeptiert also nicht.
2. Fall: $w = a^n$ mit $n > 0$. Dann $(q_A, w) \vdash^n (q_1, \lambda)$, nicht akzeptierend.
3. Fall: w beginnt mit b. Dann $(q_A, w) \vdash^* (q_3, \lambda)$, nicht akzeptierend.
4. Fall: $w = a^n b^m a v$ mit $n, m > 0$, $v \in \Sigma^*$. Auch diese Berechnung ist nicht akzeptierend: $(q_A, w) \vdash^n (q_1, b^m a v) \vdash^m (q_2, av) \vdash (q_3, v) \vdash^* (q_3, \lambda)$.

Andere Fälle gibt es nicht. Warum nicht? □

3.5 Aufgaben

Aufgabe 3.1: Denkt Euch einen möglichst originellen Automaten aus! Beschreibt den Automaten formal und durch ein Diagramm. Gebt das Problem an, das der Automat löst. Stellt dar, auf welche Weise Ihr das Problem als eine formale Sprache beschreibt.

Aufgabe 3.2: Gebt endliche Automaten an, die die Sprachen A bis G aus Aufgabe 2.4 akzeptieren.

4 Nichtdeterministische Automaten und reguläre Grammatiken

Einen Fahrkartenautomaten, der zwar Geld nimmt, bei dem es dann aber nicht klar ist, ob und welche Karte dann herauskommt, einen solchen Automaten würde ich als defekt bezeichnen. Wenn mir der Zugabfertiger dann sagen würde: „Der Automat arbeitet korrekt, denn es besteht ja immerhin die *Möglichkeit*, daß auch die richtige Karte herauskommt.“ Dann würde ich anmerken wollen, daß ich diesen Korrektheitsbegriff doch etwas bedenklich finde. Tatsächlich werden wir im folgenden „Automaten“ definieren, deren Verhalten nicht eindeutig festgelegt ist. Das Verhalten wird nicht durch die Eingabe determiniert, wir nennen sie *nichtdeterministisch*. Solche merkwürdigen Maschinen bieten sich als Hilfsmittel an, wenn wir im folgenden versuchen, das Wortproblem für reguläre Grammatiken mit endlichen Automaten zu lösen, d.h. die Äquivalenz zwischen regulären Grammatiken und endlichen Automaten zu zeigen.

4.1 Von endlichen Automaten akzeptierte Sprachen sind regulär

Wir möchten gern mit endlichen Automaten reguläre Sprachen nachprüfen (Stichwort „syntax check“); das heißt, wir möchten mit endlichen Automaten das Wortproblem für reguläre Sprachen lösen. Dazu müssen wir zu jeder regulären Grammatik G einen endlichen Automaten M konstruieren, der genau $L(G)$ akzeptiert, also die Frage „$w \in L(G)$?“ korrekt beantwortet; dann gilt $L(M) = L(G)$. Um uns zu überzeugen, daß endliche Automaten auch nicht mehr können, also ein adäquates Mittel sind, wollen wir umgekehrt zu jedem endlichen Automaten M eine reguläre Grammatik G konstruieren, die gerade $L(M)$ erzeugt; wieder gilt $L(G) = L(M)$. In beiden Fällen nennen wir G und M *äquivalent*.

Fangen wir mit dem zweiten Vorhaben an. Wir betrachten also einen endlichen Automaten $M = (Q, \Sigma, q_A, \delta, E)$. Wir können die Zustandsüberführungsfunktion δ als ein Regelsystem auffassen, mit dem wir Buchstaben löschen, d.h. wir schreiben

$$px \to q \quad \text{statt} \quad \delta(p, x) = q.$$

Aus Berechnungen werden dabei Ableitungen, bei denen wir Wörter von links nach rechts lesen und dabei löschen; also schreiben wir analog zu Abschnitt 2.3

$$pu \Rightarrow_M^* qv \quad \text{statt} \quad (p, u) \vdash_M^* (q, v).$$

Wenn wir mit diesem Regelsystem schreiben statt lesen, das heißt, die Regeln $px \to q$ durch $p \to xq$ ersetzen, erhalten wir eine reguläre Grammatik $G = (N, T, S, P)$ mit

- $N = Q$ (Zustände werden nonterminale Zeichen),
- $T = \Sigma$ (Buchstaben werden terminale Zeichen),

- $S = q_A$ (der Anfangszustand wird das Startsymbol),
- $P = \{p \to xq \mid p, q \in Q,\ x \in \Sigma,\ \delta(p, x) = q\}$ (aus Zustandsüberführungen werden Schreibregeln).

Diese Grammatik ist in Normalform (Abschnitt 2.4), es wird sogar pro Schritt genau ein Buchstabe erzeugt. Es gilt für $x \in \Sigma$, $p, q \in Q$:

$$p \Rightarrow_G xq \quad \text{gdw} \quad px \Rightarrow_M q, \text{ d.h. } \delta(p, x) = q, \tag{$*$}$$

also auch für $w \in \Sigma^*$, $p, q \in Q$ (Beweis durch Induktion nach der Länge von w, siehe unten):

$$p \Rightarrow_G^* wq \quad \text{gdw} \quad pw \Rightarrow_M^* q, \text{ d.h. } (p, w) \vdash_M^* (q, \lambda).$$

Wenn wir also mit G ein Wort w erzeugen, tauchen als „cursor“ (nonterminale Symbole rechts) genau die Zustände auf, die M beim Lesen durchläuft. Um genau die Wörter zu erzeugen, die M akzeptiert, müssen wir akzeptierende nonterminale Symbole löschen können. Also erweitern wir G um die Regeln $q \to \lambda$ für $q \in E$ und erhalten für $w \in \Sigma^*$:

$$S \Rightarrow_G^* wq \text{ für ein } q \in E \quad \text{gdw} \quad S \Rightarrow_G^* w \quad \text{gdw} \quad w \in L(G).$$

Also ist $L(G) = L(M)$, wie wir es wollten, und G ist immer noch in Normalform. Bevor wir das als unseren ersten „Satz“ (d.h. wichtige mathematische Aussage) formulieren, holen wir den oben ausgelassenen Induktionsbeweis nach.

Beginn: $w = \lambda$ (d.h. $|w| = 0$). Hier gilt $p \Rightarrow_G^* q$ gdw $p = q$ gdw $p \Rightarrow_M^* q$ (kein Schritt).

Induktionsvoraussetzung: Für $w \in \Sigma^n, p, q \in Q$ gilt:

$$p \Rightarrow_G^* wq \quad \text{gdw} \quad pw \Rightarrow_M^* q$$

Induktionsschritt: $w = xv$ mit $x \in \Sigma$, $v \in \Sigma^n$ (d.h. $|w| = n + 1$):

$$p \Rightarrow_G^* xvq \quad \text{gdw} \quad \text{es gibt } s \in Q \text{ mit } p \Rightarrow_G xs \Rightarrow_G^* xvq.$$

Nach Definition von G gilt wegen $(*)$:

$$\begin{aligned} p \Rightarrow_G xs \quad &\text{gdw} \quad \delta(p, x) = s, \text{ d.h. } px \Rightarrow_M s \\ &\text{gdw} \quad pxv \Rightarrow_M sv \text{ (da der Schritt } v \text{ nicht ändern kann).} \end{aligned}$$

Da Ableitungen in G links nichts ändern, gilt weiter:

$$xs \Rightarrow_G^* xvq \quad \text{gdw} \quad s \Rightarrow_G^* vq \quad \text{gdw} \quad sv \Rightarrow_M^* q \text{ (Induktionsvoraussetzung).}$$

Zusammen: $p \Rightarrow_G^* xvq$ gdw es ein $s \in Q$ gibt mit $pxv \Rightarrow_M sv \Rightarrow_M^* q$ gdw $pxv \Rightarrow_M^* q$.

Satz 4.1 *Aus jedem endlichen Automaten M kann man eine äquivalente reguläre Grammatik G in Normalform konstruieren, d.h. G erzeugt $L(M)$, oder $L(G) = L(M)$. Anders formuliert: Jede von einem endlichen Automaten akzeptierte Sprache ist regulär.*

Die konstruierte Grammatik braucht zum Schreiben einen Schritt mehr als der Automat zum Lesen, weil zum Schluß noch das akzeptierende Symbol gelöscht werden muß. Will man das vermeiden, kann man statt der Regeln $q \to \lambda$ im Beweis für jeden Übergang $\delta(p, x) = q$ mit $q \in E$ die Regel $p \to x$ hinzufügen; nur für $q_A \in E$ braucht man zusätzlich $S \to \lambda$. Es ist leicht einzusehen, daß der Beweis auch für diese Grammatik gilt; wie?

Für den vereinfachten Beispielautomaten M am Ende von Abschnitt 3.3 erhalten wir dabei die Grammatik G mit den Regeln

$$S \to aA, \quad A \to aA \mid bB \mid b, \quad B \to bB \mid b.$$

Die Zustände q_A, q_1, q_2 haben wir in S, A, B umbenannt. Die akzeptierende Berechnung von M für $aaabb$ sieht als Wortableitung so aus:

$$q_A aaabb \Rightarrow_M q_1 aabb \Rightarrow_M q_1 abb \Rightarrow_M q_1 bb \Rightarrow_M q_2 b \Rightarrow_M q_2.$$

Die erzeugende Ableitung von G läuft entsprechend:

$$S \Rightarrow_G aA \Rightarrow_G aaA \Rightarrow_G aaaA \Rightarrow_G aaabB \Rightarrow_G aaabb.$$

Wie erzeugt die im Beweis zuerst konstruierte Grammatik dieses Wort? Wie sehen die Ableitungen von G zu den anderen Berechnungen in 3.3 aus? Wie sehen sie für die anderen Beispiele dort aus?

4.2 Schwierigkeiten mit der umgekehrten Richtung

Jetzt machen wir uns an die eigentliche Aufgabe heran, zu einer gegebenen regulären Grammatik $G = (N, T, S, P)$ einen überprüfenden endlichen Automaten $M = (Q, \Sigma, q_A, \delta, E)$ zu konstruieren. Wir fangen an wie in 4.1 und wählen

$$Q = N, \quad \Sigma = T, \quad q_A = S.$$

Wenn wir aber δ aus P gewinnen wollen, das heißt umgekehrt zu 4.1 die terminalen Symbole in den Regeln aus P von rechts nach links schieben wollen, geraten wir in Schwierigkeiten.

Die erste ist geringfügig: P kann „große“ Regeln $Y \to uZ$ oder $Y \to u$ mit $|u| > 1$ enthalten; wir können das ganze Wort u in einem Schritt hinschreiben. Mit einem endlichen Automaten können wir Wörter aber nur Buchstabe für Buchstabe überprüfen. Ähnliche Probleme machen die „trivialen“ Regeln $Y \to Z$. Deswegen müssen wir voraussetzen, daß G in Normalform ist, daß also die Regeln auf der rechten Seite genau ein nonterminales Symbol oder λ haben. Nach Abschnitt 2.4 ist das keine Einschränkung.

Zweitens gibt es in P Regeln der Form $Y \to x$ oder $Y \to \lambda$ mit $Y \in N$, $x \in T$, mit denen wir den „cursor“ löschen und so Ableitungen in G beenden. M hat aber noch keine Endzustände. Deswegen erweitern wir Q um einen Zustand $q_E \notin N$, setzen

$$E = \{Y \in N \mid Y \to \lambda \text{ in } P\} \cup \{q_E\}$$

und definieren $\delta(Y, x) = q_E$ für jede Regel $Y \to x$ in P.

Die dritte Schwierigkeit ist ernsthaft: P kann Regeln enthalten, mit denen wir dasselbe nonterminale Symbol durch dasselbe terminale, aber unterschiedliche nonterminale ersetzen, also $Y \to xZ$ und $Y \to xZ'$ mit $Z \neq Z'$. Dann kriegen wir $\delta(Y, x) = Z$ und $\delta(Y, x) = Z'$, das geht nicht, δ wäre keine Funktion mehr.

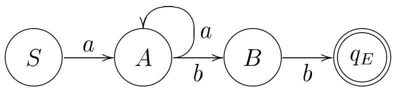
Betrachten wir als ein erstes Beispiel die Grammatik

$$S \to aA, \quad A \to aA \mid bB, \quad B \to b.$$

Für den Automaten ergibt sich dann:

δ	a	b
S	A	—
A	A	B
B	—	q_E

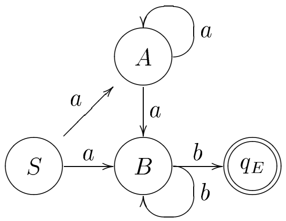
Dabei ist $E = \{q_E\}$. Der Zustandsgraph sieht auch vielversprechend aus:



Probleme gibt es aber, wenn wir die folgende Grammatik betrachten:

$$S \to aA \mid aB, \quad A \to aA \mid aB, \quad B \to bB \mid b$$

Für S müßte sowohl $\delta(S, a) = A$ als auch $\delta(S, a) = B$ gelten, δ kann daher keine Funktion mehr sein. Der Zustandsgraph macht das Problem deutlich:



Für den Zustand S etwa gibt es bei Eingabe a verschiedene Folgezustände. Auf diese Weise können wir also reguläre Grammatiken nicht so direkt in endliche Automaten gemäß der Definition aus 3.2 übersetzen. Wir definieren deshalb Automaten, die in einem Zustand bei gleicher Eingabe verschiedene mögliche Folgezustände besitzen können; wir nennen sie „nichtdeterministisch“. Das bedeutet, daß nicht vorher bestimmt werden kann, welcher der möglichen Folgezustände vom Automaten tatsächlich angenommen wird. Anders formuliert: Die Berechnung des Automaten zu einer Eingabe ist nicht eindeutig bestimmt. Der Automat rechnet nicht, er überprüft die Eingabe nicht selbst, sonders gibt nur eine Reihe von Berechnungen vor, die man ausführen könnte. In der Mathematik ist das ganz üblich. Für das Lösen von Gleichungssystemen zum Beispiel gibt es eine Reihe von Regeln für das Umformen von Ausdrücken, Kombinieren von Gleichungen u.s.w., die man geschickt anwenden muß. Man probiert solange herum, bis man eine Lösung gefunden hat oder feststellt, daß es keine gibt. Genauso ist das bei regulären Grammatiken. Nichtdeterministische endliche Automaten sind also keine echten Maschinen, eher Regelsysteme. Wir formalisieren das, indem wir statt der Funktion $\delta : Q \times \Sigma \to Q$ eine Relation

$$\Delta \subseteq Q \times \Sigma \times Q$$

benutzen, so daß es für $p \in Q$, $x \in \Sigma$ im allgemeinen mehrere (oder kein) $q \in Q$ mit $(p, x, q) \in \Delta$ gibt. Außerdem lassen wir mehrere Anfangszustände zu.

4.3 Definition: nichtdeterministischer endlicher Automat

Ein nichtdeterministischer endlicher Automat M wird durch die fünf Angaben Q, Σ, A, Δ und E spezifiziert; wir schreiben auch $M = (Q, \Sigma, A, \Delta, E)$. Dabei ist

- Q eine endliche Menge von *Zuständen*,
- Σ ein Alphabet von *Eingabesymbolen*,
- $A \subseteq Q$ die Menge der *Anfangszustände*,
- $\Delta \subseteq Q \times \Sigma \times Q$ die *Zustandsüberführungsrelation*,
- $E \subseteq Q$ die Menge der *akzeptierenden Zustände*.

Anmerkung: Jeder deterministische endliche Automat kann als Spezialfall eines nichtdeterministischen endlichen Automaten aufgefaßt werden. Wir setzen einfach $A = \{q_A\}$ und $\Delta = \{(p, x, q) \mid \delta(p, x) = q\}$; Δ ist damit „rechtseindeutig“. Da also nichtdeterministische endliche Automaten „allgemeiner“ sind, werden deterministische endliche Automaten oft gar nicht extra definiert.

4.4 Berechnung, akzeptierte Sprache

Nichtdeterministische Automaten stellen wir wie deterministische graphisch dar. Es kann jetzt aus einem Zustandsknoten heraus mehrere Pfeile mit demselben Buchstaben geben; wenn es mehrere Anfangszustände gibt, müssen wir sie kennzeichnen.

Wie im deterministischen Fall sind durch die Übergangsrelation Δ *Berechnungsschritte* von einer *Konfiguration* zur nächsten definiert:

$$(p, xv) \vdash_M (q, v) \quad \text{falls} \quad (p, x, q) \in \Delta$$

für $p, q \in Q$, $x \in \Sigma$ und $v \in \Sigma^*$. Die Definition von *Berechnungen* ($\vdash_M^*$) sowie *Anfangs-, Folge-, End-* und *akzeptierenden Konfigurationen* stimmen überein. Folgekonfigurationen und daher Berechnungen sind aber nicht eindeutig bestimmt; zur selben Eingabe kann es vom selben Zustand aus verschiedene Berechnungen geben, sogar akzeptierende und nichtakzeptierende! Das soll uns aber nicht stören. Nach Abschnitt 4.2 sind nichtdeterministische Automaten eher Regelsysteme (lesende „Grammatiken") als Maschinen; also definieren wir:

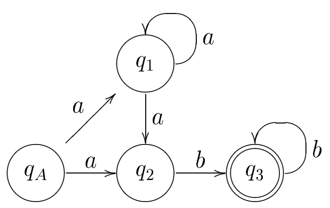
Ein Automat M *akzeptiert* eine Eingabe $w \in \Sigma^*$, wenn es dazu eine *akzeptierende Berechnung* von M *gibt* (!), d.h. eine Berechnung von einer Anfangskonfiguration für w zu einer akzeptierenden Konfiguration:

$$\text{es gibt } q_0 \in A,\ q_1 \in E \text{ mit } (q_0, w) \vdash_M^* (q_1, \lambda).$$

Es kann ruhig daneben nichtakzeptierende Berechnungen $(q_2, w) \vdash_M^* (q_3, \lambda)$ mit $q_2 \in A$, $q_3 \notin E$ geben. Die *von M akzeptierte* (oder *erkannte*) *Sprache* $L(M)$ besteht dann wieder aus allen von M akzeptierten Wörtern:

$$L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}.$$

An dem folgenden Beispiel eines nichtdeterministischen Automaten M wollen wir die Arbeitsweise und die Unterschiede zum deterministischen Fall verdeutlichen:



$$\begin{aligned} Q &= \{q_A, q_1, q_2, q_3\}, \\ \Sigma &= \{a, b\}, \\ A &= \{q_A\}, \\ E &= \{q_3\}, \\ \Delta &= \{(q_A, a, q_1), (q_A, a, q_2), (q_1, a, q_1), \\ &\qquad (q_1, a, q_2), (q_2, b, q_3), (q_3, b, q_3)\}. \end{aligned}$$

Für Eingaben, die mit b beginnen, sowie für λ terminiert M sofort, es gibt nur die Berechnungen

$$(q_A, bv) \quad \text{bzw.} \quad (q_A, \lambda)$$

der Länge 0. Für alle anderen Eingaben gibt es mehrere Berechnungen, z.B.

$$(q_A, aa) \vdash (q_2, a) \quad \text{und} \quad (q_A, aa) \vdash (q_1, a) \vdash (q_i, \lambda),\ i = 1, 2.$$

Nur für Eingaben, die mit a beginnen, irgendwann nach b wechseln und dabei bleiben, gibt es akzeptierende, aber immer auch nichtakzeptierende Berechnungen, z.B.

$$(q_A, abb) \vdash (q_2, bb) \vdash (q_3, b) \vdash (q_3, \lambda) \quad \text{und} \quad (q_A, abb) \vdash (q_1, bb).$$

Für Eingaben, die mit a beginnen, nach b wechseln, dann aber mit a fortfahren, gibt es keine akzeptierende Berechnung. Alle Berechnungen verhalten sich entweder wie die nichtakzeptierenden Berechnungen aus dem vorhergehenden Fall oder enden zwar in q_3, dort kann aber die Eingabe nicht zu Ende gelesen werden, z.B.

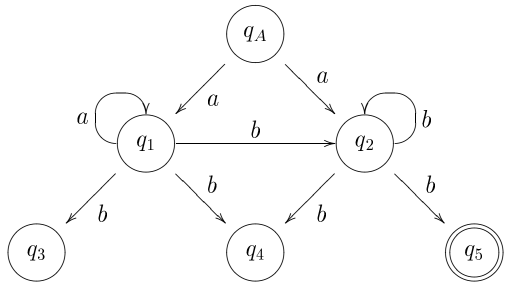
$$(q_A, aba) \vdash (q_2, ba) \vdash (q_3, a).$$

Damit haben wir bewiesen:

$$L(M) = \{a^n b^m \mid n, m \in \mathbb{N}^+\}.$$

Die Sprache kennen wir schon, oder? Übrigens ist für $w \in L(M)$ die akzeptierende Berechnung eindeutig bestimmt. (Wie sieht sie aus?) Trotzdem ist M nicht deterministisch!

Betrachten wir noch das nebenstehende („nichtdeterministischere") Beispiel. Was für Berechnungen gibt es hier? Was ist $L(M)$? Gibt es akzeptierte Eingaben mit mehr als einer akzeptierenden Berechnung? Was passiert, wenn wir q_3 und q_4 streichen? Was ist der Unterschied zum ersten Beispiel?



4.5 Reguläre Sprachen werden von nichtdeterministischen Automaten akzeptiert

Nachdem wir nichtdeterministische Automaten eingeführt haben, nehmen wir den Versuch aus 4.2 wieder auf, zu gegebener regulärer Sprache einen akzeptierenden Automaten zu konstruieren (und zwar einen nichtdeterministischen).

Satz 4.5 *Aus jeder regulären Grammatik G kann man einen äquivalenten nichtdeterministischen endlichen Automaten M konstruieren, d.h. M akzeptiert $L(G)$, oder $L(M) =$*

$L(G)$. Anders formuliert: Jede reguläre Sprache wird von einem endlichen nichtdeterministischen Automaten akzeptiert.

Beweis: Sei $G = (N, T, S, P)$ eine reguläre Grammatik in Normalform. Wir konstruieren zu G einen nichtdeterministischen endlichen Automaten $M = (Q, \Sigma, A, \Delta, E)$ nach der Methode, die bereits in 4.2 diskutiert wurde. Jedes Nonterminal von G wird zu einem Zustand von M, und wir fügen einen zusätzlichen akzeptierenden Endzustand hinzu; d.h. $Q = N \cup \{q_E\}$. Die akzeptierenden Zustände erhalten wir aus den löschenden Regeln: $E = \{q_E\} \cup \{Y \in N \mid Y \to \lambda \text{ in } P\}$. Das Startsymbol von G wird zum Anfangszustand von M, also $A = \{S\}$. Das Eingabealphabet besteht aus den Terminalsymbolen von G, also $\Sigma = T$. Die Überführungsrelation Δ wird mithilfe der Regeln aus P bestimmt:

$$\Delta = \{(p, x, q) \mid p \to xq \text{ in } P\} \cup \{(p, x, q_E) \mid p \to x \text{ in } P\}.$$

Damit beweisen wir $L(G) = L(M)$ genau wie in Abschnitt 4.1. □

4.6 Historisches

Nichtdeterministische endliche Automaten wurden von M.O. Rabin und D. Scott 1959 eingeführt. Ihre Arbeit über endliche Automaten motivieren sie dadurch, daß Turingmaschinen (siehe Kapitel 7) zu allgemein seien, um Computer modellieren zu können. Es ist also durchaus interessant, bei Turing selbst nachzulesen[3]:

> **Automatische Maschinen.** *Wenn die Bewegung einer Maschine bei jedem Schritt durch die Konfiguration* vollständig *bestimmt wird, werden wir die Maschine eine „automatische Maschine" (oder* a-*Maschine) nennen.*
> *Zu einigen Zwecken können wir Maschinen (Wahl-Maschinen oder* w-*Maschinen) verwenden, deren Bewegung durch die Konfiguration nur teilweise bestimmt wird. Wenn eine solche Maschine eine dieser mehrdeutigen Konfigurationen erreicht, kann sie nicht weiterlaufen, bis ein externer Operator eine willkürliche Wahl getroffen hat. Dies wäre der Fall, wenn wir Maschinen zur Behandlung axiomatischer Systeme benutzen würden. In diesem Artikel behandle ich nur automatische Maschinen und lasse das Präfix* a- *deshalb oft weg.*

Dieses Zitat ist in mehrfacher Hinsicht interessant:

- Deterministische Automaten (Turingmaschinen) nennt er automatische Maschinen. Er hat also eine *Maschinen-Sichtweise* auf „Automaten".

[3] On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society (2)42, 1937. Auch in dem Sammelband *The Undecidable*, M. Davis (Ed.), New York, 1965. Deutsch in Alan Turing, *Intelligence Service*.

- Die deterministische Arbeitsweise ist wichtiger Bestandteil der Definition von Automaten. So ist es auch folgerichtig, daß bei mehreren möglichen Folgezuständen die Auswahl von außen (etwa durch einen Menschen) durchgeführt werden muß. Das einzige Beispiel, bei dem er solche Automaten als sinnvoll ansieht, sind *axiomatische Systeme* (also *Ersetzungssysteme*).

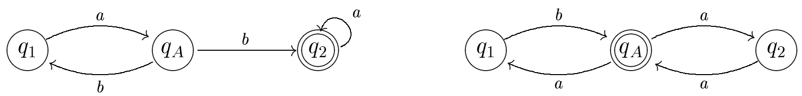
Rabin und Scott haben eine eher mathematische Sichtweise auf Automaten. Von ihnen stammt die Bezeichnung „nichtdeterministischer endlicher Automat" und die heutzutage übliche Definition. Sie verändern und verallgemeinern endliche Automaten – lassen z.B. von rechts nach links oder beliebig die Richtung wechselnd oder mit mehreren „Köpfen" lesen – und zeigen mit Hilfe nichtdeterministischer Automaten, daß die neuen Typen nicht mehr oder weniger können.

4.7 Aufgaben

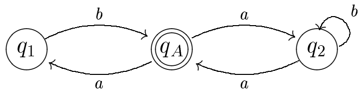
Aufgabe 4.1: Welche der Sprachen aus Aufgabe 2.4 sind nichtdeterministisch einfacher zu akzeptieren als deterministisch?

Aufgabe 4.2: Gebt einen deterministischen und einen nichtdeterministischen Automaten an, der jeweils die Menge der geraden Zahlen in Dezimaldarstellung ohne führende Nullen akzeptiert. (Vgl. Aufgabe 2.3.)

Aufgabe 4.3: Gebt die Sprachen über $\{a, b\}$ an, die die durch folgende Zustandsgraphen gegebenen endlichen Automaten (jeweils mit Anfangszustand q_A) akzeptieren.



Aufgabe 4.4: Versucht einmal, die von diesem Automaten erkannte Sprache verbal zu beschreiben! (Das ist ein gutes Beispiel dafür, daß diese Aufgabe selbst für Automaten mit wenigen Zuständen keineswegs trivial ist. Oftmals ist die Angabe eines Automaten gerade die beste Beschreibung der von ihm erkannten Sprache.) Anfangszustand ist wieder q_A, das Eingabealphabet $\Sigma = \{a, b\}$.

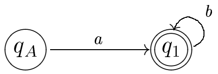


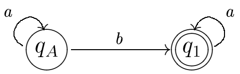
Aufgabe 4.5: Welche Sprache erzeugt diese Grammatik aus dem Startsymbol S?

$$\begin{aligned} S &\to aS \mid bS \mid aA \\ A &\to bB \\ B &\to aB \mid bB \mid a \mid b \end{aligned}$$

Konstruiert daraus einen endlichen Automaten, der dieselbe Sprache akzeptiert.

Aufgabe 4.6: Diese beiden Automaten M_1 und M_2 haben jeweils den Anfangszustand q_A.





a) Welche Sprachen akzeptieren M_1 und M_2?

b) Konstruiert einen nichtdeterministischen Automaten, der $L(M_1) \cup L(M_2)$ akzeptiert ...

c) ... und dann einen deterministischen für denselben Zweck.

Aufgabe 4.7: Gegeben ist ein nichtdeterministischer endlicher Automat. Zeigt, daß es einen äquivalenten Automaten gibt, der folgende besondere Form hat: In die Anfangszustände führen keine Kante hinein und aus den akzeptierenden Zuständen führen keine Kanten hinaus.

5 Endliche Automaten und reguläre Sprachen

In diesem Kapitel werden wir zeigen, daß es für jeden nichtdeterministischen endlichen Automaten einen äquivalenten deterministischen gibt.

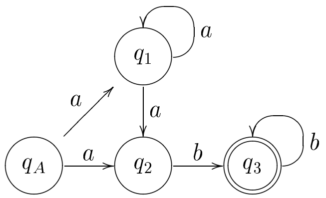
5.1 Wie machen wir nichtdeterministische endliche Automaten deterministisch?

Die Beweisidee ist recht einfach. Bei einem nichtdeterministischen endlichen Automaten sind zwar die Folgezustände nicht eindeutig bestimmt, aber das ist gar nicht nötig, um sein Akzeptieren zu überprüfen. M akzeptiert eine Eingabe w, wenn es eine Berechnung über w von einem Anfangs- zu einem akzeptierenden Zustand gibt, d.h. wenn von den Anfangszuständen aus bei Eingabe von w mindestens ein akzeptierender Zustand *erreichbar* ist:

$$\text{es gibt } q_0 \in A,\ q_1 \in E \text{ mit } (q_0, w) \vdash_M^* (q_1, \lambda).$$

Für jede Menge Q' von Zuständen und jedes Eingabesymbol x ist die *Menge* der Zustände, die M beim Lesen von x von Q' aus erreichen kann, eindeutig bestimmt. Also betrachten wir nicht die verschiedenen Berechnungen, die es zu einer Eingabe w gibt, der Reihe nach (Längsschnittuntersuchung), sondern wir hangeln uns von A ausgehend Buchstabe für Buchstabe von einer Menge von erreichbaren Zuständen zur nächsten (Querschnittuntersuchung); zum Schluß schauen wir nach, ob wir unter den erreichten einen akzeptierenden Zustand von M erwischt haben. Der zu konstruierende deterministische Automat sollte daher als Zustände Mengen von Zuständen des gegebenen nichtdeterministischen Automaten haben.

Betrachten wir als Beispiel den ersten Automaten M aus Abschnitt 4.4 (mit Anfangszustand q_A), der $L = \{a^n b^m | n, m \in \mathbb{N}^+\}$ akzeptiert.



Als Anfangszustand unseres deterministischen Automaten wählen wir die Menge $\{q_A\}$. Von q_A aus sind mit a die Zustände q_1 und q_2 erreichbar, mit b kein Zustand. Also setzen wir

$$\delta(\{q_A\}, a) = \{q_1, q_2\}, \quad \delta(\{q_A\}, b) = \emptyset.$$

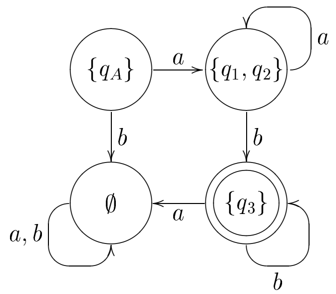
Von q_1 aus sind mit a wieder q_1 und q_2 erreichbar, von q_2 aus mit a gar nichts; mit b erreichen wir nur von q_2 aus etwas, nämlich q_3. Also

$$\delta(\{q_1, q_2\}, a) = \{q_1, q_2\}, \quad \delta(\{q_1, q_2\}, b) = \{q_3\}.$$

Schließlich ist

$$\delta(\{q_3\}, b) = \{q_3\},$$

in allen anderen Fällen liefert δ die leere Menge. Da nur $\{q_3\}$ den akzeptierenden Zustand q_3 von M enthält, erklären wir $\{q_3\}$ zum akzeptierenden Zustand des konstruierten Automaten. Im Bild:



Das ist genau der Automat aus 3.3. Nennen wir ihn M_d, so gilt:

$$w \in L(M_d) \quad \text{gdw} \quad (\{q_A\}, w) \vdash^*_{M_d} (\{q_3\}, \lambda) \quad \text{gdw} \quad (q_A, w) \vdash^*_M (q_3, \lambda) \quad \text{gdw} \quad w \in L(M).$$

5.2 Deterministische und nichtdeterministische endliche Automaten sind äquivalent

Satz 5.2: *Aus jedem nichtdeterministischen endlichen Automaten M_n kann man einen äquivalenten deterministischen endlichen Automaten M_d konstruieren, d.h. es gilt $L(M_n) = L(M_d)$.*

Beweis: Es sei $M_n = (Q_n, \Sigma, A_n, \Delta, E_n)$. Für $Q' \subseteq Q_n$ und $w \in \Sigma^*$ bezeichnen wir mit

$$\mathrm{Erb}(Q', w) = \{q \in Q_n \mid \text{es gibt } p \in Q' \text{ mit } (p, w) \vdash^*_{M_n} (q, \lambda)\}$$

die Menge der Zustände, die M_n von Q' aus bei Eingabe w erreichen kann. Zu M_n konstruieren wir einen deterministischen Automaten $M_d = (Q_d, \Sigma, A_n, \delta, E_d)$ wie im Abschnitt 5.1:

$$\begin{aligned} Q_d &= \mathcal{P}(Q_n) = \{Q' \mid Q' \subseteq Q_n\} \text{ (Potenzmenge)}, \\ E_d &= \{Q \subseteq Q_n \mid Q \cap E_n \neq \emptyset\}, \\ \delta(Q', x) &= \mathrm{Erb}(Q', x) \quad \text{für die Überführungsfunktion } \delta : Q_d \times \Sigma \to Q_d. \end{aligned}$$

Es bleibt zu zeigen, daß beide Automaten dieselbe Sprache akzeptieren, also $L(M_d) = L(M_n)$. Wir zeigen dazu zuerst für $Q' \subseteq Q_n$, $w \in \Sigma^*$:

$$\hat{\delta}(Q', w) = \mathrm{Erb}(Q', w).$$

Das ist nicht überraschend, denn δ haben wir so definiert und $\hat{\delta}$ ist die Fortsetzung von δ auf Wörter (Abschnitt 3.4). Wir beweisen die Behauptung durch Induktion nach der Länge von w, d.h. wir zeigen sie erst für $w = \lambda$ und dann schrittweise für immer längere Wörter.

Beginn: $w = \lambda$. Hier ist $\hat{\delta}(Q', \lambda) = Q' = \mathrm{Erb}(Q', \lambda)$.

Induktionsschritt: Sei $w = xv$. Sei die Gleichung für v schon bewiesen (Induktionsvoraussetzung). Dann gilt

$$\begin{aligned} &\hat{\delta}(Q', xv) \\ &\quad = \hat{\delta}(\delta(Q', x), v) \quad \text{(Definition von } \hat{\delta}) \\ &\quad = \mathrm{Erb}(\delta(Q', x), v) \quad \text{(Induktionsvoraussetzung)} \\ &\quad = \{\, q \in Q_n \mid \text{es gibt } s \in \delta(Q', x) \text{ mit } (s, v) \vdash^*_{M_n} (q, \lambda) \,\} \quad \text{(Definition von Erb)} \\ &\quad = \{\, q \in Q_n \mid \text{es gibt } p \in Q',\ s \in Q_n \text{ mit } (p, x) \vdash_{M_n} (s, \lambda),\ (s, v) \vdash^*_{M_n} (q, \lambda) \,\} \quad \text{(Def. } \delta) \\ &\quad = \{\, q \in Q_n \mid \text{es gibt } p \in Q',\ s \in Q_n \text{ mit } (p, xv) \vdash_{M_n} (s, v) \vdash^*_{M_n} (q, \lambda) \,\} \\ &\quad = \{\, q \in Q_n \mid \text{es gibt } p \in Q' \text{ mit } (p, xv) \vdash^*_{M_n} (q, \lambda) \,\} \quad \text{(Definition von } \vdash^*_{M_n}) \\ &\quad = \mathrm{Erb}(Q', xv) \quad \text{(Definition von Erb)} \end{aligned}$$

Die unkommentierte Gleichung gilt, weil

$$(p, x) \vdash_{M_n} (s, \lambda) \quad \text{gdw} \quad (p, xv) \vdash_{M_n} (s, v).$$

Jetzt beweisen wir $L(M_d) = L(M_n)$:

$$\begin{aligned} w \in L(M_d) \quad &\text{gdw} \quad \hat{\delta}(A_n, w) \in E_d \\ &\text{gdw} \quad \hat{\delta}(A_n, w) \cap E_n \neq \emptyset \\ &\text{gdw} \quad \text{es gibt } q \in \mathrm{Erb}(A_n, w) \cap E_n \\ &\text{gdw} \quad \text{es gibt } q \in E_n,\ p \in A_n \text{ mit } (p, w) \vdash^*_{M_n} (q, \lambda) \\ &\text{gdw} \quad M_n \text{ akzeptiert } w \\ &\text{gdw} \quad w \in L(M_n) \end{aligned}$$

□

Der Automat M_d im Beweis hat alle Teilmengen der Zustandsmenge Q_n von M_n als Zustände; das sind sehr viele, nämlich 2^k für $|Q_n| = k$. Es lohnt sich also, wie im Beispiel in Abschnitt 5.1, nur die erreichbaren Teilmengen zu konstruieren. Die anderen braucht man nicht (warum nicht?), und man kriegt oft viel kleinere Automaten. Muß man den Automaten nicht öfters verwenden, genügt es sogar, zu einer konkreten Eingabe nur die dabei erreichbaren Zustände zu berechnen und auch immer nur einen zu speichern.

5.3 Äquivalenz regulärer Grammatiken und endlicher Automaten

Wir können jetzt die bisherigen Ergebnisse so zusammenfassen:

Satz: *Sei Σ ein Alphabet und $L \subseteq \Sigma^*$. Die folgenden drei Aussagen sind äquivalent:*

(1) L ist regulär.

(2) Es gibt einen deterministischen endlichen Automaten, der L akzeptiert.

(3) Es gibt einen nichtdeterministischen endlichen Automaten, der L akzeptiert.

Beweis: Nach Satz 4.1 folgt (1) aus (2); nach Satz 4.5 folgt (3) aus (1); und nach Satz 5.2 sind (2) und (3) äquivalent. □

Tatsächlich ist der Beweis „konstruktiv“: Man kann aus jeder der drei Darstellungen einer Sprache (Grammatik, deterministischer Automat, nichtdeterministischer Automat) die anderen konstruieren.

Reguläre Grammatiken werden in der Informatik benutzt, wie in Abschnitt 2.5 dargestellt, um einfache Ausdrücke in Programmiersprachen zu definieren. Die Korrektheit solcher Ausdrücke kann man daher mit deterministischen endlichen Automaten überprüfen. Will man die Automaten aus der Grammatik konstruieren, braucht man nichtdeterministische Automaten als Zwischenstufe. Wichtig ist dabei, möglichst kleine Automaten zu erhalten; damit haben wir uns hier nicht befaßt. (Vergleiche aber das Ende des vorherigen Abschnitts.)

Nichtdeterministische Automaten sind wichtige Hilfsmittel, um die Regularität von Mengen zu beweisen. Sie sind oft kleiner und meist leichter zu finden als ihre deterministischen Verwandten. Der folgende Satz ist ein Beispiel dafür.

Satz 5.3: *Die Klasse der regulären Sprachen ist abgeschlossen unter Booleschen und Kleeneschen Operationen (Abschnitt 1.4), d.h. wenn L_1 und L_2 regulär sind, so auch $\overline{L_1}$ (Komplement), $L_1 \cap L_2$ (Durchschnitt), $L_1 \cup L_2$ (Vereinigung), $L_1 \circ L_2$ (Verkettung), L_1^n (Potenz), L_1^* (Stern).*

Beweis als Übung bzw. Projektaufgabe. Man muß akzeptierende Automaten geeignet „zusammenschalten“. Es empfiehlt sich, zuerst zu beweisen, daß man Anfangs- und akzeptierende Zustände „auslagern“ kann, d.h. daß keine Pfeile in die Anfangszustände hinein oder aus den akzeptierenden Zuständen herausführen, siehe Aufgabe 4.7. – Alternativ kann man in nichtdeterministischen Automaten „spontane“ Übergänge $p \to q$ zulassen, in denen kein Symbol gelesen wird (formal wird Δ auf $Q \times Q$ erweitert). Diese spontanen Übergänge entsprechen den trivialen Regeln aus Abschnitt 2.4. Sie lassen sich wie dort eliminieren – eine gute Übungsaufgabe.

5.4 Aufgaben

Aufgabe 5.1: Macht den Automaten aus Aufgabe 4.4 mit dem Verfahren aus 5.2 deterministisch. Wird er einfacher oder komplizierter? Ist er einfacher oder schwieriger zu verstehen?

Aufgabe 5.2: Die folgende Grammatik G hat das Startsymbol S:

$$S \to ab \mid ba \mid aAb \mid bAa$$
$$A \to aA \mid bA \mid a \mid b$$

a) Warum ist G keine reguläre Grammatik?

b) Welche Sprache erzeugt G? Ist $L(G)$ regulär? Wenn ja (eine rhetorische Frage), gebt eine reguläre Grammatik G' an, die $L(G)$ erzeugt.

c) Konstruiert aus G' einen nichtdeterministischen endlichen Automaten, der $L(G)$ akzeptiert und daraus einen deterministischen für denselben Zweck.

Aufgabe 5.3: Gebt zu den Automaten aus Aufgabe 4.3 deterministische endliche Automaten an, die dieselben Sprachen akzeptieren. Ihr könnt sie ad hoc konstruieren oder die gegebenen Automaten deterministisch machen. Geht beide Wege und vergleicht die Ergebnisse.

Aufgabe 5.4: Sei Σ ein Alphabet. Beweist, daß reguläre Ausdrücke über Σ reguläre Sprachen beschreiben.

Im folgenden wird erstmal definiert, was reguläre Ausdrücke sind und welche Sprachen sie beschreiben. Danach gibt es ein paar Hinweise, wie Ihr an diese Aufgabe herangehen solltet.

Definition (Reguläre Ausdrücke über Σ und die dadurch definierten Sprachen).

1. Einfache Ausdrücke: $\underline{\emptyset}, \underline{\lambda}$ und $\underline{a}$ für jedes $a \in \Sigma$ sind reguläre Ausdrücke über Σ.
2. Zusammengesetzte Ausdrücke: Sind α und β reguläre Ausdrücke über Σ, so auch $\underline{(}\alpha \underline{\circ} \beta\underline{)}, \underline{(}\alpha \underline{\cup} \beta\underline{)}$ und $\alpha^{\underline{\star}}$.

Für jeden regulären Ausdruck γ über Σ definieren wir eine Sprache $L(\gamma) \subseteq \Sigma^\star$:

1. Einfache Ausdrücke: $L(\underline{\emptyset}) = \emptyset, L(\underline{\lambda}) = \{\lambda\}$ und $L(\underline{a}) = \{a\}$ für $a \in \Sigma$.
2. Zusammengesetzte Ausdrücke:
   $L\big(\underline{(}\alpha \underline{\circ} \beta\underline{)}\big) = L(\alpha) \circ L(\beta), L\big(\underline{(}\alpha \underline{\cup} \beta\underline{)}\big) = L(\alpha) \cup L(\beta), L(\alpha^{\underline{\star}}) = L(\alpha)^\star$.

Formal sind das „induktive“ Definitionen: Mit der ersten könnt Ihr aus einfachen Ausdrücken (Teil 1) immer kompliziertere zusammensetzen (Teil 2). Mit der zweiten könnt Ihr ganz entsprechend die definierte Sprache für einfache Ausdrücke (Teil 1) und daraus Schritt für Schritt für zusammengesetzte (Teil 2) bestimmen.

Die Zeichen $\emptyset, \lambda, \ldots, \star$ stehen dabei, wenn sie unterstrichen sind (hier links vom Gleichheitszeichen) und wenn sie nicht unterstrichen sind (also rechts vom Gleichheitszeichen), für ganz verschiedene Dinge: Links sind es Symbole („Buchstaben“) in regulären Ausdrücken, rechts stehen sie für Mengen ($\emptyset$), bestimmte Wörter (λ, a) und Mengenoperationen ($\circ, \cup, \star$). Häufig werden jedoch für beide Dinge dieselben Zeichen verwendet.

Macht Euch die Definitionen zunächst an Beispielen klar, d.h. sammelt reguläre und nichtreguläre Ausdrücke und rechnet für die regulären Ausdrücke die dadurch definierten Sprachen aus, und beschreibt reguläre Sprachen aus dem Skript oder aus den Aufgaben mit regulären Ausdrücken.

Beweist dann, daß alle regulären Ausdrücke reguläre Sprachen beschreiben, d.h. daß $L(\gamma)$ für jeden regulären Ausdruck γ eine reguläre Sprache ist.

Für die einfachen Ausdrücke ist das leicht. Für die zusammengesetzten müßt Ihr zeigen, daß die durch reguläre Ausdrücke definierten Sprachen unter $\circ, \cup$ und $\star$ „abgeschlossen“ sind, d.h. wenn für reguläre Ausdrücke α und β die Sprachen $L(\alpha)$ und $L(\beta)$ regulär sind, so sind auch die Sprachen $L\big(\underline{(}\alpha \underline{\circ} \beta\underline{)}\big), L\big(\underline{(}\alpha \underline{\cup} \beta\underline{)}\big)$ und $L(\alpha^{\underline{\star}})$ regulär. Das folgt aus Teil 2 der Definition von $L(\gamma)$ und aus Teil 1 der ersten Projektaufgabe bzw. Satz 5.3. Damit habt Ihr eine Methode, für immer längere reguläre Ausdrücke reguläre Grammatiken oder endliche Automaten zu konstruieren, die die durch den Ausdruck definierte Sprache erzeugen bzw. akzeptieren. (Formal ist das ein Beweis durch „strukturelle“ Induktion oder durch vollständige Induktion nach der Zahl der Verknüpfungszeichen in einem regulären Ausdruck.)

6 Kontextfreie Grammatiken und Sprachen

In dem folgenden Abschnitt führen wir eine Erweiterung der regulären Grammatiken ein. Die kontextfreien Grammatiken und Sprachen haben in der Informatik eine besondere Bedeutung erlangt. Es gibt effiziente Algorithmen fürs Akzeptieren (einer Teilklasse) von kontextfreien Sprachen. Die Syntax vieler Programmiersprachen läßt sich (teilweise) durch kontextfreie Grammatiken beschreiben.

6.1 Reguläre versus kontextfreie Grammatiken

In den Ableitungen regulärer Grammatiken durfte nur das letzte Zeichen ersetzt werden. Wir hatten dazu nonterminale Symbole eingeführt und die Regeln so eingeschränkt, daß jeweils nur das letzte Symbol der rechten Seite nonterminal sein durfte. Bei kontextfreien Grammatiken fällt diese Einschränkung weg; die rechte Seite ist jetzt beliebig, und Regeln dürfen jetzt überall im Wort angewendet werden. Für das Beispiel

$$S \to SS \mid aSa \mid bSb \mid a \mid b$$

gibt es Ableitungen wie etwa

$$\begin{aligned} S &\Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabSSbaa \Rightarrow aabbSbSbaa \\ &\Rightarrow aabbaSabSbaa \Rightarrow aabbababSbaa \Rightarrow^2 aabbababababaa. \end{aligned}$$

Wie sieht die dadurch erzeugte Sprache aus? Mit kontextfreien Grammatiken können wir zum Beispiel korrekt geklammerte Ausdrücke über dem Terminalalphabet $\{a, (,)\}$ erzeugen:

$$S \to SS \mid (S) \mid a$$

Eine typische Ableitung ist hier

$$\begin{aligned} S &\Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow ((S)a) \Rightarrow (((S))a) \\ &\Rightarrow (((SS))a) \Rightarrow ((((S)S))a) \Rightarrow ((((a)S))a) \Rightarrow ((((a)(S)))a) \Rightarrow ((((a)(a)))a). \end{aligned}$$

Daran wird auch deutlich, warum kontextfreie Grammatiken die Syntax von Programmiersprachen mit Blockstrukturierung so gut beschreiben können, wie das folgende Beispiel über dem Terminalalphabet $\{\texttt{begin}, \texttt{end}, ;\}$ zeigt:

$$S \to S; S \mid \texttt{begin}\ S\ \texttt{end} \mid \ldots$$

Man schreibt dann meist := statt $\to$ und spricht von der *Backus-Naur-Form*, weil John Backus und Peter Naur sie 1958 benutzt haben, um die Syntax der neuen Sprache ALGOL zu definieren.

Bei arithmetischen Ausdrücken über $\{+, -, *, /, (,), 0, 1, \ldots, 9\})$ ist es im Prinzip nicht anders:

$$\begin{array}{rcl} S & \to & (S+S) \mid (S-S) \mid (S*S) \mid (S/S) \\ S & \to & 1Z \mid 2Z \mid \ldots \mid 9Z \mid 0 \mid 1 \mid \ldots \mid 9 \\ Z & \to & 0Z \mid 1Z \mid \ldots \mid 9Z \mid 0 \mid 1 \mid \ldots \mid 9 \end{array}$$

Probiert selbst ein paar Ableitungen aus. Mit kontextfreien Grammatiken lassen sich aber nicht alle Sprachen erzeugen. Die Sprache

$$\{a^n b^n c^n \mid n \in \mathbb{N}\}$$

kann von keiner kontextfreien Grammatik erzeugt werden. (Aber sie ist immerhin kontext*sensitiv*, siehe Abschnitt 6.5.) Dies liegt daran, daß bei der Ersetzung die Symbole, die links und rechts vom ersetzten Symbol stehen, nicht berücksichtigt werden. Ein Beispiel für eine nicht kontextfreie Regel ist

$$aSa \to aba.$$

Bei kontextfreien Regeln steht auf der linken Seite immer nur *ein* nonterminales Symbol:

$$\underbrace{\quad}_{\text{kein}} S \underbrace{\quad}_{\text{Kontext}} \to aSa$$

Daraus folgt $|u| \leq |v|$ für jede kontextfreie Regel $u \to v$, mit Ausnahme der Regeln $X \to \lambda$.

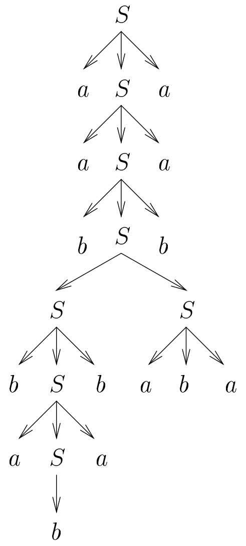
6.2 Definition: kontextfreie Grammatik

Eine kontextfreie Grammatik G wird durch vier Angaben spezifiziert; wir schreiben auch $G = (N, T, S, P)$. Dabei ist

- N ein Alphabet mit *nonterminalen Symbolen*,
- T ein Alphabet mit *terminalen Symbolen*,
- $S \in N$ das *Startsymbol*,
- P eine Menge von *Produktionen* der Form $X \to v$ mit $X \in N$ und $v \in (N \cup T)^*$.

Die Alphabete N und T müssen wieder disjunkt sein.

6.3 Ableitungsbaum, Links-Ableitung, kontextfreie Sprache

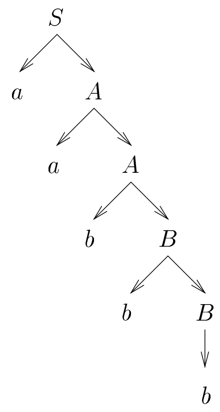


Wir haben Ableitungen bisher immer „linear“, d.h. als Kette von einzelnen Ableitungsschritten aufgeschrieben. Dies ist bei kontextfreien Grammatiken nicht mehr übersichtlich, wie wir in 6.1 gesehen haben. Im folgenden wollen wir deshalb Bäume zur Darstellung von Ableitungen verwenden.
Der *Ableitungsbaum* für die erste Ableitung in 6.1 ist hier dargestellt. Etwas formaler beschrieben: Wir markieren die Knoten mit Symbolen aus $N \cup T$. Für jeden Ableitungsschritt gibt es für jedes Symbol der rechten Regelseite eine neue Kante. Diese Kanten beginnen alle in dem Knoten, dessen Markierung das ersetzte nonterminale Symbol ist. Wir markieren die Zielknoten folgendermaßen: Ist

$$X \to v_1 v_2 \ldots v_n$$

die verwendete Produktion, dann erhalten die Zielknoten die Markierungen $v_1, v_2, \ldots, v_n$ (von links nach rechts). Das erzeugte Wort können wir an den Blättern ablesen, wenn wir den Ableitungsbaum „leftmost-depthfirst“ traversieren. In dem links abgebildeten Ableitungsbaum wurde z.B. das Wort *aabbababababaa* erzeugt.

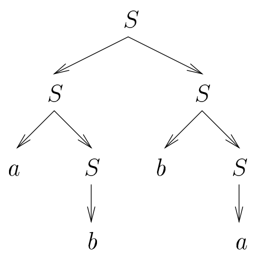
Für reguläre Grammatiken sind die entsprechenden Ableitungsbäume recht langweilig. Sie unterscheiden sich kaum von der linearen Darstellung.



Ableitungsbäume eignen sich als Hilfsmittel zur Darstellung von Ableitungen. Aber die Reihenfolge der Ersetzungsschritte wird damit nicht dargestellt.

Für die Grammatik

$$S \to SS \mid aS \mid bS \mid a \mid b$$



gibt es mehrere mögliche Ableitungen zu nebenstehendem Ableitungsbaum, unter anderen

(1) $S \Rightarrow SS \Rightarrow aSS \Rightarrow abS \Rightarrow abbS \Rightarrow abba$,

(2) $S \Rightarrow SS \Rightarrow aSS \Rightarrow aSbS \Rightarrow abbS \Rightarrow abba$,

(3) $S \Rightarrow SS \Rightarrow SbS \Rightarrow Sba \Rightarrow aSba \Rightarrow abba$.

Für manche Zwecke ist es nützlich, eine Ableitung eindeutig festzulegen: Die Ableitung, in der immer zuerst das am weitesten links stehende nonterminale Symbol ersetzt wird, nennen wir *Links-Ableitung*. (Wer mehr auf rechts steht, kann analog Rechts-Ableitungen betrachten. Was wären Mitte-Ableitungen?) Im Beispiel oben ist dies die Ableitung (1). Wir halten das als Lemma (oder Hilfssatz) fest:

Lemma 6.3 *Läßt sich ein Wort w mit einer kontextfreien Grammatik G ableiten, so gibt es auch eine Links-Ableitung für w mit G.*

Ähnlich wie für reguläre Grammatiken definieren wir die von einer kontextfreien Grammatik G *erzeugte Sprache* als

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

Dabei sei $\Rightarrow_G^*$ analog zu den regulären Grammatiken definiert; für $v, w \in (N \cup T)^*$ heißt $v \Rightarrow_G^* w$: es gibt eine Ableitung von v nach w mit G. Eine Sprache $L \subseteq \Sigma^*$ heißt *kontextfrei*, wenn es eine kontextfreie Grammatik gibt, die L erzeugt. Jede reguläre Sprache ist kontextfrei (warum?). Die Klasse der kontextfreien Sprachen bezeichnen wir mit CFL (für *Context Free Languages*).

Wir nennen noch zwei Beispiele für besonders „typische“, nämlich nicht reguläre, kontextfreie Sprachen:

- Die Sprache $L_1 = \{a^n b^n \mid n \in \mathbb{N}^+\}$ wird von der Grammatik G_1 mit den Regeln $S \to aSb \mid ab$ erzeugt.
- Die Sprache $L_2 = \{ww^{-1} \mid w \in \{a,b\}^+\}$ aller Palindrome gerader Länge über dem Alphabet $\{a,b\}$ wird von der Grammatik G_2 mit den Regeln $S \to aSa \mid bSb \mid aa \mid bb$ erzeugt. Dabei ist w^{-1} das gespiegelte Wort zu w, siehe 1.4, z.B. $(abb)^{-1} = bba$.

Um $L_i = L(G_i)$ für $i \in \{1,2\}$ zu zeigen, könnt Ihr wie in Abschnitt 2.3 bei den regulären Grammatiken vorgehen. Zum Beispiel betrachtet Ihr im ersten Beispiel zuerst nur die erste Regel und beweist (durch Induktion nach n), daß Ableitungen der Länge n, die von S ausgehen, das Wort $a^n S b^n$ erzeugen. Daraus folgt mit Hilfe der zweiten Regel $L(G_1) \subseteq L_1$. Die Umkehrung zeigt Ihr genauso durch Induktion nach der Wortlänge. Die Beweise im

zweiten Beispiel sind analog. Tatsächlich sind die Grammatiken so einfach, daß Ihr z.B. für alle $w \in T^*$ direkt

$$S \Rightarrow_{G_1}^n w \quad \text{gdw} \quad w = a^n b^n$$

durch Induktion nach n beweisen könnt. Dazu müßt Ihr im Induktionsschritt die Ableitungen nicht am Ende, sondern am Anfang verkürzen bzw. verlängern und in „Kontexte“ (Randbuchstaben) einbetten bzw. aus ihnen herausnehmen. Wenn Ihr das macht, seht Ihr, warum es wichtig ist, daß die Regeln „kontextfrei“ sind.

Die Sprachen $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ und $\{ww \mid w \in \{a,b\}^*\}$ sind typische *nicht* kontextfreie Sprachen. Darum geht es im übernächsten Abschnitt.

6.4 Nicht reguläre Sprachen

Will man von einer bestimmten Sprache zeigen, daß sie regulär ist, dann kann man dies tun, indem man eine reguläre Grammatik angibt, die die gegebene Sprache erzeugt, oder einen endlichen Automaten, der die Sprache erkennt. Wie aber läßt sich zeigen, daß eine Sprache *nicht* regulär ist, also von *keiner* regulären Grammatik erzeugt wird? Dazu benutzt man sogenannte „Pumping-Lemma-Argumente“ wie im folgenden Beweis.

Satz 6.4. *Die Sprache $L = \{a^m b^m \mid m \in \mathbb{N}\}$ ist nicht regulär.*

Beweis. Zum Zwecke des Widerspruchs nehmen wir an, die Sprache L wäre regulär. Dann gäbe es einen endlichen deterministischen Automaten M mit $L = L(M)$. Dieser Automat habe die Zustandsmenge Q.

Betrachten wir nun ein sehr langes Wort $w \in L$. Beispielsweise $w = a^n b^n$, wobei $n > |Q|$ eine beliebige große Zahl ist. Wir werden jetzt gleich argumentieren, daß L neben w auch noch einige weitere Wörter enthalten muß.

Der Automat M akzeptiert w, denn er akzeptiert ja genau die Wörter aus L. Sei nun $(q_0, a^n b^n) \vdash_M (q_1, a^{n-1} b^n) \vdash_M (q_2, a^{n-2} b^n) \vdash_M^* (q_{2n}, \lambda)$ die Berechnung des Automaten bei Eingabe w. Der Automat durchläuft also nacheinander die Zustände $q_0, q_1, \ldots, q_{2n}$. *Dies sind aber mehr Zustände als es überhaupt unterschiedliche Zustände gibt.* Also muß mindestens ein Zustand zweimal durchlaufen werden. Es gibt also Indizes i, j mit $i < j$, so daß $q_i = q_j$.

Anschaulich gesprochen enthält die Zustandsfolge eine *Schleife*. Das Wort u enthalte die ersten i Symbole des Wortes w, das Wort z die Symbole zwischen dem i-ten und dem j-ten Symbol und das Wort v die restlichen Symbole. Wir zerlegen also $w = uzv$. Während der Automat die Schleife durchläuft „sieht“ er genau das Wort z. Etwas formaler: $(q_i, z) \vdash_M^* (q_j, \lambda)$.

Wenn der Automat das j-te Zeichen erreicht hat und *nochmal* das Wort z käme, so würde der Automat wiederum im Zustand $q_j = q_i$ enden. Der Automat würde also auch das

Wort $uzzv$ akzeptieren, denn für ihn sind uzv und $uzzv$ gewissermaßen ununterscheidbar. Ebenso würde der Automat uz^kv für jedes beliebige k akzeptieren.

Wir haben nun (fast) den gewünschten Widerspruch gefunden: Mit dem langen Wort $uzv = a^nb^n$ müßte L auch die „aufgepumpten" Wörter uz^kv enthalten. Dies kann aber nicht sein. Betrachten wir die verschiedenen Möglichkeiten:

- z enthält sowohl a's als auch b's. Dann steht in uz^2v aber mindestens ein b links von einem a, das Wort uz^2v ist also sicher *nicht* in L, ein Widerspruch.
- z enthält nur a's, sagen wir $z = a^p$. Dann ist aber $uz^2v = a^{n+p}b^n$ sicher *nicht* in L (da $p > 0$, also $n + p \neq n$ ist), ein Widerspruch.
- z enthält nur b's. Wir argumentieren wie im zweiten Fall.

In allen Fällen ergab sich ein Widerspruch, L ist also nicht regulär. □

Fassen wir die zentrale Ideen des Widerspruchsbeweises nochmal zusammen:

1. Wir betrachten ein hinreichend langes Wort $w \in L$. Dieses muß länger als die Anzahl der Zustände eines Automaten M sein, der L erkennt.
2. Da das Wort so lang ist, muß der Automat bei Eingabe der Wortes eine Schleife durchlaufen.
3. Den Teil der Wortes, bei dem der Automat die Schleife durchläuft, kann man beliebig oft wiederholen („aufpumpen") und erhält immer Wörter, die noch in der Sprache $L(M)$, also auch in L sein müssen.
4. Den Widerspruch erhält man dann, wenn eine aufgepumpte Version des Wortes nicht in der Sprache L ist.

Wichtig ist, daß man im allgemeinen nicht weiß, *wo* der Automat die Schleife durchläuft. Deshalb haben wir auch die verschiedenen Fälle unterschieden und erhalten, daß egal wo die Schleife durchlaufen wird, aufgepumpte Versionen des Wortes a^nb^n nicht in L liegen können.

Leider gibt es Sprachen, die nicht regulär sind, wo wir dies aber mit obigem Argument nicht beweisen können. Sehen wir uns etwa die Sprache

$$L_2 = \{\, w \in \{a,b\}^* \mid w \text{ enthält gleich viele } a\text{'s wie } b\text{'s} \,\}$$

an. Tatsächlich lassen sich alle Wörter $w \in L_2$ ab Länge zwei in $w = uzv$ zerlegen, so daß sie aufgepumpt werden können. Wie? Nun, jedes nichtleere Wort in L_2 enthält entweder ab oder ba als Teilwort; wählt man $z = ab$ bzw. $z = ba$ und u und v entsprechend als linken und rechten Kontext, dann ist z ein nichtleeres Wort und $uz^kv \in L_2$ für alle $k \in \mathbb{N}$. Schade.

Aber die Sache ist nicht aussichtslos, wir müssen unser Argument nur etwas nachbessern; und tatsächlich gibt der beschriebene Beweis auch mehr her: Wir wählen u und z einfach so klein wie irgend möglich. Es soll also z das Wort sein, das bei der *ersten* Schleife gelesen wird. Dann ist klar, daß uz höchstens die Länge $|Q|$ hat, denn die erste Schleife ist spätestens dann vollendet, wenn alle Zustände durchlaufen sind.

Jetzt können wir zeigen, daß L_2 nicht regulär ist. Wäre die Sprache regulär, dann müßten sich hinreichend lange $w \in L_2$ als uzv schreiben lassen, wobei $|uz| \leq n$ und $n = |Q|$. Nun ist sicherlich $w = a^nb^n \in L_2$. Wegen $|uz| \leq n$ besteht z nur aus a's! Dann ist aber $uz^2v \notin L_2$, denn uz^2v enthält ja mehr a's als b's.

Übrigens läßt sich so auch der Beweis vereinfachen, daß die Sprache $\{a^mb^m \mid m \in \mathbb{N}\}$ nicht regulär ist. Wir wählen wie eben $w = a^nb^n$ und garantieren so, daß z nur a's und keine b's enthält; von den drei betrachteten Fällen bleibt dann lediglich der zweite übrig.

6.5 Nicht kontextfreie Sprachen

Wie schon erwähnt gibt es Sprachen, die nicht kontextfrei sind. Ein Beispiel einer solchen Sprache ist

$$L = \{a^nb^nc^n \mid n \in \mathbb{N}\}.$$

Daß diese Sprache nicht kontextfrei ist, läßt sich mit einem entsprechend komplizierteren Pumping-Lemma-Argument für kontextfreie Sprachen beweisen. Auf dieses Beweisverfahren werden wir hier aber nicht eingehen.

Obwohl diese Sprache nicht kontextfrei ist, läßt sich immerhin eine Grammatik mit schönen Eigenschaften für sie angeben.

Wie könnte eine Grammatik aussehen, die L erzeugt? Eine Idee ist, zuerst Satzformen $a^n(BC)^n$ zu erzeugen – das geht kontextfrei, nämlich wie? –, darin die nonterminalen Symbole zu sortieren, z.B. B nach links – dazu muß man C und B vertauschen, wie geht das? – und dabei oder danach B durch b und C durch c zu ersetzen. Das ist eine schöne Übungsaufgabe, mit einigen Fallen. Probiert Eure Lösungsansätze immer an genug Beispielen aus, bis Ihr sicher seid. Dann beweist die Korrektheit, nach der bewährten Art (zwei Richtungen, stückweise, mit Induktion).

Wahrscheinlich gilt für die Regeln $u \to v$ in Euren Lösungen $|u| \leq |v|$ (bis auf die Löschregel $S \to \lambda$), d.h. Wörter werden beim Ableiten nicht kürzer. Solche Regeln und Grammatiken und dadurch erzeugte Sprachen heißen *kontextsensitiv*; für Sprachen, die λ enthalten, braucht man zusätzlich die Regel $S \to \lambda$ für das Startsymbol S. Der Name kommt daher, daß man jede kontextsensitive Sprache durch Regeln der Form

$$uXv \to uwv$$

mit $X \in N$, $u, v, w \in (N \cup T)^*$ erzeugen kann, bei denen man X nur „im Kontext" u_v durch w ersetzen darf.

6.6 Wortproblem, akzeptierende Automaten

Für kontextfreie und sogar für kontextsensitive Sprachen ist das Wortproblem entscheidbar (Abschnitt 2.5). Die Beweisidee ist für beide Typen dieselbe: Um für eine gegebene kontextsensitive Grammatik $G = (N, T, S, P)$ und ein Wort $w \in T^*$ zu entscheiden, ob $w \in L(G)$ ist, erzeugt man systematisch alle Ableitungen $S \Rightarrow_G^* v$ mit $|v| \leq |w|$. Anders als mit regulären erzeugt man mit kontextsensitiven Regeln nicht notwendig terminale Symbole (vgl. das Beispiel oben). Die betrachteten Ableitungen $S \Rightarrow_G^* v$ sind also i.a. länger als $|w|$, aber nicht beliebig lang. Wie beim Pumping-Lemma haben sehr lange Ableitungen Wiederholungen, die man ausschneiden könnte; so kann man die Länge der zu betrachtenden Ableitungen beschränken. Daher bricht das Verfahren ab, wenn man eine Ableitung für w gefunden oder alle in Frage kommenden Ableitungen vergeblich durchgemustert hat. Der Beweis wird einfacher, wenn man *Normalformen* einführt; für kontextfreie Normalformen siehe z.B. Abschnitt 1.3.1 in dem Buch von Schöning.

Wie im regulären Fall gibt es auch hier zu den erzeugenden Grammatiken akzeptierende Automaten, mit denen man das Wortproblem lösen kann. Im kontextfreien Fall sind das die (allerdings nichtdeterministischen!) *Kellerautomaten*; die sollt ihr in einer Projektaufgabe selber erfinden. Im kontextsensitiven Fall sind es die *linear beschränkten Turingmaschinen*; die kommen im nächsten Kapitel vor.

6.7 Aufgaben

Aufgabe 6.1: a) Welche Sprache erzeugt die Grammatik

$$S \to a \mid b \mid aa \mid bb \mid aSa \mid bSb$$

aus dem Startsymbol S? Warum ist die Grammatik nicht regulär? Warum ist die erzeugte Sprache nicht regulär?

b) Beantwortet dieselben Fragen für die Grammatik

$$\begin{aligned} S &\to BaSbA \mid aSbA \mid BaSb \mid aSb \mid BcA \mid c \\ A &\to aA \mid a \\ B &\to Bb \mid b \end{aligned}$$

Aufgabe 6.2: Welche der drei folgenden Grammatiken sind regulär? Welche der drei (jeweils aus dem Startsymbol S) erzeugten Sprachen sind regulär?

$$\begin{array}{lll} S \to AB & S \to aA \mid ab & S \to aS \mid bA \\ A \to aA \mid a & A \to Sa & A \to bA \mid b \\ B \to bB \mid b & B \to bB \mid b & B \to bB \mid b \end{array}$$

Aufgabe 6.3: Schreibt kontextfreie Grammatiken, die die folgenden Sprachen erzeugen:

$$\begin{aligned} L_1 &= \{ww^{-1} \mid w \in \{a, b\}^*\} \\ L_2 &= \{a^n b^m \mid n, m \in \mathbb{N},\ n \neq m\} \end{aligned}$$

Beweist nun, daß L_2 nicht regulär ist. Dies ist nicht ganz einfach, denn die Tricks, die wir bei der Sprache $\{a^n b^n \mid n \in \mathbb{N}\}$ benutzt haben, funktionieren hier nicht.

Die Idee ist, einen Widerspruchsbeweis zu führen. Nehmt an, daß L_2 regulär wäre. Nun könnt Ihr die Abschlußeigenschaften der regulären Sprachen anwenden. Zunächst folgt, daß auch $\overline{L_2}$ regulär wäre. Beachtet, daß das Komplement von L_2 *nicht* die Sprache $\{a^n b^n \mid n \in \mathbb{N}\}$ ist, sondern eine Obermenge hiervon. Da die regulären Sprachen auch unter Schnitt abgeschlossen sind, läßt sich nun aber trotzdem geschickt folgern, daß mit $\overline{L_2}$ auch $\{a^n b^n \mid n \in \mathbb{N}\}$ regulär sein müßte. Dies liefert aber einen Widerspruch (warum?).

Aufgabe 6.4: Welche Sprache erzeugt die Grammatik $S \to aSb \mid ab$ aus S?

Aufgabe 6.5: Welche der folgenden Sprachen über dem Alphabet $\Sigma = \{a, b\}$ ist nicht regulär, welche ist regulär? Falls eine Sprache nicht regulär ist, beweist dies mit der Methode aus Satz 6.4. Falls eine Sprache regulär ist, gebt eine reguläre Grammatik oder einen endlichen Automaten für sie an an.

1. $L_1 = \{a^{2n} \mid n \in \mathbb{N}\}$
2. $L_2 = \{a^m b^n a^{n+m} \mid n, m \in \mathbb{N}\}$
3. $L_3 = \{w \in \Sigma^* \mid w \text{ enthält } aaa \text{ nicht als Teilwort}\}$
4. $L_4 = \{w \in \Sigma^* \mid w = w^{-1}\}$
5. $L_5 = \{a^{n^2} \mid n \in \mathbb{N}\}$

Aufgabe 6.6: Sei M ein endlicher Automat mit Zustandsmenge Q. Beweist:

$L(M)$ ist genau dann unendlich, wenn es ein $w \in L(M)$ gibt mit $|w| \geq |Q|$.

Aufgabe 6.7: Entwerft eine kontextfreie Grammatik G_{HTML} für „Mini-HTML". Die Sprache $L(G_{\mathrm{HTML}})$ soll korrekte HTML-Texte enthalten. Fehlerhafte HTML-Texte sollen hingegen nicht Wörter der Sprache sein. So sollte der Beispieltext unten ein Wort der Sprache $L(G_{\mathrm{HTML}})$ sein. Natürlich braucht ihr keine Grammatik für die gesamte HTML konstruieren, aber das Beispiel unten sollte sich schon ableiten lassen.

HTML steht für „Hypertext Markup Language" und ist die derzeitige Sprache des World Wide Web. Eine HTML-Seite besteht aus einer großen Menge verschachtelter Abschnitte. Ein Abschnitt wird durch ein sogenanntes *öffnendes Tag* eröffnet. Diese öffnenden Tags stehen immer in spitzen Klammern, wie zum Beispiel `<html>` oder `<title>`. Nach dem öffnenden Tag kommt eine beliebige Menge Text, der wieder Abschnitte enthalten darf. Am Ende eines Abschnitts muß passend zum öffnenden Tag ein *schließendes Tag* stehen. Es ist auch von spitzen Klammern umschlossen, aber zusätzlich ist noch ein Schrägstrich vorangestellt, wie in `</html>` oder `</title>`.

Wenn Ihr genauer wissen wollt, wie HTML funktioniert, dann schaut doch mal auf den Webseiten des World Wide Web Consortiums vorbei unter `www.w3c.org`.

Hier noch der Beispiel-HTML-Text, der sich aus Eurer Grammatik ableiten lassen sollte:

```
<html>
<head>
  <title> Beispiel f&uuml;r eine Webseite. </title>
</head>
<body>
  <p> Toll, das ist eine Webseite. </p>
  <p> Hier ist ein zweiter Absatz. </p>
</body>
</html>
```

Der folgende Text sollte sich nicht aus Eurer Grammatik ableiten lassen, da die Tags falsch verschachtelt sind: `<body> <p> Hallo </body> </p>`.

7 Deterministische Turingmaschinen

Alle bisher betrachteten Automaten „stammen" vom Konzept her, nicht historisch, von den Turingmaschinen „ab". Diese sind ein wichtiges Berechnungsmodell in der Algorithmen- und Komplexitätstheorie. In der Theorie formaler Sprachen sind sie das Automatenmodell für das Akzeptieren der Sprachen, die von allgemeinen Chomsky-Grammatiken erzeugt werden. Unserem „roten Faden" folgend werden wir erst deterministische, dann nichtdeterministische Turingmaschinen und dann allgemeine Grammatiken behandeln und dabei zeigen, daß alle drei Modelle äquivalent sind.

7.1 Was sind Turingmaschinen?

In seinem Aufsatz „On Computable Numbers, with an Application to the Entscheidungsproblem" von 1936 begründet Alan M. Turing seine Definition der „automatischen Maschinen" (die Church schon 1937 in seiner Besprechung Turingmaschinen nannte). Turing stellt sich einen Menschen vor, der (allein im stillen Kämmerlein ...) rechnet. Von „außen" können wir erkennen, daß er auf einem karierten Blatt Papier Zeichen schreibt, sie wegradiert und sie evtl. durch andere Zeichen ersetzt. Welche Zeichen er schreibt, hängt von den schon geschriebenen Zeichen ab und von dem, „was in seinem Kopf passiert". Turing nennt das den „Geistes-Zustand" (state of mind). Da das Volumen so eines Wissenschaftlerkopfes endlich ist, kann der auch nur endlich viele „Geistes-Zustände" annehmen. (Turing argumentiert dabei richtig psychologisch.) Mit der gleichen Argumentation rechtfertigt er, daß eine endliche Menge von Symbolen (auf dem Papier) zum Rechnen ausreicht. (Wenn Ihr mir nicht glaubt, daß Turing seine Definition so rechtfertigt, dann könnt Ihr ja 7.6 lesen.) Turingmaschinen sind (grob betrachtet) endliche Automaten mit einigen zusätzlichen Ausstattungen und „Fähigkeiten":

- Es gibt „Bänder", die in „Felder" unterteilt sind. Auf jedes Feld kann mit einem „Schreib-Lesekopf" ein Zeichen „geschrieben" oder „gelesen" werden. Die gespeicherten Zeichen können überschrieben werden. Der Schreib-Lesekopf kann sich auf dem Band in beide Richtungen (also nach „rechts" oder „links" oder „vorwärts" oder „rückwärts") bewegen. Diese Bänder sind nach beiden Seiten nicht beschränkt.
- Die Eingabe steht auch auf einem solchen Band und kann in beide Richtungen beliebig gelesen werden. (Fassen wir endliche Automaten als Turingmaschinen auf, so haben sie nur dieses Band und können es nur von links nach rechts lesen.)

7.2 Welches „Modell" verwenden wir?

Es gibt die unterschiedlichsten Möglichkeiten, Turingmaschinen zu definieren. Mal ist das Eingabeband nur auf einer Seite unbeschränkt, mal darf die Eingabe nicht überschrieben werden. Manchmal haben Turingmaschinen zusätzliche „Hilfsbänder“ und manchmal eben nicht. Im folgenden beschreiben wir das „Modell“, das wir in dieser LV benutzen. Die genauen Definitionen kommen danach.

- Die Turingmaschine hat eine Anzahl von beidseitig unbeschränkten *Bändern* mit jeweils einem *Schreib-Lesekopf*. Die Zahl der Bänder ist für alle Eingaben gleich.
- Eins der Bänder dient als *Eingabeband*. Es wird, wenn es weitere Bänder gibt, i.a. nur gelesen und nicht beschrieben. Zu Beginn der Berechnung steht der Kopf dieses Bandes auf dem ersten Symbol der Eingabe, das ist i.a. das am weitesten links stehende Zeichen.
- Die übrigen Bänder sind *Arbeitsbänder*; sie dienen als „Speicher“ und können deshalb beliebig beschrieben und gelesen werden.
- Bis auf die Eingabe sind die Felder aller Bänder mit einem speziellen Zeichen ($\Box$ für *Blanksymbol*) initialisiert. Die Turingmaschine kann dadurch den linken und den rechten Rand der Eingabe erkennen.
- Die *Überführungsfunktion* δ ist eine partielle Funktion, die zu einem Zustand und gelesenen Symbolen die zu schreibenden Symbole, die Kopfbewegungen und den Folgezustand angibt, falls sie definiert ist. Die Turingmaschine „hält an“, wenn δ nicht mehr angewendet werden kann, d.h. wenn sie in einer *Endkonfiguration* angekommen ist.
- Es gibt eine Menge von *akzeptierenden* Zuständen. Die Turingmaschine *akzeptiert* die Eingabe, wenn sie in einer akzeptierenden Endkonfiguration anhält.

Die Bestandteile einer Turingmaschine werden wir in 7.3 formal definieren, wie sie zusammenwirken in 7.4.

7.3 Definition: deterministische Turingmaschine

Eine deterministische Turingmaschine (DTM) M wird durch acht Angaben spezifiziert; wir schreiben auch $M = (Q, \Sigma, \Gamma, \Box, k, \delta, q_A, E)$. Dabei ist

- Q eine endliche Menge von *Zuständen*,
- Σ ein Alphabet von *Eingabesymbolen*,
- Γ mit $\Sigma \subset \Gamma$ ein Alphabet mit zusätzlichen Symbolen (*Bandalphabet*),
- $\Box \in \Gamma \setminus \Sigma$ das *Blanksymbol* zur Darstellung leerer Felder,

- $k \in \mathbb{N}^+$ die *Anzahl der Bänder*,
- $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, N, R\}^k$ die partielle *Überführungsfunktion*,
- $q_A \in Q$ der *Anfangszustand*,
- $E \subseteq Q$ die Menge der *akzeptierenden Zustände*.

7.4 Konfiguration, Berechnung, akzeptierte Sprache

Durch die Definition haben wir die Bestandteile einer Turingmaschine festgelegt, aber nicht gesagt, wie sie mit der intuitiven Vorstellung von Turingmaschinen zusammenhängen. So gibt es im Wertebereich von δ diese mysteriösen Buchstaben L, N und R. Wir können nur spekulieren, daß sie etwas mit Links, Nicht und Rechts zu tun haben. Nach den Erfahrungen mit endlichen Automaten ist es aber leicht, informell zu beschreiben, wie eine DTM M mit der Überführungsfunktion δ „arbeitet“: Es sei M im Zustand q und unter den Schreib-Leseköpfen auf den k Bändern stehen die Symbole $a_1, \ldots, a_k \in \Gamma$. Wenn

$$\delta(q, a_1, \ldots, a_k) = (q', b_1, \ldots, b_k, B_1, \ldots, B_k)$$

ist, dann ersetzt M für $i = 1, \ldots, k$ auf dem i-ten Band a_i durch $b_i \in \Gamma$, bewegt den Kopf nach $B_i \in \{L\text{inks}, N\text{icht}, R\text{echts}\}$ und geht in den Zustand $q' \in Q$; wenn $\delta(q, a_1, \ldots, a_k)$ nicht definiert ist, bleibt M stehen. In einem solchen *Berechnungsschritt* werden also der Zustand, die Bandinhalte und die Positionen der Schreib-Leseköpfe verändert. Diese machen zusammen die jeweilige *Konfiguration* von M aus. Berechnungsschritte und Folgen davon definieren also Relationen auf den Konfigurationen, die wir wieder mit $\vdash_M$ und $\vdash_M^*$ bezeichnen. (Da M deterministisch ist, sind dadurch tatsächlich Funktionen gegeben; wir können also z.B. von *der Folgekonfiguration* sprechen.) Diese *Berechnungen* werden wir wieder von Hand nachvollziehen, Ihr könntet aber auch ein Simulationsprogramm (einen „Interpreter“) für Euren Rechner schreiben (oder aus dem Netz ziehen, z.B. das *Java Formal Language & Automata Package* (JFLAP)[4]).

Das wollen wir jetzt formaler machen. Um die Definition übersichtlicher zu gestalten, tun wir das nur für Turingmaschinen ohne Arbeitsbänder, also für $k = 1$. Die Definition läßt sich leicht auf Turingmaschinen mit einer beliebigen Zahl von Arbeitsbändern übertragen. Bandinhalte stellen wir als Wörter über Γ dar; zum Beispiel repräsentieren wir das Band

$$\cdots \boxed{\Box}\boxed{\Box}\boxed{a_1}\boxed{a_2}\boxed{a_3}\boxed{a_4}\boxed{\Box}\boxed{\Box} \cdots$$

durch das Wort $a_1a_2a_3a_4$. Dabei können die a_i auch $\Box$ sein. Das Band hat in beiden Richtungen unendlich viele Felder, die $\Box$ enthalten. Davon berücksichtigen wir immer nur endlich viele.

[4] `http://www.cs.duke.edu/~rodger/tools/jflap/index.html`

Zur Darstellung der Position des Schreib-Lesekopfes fügen wir das Sonderzeichen $\triangleright$, das nicht in Γ enthalten sein soll, links vom gelesenen Symbol in das Bandwort ein. Steht der Kopf des oben dargestellten Bandes auf dem Zeichen a_2, dann wird dies durch das Wort

$$a_1 \triangleright a_2 a_3 a_4$$

repräsentiert.

Definition: Paare aus Zustand und Bandinhalt bilden die *Konfigurationen* der Maschine M, sie haben also die Form

$$(q, u \triangleright v) \quad \text{mit } q \in Q,\ u, v \in \Gamma^*.$$

Jetzt können wir die Berechnungsschritte von M als zweistellige Relation $\vdash_M$ auf den Konfigurationen von M definieren; immer sei $q, q' \in Q$, $a, b, c \in \Gamma$, $u, v \in \Gamma^*$ und

$$\delta(q, a) = (q', b, B) \quad \text{mit} \quad B \in \{L, N, R\}.$$

Zuerst behandeln wir den Fall, bei dem der Kopf nicht ganz am Ende steht.

$$\begin{aligned}
(q, uc \triangleright av) &\vdash_M (q', u \triangleright cbv) && \text{für } B = L,\\
(q, \triangleright av) &\vdash_M (q', \triangleright \Box bv) && \text{für } B = L,\\
(q, u \triangleright av) &\vdash_M (q', u \triangleright bv) && \text{für } B = N,\\
(q, u \triangleright av) &\vdash_M (q', ub \triangleright v) && \text{für } B = R.
\end{aligned}$$

Falls wir am rechten Rand stehen, treten folgende Fälle ein für $\delta(q, \Box) = (q', b, B)$

$$\begin{aligned}
(q, uc \triangleright) &\vdash_M (q', u \triangleright cb) && \text{für } B = L,\\
(q, \triangleright) &\vdash_M (q', \triangleright \Box b) && \text{für } B = L,\\
(q, u \triangleright) &\vdash_M (q', u \triangleright b) && \text{für } B = N,\\
(q, u \triangleright) &\vdash_M (q', ub \triangleright) && \text{für } B = R.
\end{aligned}$$

Definition:

- Die Konfiguration $k_A = (q_A, \triangleright w, \triangleright, \ldots, \triangleright)$ nennen wir *Anfangskonfiguration* von M zur Eingabe w. Gilt $k \vdash_M k'$, so nennen wir k' *Folgekonfiguration* von k. Eine Konfiguration ohne Folgekonfiguration nennen wir *Endkonfiguration*.

- Eine endliche oder unendliche Folge $k_0 \vdash_M k_1 \vdash_M \cdots k_i \vdash_M k_{i+1} \vdash_M \cdots$ von Konfigurationen nennen wir *Berechnung* von M. Die *Länge* einer endlichen Berechnung ist die Zahl ihrer Schritte; eine Berechnung, die keine Schritte macht und nur aus einer Konfiguration besteht, hat also die Länge 0. Wir schreiben $k \vdash_M^* k'$, wenn die Berechnung von M, die mit k beginnt, zu k' führt.

  Eine Berechnung, die in einer Endkonfiguration endet, heißt *terminierend*. Die (eindeutig bestimmte, terminierende oder unendliche) Berechnung von M, die mit der Anfangskonfiguration zur Eingabe w beginnt, nennen wir *Berechnung zur Eingabe* w.

- Eine Endkonfiguration mit Zustand aus E heißt *akzeptierend*. Eine terminierende Berechnung, die in einer akzeptierenden Endkonfiguration endet, heißt *akzeptierend*. *M akzeptiert $w \in \Sigma^*$*, wenn die Berechnung von M zur Eingabe w akzeptierend ist. Die von M *akzeptierte Sprache* ist $L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$. Eine Sprache heißt *akzeptierbar*, wenn es eine DTM gibt, die sie akzeptiert.

Im Gegensatz zu den endlichen Automaten können Turingmaschinen beliebig lange laufen, z.B. mit dem Schreib-Lesekopf über die Eingabe hinaus, ohne anzuhalten. Erst wenn für einen Zustand q und für gelesene Symbole $a_1, \ldots, a_k$ die Funktion $\delta(q, a_1, \ldots, a_k)$ nicht definiert ist, dann ist eine Endkonfiguration erreicht, die Maschine hält an.

Fassen wir nochmal zusammen, was eine Turingmaschine M bei Eingabe eines Wortes $w \in \Sigma^*$ alles machen kann:

- Sie kann anhalten (terminieren) und das in einem akzeptierenden Zustand. Dann wird das Wort w akzeptiert und $w \in L(M)$.

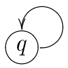
- Sie kann anhalten (terminieren), aber das in einem nicht akzeptierenden Zustand. Dann wird das Wort w nicht akzeptiert und $w \notin L(M)$.

- Sie kann nicht anhalten. Dann wird das Wort nicht akzeptiert und $w \notin L(M)$.

Turingmaschinen ohne Arbeitsband heißen *Ein-Band-Maschinen*; sie führen alle Berechnungen auf dem Eingabeband durch, insbesondere beschreiben sie es. Weil sie keine Arbeitsbänder haben, arbeiten sie umständlich und sind kompliziert, sie lassen sich aber noch einigermaßen übersichtlich mit Zustandsgraphen darstellen. Wir stellen Übergänge

$$\delta(q, a) = (q', b, B) \quad \text{durch} \quad \textcircled{q} \xrightarrow{a,b,B} \textcircled{q'}$$

dar. Für akzeptierende Zustände verwenden wir wie bei den endlichen Automaten $\circledcirc$. Abkürzend schreiben wir x als Platzhalter für ein beliebiges Eingabezeichen; ist $\Sigma = \{a, b, c\}$, so steht also

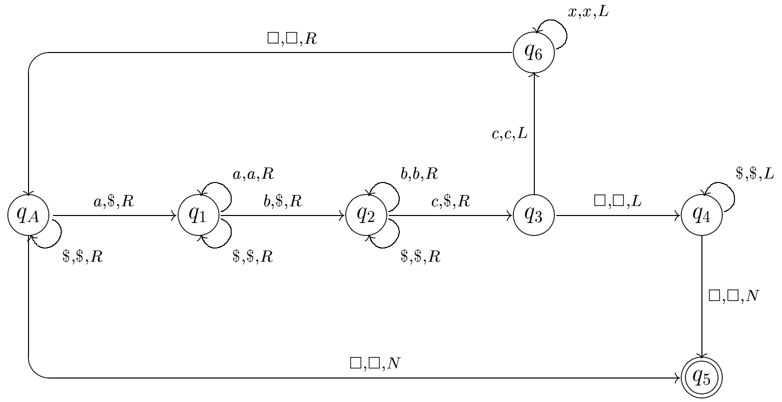

x, x, L

für die drei Übergänge $\delta(q, a) = (q, a, L)$, $\delta(q, b) = (q, b, L)$, $\delta(q, c) = (q, c, L)$. Diese Darstellung läßt sich auf Turingmaschinen mit wenigen Arbeitsbändern verallgemeinern.

Als Beispiel betrachten wir eine deterministischen Turingmaschine, die die Sprache

$$\{a^n b^n c^n \mid n \in \mathbb{N}\}$$

akzeptiert, und zwar $M = (\{q_A, q_1, q_2, q_3, q_4, q_5, q_6\}, \{a, b, c\}, \{a, b, c, \Box, \$\}, \Box, 1, \delta, q_A, \{q_5\})$, wobei δ durch das folgende Diagramm gegeben ist:



M ersetzt in einer Schleife jeweils ein a, ein b und ein c durch $\$$; wenn im letzten Schleifendurchlauf das Wort nur aus $\$$'s besteht, dann wird die Eingabe akzeptiert. Betrachten wir die Berechnung für die Eingabe $aaabbbccc$:

$(q_A, \triangleright aaabbbccc)$	Im Zustand q_A wird ein a durch $\$$ ersetzt.
$\vdash$	
$(q_1, \$\triangleright aabbbccc)$	Im Zustand q_1 wird der Schreib-Lesekopf auf das erste b positioniert.
$\vdash$	
$(q_1, \$a\triangleright abbbccc)$	
$\vdash$	
$(q_1, \$aa\triangleright bbbccc)$	Das b wird dann durch $\$$ ersetzt.
$\vdash$	
$(q_2, \$aa\$\triangleright bbccc)$	Im Zustand q_2 wird das erste c gesucht ...
$\vdash$	
$(q_2, \$aa\$b\triangleright bccc)$	
$\vdash$	
$(q_2, \$aa\$bb\triangleright ccc)$	... und durch $\$$ ersetzt.
$\vdash$	
$(q_3, \$aa\$bb\$\triangleright cc)$	Im Zustand q_3 wird geprüft, ob das Ende der Eingabe erreicht wurde.
$\vdash$	
$(q_6, \$aa\$bb\triangleright\$cc)$	Im Zustand q_6 läuft der Schreib-Lesekopf auf den Wortanfang zurück.
$\vdash$	
$(q_6, \$aa\$b\triangleright b\$cc)$	
$\vdash$	
$(q_6, \$aa\$\triangleright bb\$cc)$	
$\vdash$	
$(q_6, \$aa\triangleright\$bb\$cc)$	
$\vdash$	
$(q_6, \$a\triangleright a\$bb\$cc)$	
$\vdash$	

$(q_6, \$\triangleright aa\$bb\$cc)$	
$\vdash$	
$(q_6, \triangleright\$aa\$bb\$cc)$	
$\vdash$	
$(q_6, \triangleright\Box\$aa\$bb\$cc)$	Das $\Box$ signalisiert den Wortanfang.
$\vdash$	
$(q_A, \Box\triangleright\$aa\$bb\$cc)$	Die $\$$'s werden in q_A überlesen.
$\vdash$	
$(q_A, \Box\$\triangleright aa\$bb\$cc)$	
$\vdash$	
$(q_1, \Box\$\$\triangleright a\$bb\$cc)$	Da sich die Rechnung wiederholt, kürze ich die Darstellung etwas ab ...
$\vdash_*$	
$(q_1, \Box\$\$a\$\triangleright bb\$cc)$	
$\vdash$	
$(q_2, \Box\$\$a\$\$\triangleright b\$cc)$	
$\vdash_*$	
$(q_2, \Box\$\$a\$\$b\$\triangleright cc)$	
$\vdash$	
$(q_3, \Box\$\$a\$\$b\$\$\triangleright c)$	
$\vdash$	
$(q_6, \Box\$\$a\$\$b\$\triangleright\$c)$	
$\vdash_*$	
$(q_6, \triangleright\Box\$\$a\$\$b\$\$c)$	
$\vdash$	
$(q_A, \Box\triangleright\$\$a\$\$b\$\$c)$	Jetzt wird noch mehr abgekürzt ...
$\vdash_*$	
$(q_3, \Box\$\$\$\$\$\$\$\$\$\triangleright)$	
$\vdash$	
$(q_4, \Box\$\$\$\$\$\$\$\$\triangleright\$)$	Im Zustand q_4 wird jetzt geprüft, ob die Anzahlen gleich waren, ...
$\vdash_*$	
$(q_4, \triangleright\Box\$\$\$\$\$\$\$\$\$)$	... was hier der Fall war.
$\vdash$	
$(q_5, \triangleright\Box\$\$\$\$\$\$\$\$\$)$	

Bemerkungen:

- Das ist natürlich noch kein Beweis, daß die Maschine korrekt ist. Der geht nach bewährtem Muster: Außer Eingaben $\in L$ auch solche $\notin L$ betrachten; Berechnungen von Teilmaschinen untersuchen; Beweis über Teilberechnungen mit Induktion.
- Die Turingmaschine benutzt zum Akzeptieren eines Wortes nur die Felder der Eingabe, sie benötigt keinen zusätzlichen Speicherplatz. (Nichtdeterministische) Turingmaschinen mit dieser Einschränkung bilden einen eigenen Automatentyp (LBA: *L*inear *B*eschränkte *A*utomaten). Sie akzeptieren genau die kontextsensitiven Sprachen; siehe Abschnitte 6.5 und 6.6.
- An dieser Turingmaschine wird deutlich, warum das Eingabealphabet Σ in der Spezifikation der Turingmaschine enthalten sein muß. Für $\Sigma = \{a, b, c\}$ akzeptiert M die

Sprache $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. Die „gleiche“ Turingmaschine, aber mit dem erweiterten Alphabet $\Sigma' = \{a, b, c, \$\}$, akzeptiert die Sprache $\{\$^i a^n \$^j b^n \$^k c^n \mid i, j, k, n \in \mathbb{N}\}$.

7.5 Arbeiten mit Turingmaschinen

Es gibt nur einen Weg, wirklich etwas über Turingmaschinen zu lernen: selbst viele konstruieren und beweisen, welche Sprache sie akzeptieren. Ein gutes Hilfsmittel ist dabei, aus einfachen Maschinen kompliziertere zusammenzusetzen; auch das probiert man am besten selbst aus. Da Turingmaschinen ihre Eingaben und Zwischenergebnisse speichern und wiederverwenden können, ist das i.a. einfacher als bei endlichen Automaten. Zum Beispiel kann man Turingmaschinen auf mehreren Bändern parallel rechnen lassen. Insbesondere kann man zwei Turingmaschinen „parallel schalten“. Sind z.B. M_1 und M_2 zwei Ein-Band-Maschinen mit Zuständen Q_i, Symbolen Γ_i, Überführungen δ_i für $i = 1, 2$, so ist M mit $Q = Q_1 \times Q_2$, $\Gamma = \Gamma_1 \cup \Gamma_2$ und

$$\delta\big((q_1, q_2), a_1, a_2\big) = \big((q_1', q_2'), b_1, b_2, B_1, B_2\big),$$

falls

$$\delta_i(q_i, a_i) = (q_i', b_i, B_i) \text{ für } i = 1, 2$$

eine Zwei-Band-Maschine, die auf dem einen Band wie M_1 und auf dem anderen wie M_2 arbeitet. Wie macht man das mit der Eingabe? Und mit dem Anfangen und Akzeptieren? – Viel einfacher ist es, Turingmaschinen „hintereinander zu schalten“. Aber nur, wenn die erste für jede Eingabe terminiert; sonst gibt es unlösbare Probleme. Welche?

Beide Methoden könnt Ihr an dem folgenden Satz ausprobieren, der einen Teil von Satz 5.3 (für reguläre Sprachen) verallgemeinert.

Satz 7.5.1: *Die akzeptierbaren Sprachen sind abgeschlossen unter Durchschnitt und Vereinigung (Abschnitt 1.4), d.h. wenn L_1 und L_2 akzeptierbar sind, so auch $L_1 \cap L_2$ und $L_1 \cup L_2$.*

Der Satz gilt auch für die Kleeneschen Operationen, ist aber mit Hilfe nichtdeterministischer Turingmaschinen viel einfacher zu beweisen; siehe Abschnitt 8.4. Für das Komplement ist der Satz falsch! Das liegt an den nichtterminierenden Maschinen; siehe Kapitel 11.

Beim Konstruieren von Turingmaschinen braucht man oft Teilmaschinen, die gewisse Zwischenergebnisse erzielen und „abliefern“, die also Funktionen berechnen. Tatsächlich werden Turingmaschinen oft so definiert und unsere „Akzeptoren“ als spezielle „Berechner“ aufgefaßt, die „ja“ und „nein“ abliefern.

Definition: Seien Σ_1 und Σ_2 zwei Alphabete, sei $f : \Sigma_1^* \to \Sigma_2^*$ eine partielle Funktion. Eine *Rechen-Turingmaschine für f* ist eine DTM M mit folgenden Eigenschaften:

- M hat neben Σ_1 als Eingabealphabet Σ_2 als *Ausgabealphabet*. Die *Ausgabe* einer terminierenden Berechnung ist ein Wort aus Σ_2^*, das bei Mehrbandmaschinen auf einem besonderen *Ausgabeband*, bei Ein-Band-Maschinen auf einem besonders markierten Teil des Bandes steht.
- Wenn $f(w) = v$ für $w \in \Sigma_1^*$, $v \in \Sigma_2^*$, dann terminiert M bei Eingabe w mit Ausgabe v. Wenn $f(w)$ nicht definiert ist, terminiert M bei Eingabe w nicht.

Eine Funktion ist *berechenbar*, wenn es eine DTM gibt, die sie berechnet.

Eine triviale Funktion, die Ihr in den Übungsaufgaben dauernd „berechnet“, ist die Identität $f(w) = w$ („Kopieren“); fast so trivial ist die Spiegelfunktion $f(w) = w^{-1}$ (Abschnitt 1.4) zu berechnen. Andere berechenbare Funktionen sind die arithmetischen Funktionen wie Addition oder Multiplikation auf Dezimal-, oder Binär-, oder Unärzahlen. Tatsächlich werdet Ihr keine nichtberechenbaren Funktionen finden; das schaffen wir erst in Kapitel 11. Wie Akzeptieren und Berechnen zusammenhängen, könnt Ihr Euch selbst überlegen; auch damit beschäftigen wir uns genauer in Kapitel 11.

Für konkrete Konstruktionen sind Mehrbandmaschinen meist viel einfacher, wie Ihr an den Übungsaufgaben gemerkt habt. Für allgemeine Beweise wie bei Satz 9.3.2 sind Ein-Band-Maschinen oft angenehmer. Tatsächlich kann man sich auf solche beschränken:

Satz 7.5.2: *Zu jeder DTM mit Arbeitsbändern kann man eine Ein-Band-Maschine konstruieren, die dieselbe Sprache akzeptiert.*

Beweis: Sei $M = (Q, \Sigma, \Gamma, \Box, k, \delta, q_A, E)$ eine DTM mit $k > 1$ Bändern. Wir konstruieren eine DTM M_1, die die Berechnungen von M auf einem Band simuliert, auf dem die k Bandinhalte von M hintereinanderstehen. Dazu erweitern wir Γ um die Symbole $\triangleright$ und $|$. Wenn $(q, v_1, \ldots, v_n)$ eine Konfiguration von M mit $v_i \in (\Gamma \cup \{\triangleright\})^*$ wie in Abschnitt 7.4 ist, steht $v_1| \ldots |v_n$ auf dem Band von M_1 und q ist im Zustand von M_1 „gespeichert“. Um einen Schritt von M zu simulieren, läuft M_1 einmal von links nach rechts über ihr Band und „merkt sich“ (auch in ihren Zuständen) die k Symbole unter den Köpfen von M; wenn sie rechts angekommen ist, läuft sie zurück nach links und verändert dabei die Konfigurationen, wie es die Überführungsfunktion δ vorschreibt, die in M_1 „einprogrammiert“ ist. Wenn dabei Konfigurationen länger werden, kopiert M_1 beim Laufen alle Symbole entsprechend viele Felder weiter (das entfällt, wenn man die Bänder von M nicht neben-, sondern „untereinander“ auf das Band von M_1 legt – ein alternativer Beweis, bei dem man $(\Gamma \cup \{\Box, \triangleright\})^k$ als Bandalphabet von M_1 benutzt, um die Symbole und die Kopfpositionen von M zu speichern.) Zu Beginn wandelt M_1 selbst die Eingabe in die Anfangskonfiguration von M um. (Wie geht das?) M_1 terminiert und akzeptiert genau dann, wenn M es tun würde. $\Box$

7.6 Alan Turing und der Geisteszustand

In diesem Anhang zitieren wir aus dem Artikel *On Computable Numbers, with an Application to the Entscheidungsproblem* von Alan Turing, den wir schon in Abschnitt 4.6

erwähnt haben. Der Artikel ist 1936 eingereicht, aber erst 1937 erschienen[5]. Die deutsche Übersetzung stammt von Bernhard Siegert, wir haben sie dem Buch *Alan Turing – Intelligence Service*[6] entnommen[7].

Das erste Zitat ist so etwas wie eine „Definition" einer Turingmaschine. (Irgendwelche Irgendwie-Tupel kommen in dem Artikel nicht vor!). Was wir heutzutage Zustand nennen, nennt Turing „m-Zustand" („m" wie „machine"); er leitet sie (im zweiten Zitat) nämlich von Geistes„zuständen" („states of mind") her.

Wir können einen Mann, der gerade eine reelle Zahl berechnet, mit einer Maschine vergleichen, die nur über eine endliche Zahl von Zuständen[8] $q_1, q_2, \ldots, q_R$ verfügt, die ihre „m-Zustände" heißen sollen. Die Maschine wird von einem „Band" versorgt, das (analog zum Papier) durch sie hindurchläuft und in Sektionen („Felder" genannt) aufgeteilt ist, von denen jedes ein „Symbol" tragen kann. Zu jedem gegebenen Zeitpunkt ist es genau ein Feld, etwa das r-te mit dem Symbol $\mathcal{S}(r)$, das „in der Maschine" ist. Wir können dieses Feld das „abgetastete Feld" nennen. Das Symbol auf dem abgetasteten Feld kann „abgetastetes Symbol" heißen. Das „abgetastete Symbol" ist das einzige, dessen sich die Maschine sozusagen „direkt bewußt" ist. Durch Änderung ihres m-Zustands jedoch kann die Maschine einige der Symbole, die sie vorher „gesehen" (abgetastet) hat, effektiv erinnern. Das mögliche Verhalten der Maschine zu jedem Zeitpunkt wird bestimmt vom m-Zustand q_n und dem abgetasteten Symbol $\mathcal{S}(r)$. Dieses Paar $q_n, \mathcal{S}(r)$ soll die „Konfiguration" heißen: dergestalt bestimmt die Konfiguration das mögliche Verhalten der Maschine. In einigen der Konfigurationen, in denen das abgetastete Feld leer ist (d.h. kein Symbol trägt), schreibt die Maschine ein neues Symbol aufs abgetastete Feld: in anderen Konfigurationen tilgt sie das abgetastete Symbol. Die Maschine kann auch das Feld ändern, das sie abtastet, aber nur durch Verschiebung um eine Stelle nach rechts oder links. Zuzüglich zu jeder dieser Operationen kann die m-Konfiguration geändert werden. Einige der niedergeschriebenen Symbole werden die Folge von Ziffern bilden, die den Dezimalausdruck der berechneten reellen Zahl darstellt. Die anderen sind einfach Stichworte zur „Gedächtnisstütze". Und nur diese Stichworte sind von Tilgungen betroffen.

In dem zweiten Zitat begründet Turing seine Definitionen.

Rechnungen werden für gewöhnlich in der Weise ausgeführt, daß bestimmte Symbole auf ein Stück Papier geschrieben werden. Wir wollen annehmen, daß dieses Stück Papier kariert ist, wie das Rechenheft eines Kindes. Beim elementaren Rechnen wird zuweilen die Zweidimensionalität des Papiers ausgenutzt. Aber diese Verwendungsart ist keineswegs unvermeidlich, und ich denke, man wird sich darüber einig sein, daß die Zweidimensionalität des Papiers für die Rechnung nicht wesentlich ist. Ich gehe daher davon aus, daß die

[5] Proceedings of the London Mathematical Society (2) 42, 1937.

[6] Brinkmann & Bose, Berlin, 1987.

[7] Allerdings werden dort sowohl „Zustand" also auch „Konfiguration" immer mit „Zustand" übersetzt; das wurde jetzt wieder dem Original angeglichen.

[8] Bei Turing „conditions".

Rechnung auf eindimensionalem Papier ausgeführt wird, d.h. auf einem durch Felder unterteilten Band. Ich werde außerdem unterstellen, daß die Anzahl der Symbole, die gedruckt werden können, endlich ist. Ließen wir unendlich viele Symbole zu, gäbe es Symbole, die sich nur noch in einem willkürlich kleinen Maße voneinander unterscheiden[9]. Die Auswirkungen dieser Einschränkung des Symbolvorrats sind nicht sehr entscheidend. Es bleibt immer möglich, Symbolfolgen anstelle einfacher Symbole zu verwenden. So wird eine arabische Ziffer wie 17 oder 999999999999999 gewöhnlich wie ein einfaches Symbol behandelt. Ähnlich werden Wörter in jeder europäischen Sprache wie einfache Symbole behandelt (Chinesisch bemüht sich dagegen um eine abzählbare Unendlichkeit von Symbolen). Von unserem Standpunkt aus bestehen die Unterschiede zwischen einfachen und zusammengesetzten Symbolen darin, daß zusammengesetzte Symbole nicht mit einem Blick erfaßt werden können, wenn sie zu lang sind. Das stimmt mit der Erfahrung überein. Wir können nicht auf einen Blick sagen, ob 9999999999999999 und 999999999999999 gleich sind.
Das Verhalten des Rechnenden wird zu jedem Zeitpunkt durch die wahrgenommenen Symbole und durch seinen momentanen „Geisteszustand" bestimmt. Wir gehen davon aus, daß es eine Grenze G für die Anzahl der Symbole gibt, die der Rechnende in einem Augenblick erfassen kann. Will er mehr erfassen, muß er sich einer Reihe aufeinanderfolgender Wahrnehmungsakte bedienen. Wir gehen außerdem davon aus, daß die Zahl der Geisteszustände, die berücksichtigt werden müssen, endlich ist. Die Gründe hierfür sind dieselben wie für die Einschränkung des Symbolvorrats. Ließen wir eine unendliche Anzahl von Geisteszuständen zu, lägen einige von ihnen „willkürlich dicht" beieinander und würden verwechselt. Auch hier beeinträchtigt die Beschränkung die Rechnung nicht ernsthaft, da kompliziertere Geisteszustände unnötig werden, indem einfach mehr Symbole auf das Band geschrieben werden.
Wir stellen uns die vom Rechnenden durchgeführten Operationen in „einfache Operationen" aufgeteilt vor, die so elementar sind, daß es schwer fällt, sie sich noch weiter aufgespalten vorzustellen. Jede dieser Operationen besteht in irgendeiner Veränderung im physikalischen System, das vom Rechnenden und seinem Band gebildet wird. Wir kennen den Zustand des Systems, wenn wir die Symbolfolge auf dem Band, die Symbole, die vom Rechnenden (gegebenenfalls mittels eines besonderen Befehls) wahrgenommen werden, und den Geisteszustand des Rechnenden kennen. Wir nehmen an, daß in einer einfachen Operation nicht mehr als ein Symbol verändert wird. Alle anderen Veränderungen können auf einfache Veränderungen dieser Art zurückgeführt werden. Bei den Feldern, deren Symbole auf diese Weise verändert werden können, handelt es sich um dasselbe Problem wie bei den wahrgenommenen Feldern. Ohne an Allgemeingültigkeit zu verlieren, können wir da-

[9] *Wenn wir ein Symbol buchstäblich als etwas Gedrucktes auf einem Feld verstehen, dann können wir annehmen, daß das Feld $0 \leq x \leq 1, 0 \leq y \leq 1$ ist. Das Symbol wird als Menge von Punkten in diesem Feld definiert, nämlich als die Menge der Punkte, die mit Druckerschwärze gefärbt sind. Wenn diese Mengen innerhalb der Grenze des Meßbaren bleiben sollen, können wir den „Abstand" zwischen zwei Symbolen als den zur Umformung eines Symbols in ein anderes erforderlichen Aufwand definieren, wenn der Aufwand, die Einheitsfläche von Druckerschwärze um den Einheitsabstand zu verschieben, Eins beträgt und unendlich viel Schwärze bei $x = 2, y = 0$ zur Verfügung steht. Mit dieser Topologie bilden die Symbole einen bedingt kompakten Raum.*

her unterstellen, daß Felder, deren Symbole verändert werden, immer „wahrgenommene“ Felder sind.
Die einfachen Operationen müssen neben diesen Symbolveränderungen auch Distributionswechsel der wahrgenommenen Felder umfassen. Der Rechnende muß die neu wahrgenommenen Felder augenblicklich erkennen können. Ich halte es für vernünftig, zu unterstellen, daß es nur Felder sein können, deren Abstand von den nächstliegenden der unmittelbar vorher wahrgenommenen Felder eine bestimmte Größe nicht überschreitet. Wir wollen festlegen, daß alle neu wahrgenommenen Felder innerhalb eines Abstands von L Feldern vom unmittelbar zuvor wahrgenommenen Feld liegen.
Man könnte im Zusammenhang mit der „augenblicklichen Erkennbarkeit“ auf den Gedanken kommen, daß es noch andere augenblicklich erkennbare Felder gibt. Insbesondere könnten durch spezielle Symbole markierte Felder für augenblicklich erkennbar gehalten werden. Da aber nun diese Felder nur durch ein einziges Symbol markiert sind, kann es nur eine endliche Anzahl von ihnen geben, und wir sollten unsere Theorie nicht dadurch verwirren, indem wir diese markierten Felder mit den wahrgenommenen Feldern zusammenwerfen. Wenn sie allerdings durch eine Folge von Symbolen markiert sind, können wir die Erkennungsoperation nicht als eine einfache ansehen. Dies ist ein grundlegender Punkt, der einer Veranschaulichung bedarf. In den meisten mathematischen Aufsätzen werden die Gleichungen und Theoreme durchnumeriert. Gewöhnlich gehen die Zahlen nicht über (etwa) 1000 hinaus. Es ist daher möglich, ein Theorem auf einen Blick anhand seiner Nummer zu erkennen. Aber falls ein Aufsatz sehr lang wäre, könnten wir zum Theorem 157767733443477 gelangen; und weiter unten im Aufsatz würden wir lesen: „... also ergibt sich (nach Anwendung des Theorems 157767733443477) ... “. Um sicher zu gehen, daß es das entsprechende Theorem ist, müßten wir die beiden Zahlen Ziffer für Ziffer vergleichen, wobei wir möglicherweise die Ziffern mit Bleistift abhaken würden, um sicherzugehen, daß wir keine doppelt gezählt haben. Sollte trotzdem noch die Meinung vertreten werden, daß es andere „augenblickliche erkennbare“ Felder gibt, so wird das solange nicht meine Behauptung beeinträchtigen, wie durch irgendein Verfahren, das mein Maschinentyp beherrscht, diese Felder aufgefunden werden können.
Die einfachen Operationen müssen daher folgendes umfassen:

(a) Veränderungen des Symbols auf einem der wahrgenommenen Felder.

(b) Veränderungen eines der wahrgenommenen Felder in ein anderes Feld im Abstand von L Feldern von einem der vorher wahrgenommenen Felder.

Es ist möglich, daß einige dieser Veränderungen notwendig eine Veränderung des Geisteszustands implizieren. Als allgemeinste einfache Operationen müssen daher die beiden folgenden gelten:

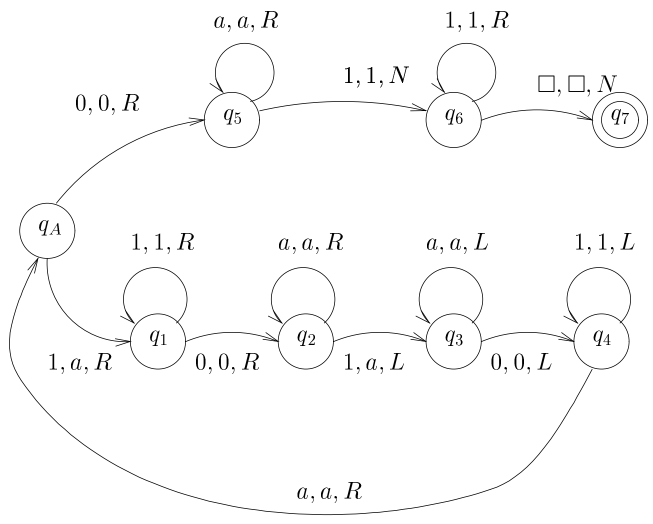
(A) Eine mögliche Symbolveränderung (a) zusammen mit einer möglichen Veränderung des Geisteszustands.

(B) Eine mögliche Veränderung (b) wahrgenommener Felder zusammen mit einer möglichen Veränderung des Geisteszustandes.

Die tatsächlich durchgeführte Operation wird durch den Geisteszustand des Rechnenden und durch die wahrgenommenen Symbole bestimmt. Insbesondere bestimmen sie den Geisteszustand des Rechnenden, nachdem die Operation ausgeführt worden ist.
Wir können nun eine Maschine konstruieren, die die Arbeit dieses Rechnenden tut. Jedem Geisteszustand des Rechnenden entspricht eine „m-Konfiguration“ der Maschine. Entsprechend den vom Rechnenden wahrgenommenen G Feldern tastet die Maschine G Felder ab. Während jeder ihrer Züge kann die Maschine ein Symbol auf einem abgetasteten Feld verändern oder kann jedes der abgetasteten Felder in ein anderers, nicht weiter als L Felder von einem der anderen abgetasteten Felder entferntes, verändern. Der ausgeführte Zug und die nachfolgende Konfiguration werden durch das abgetastete Symbol und durch die m-Konfiguration bestimmt.

7.7 Aufgaben

Aufgabe 7.1: Diese deterministische Turingmaschine M mit Eingabealphabet $\Sigma = \{0, 1\}$ und Bandalphabet $\Gamma = \{0, 1, a, \Box\}$ hat kein Arbeitsband:



Schreibt die Berechnung von M für die Eingabe 1101111 auf, ebenso für 11101. Welche Sprache akzeptiert M?

Aufgabe 7.2: Gebt eine deterministische Turingmaschine an, die die Sprache

$$L = \{w\#w^{-1}) \mid w \in \{a,b\}^*\}$$

über dem Alphabet $\{a, b, \#\}$ akzeptiert. Schreibt die Berechnungen von M für die Eingaben $ab\#ba$ und $abb\#a$ auf.

Aufgabe 7.3: Gebt eine deterministische Turingmaschine M mit einem Arbeitsband an, die ein Wort der Form 1^n mit $n > 0$ kopiert: Für M soll

$$(q_A, \triangleright 1^n, \triangleright) \vdash_M^* (q_E, 1^n01^n\triangleright, \Box 1^n \triangleright)$$

gelten; q_A ist der Anfangszustand, $(q_E, 1^n01^n\triangleright, \Box 1^n\triangleright)$ ist eine Endkonfiguration, und q_E der einzige akzeptierende Zustand.

Aufgabe 7.4: Gebt eine deterministische Turingmaschine M ohne Arbeitsband an, die ein Wort der Form 1^n mit $n > 0$ kopiert. Für M soll

$$(q_A, \triangleright 1^n) \vdash_M^* (q_E, \Box 1^n01^n\triangleright)$$

gelten; q_A ist der Anfangszustand, $(q_E, \Box 1^n01^n\triangleright)$ ist eine Endkonfiguration und q_E der einzige akzeptierende Zustand.

Aufgabe 7.5: Gebt eine deterministische Turingmaschine an, die die Sprache

$$\{a^nb^mc^k \mid n, m, k \in \mathbb{N},\ n = m \text{ oder } n = k\}$$

akzeptiert.

8 Nichtdeterministische Turingmaschinen

Alan M. Turing hatte sich bei seinen Überlegungen auf „Automatische Maschinen“, also deterministische Turingmaschinen beschränkt; siehe Abschnitt 4.6. In seinem Artikel von 1936 begründet er seine Überzeugung, daß deterministische Turingmaschinen ein adäquates Modell für das menschliche Rechnen sind. Da jede deterministische Turingmaschine ein Spezialfall einer nichtdeterministischen Turingmaschine (bei Turing: Wahl–Maschine) ist, ergab sich für ihn also auch nicht die Notwendigkeit, eine Theorie für solche Maschinen zu entwickeln.

Wir beweisen in diesem Abschnitt, daß nichtdeterministische Turingmaschinen tatsächlich nicht „mehr“ Sprachen akzeptieren können: Zu jeder nichtdeterministischen Turingmaschine M gibt es eine äquivalente deterministische Turingmaschine M', d.h. $L(M) = L(M')$.

Trotzdem sind wie bei den endlichen Automaten nichtdeterministische Maschinen wichtig: Sie sind meist einfacher und erlauben einfachere Beweise. Außerdem können wir an dem Äquivalenzbeweis erkennen, daß nichtdeterministische Turingmaschinen „schneller“ sein können, so wie nichtdeterministische endliche Automaten „kleiner“. In der Komplexitätstheorie, in der es um den Aufwand für das Akzeptieren von Sprachen geht, ist die Betrachtung nichtdeterministischer Turingmaschinen also wieder interessant; siehe Kapitel 12.

8.1 Definition: nichtdeterministische Turingmaschinen

Es gibt viele Möglichkeiten, nichtdeterministische Turingmaschinen zu definieren. Dabei geht es immer darum, daß es zu einer Konfiguration mehrere mögliche Nachfolgekonfigurationen gibt. Wir ersetzen wie bei den endlichen Automaten die Überführungsfunktion durch eine Relation.

Eine nichtdeterministische Turingmaschine (NTM) M wird also durch acht Angaben spezifiziert; wir schreiben auch $M = (Q, \Sigma, \Gamma, \Box, k, \Delta, q_A, E)$. Dabei ist

- Q eine endliche Menge von *Zuständen*,
- Σ ein Alphabet von *Eingabesymbolen*,
- Γ mit $\Sigma \subset \Gamma$ ein Alphabet mit zusätzlichen Symbolen (*Bandalphabet*),
- $\Box \in \Gamma \setminus \Sigma$ das *Blanksymbol*,
- $k \in \mathbb{N}^+$ die *Anzahl der Bänder*,
- $\Delta \subseteq Q \times \Gamma^k \times Q \times \Gamma^k \times \{L, N, R\}^k$ die $3k + 2$-stellige *Überführungsrelation*,

- $q_A \in Q$ der *Anfangszustand*,
- $E \subseteq Q$ die Menge der *akzeptierenden Zustände*.

8.2 Berechnung, akzeptierte Sprache, Berechnungsbaum

Wir können die Definitionen aus 7.4 mit wenigen Änderungen übernehmen. Konfigurationen sind dieselben, und wir wenden Δ genauso auf sie an wie δ in 7.4. Der einzige Unterschied: Da Δ eine Relation und keine Funktion ist, sind Folgekonfigurationen und Berechnungen nicht mehr eindeutig bestimmt. Konfigurationen, Berechnungen usw. definieren wir wie in 7.4 mit folgenden Änderungen (fett):

- **Jede** (terminierende oder unendliche) Berechnung von M, die mit der Anfangskonfiguration zur Eingabe w beginnt, nennen wir *Berechnung zur Eingabe w*.
- *M akzeptiert $w \in \Sigma^*$*, wenn es eine akzeptierende Berechnung zur Eingabe w **gibt**.

Wie bei nichtdeterministischen endlichen Automaten kann es zu einer Eingabe ganz unterschiedliche Berechnungen geben, akzeptierende und verwerfende, hier sogar auch terminierende und nichtterminierende. M akzeptiert, wenn es eine akzeptierende Berechnung gibt.

Noch mehr als bei nichtdeterministischen endlichen Automaten ist es für die Analyse von NTMn und für Beweise oft nützlich, alle Berechnungen von M zu einer Eingabe als Baum darzustellen.

- Sei K_M die Menge aller Konfigurationen einer NTM M. Für eine Konfiguration $k_0 \in K_M$ ist der *Berechnungsbaum von M zu k_0* folgendermaßen definiert: Die Knoten sind mit Konfigurationen von M markiert. An der Wurzel steht k_0. Ein Knoten mit der Marke $k \in K_M$ hat so viele Nachfolgerknoten wie k Folgekonfigurationen hat; mit denen sind sie auch markiert.
- Für eine Eingabe w ist der *Berechnungsbaum von M zu w* der Berechnungsbaum von M zur Anfangskonfiguration für w.

Eine Berechnung von M, die mit k_0 beginnt, ist ein Pfad im Berechnungsbaum zu k_0. Man spricht daher auch von *Berechnungspfaden* statt Berechnungen und überträgt die Definitionen auf Pfade.

Ein Knoten in einem Berechnungsbaum von M hat nur endlich viele direkte Nachfolger, nämlich höchstens $|Q|^k \cdot |\Gamma|^k \cdot 3^k$ viele, wenn M k Bänder, Zustände Q und das Bandalphabet Γ hat (warum?). Die Zahl der direkten Nachfolger eines Knotens heißt sein Verzweigungsgrad, ihre Maximalzahl der *Verzweigungsgrad* des Berechnungsbaums. Wenn Δ zum Beispiel höchstens zwei Folgekonfigurationen zuläßt (binäre Wahl), dann sind die Berechnungsbäume binär (Verzweigungsgrad 2). Die Berechnungsbäume von DTMn sind linear (Verzweigungsgrad 1).

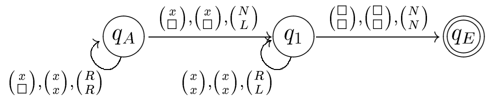
8.3 Beispiele für nichtdeterministische Turingmaschinen

Die Ein-Band-Maschine für die Sprache $L_1 = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ in Abschnitt 7.4 ist recht groß und kompliziert und auch zeitaufwendig. Aber mit einem Arbeitsband geht es ganz einfach: Die a's auf das Arbeitsband kopieren und dann mit den b's und c's vergleichen; das geht in einem Durchgang, also mit n Schritten. (Wieviel Schritte braucht die Maschine in 7.4 ungefähr? Solche Fragen betrachten wir in Kapitel 12.) Nichtdeterministisch ist das nicht einfacher.

Auch bei $L_2 = \{v\$v \mid v \in \Sigma^*\}$ oder $L_3 = \{v\$v^{-1} \mid v \in \Sigma^*\}$ (v^{-1} ist das Spiegelwort zu v, Abschnitt 1.4) hilft Nichtdeterminismus nichts. Wie sieht eine einfache DTM mit einem Arbeitsband dafür aus? Anders wird es, wenn wir das Trennzeichen \$ weglassen.

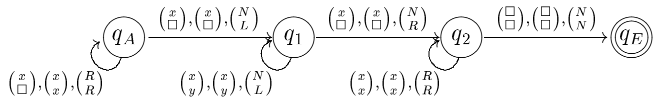
Eine Turingmaschine für $L_4 = \{vv \mid v \in \Sigma^+\}$ oder $L_5 = \{vv^{-1} \mid v \in \Sigma^+\}$ muß die Mitte einer Eingabe $w \in \Sigma^*$ kennen, um die vordere und die hintere Hälfte vergleichen zu können. Tatsächlich geht das deterministisch einfach; die Idee wird nicht verraten. Aber nichtdeterministisch ist es ganz leicht: Eine NTM für L_4 oder L_5 kopiert den Anfang der Eingabe auf das Arbeitsband, bricht das irgendwann nichtdeterministisch ab und vergleicht den Rest der Eingabe mit dem Kopierten (in der richtigen Richtung), z.B. für L_5:

M_5:



und für L_4:

M_4:



Hier stehen x und y für beliebigen Buchstaben aus Σ, innerhalb eines Übergangs überall für dieselben. Jeder solche Pfeil stellt also $|\Sigma|$ oder $|\Sigma|^2$ viele Übergänge dar.

Wir führen nun exemplarisch einen Korrektheitsbeweis für die Maschine M_4.

Behauptung: *Es gilt $L_4 = L(M_4)$.*

Wir zeigen, daß M_4 tatsächlich genau die Wörter aus L_4 zu akzeptieren. Es darf kein Wort aus L_4 geben, das M_4 nicht akzeptiert; das wird überprüft, wenn wir $L_4 \subseteq L(M_4)$ zeigen. Umgekehrt darf M_4 keine Wörter akzeptieren, die nicht in L_4 enthalten sind, was wir überprüfen, wenn wir $L(M_4) \subseteq L_4$ zeigen.

Erste Beweisrichtung $L_4 \subseteq L(M_4)$. Um $L_4 \subseteq L(M_4)$ zu zeigen, muß zu jedem Wort w aus L_4 eine akzeptierende Berechnung in M_4 angegeben werden, d.h. eine Berechnung, die in q_A mit w auf dem Eingabeband startet und in q_E, also dem akzeptierenden Endzustand von M_4, endet.

Sei $w \in L_4$. Nach Definition von L_4 existiert dann ein $v \in \Sigma^+$, so daß $w = vv$. Die einzelnen Zeichen von v wollen wir v_1 bis v_n nennen, also $v = v_1 \dots v_n$ mit $n = |v|$ und $v_i \in \Sigma$ für $1 \leq i \leq n$. Die Maschine M_4 akzeptiert dann die Eingabe w auf folgendem Berechnungspfad:

$$\begin{aligned}
(q_A, \rhd w, \rhd) &= (q_A, \rhd v_1 \dots v_n v_1 \dots v_n, \rhd) \\
&\vdash^n (q_A, v_1 \dots v_n \rhd v_1 \dots v_n, v_1 \dots v_n \rhd) \\
&\vdash (q_1, v_1 \dots v_n \rhd v_1 \dots v_n, v_1 \dots v_{n-1} \rhd v_n) \\
&\vdash^n (q_1, v_1 \dots v_n \rhd v_1 \dots v_n, \rhd \Box v_1 \dots v_n) \\
&\vdash (q_2, v_1 \dots v_n \rhd v_1 \dots v_n, \rhd v_1 \dots v_n) \\
&\vdash^n (q_2, v_1 \dots v_n v_1 \dots v_n \rhd, v_1 \dots v_n \rhd) \\
&\vdash (q_E, v_1 \dots v_n v_1 \dots v_n \rhd, v_1 \dots v_n \rhd) \\
&\ \text{Also gilt } w \in L(M_4)
\end{aligned}$$

Damit ist $L_4 \subseteq L(M_4)$ gezeigt.

Zweite Beweisrichtung $L(M_4) \subseteq L_4$. Um diese Richtung zu zeigen, muß für jedes Wort w aus $L(M_4)$, also jedes Wort, daß von M_4 akzeptiert wird, gezeigt werden, daß es auch in L_4 enthalten ist. Dazu muß aus der akzeptierenden Berechnung für w auf die Form von w geschlossen werden.

Sei $w \in L(M_4)$, d.h. es existiert eine Berechnung $(q_A, \rhd w, \rhd) \vdash^* (q_E, \alpha, \beta)$, wobei $\alpha, \beta \in \Gamma^*$. Um diese akzeptierende Berechnung analysieren zu können, betrachten wir zunächst Teilberechnungen von M_4. Eigentlich interessieren uns nur die Teilberechnungen, die in der Berechnung tatsächlich vorkommen, aber es ist – so wie hier – oft einfacher, erstmal etwas mehr zu beweisen. Wir betrachten deshalb die folgenden drei allgemeinen Fälle:

1. n-maliger Durchlauf der Schleife in q_A:

   Für $u, v, w \in \Sigma^*$ mit $|w| = n$ und $v' \in \Gamma^*$ gilt:

   $$(q_A, v \rhd wu, v' \rhd) \quad \vdash^n \quad (q_A, vw \rhd u, v'w \rhd)$$

   Genauer beweist man das durch Induktion über die Länge von w: Falls $w = \lambda$ so entspricht die Zielkonfiguration gerade der Ausgangskonfiguration, für den Induktionsschritt geht man davon aus, daß dieser Zusammenhang bereits für w mit $|w| = n$ gezeigt ist. Dann wird w um ein Zeichen verlängert, die Berechnung einen Schritt weitergeführt und gezeigt, daß dann die Behauptung immer noch gilt.

2. n-maliger Durchlauf der Schleife in q_1:

   Für $u, v \in \Sigma^*$, $w, w', w'' \in \Gamma^*$ mit $|w| = n$ gilt:

   $$(q_1, u \rhd v, w'w \rhd w'') \quad \vdash^n \quad (q_1, u \rhd v, w' \rhd ww'')$$

   Auch diese Behauptung müßte noch mit vollständiger Induktion über die Länge von w bewiesen werden, ähnlich wie oben beschrieben.

3. n-maliger Durchlauf der Schleife in q_2:

   Für $u, v, w \in \Sigma^*$ mit $|w| = n$, $w', w'' \in \Gamma^*$ gilt:

   $$(q_2, u \rhd wv, w' \rhd ww'') \quad \vdash^n \quad (q_2, uw \rhd v, w'w \rhd w'')$$

   Auch hier gilt wieder das oben Gesagte: eigentlich folgt hier ein Beweis der Behauptung durch vollständige Induktion, den ich hier aber nicht ausführe. Das bedeutet natürlich, daß ihr bei entsprechenden Beweisen in euren Projektaufgaben auch auf die Beweise solcher Details – was passiert beim n-maligen Durchlauf einer Schleife – per Induktion verzichten könnt.

Nun haben wir die wesentlichen Bestandteile einer akzeptierenden Berechnung analysiert. Nun fehlen noch die Details:

Sei also $w \in L(M_4)$ und $w = w_1 \dots w_n$. (Falls $n = 0$, so ist $w_1 \dots w_n = \lambda$.) Dann hat jede akzeptierende Berechnung die Form:

$$\begin{aligned}
(q_A, \rhd w_1 \dots w_n, \rhd) &\vdash^k (q_A, w_1 \dots w_k \rhd w_{k+1} \dots w_n, w_1 \dots w_k \rhd) && (1)\\
&\vdash (q_1, w_1 \dots w_k \rhd w_{k+1} \dots w_n, w_1 \dots w_{k-1} \rhd w_k) && (2)\\
&\vdash^k (q_1, w_1 \dots w_k \rhd w_{k+1} \dots w_n, \rhd \Box w_1 \dots w_k) && (3)\\
&\vdash (q_2, w_1 \dots w_k \rhd w_{k+1} \dots w_n, \rhd w_1 \dots w_k) && (4)\\
&\vdash^k (q_2, w_1 \dots w_{2k} \rhd w_{2k+1} \dots w_n, w_1 \dots w_k \rhd) && (5)\\
&\vdash (q_E, w_1 \dots w_{2k} \rhd w_{2k+1} \dots w_n, w_1 \dots w_k \rhd) && (6)
\end{aligned}$$

Dabei folgt Schritt 1 aus der Behauptung 1 für alle $k \in \mathbb{N}$ mit $0 \leq k \leq n$. Schritt 2 ist allerdings nur möglich, falls $w_{k+1} \dots w_n \neq \lambda$, d.h. falls $w_{k+1} \in \Sigma$. Daraus folgt also, daß die Berechnung bis zum Schritt 2 nur noch für $0 \leq k < n$ möglich ist, womit insbesondere $n > 0$ folgt. Schritt 3 gilt wiederum für alle k mit $0 \leq k < n$ wegen Behauptung 2. Diese Schleife muß k mal durchlaufen werden, da sonst Schritt 4 nicht möglich wäre. Schritt 5 folgt nun aus Behauptung 3. Dazu muß jedoch $w_{k+1} \dots w_{2k} = w_1 \dots w_k$ gelten. Schritt 6 ist nur möglich, falls $w_{2k+1} \dots w_n = \lambda$, d.h. falls $2k = n$ (und $k > 0$, da $n > 0$). Damit folgt nun für w: $w = w_1 \dots w_n = w_1 \dots w_k w_{k+1} \dots w_{2k}$ und $w_1 \dots w_k = w_{k+1} \dots w_{2k}$, d.h. $w = vv$ mit $v = w_1 \dots w_k$ für $k > 0$. Damit gilt $v \in \Sigma^+$, woraus $w \in L_4$ folgt. Damit ist also auch $L(M_4) \subseteq L_4$ gezeigt. $\Box$

Die Idee der Konstruktion von M_4 und M_5 läßt sich leicht verallgemeinern. Haben wir zum Beispiel zwei akzeptierende Turingmaschinen M und N für die Sprachen A und B, so gewinnen wir eine akzeptierende Maschine für die Verkettung $A \circ B$, indem wir M und N nichtdeterministisch den vorderen bzw. den hinteren Teil der Eingabe prüfen lassen, sie also so „nichtdeterministisch hintereinanderschalten“. (Deterministisch ist das nur zu machen, indem wir systematisch alle Zerlegungen $w = uv$ erzeugen und u durch M und v durch N prüfen lassen; das ist umständlich und zeitaufwendig.) Damit können wir Satz 7.5.1 verbessern, für die Vereinigung wird der Beweis sogar leichter.

Satz 8.3: *Die nichtdeterministisch akzeptierbaren Sprachen sind abgeschlossen unter Durchschnitt und Vereinigung und den Kleeneschen Operationen Verkettung, Potenz und Stern.*

Der Satz gilt auch für die deterministisch akzeptierbaren Sprachen, ist aber schwieriger zu beweisen (s.o.); wir verschieben das auf den nächsten Abschnitt.

Zum Schluß betrachten wir noch ein schwierigeres Beispiel: die Primzahlen. Eine natürliche Zahl $n > 1$ heißt *prim*, wenn sie keine echten Teiler hat, d.h. Teiler m mit $1 < m < n$. (m teilt n, wenn es ein l gibt mit $l \cdot m = n$.) Sonst heißt n *nichtprim* oder *zerlegbar*. Primalität kann man (mit Turingmaschinen) prüfen, indem man systematisch alle Zahlen m mit $1 < m < n$ (auf einem Arbeitsband) erzeugt und testet, ob eine davon n teilt. (Bei dezimaler oder binärer Zahldarstellung muß man dazu dividieren oder wenigstens multiplizieren können. Bei unären Zahlen – n wird durch 1^n dargestellt – genügt es, m immer wieder entlang n „abzutragen“ und am Schluß zu sehen, ob es „aufgeht“.) Wenn man so einen Teiler von n findet, ist n zerlegbar, sonst prim.

Wie Nichtdeterminismus helfen könnte, um Primzahlen zu akzeptieren, ist nicht so leicht zu sehen. (Tatsächlich geht es mit einer sehr schwierigen mathematischen Konstruktion.) Aber die zerlegbaren kriegt man ganz leicht: Nichtdeterministisch eine Zahl m mit $1 < m < n$ auf das Arbeitsband schreiben und akzeptieren, wenn sie n teilt; fertig.

Aufgabe: *Konstruiert eine NTM (eine DTM), die genau die zerlegbaren Zahlen akzeptiert. Macht das zuerst für unäre, dann für binäre Zahlen.*

Jetzt haben wir ein Gespür dafür, wann Nichtdeterminismus nützlich ist: Wenn eine Sprache gegeben ist als

$$L = \{w \in \Sigma^* \mid \exists x\colon \text{ Bedingung}(x, w)\},$$

dann ist es i.a. leichter, eine NTM M zu konstruieren, die L akzeptiert: Bei Eingabe w „rät“ M (d.h. schreibt nichtdeterministisch) ein Wort x und testet die Bedingung für x und w. Eine DTM für L muß i.a. alle x systematisch durchsuchen. Unsere Beispiele waren

$$\begin{aligned} L_4 &= \{w \in \Sigma^* \mid \exists v\colon w = vv\},\\ L_5 &= \{w \in \Sigma^* \mid \exists v\colon w = vv^{-1}\},\\ \textsc{NichtPrim} &= \{w \in \mathbb{N} \mid \exists v\colon 1 < v < w \text{ und } v \text{ teilt } w\ \}. \end{aligned}$$

Findet weitere; statt x dürfen auch mehrere Variable stehen, die NTM rät dann mehrere Wörter. Aus der Sicht des Rechenaufwandes befassen wir uns mit dem Problem in Kapitel 12.

8.4 Deterministische und nichtdeterministische Turingmaschinen akzeptieren dieselben Sprachen

DTMn können wir als spezielle NTMn auffassen; wir müssen nur die Überführungsrelation aus der -funktion definieren, z.B. bei einer Ein-Band-Maschine:

$$(q, a, q', b, B) \in \Delta \quad \text{gdw} \quad \delta(q, a) = (q', b, B).$$

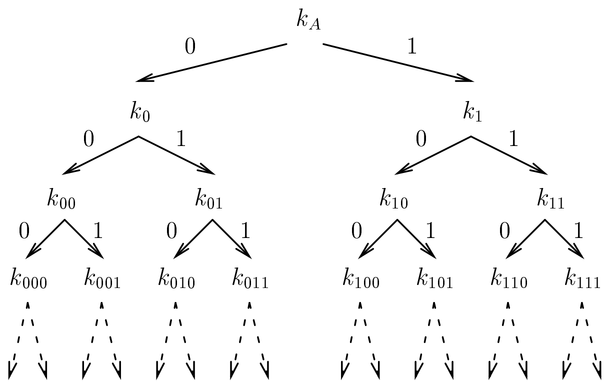
Wir müssen also nur die eine Richtung beweisen:

Satz: 8.4.1 *Zu jeder nichtdeterministischen Turingmaschine M_n kann man eine äquivalente deterministische Turingmaschine M_d konstruieren, d.h. $L(M_n) = L(M_d)$.*

Wir werden nur die Beweisidee angeben, aber die so ausführlich beschreiben, daß Ihr auf dieser Grundlage den Beweis selbst führen könnt.

Beweisidee: Sei M_n eine NTM. Wir konstruieren eine deterministische Turingmaschine M_d, die für eine Eingabe w den Berechnungsbaum von M_n zu w mittels *Breitensuche* traversiert. Sobald sie dabei eine akzeptierende Endkonfiguration findet, akzeptiert sie; sonst terminiert sie nicht. Also akzeptiert M_d bei Eingabe w genau dann, wenn es einen akzeptierenden Berechnungspfad im Berechnungsbaum von M_n bei Eingabe w gibt, genau dann also, wenn M_n die Eingabe w akzeptiert. Deshalb ist $L(M_n) = L(M_d)$.

Jetzt müssen wir also „nur noch“ die Turingmaschine M_d „programmieren“. Zur Vereinfachung nehmen wir zunächst an, daß M_n in jedem Schritt höchstens zweifach verzweigt, daß also die Berechnungsbäume binär sind. (Der allgemeine Fall ist dann leicht.) Wir vereinfachen weiter, indem wir den Baum „voll“ machen (jeder Knoten hat zwei oder keinen Nachfolgerknoten): Hat ein Knoten nur einen Nachfolgerknoten, so verdoppeln wir diesen. Dann sieht der Berechnungsbaum von M_n zur Eingabe w so aus:



Damit beschreiben Binärwörter $z \in \{0,1\}^*$ die endlichen Berechnungspfade von M_n. Also kann M_d jetzt mit Hilfe eines Binärwortes z einen Berechnungspfad von M_n nachvollziehen (simulieren): Sie „verhält“ sich so wie M_n, folgt aber dem Berechnungspfad z, d.h. wählt (deterministisch!) jeweils die Konfiguration, die durch z vorgeschrieben wird.

Etwas genauer: Es sei $M_n = (Q, \Sigma, \Gamma, \square, k, \Delta, q_A, E)$. Wir konstruieren zuerst eine DTM

$$M' = (Q, \Sigma, \Gamma \cup \{1, 0\}, \square, k+1, \delta', q_A, E).$$

Auf dem zusätzlichen Arbeitsband von M' steht bei Beginn der Berechnung ein Binärwort z. M' simuliert den Berechnungspfad von M_n entlang z. (Wir werden M' später als „Unterprogramm“ benutzen: das Binärwort wird dann vor dem Start von M' von der aufrufenden Turingmaschine geschrieben.) Die Überführungsfunktion δ' von M' gewinnen wir aus der Überführungsrelation Δ von M_n: Für $q \in Q$, $a_1, \ldots, a_k \in \Gamma$ seien mit $r \in Q$, $b_1, \ldots, b_k \in \Gamma$, $B_1, \ldots, B_k \in \{L, N, R\}$ und $s \in Q$, $c_1, \ldots, c_k \in \Gamma$, $C_1, \ldots, C_k \in \{L, N, R\}$ die beiden möglichen Rechenschritte mittels Δ gegeben. Dann definieren wir:

$$\begin{aligned}
\delta'(q, a_1, \ldots, a_k, 0) &= (r, b_1, \ldots, b_k, 0, B_1, \ldots, B_k, R),\\
\delta'(q, a_1, \ldots, a_k, 1) &= (s, c_1, \ldots, c_k, 1, C_1, \ldots, C_k, R).
\end{aligned}$$

Falls Δ keine Fortsetzungen zuläßt, sei $\delta'(q, a_1, \ldots, a_k, i)$ nicht definiert.

Wenn M' bei der Simulation eine akzeptierende Endkonfiguration von M_n erreicht, dann akzeptiert sie (weil sie dieselben akzeptierenden Zustände wie M_n hat). Wenn M' eine nichtakzeptierende Endkonfiguration von M_n erreicht, bricht sie die Simulation zu z ab. Wenn schließlich das Binärwort „zu Ende“ ist, ohne daß eine Endkonfiguration von M_n erreicht wurde, dann wurde die entsprechende Berechnung von M_n nur teilweise simuliert. (Hier werden wir M' später noch abändern müssen, um sie als Interpreterprogramm verwenden zu können.)

Wie programmieren wir die Breitensuche? Für den Berechnungsbaum meinen wir damit eine Folge von Simulationen:

$$\begin{aligned}
&k_A,\\
&k_A \vdash k_0,\\
&k_A \vdash k_1,\\
&k_A \vdash k_0 \vdash k_{00},\\
&k_A \vdash k_0 \vdash k_{01},\\
&k_A \vdash k_1 \vdash k_{10},\\
&k_A \vdash k_1 \vdash k_{11},\\
&k_A \vdash k_0 \vdash k_{00} \vdash k_{000} \qquad \ldots
\end{aligned}$$

Diese erreichen wir dadurch, daß M_d in einer Schleife nacheinander *alle* Binärwörter erzeugt, z.B. in Standardreihenfolge. In jedem Schleifendurchlauf wird die Maschine M' mit dem jeweils aktuellen Binärwort (und der Eingabe) gestartet, also „der Reihe nach“ mit $\lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$. Wenn M' einen Binärstring „abgearbeitet“ hat, ohne in

einen akzeptierenden Endzustand von M_n zu gelangen (siehe die beiden letzten Fälle oben), dann erzeugt M_d das nächste Binärwort. (M_d muß natürlich vorher alles schön initialisieren, Bänder löschen usw.) Wenn M_n die Eingabe nicht akzeptiert, gibt es kein Binärwort, das einen akzeptierenden Berechnungspfad beschreibt, M_d terminiert also nicht.

Damit ist die Konstruktion von M_d hoffentlich ausführlich genug beschrieben; fehlende Details, z.B. das Erzeugen der Binärwörter, könnt Ihr jetzt alleine durchführen. Eine NTM M_n mit Verzweigungsgrad $k > 2$ simuliert man genauso mit Hilfe von Wörtern $z \in \{0, 1, \ldots, k-1\}^*$. Damit ist der Beweis vollständig. Also gilt:

Satz 8.4.2: *Deterministische und nichtdeterministische Turingmaschinen akzeptieren dieselben Sprachen.*

8.5 Aufgaben

Aufgabe 8.1: Gebt eine nichtdeterministische Turingmaschine für die Sprache aus Aufgabe 7.5 an.

Aufgabe 8.2: Gebt eine nichtdeterministische Turingmaschine an, die die Sprache PAL akzeptiert.

$$\mathrm{PAL} = \{wxw^{-1} \mid w \in \Sigma^* \text{ und } x \in \Sigma\} \cup \{ww^{-1} \mid w \in \Sigma^*\}$$

Was ändert sich, wenn Ihr $*$ durch $+$ ersetzt?

Aufgabe 8.3: Wie ändert sich die nichtdeterministische Turingmaschine für die Sprache L_4 aus Abschnitt 8.3, wenn Ihr $+$ durch $*$ ersetzt?

9 Allgemeine Grammatiken und akzeptierbare Sprachen

In diesem Abschnitt betrachten wir (formale) Grammatiken ohne Einschränkungen. Sie erzeugen genau die Sprachen, die von Turingmaschinen akzeptiert werden. Da Ihr nun schon einige Erfahrung mit Grammatiken und Automaten habt, werden wir uns in diesem Kapitel kurz fassen.

9.1 Definition: allgemeine Chomsky-Grammatik

Eine (allgemeine Chomsky-)*Grammatik* G wird durch vier Angaben spezifiziert; wir schreiben auch $G = (N, T, S, P)$. Dabei ist

- N ein Alphabet mit *nonterminalen Symbolen*,
- T ein Alphabet mit *terminalen Symbolen*,
- $S \in N$ das *Startsymbol*,
- P eine Menge von *Produktionen* der Form $u \to v$ mit $u, v \in (N \cup T)^*$, wobei in u mindestens ein Nonterminal vorkommt.

N und T sind disjunkt.

9.2 Erzeugte Sprache

Bei den allgemeinen Chomsky-Grammatiken ist nun (fast) alles erlaubt. (Auch die Einschränkung, daß in u mindestens ein Nonterminal vorkommt, könnten wir weglassen. Wir tun es nicht, weil wir mit Chomsky die Nonterminale als Steuerzeichen in Ableitungen auffassen; Ableitungen sollten daher abbrechen, wenn keine Nonterminale mehr übrig sind.) Eine Regel $u \to v$ auf ein Wort w anwenden heißt, in w das Teilwort u durch v zu ersetzen: $w = w_1uw_2 \Rightarrow w_1vw_2$. Wie bisher nennen wir Folgen solcher Ersetzungsschritte *Ableitungen* und schreiben

$$w \Rightarrow_G^* w',$$

wenn es eine Ableitung von w nach w' mit G gibt. Die von einer Grammatik G *erzeugte Sprache* besteht aus den Wörtern über T, die aus S ableitbar sind:

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

Eine Sprache L heißt *erzeugbar*, wenn es eine Grammatik G mit $L = L(G)$ gibt.

Die verschiedenen Grammatiktypen kann man hinsichtlich ihrer Einschränkungen linear anordnen: regulär, kontextfrei, kontextsensitiv, allgemein; in der Reihenfolge haben wir sie auch eingeführt. Da die Grammatiken von links nach rechts mächtiger werden, werden sie und die zugehörigen Sprachen von links nach rechts immer mehr; das haben wir jeweils gezeigt, für die letzte Stufe steht es noch aus. Eine solche Kette von (echten) Inklusionen nennt man eine (echte) *Hierarchie* und stellt sie als von unten nach oben geordnet dar: Das Allgemeinste, Stärkste, Mächtigste ist in unserer Kultur das Beste und ist immer oben; vgl. dazu das schöne Buch *Metaphors We Live By* von George Lakoff und Mark Johnson. Weil Noam Chomsky die formalen Grammatiken eingeführt hat, heißt die Hierarchie in der Theorie der formalen Sprachen „Chomsky-Hierarchie“. Chomsky hat die Grammatik- und Sprachtypen von oben nach unten durchnumeriert. Allgemeine Grammatiken und erzeugbare Sprachen nennt man daher oft vom *Typ 0*.

Typ 0: allgemeine Chomsky-Grammatiken und erzeugbare Sprachen.
Typ 1: kontextsensitive Grammatiken und Sprachen
Typ 2: kontextfreie Grammatiken und Sprachen
Typ 3: reguläre Grammatiken und Sprachen

Die Grammatik- und Sprachklassen sind jeweils echte Teilmengen voneinander:

$$\text{Typ-3} \subset \text{Typ-2} \subset \text{Typ-1} \subset \text{Typ-0}$$

9.3 Äquivalenz von Grammatiken und Turingmaschinen

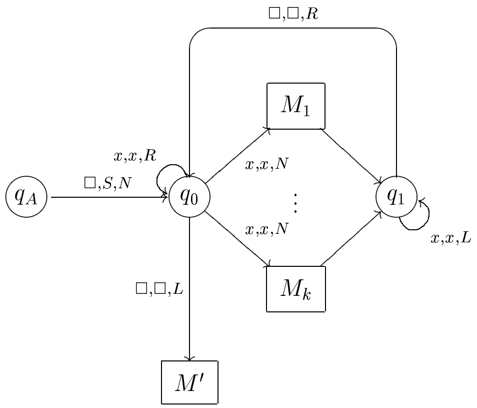
In diesem Abschnitt beschränken wir uns wieder auf die Darstellung der Beweisideen.

Satz 9.3.1: *Zu jeder allgemeinen Grammatik G kann man eine äquivalente Turingmaschine M konstruieren, d.h. $L(G) = L(M)$.*

Beweis: Die Beweisidee ist einfach: Zu einer Grammatik G konstruieren wir eine NTM M mit einem Arbeitsband, die die Regeln von G „kennt“, d.h. einprogrammiert hat. Auf dieses Arbeitsband schreibt M das Startsymbol und leitet daraus mit den Regeln von G (irgend)ein Wort ab. Danach vergleicht sie die Eingabe mit dem abgeleiteten Wort. Wenn beide Wörter gleich sind, akzeptiert sie; sonst terminiert sie nichtakzeptierend.

M produziert Ableitungen schrittweise. In jedem solchen „Schritt“ wendet sie nichtdeterministisch eine Regel von G auf das schon abgeleitete Wort w' an. Dazu durchläuft sie w' von links nach rechts, wählt nichtdeterministisch irgendwann irgendeine Regel $u \to v$ und ersetzt an der Stelle in w' das Teilwort u, wenn es dort steht, durch v. (Dabei muß sie für $|v| \neq |u|$ den Rest von w' nach rechts oder links verschieben. Dazu benutzt sie ein weiteres Arbeitsband.) Danach läuft sie an den Anfang von w' zurück und beginnt von vorne. Wenn an der Stelle nicht u steht, terminiert sie nichtakzeptierend. Wenn sie beim Suchen das Wort w' nach rechts verläßt, vergleicht sie w' mit der Eingabe.

Wenn G nun k Regeln hat, besteht M aus k Teilmaschinen $M_1, \dots, M_k$ zum Anwenden der Regeln, der Maschine M' zum Vergleichen, zwei „Lauf“-Zuständen und dem Anfangszustand, in dem S auf das Arbeitsband geschrieben wird. Schematisch (die Pfeile beziehen sich nur auf das Ableitungsband, x steht für Symbole aus $N \cup T$):



Die Details, insbesondere die Regelmaschinen M_i, könnt Ihr selber ausführen. Ihr solltet M unbedingt erst für ein konkretes Beispiel konstruieren, dabei aber wirklich die Grammatik benutzen, auch wenn es leichter ist, für die erzeugte Sprache direkt eine akzeptierende NTM zu konstruieren.

Nachdem nun also die Maschine M konstruiert ist, müssen wir noch zeigen, daß sie tatsächlich korrekt arbeitet. Dazu sind zwei Richtungen zu zeigen: Falls w ableitbar ist, so muß es von M akzeptiert werden; und falls w akzeptiert wird, so muß es ableitbar sein.

Für die erste Richtung sei w ableitbar. Dann gibt es eine Ableitungskette, an deren Ende w steht. Um zu zeigen, daß M das Wort w akzeptiert, genügt es nach Definition, eine akzeptierende Berechnung zu finden. Diese akzeptierende Berechnung ist die folgende: Die Maschine vollführt auf ihrem Arbeitsband einfach die Ableitungsschritte nach, die in der Ableitung von w auftauchen. Auf dem Arbeitsband stehe beispielsweise der Kopf am Anfang des Wortes xuy und der aktuelle Ableitungsschritt laute $xuy \Rightarrow_G xvy$ mittels Regel i. In der akzeptierenden Berechnung wird nun so lange die Schleife im Zustand q_0 durchlaufen, bis der Kopf den Anfang von u erreicht hat. Dann geht die akzeptierende Berechnung weiter, indem nichtdeterministisch in die Maschine M_i verzweigt wird. Diese ersetzt dann u durch v. Sind alle Ableitungsschritte nachvollzogen, so steht auf dem Arbeitsband das Wort w und der Vergleicher M' läuft bis zum Ende durch. Das Wort wird also akzeptiert.

Für die zweite Richtung werde nun w akzeptiert. Es gibt also eine Konfigurationsfolge, die mit der Anfangskonfiguration $(q_a, \triangleright w, \dots)$ startet und die in einer akzeptierenden Konfiguration endet. Zunächst zerlegt man die Berechnung in Teilstücke. Ein solches Teilstück reicht immer von dem Erreichen des Zustands q_0 bis zum Erreichen des Zustands q_1. Aufgrund der Konstruktion der Maschine ist nun klar, daß auf einem solchen Teilstück genau ein Wort u auf dem Arbeitsband durch ein Wort v ersetzt wird mit $u \Rightarrow_G v$. Deshalb steht am Ende des letzten Teilstücks ein Wort w' auf dem Arbeitsband, das sich in endlich vielen Schritten aus S ableiten läßt, also $S \Rightarrow_G^* w'$. Da w akzeptiert wurde, muß auch der

Vergleicher am Ende durchgelaufen sein. Also gilt $w' = w$ und somit $S \Rightarrow_G^* w$, was zu beweisen war. □

Alternativ könnt Ihr die Regeln von G umdrehen, d.h. $u \to v$ durch $v \to u$ ersetzen, und die Ableitungen rückwärts von der Eingabe aus statt vorwärts von S aus produzieren lassen. Eine solche Maschine M kopiert die Eingabe auf das Arbeitsband, wendet darauf die umgedrehten Regeln an und akzeptiert, wenn sie S erhalten hat. Dazu müßt Ihr Euch vergewissern, daß die Grammatik $\overline{G}$ mit den umgedrehten Regeln gerade die umgedrehten Ableitungen von G zuläßt (Induktion nach der Länge von Ableitungen), insbesondere gilt

$$u \Rightarrow_G^* v \quad \text{gdw} \quad v \Rightarrow_{\overline{G}}^* u.$$

Das Umdrehen von Regeln ist zunächst befremdlich, aber das Suchverfahren ist viel natürlicher: Statt ohne Rücksicht auf das gegebene Wort w wild drauf los abzuleiten und dann zu prüfen, ob man w erwischt hat, versucht man von w ausgehend rückwärts eine Ableitung für w aufzubauen. Beide Verfahren sind nichtdeterministisch, aber aus dem zweiten läßt sich eher eine Heuristik machen. Die Idee läßt sich auch bei den eingeschränkten Grammatiken anwenden; geht daraufhin Kapitel 2–5 und 6 noch einmal durch.

Satz 9.3.2: *Zu jeder nichtdeterministischen Turingmaschine M kann man eine äquivalente allgemeine Grammatik G konstruieren, d.h. $L(M) = L(G)$.*

Beweis: Sei M eine Ein-Band-Maschine mit Zustandsmenge Q, Eingabe- und Bandalphabet Σ bzw. Γ und Überführungsrelation Δ. Die Beweisidee ist, die Übergänge, die Δ erlaubt, als Regeln aufzuschreiben. Mit dieser Grammatik können wir Berechnungen von M simulieren: Rechenschritte werden zu Ableitungsschritten. Dazu schreiben wir in den Konfigurationen von M statt $\triangleright$ den jeweiligen Zustand von M, also

$$uqv \quad \text{statt} \quad (q, u \triangleright v).$$

Sonst wären Zustand und gelesenes Symbol beliebig weit auseinander, Regeln können aber nur an *einer* Stelle im Wort angewendet werden. Die Regeln sehen dann so aus:

$$\begin{aligned} qa &\to q'b & \text{falls} \quad & \Delta(q, a, q', b, N), \\ qa &\to bq' & \text{falls} \quad & \Delta(q, a, q', b, R), \\ cqa &\to q'cb & \text{falls} \quad & \Delta(q, a, q', b, L). \end{aligned}$$

Die Fälle, in denen M an den Rand der Konfiguration kommt, behandeln wir am besten, indem wir die Konfiguration an den Rändern markieren, z.B. mit [und], und entsprechende Regeln hinzufügen, z.B.

$$[qa \to [q'\Box b \quad \text{falls} \quad \Delta(q, a, q', b, L).$$

Für diese Grammatik G_1 erhalten wir für Konfigurationen k, k':

$$k \vdash_M^* k' \quad \text{gdw} \quad [k] \Rightarrow_{G_1}^* [k'].$$

Jetzt drehen wir G_1 um (vgl. den Absatz oben) und ergänzen Regeln, die akzeptierende Endkonfigurationen erzeugen und in Anfangskonfigurationen $[q_A w]$ die Ränder $[q_A$ und $]$ löschen (wie geht das?). G besteht aus all diesen Regeln und den Alphabeten $T = \Sigma$ und $N = (\Gamma \setminus \Sigma) \cup Q \cup \{[,]\}$, wobei wir Q und Γ als disjunkt voraussetzen. Dann gilt:

$$\begin{array}{lll} w \in L(M) & \text{gdw} & q_A w \vdash_M^* k \text{ für eine akzeptierende Endkonfiguration } k \\ & \text{gdw} & [q_A w] \Rightarrow_{G_1}^* [k] \text{ für eine akzeptierende Endkonfiguration } k \\ & \text{gdw} & S \Rightarrow_G^* [k] \Rightarrow_G^* [q_A w] \Rightarrow_G^* w \text{ für eine akzeptierende Endkonf. } k \\ & \text{gdw} & S \Rightarrow_G^* w \\ & \text{gdw} & w \in L(G) \end{array}$$

□

Wem rückwärts laufende Turingmaschinen ein Greuel sind, der kann mit G_1 arbeiten, muß dann aber zu Beginn ein beliebiges Wort w erzeugen (am besten gleich doppelt), daraus eine „Anfangskonfiguration“ $[q_A w]w$ herstellen, am Ende der Simulation, falls M akzeptiert, die Endkonfiguration $[k]$ löschen, so daß w übrigbleibt.

Satz 9.3.3: *(Nichtdeterministische) Turingmaschinen akzeptieren genau die Sprachen, die von allgemeinen Grammatiken erzeugt werden.*

9.4 Aufgaben

Aufgabe 9.1: Konstruiert zu der (kontextfreien) Grammatik G mit den Regeln

$$S \to aSa \mid bSb \mid \lambda$$

nach dem Beweis von Satz 9.3.1 eine NTM M, die $L(G)$ akzeptiert. Gebt zu den Eingaben λ, aba, $abba$ je eine Berechnung von M an, eine davon akzeptierend, wenn möglich. Gebt eine einfachere NTM M' an, die $L(G)$ akzeptiert. Dann eine DTM M''. Macht M deterministisch (s. 8.4) und vergleicht.

Aufgabe 9.2: Vervollständigt den Beweis von Satz 9.3.1.

Aufgabe 9.3: Konstruiert zu der NTM M' aus Aufgabe 1 nach dem Beweis von Satz 9.3.2 eine Grammatik G', die $L(M')$ erzeugt. Gebt Ableitungen der Wörter λ und $abba$ mit G' an. Findet Ihr auch eine für aba? Jetzt dasselbe für die DTM M''. Ist die dabei konstruierte Grammatik G'' „deterministischer“ als G'? Sind die Grammatiken regulär, kontextfrei, kontextsensitiv?

Aufgabe 9.4: Vervollständigt den Beweis von Satz 9.3.2.

Aufgabe 9.5: Beweist Satz 9.3.3.

10 Die Churchsche These

In der Einleitung haben wir erwähnt, daß im Jahr 1936 eine ganze Reihe von Mathematikern und Logikern Berechnungsformalismen veröffentlichten, die sich alle als gleichwertig erwiesen; einer davon sind die Turingmaschinen. Seit 1936 sind viele weitere Berechnungsformalismen vorgestellt worden, alle äquivalent zu den schon behandelten. Dazu gehören die formalen Grammatiken, die Chomsky 1956 einführte; in Abschnitt 9.3 ist ein Beweis skizziert, daß Typ-0-Grammatiken und Turingmaschinen äquivalent sind. Daher ist man sich seit langem einig: Es gibt keinen umfassenderen (und keinen engeren) Sinn von ‘berechenbar’; alle Probleme, die überhaupt durch Rechnung lösbar sind, können (zum Beispiel) durch Turingmaschinen gelöst werden. Als These hat das schon 1936 der amerikanische Logiker Alonzo Church formuliert, für den Formalismus der „rekursiven Funktionen“, also für Funktionen statt Probleme.

Churchsche These (für Turingmaschinen): *Alle Funktionen, die überhaupt (in irgendeinem mathematischen Sinn) berechenbar sind, sind mit Turingmaschinen (oder einem der anderen äquivalenten Formalismen) berechenbar und umgekehrt.*

Eine weitere Begründung für die Churchsche These ist neben der Äquivalenz und der „Universalität“ so vielfältiger Berechnungsformalismen ihre „Robustheit“: man kann sie alle stark verändern, ohne daß sie „Berechnungskraft“ verlieren oder gewinnen. So kann man Turingmaschinen auf einem, auf zwei, auf einem halben (das heißt, nur nach einer Seite unbeschränkten) oder auf 17 Bändern rechnen lassen; ebenso auf 2-dimensionalen Rechenfeldern oder im 4-dimensionalen Raum; man kann sie statt Buchstaben ganze Wörter lesen und schreiben lassen; man kann sie deterministisch oder nichtdeterministisch rechnen lassen – die Probleme, die man damit lösen kann, ändern sich nicht. (Für aufwandsbeschränkte Rechnungen gilt das nicht unbedingt; die Frage in Kapitel 12 ist gerade, ob deterministische und nichtdeterministische Turingmaschinen in polynomialer Zeit das gleiche können.)

Beweisen kann man die Churchsche These nicht; denn darin ist ein präzise definierter Begriff (Turing-berechenbar) mit einem vagen (überhaupt berechenbar) verknüpft. Trotzdem ist die Churchsche These eine mathematische Aussage, nicht eine tief philosophische über menschliche Fähigkeiten und Beschränktheiten. ‘Berechenbar’ ist ein mathematischer Begriff, auch wenn darin auf beliebige Formalismen Bezug genommen wird und nicht auf einen bestimmten wie in ‘Turing-berechenbar’. Erstaunlicher und diskussionswürdiger als die Churchsche These sind die Äquivalenzbeweise, die dahinterstecken: jede Rechnung, die Mathematiker überhaupt durchführen können, läßt sich auf einer Turingmaschine simulieren. Damit wird Rechnen auf das Manipulieren (Lesen, Löschen, Schreiben) von Symbolen reduziert. Manche Vertreter des Gebietes „Künstliche Intelligenz“ schließen daraus, daß man Denken auf Symbolmanipulation reduzieren und auf Rechnern simulieren könne. Wer die Überlegungen aus der Einleitung zu den nichtmathematischen Grundlagen der Informatik nachvollzieht, sieht zu einer solchen Hoffnung oder Befürchtung keinen Anlaß. Wie kann man auf die Idee kommen, Maschinen könnten oder sollten denken? Denken ist kein formales Spiel, auch rationale Überlegungen sind immer mit Gefühlen von ‘gut’ oder ‘böse’, ‘häßlich’ und ‘schön’, also mit Werten, verflochten. Wer könnte sonst Angst vor der Mathematik haben oder sich an eleganten Beweisen begeistern? Denken geschieht nicht im „Gehirnkästchen“; gerade im Begrifflichen sind wir auf Sprache, auf andere Menschen, auf deren Entwicklung angewiesen. Denken ist kein individuelles Unterfangen, sondern steckt in jeder verantwortlichen menschlichen Handlung.

11 Entscheidbarkeit

In diesem Kapitel wollen wir uns mit der Frage beschäftigen, ob es „Aufgaben“ oder „Probleme“ gibt, die Computer nicht lösen können. Dabei geht es uns nicht um Grundfragen der KI, etwa inwieweit menschliche Eigenschaften, Verhaltensweisen und Fähigkeiten durch Computer nachgebildet oder ersetzt werden können; dazu steht ein bißchen im vorigen Kapitel. Wir werden uns hier auf „Rechenaufgaben“ beschränken.

Viele Leute glauben, daß es ausreiche, eine Aufgabe nur genau genug zu beschreiben; dann sei es immer möglich, ein Programm dafür zu schreiben. Es gebe allenfalls Probleme mit zu langer Rechenzeit oder zu geringer Speicherkapazität. Im folgenden werden wir zeigen, daß es Aufgaben gibt, die *prinzipiell* nicht lösbar sind. Dazu müssen wir präzisieren, was wir unter „Aufgaben lösen“ verstehen.

11.1 Akzeptieren versus Entscheiden

Bisher haben wir uns um Rechenaufgaben nicht viel gekümmert. Wir haben reguläre Sprachen als Wortmengen eingeführt, die durch gewisse einfache („reguläre“) Regelsysteme („Grammatiken“) definierbar („erzeugbar“) sind (Kap. 2). Für solche formal definierten Sprachen stellte sich das *Wortproblem* (2.5): Gibt es ein Verfahren, das für reguläre Grammatiken G und Wörter w feststellt, ob w durch G erzeugbar ist? Nach Abschnitt 5.3 können wir die Frage mit endlichen Automaten beantworten, das Wortproblem für reguläre Grammatiken ist also lösbar; nichtdeterministische endliche Automaten tauchten dabei als Hilfsmittel auf. Allerdings haben wir dann endliche Automaten unabhängig von Grammatiken benutzt, um Probleme zu lösen: Enthält das Wort w eine gerade Anzahl von Nullen? Ein Automat, der die Sprache $\{w \in \{0,1\}^* \mid |w|_0 \text{ gerade}\}$ akzeptiert, löst das Problem.

Ähnlich haben wir in Kapitel 6 kontextfreie Sprachen durch Grammatiken definiert. Die dazu passenden Maschinen heißen Kellerautomaten; bei uns kommen sie nur in Aufgaben vor. Wieder kann man Kellermaschinen als „Entscheider“ benutzen: Ist das Eingabewort ein korrekt geklammerter Ausdruck (6.1)? Ein Programm? Allerdings sind diese Maschinen nichtdeterministisch, und dieser „Schönheitsfehler“ ist nicht behebbar.

Im dritten Teil sind wir umgekehrt vorgegangen: Wir haben Turingmaschinen zum Lösen von Problemen (und Berechnen von Funktionen) eingeführt. Nichtdeterminismus war dabei kein Problem, weil eliminierbar (Kap. 8), endliche und Kellerautomaten wurden zum Spezialfall. Turingmaschinen akzeptieren genau die Sprachen, die durch allgemeine Grammatiken erzeugbar sind, sind also ganz „allgemeine Automaten“. Trotzdem haben wir in Kap. 9 kein Wort über das Wortproblem verloren. Das geschah nicht versehentlich: Das Wortproblem für allgemeine Grammatiken ist nicht lösbar! Wir können zwar zu jeder Grammatik G eine äquivalente TM M konstruieren (Satz 9.3.1). Wir können also auf die

Frage „Ist w durch G erzeugbar?“ die Maschine M mit Eingabe w starten und die Antwort „M akzeptiert w (nicht)“ benutzen. Wir kriegen aber nicht immer eine Antwort.

Das sah zunächst wie ein Problem des Nichtdeterminismus aus. Wenn wir eine akzeptierende Berechnung von M zur Eingabe w haben, sind wir fertig: $w \in L(M)$, also $w \in L(G)$. Eine nicht akzeptierende Berechnung läßt dagegen offen, ob es nicht andere, akzeptierende gibt. Aber wir können zu M ja eine äquivalente deterministische Maschine M' konstruieren (Abschnitt 8.4) und die benutzen. Dann ist ihre Antwort eindeutig: Entweder ist die Berechnung zu w akzeptierend oder nicht. Das Problem liegt also an der Termination: Eine terminierende Berechnung von M' liefert eine eindeutige Antwort; aber solange M' rechnet, wissen wir die Antwort nicht, wissen nicht einmal, ob M' jemals terminieren wird.

Dem Phänomen des anscheinend unvermeidbaren Nichtterminierens sind wir schon mehrfach begegnet. Zuerst bei der gerade erwähnten Konstruktion einer äquivalenten DTM M' zu einer NTM M: Die DTM durchsucht den Berechnungsbaum der NTM nach einer akzeptierenden (insbesondere terminierenden) Berechnung. Wenn es eine gibt, findet M' sie; wenn nicht, sucht sie ewig weiter. Wenn die maximale Länge akzeptierender Berechnungen von M aus der Eingabe bestimmbar ist, kann M' die Breitensuche abbrechen, wenn sie diese Länge erreicht hat. Deswegen lassen wir bei konkreten Problemen auch nichtdeterministische Maschinen zur Problemlösung zu. Z.B. haben wir in Abschnitt 8.3 eine NTM betrachtet, die die „Spiegelwörter“ vv^{-1} akzeptiert. Für jede Eingabe w der Länge n hat sie n verschiedene Berechnungen, alle von einer Länge $\leq 2n$; die könnte eine DTM alle durchsuchen und so entscheiden. Wenn wir eine solche Längenbeschränkung nicht haben, können wir nicht erreichen, daß M' terminiert.

Ähnlich bei der Konstruktion einer äquivalenten NTM M zu einer Grammatik G: Die NTM sucht zu einer Eingabe w nach Ableitungen von G für w. Wenn es eine gibt, findet M sie; wenn nicht, sucht sie ewig weiter. Wieder können wir die Suche abbrechen, wenn wir die Länge der Ableitung für w beschränken können. Deswegen ist das Wortproblem für kontextsensitive (erst recht für kontextfreie oder reguläre) Sprachen mit einem deterministischen Verfahren lösbar: Wörter werden beim Ableiten nicht kürzer; deswegen können wir Ableitungen abbrechen, wenn das abgeleitete Wort länger als w wird (siehe Abschnitt 6.5). Bei allgemeinen Grammatiken können wir das nicht.

Das Problem, auf das wir immer wieder stoßen, ist also das *Halteproblem* (für deterministische Turingmaschinen): „Terminiert die DTM M bei Eingabe w?“ Könnten wir das lösen, so auch das Halteproblem für NTMn (über 8.4), das Wortproblem für Turingmaschinen „Akzeptiert M die Eingabe w?“ und das Wortproblem für allgemeine Grammatiken (über 9.3).

In diesem Abschnitt betrachten wir erst einmal die entscheidbaren Probleme. Aufgrund der Churchschen These (Kap. 10) können wir dabei beliebige Formalismen zum Berechnen und Akzeptieren verwenden. Wir benutzen Turingmaschinen, weil sie sich in der Komplexitätstheorie eingebürgert haben. Turingmaschinen arbeiten auf Wörtern; deswegen stellen wir *Probleme* als Sprachen über einem geeigneten Alphabet dar.

Zum Entscheiden von Problemen brauchen wir Turingmaschinen, die für alle Wörter über ihrem Eingabealphabet terminieren. Außerdem sollten sie deterministisch sein; denn dann gibt es für jede Eingabe genau eine Berechnung, akzeptierend oder nichtakzeptierend, auf jeden Fall terminiert sie. Also liefern solche Maschinen für jede Eingabe eine klare Antwort: JA oder NEIN. Deswegen führen wir dafür folgende Sprechweisen ein.

Definition: Eine Turingmaschine M, die eine Sprache L akzeptiert, nennen wir einen *Akzeptor* für L. Die Sprache heißt dann *akzeptierbar*. Unter besonderen Umständen ist ein Akzeptor sogar ein *Entscheider* für L, falls nämlich M deterministisch ist und für jede Eingabe terminiert. Die Sprache heißt dann *entscheidbar*.

Wir sagen, daß ein Entscheider M für L *die Sprache L entscheidet* bzw. das durch L repräsentierte Problem *löst*. Weiter sagen wir, daß die nichtakzeptierenden Zustände und Berechnungen von M *verwerfend* sind und daß M Eingaben *verwirft*, die sie nicht akzeptiert.

Beispiele: Alle bisher betrachteten Sprachen wie die Spiegelwörter ($w = vv^{-1}$), Palindrome ($w = w^{-1}$), Primzahlen (8.3) sind entscheidbar. Tatsächlich sind alle kontextsensitiven (erst recht alle kontextfreien, alle regulären, alle endlichen) Sprachen entscheidbar, da das Wortproblem dafür lösbar ist. Zu einer gegebenen Turingmaschine M ist entscheidbar, ob ein Wort eine (Anfangs-, End-, akzeptierende) Konfiguration für M darstellt und ob eine gegebene endliche Berechnung von M terminierend (akzeptierend) ist. Achtung: In den nächsten beiden Abschnitten werden wir zeigen, daß die scheinbar nur wenig allgemeinere Frage, ob sich ein gegebenes endliches Anfangsstück einer Berechnung zu einer terminierenden Berechnung fortsetzt, i.a. nicht entscheidbar ist – auch nicht, wenn M deterministisch ist.

Bevor wir zum Halteproblem kommen, fassen wir in zwei Sätzen einige schöne Eigenschaften der entscheidbaren Sprachen zusammen. Den ersten Satz haben wir für die regulären Sprachen in 5.3 formuliert, für die akzeptierbaren in 8.3 (nicht fürs Komplement, dafür ist er falsch, siehe 11.3). Die dabei gelernten Techniken zur Konstruktion von Maschinen können wir übertragen. Aufpassen: Die gegebenen Maschinen sind hier Entscheider, das vereinfacht einiges; aber die konstruierten müssen es auch sein, das macht teilweise Mühe. Den zweiten Satz brauchen wir in Abschnitt 11.3.

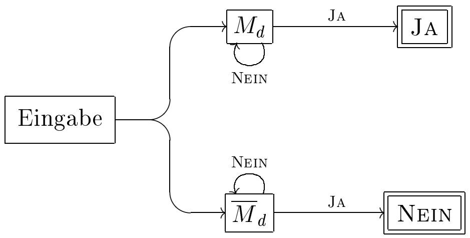
Satz 11.1.1: *Die entscheidbaren Sprachen sind abgeschlossen unter Booleschen und Kleeneschen Operationen (Abschnitt 1.4), d.h. wenn L_1 und L_2 entscheidbar sind, so auch $\overline{L_1}$, $L_1 \cap L_2$, $L_1 \cup L_2$, $L_1 \circ L_2$, L_1^n für $n \in \mathbb{N}$ und L_1^*.*

Satz 11.1.2: *Eine Sprache L ist entscheidbar genau dann, wenn L und $\overline{L}$ akzeptierbar sind.*

Beweis: Wenn M ein Entscheider für L ist, dann auch ein Akzeptor für L. Und wenn wir in M die akzeptierenden und die verwerfenden Zustände vertauschen, erhalten wir einen Akzeptor (sogar einen Entscheider) für $\overline{L}$.

Die Umkehrung ist aufwendiger zu beweisen: Seien M und $\overline{M}$ Akzeptoren für L bzw. $\overline{L}$. Wir machen die Maschinen mit Satz 8.4 deterministisch und schalten die konstruierten

Maschinen M_d und $\overline{M}_d$ parallel (siehe Abschnitt 7.5), lassen sie also auf getrennten Bändern gleichzeitig arbeiten.



M_d und $\overline{M}_d$ müssen nicht immer terminieren, aber bei jeder Eingabe w akzeptiert genau eine von beiden: M_d bei $w \in L$, $\overline{M}_d$ bei $w \notin L$. Entsprechend akzeptieren oder verwerfen wir. Wenn M_d oder $\overline{M}_d$ nichtakzeptierend terminieren will, lassen wir sie in eine Endlosschleife laufen. Die so konstruierte Maschine entscheidet L. ☐

11.2 Das Halteproblem ist nicht entscheidbar

Wir wollen also beweisen, daß das Halteproblem nicht entscheidbar ist. Das *Halteproblem* ist gegeben durch die Frage: „Terminiert (hält) die DTM M bei Eingabe w?" Wir befassen uns also im Rest des Abschnitts nur noch mit deterministischen Maschinen; das ist keine Einschränkung (siehe Schluß des Abschnitts). Als Aufgabe läßt sich das Halteproblem so formulieren:

Eingabe: Eine Turingmaschine M und ein Wort w.

Frage: Terminiert (hält) die Berechnung von M bei Eingabe w?

Da die von uns betrachteten „Aufgaben" von Turingmaschinen „gelöst" werden sollen, müssen die Eingaben als Wörter dargestellt werden; die Antwort auf die Frage besteht im Akzeptieren oder Verwerfen. In Kap. 7 haben wir schon Beispiele von Turingmaschinen und Eingaben dafür aufgeschrieben, also als Wörter über dem Alphabet des Textsystems dargestellt. In der gleichen Weise können wir Turingmaschinen als Eingaben für das Halteproblem darstellen. Allerdings müssen wir zur Darstellung aller möglichen Eingaben dasselbe (zuvor fest gewählte) Alphabet benutzen. Die Turingmaschinen enthalten aber ihrerseits wieder Alphabete, das Eingabe- und das Bandalphabet. Es gibt unendlich viele Turingmaschinen, die jeweils verschiedene Alphabete verwenden. Wir müssen also unendlich viele verschiedene Alphabete mit den Zeichen eines fest gewählten, endlichen (!) Alphabets darstellen. Wir kodieren deshalb die Zeichen der darzustellenden Alphabete in Wörter. Die gleichen Überlegungen gelten auch für die Zustandsmengen und die Zahl der Arbeitsbänder. Einen Vorschlag für ein solches „universelles" Alphabet machen wir im letzten Abschnitt, wenn wir einen Akzeptierer für das Halteproblem skizzieren.

Wir wollen beweisen, daß das Halteproblem nicht lösbar ist, daß es also keine DTM gibt, die die Frage entscheidet. Dafür müssen wir uns nicht auf ein universelles Alphabet festlegen. Wir wählen irgendeins und nennen es Γ_{HP}. Im folgenden betrachten wir nur DTMn mit Γ_{HP}

als Eingabealphabet. Für Wörter $v, w \in \Gamma_{\mathrm{HP}}^*$ bedeutet dann „v terminiert bei Eingabe w": v ist die Darstellung (in Γ_{HP}) einer DTM M mit Γ_{HP} als Eingabealphabet, die bei Eingabe w terminiert. Achtung: w ist ein beliebiges Wort über Γ_{HP}, es kann insbesondere selbst eine DTM darstellen; durch v ist aber eindeutig festgelegt, wie die Buchstaben der Eingabe w von M verarbeitet werden. Damit können wir das Halteproblem als eine Relation auf Γ_{HP}^* repräsentieren:

$$\begin{aligned}\mathrm{HP} = \{(v, w) \in \Gamma_{\mathrm{HP}}^* \times \Gamma_{\mathrm{HP}}^* \mid\ & v \text{ stellt eine DTM über } \Gamma_{\mathrm{HP}} \text{ dar,} \\ & \text{die bei Eingabe } w \text{ terminiert}\}.\end{aligned}$$

Um zu beweisen, daß HP unentscheidbar ist, betrachten wir das Selbstanwendbarkeitsproblem von HP:

$$\mathrm{SP} = \{w \in \Gamma_{\mathrm{HP}}^* \mid (w, w) \in \mathrm{HP}\}.$$

Damit haben wir statt einer Relation eine Menge, also eine Sprache, wie wir es gewohnt sind. — Das Paar (w, w) sieht zunächst befremdlich aus: Eine Turingmaschine, die sich selbst als Eingabe bekommt? In der Informatik sind aber solche „Selbstbezüglichkeiten" ganz üblich: Mit den meisten textverarbeitenden Programmen kann man ihren eigenen Kode manipulieren. Und Turingmaschinen sind einfache textverarbeitende Programme.

Wäre HP entscheidbar, so auch SP. Hätten wir nämlich eine DTM M_{HP}, die HP entscheidet, könnten wir daraus eine DTM M_{SP} basteln, die SP entscheidet: M_{SP} verdoppelt eine Eingabe w, übergibt sie an M_{HP} und übernimmt die Antwort.

Es genügt also zu beweisen, daß SP unentscheidbar ist. Das zeigen wir mit einem Widerspruchsbeweis: Wir nehmen an, es gäbe eine DTM M_{SP}, die SP entscheidet, und leiten daraus einen Widerspruch ab. Dazu wird M_{SP} „umgedreht": Wir fügen Übergänge hinzu, die akzeptierende Endkonfigurationen in sich selbst übergehen lassen; statt zu akzeptieren, terminiert sie also nicht. Die neue DTM nennen wir M'. Für sie gilt:

M' terminiert bei Eingabe w genau dann nicht, wenn

M_{SP} die Eingabe w akzeptiert, also genau dann, wenn

$(w, w) \in \mathrm{HP}$, also genau dann (nach Def. von HP), wenn

w eine DTM über Γ_{HP} darstellt, die bei Eingabe w terminiert.

M' ist eine DTM über Γ_{HP}; es sei $w_0 \in \Gamma_{\mathrm{HP}}^*$ ihre Darstellung. Geben wir M' ihre eigene Darstellung ein, setzen also oben w_0 für w ein, erhalten wir

M' terminiert bei Eingabe w_0 genau dann nicht, wenn

M' bei Eingabe w_0 terminiert.

Das ist ein Widerspruch; fertig.

Satz 11.2.1: *Das Halteproblem ist nicht entscheidbar.*

Folgerung 11.2.2: *Ebenfalls unentscheidbar sind:*

1) Das Halteproblem für NTMn.

2) Das Akzeptierungsproblem für DTMn oder NTMn.

3) Das Wortproblem für allgemeine erzeugbare Sprachen.

11.3 Das Halteproblem ist akzeptierbar

Nach dem eben Bewiesenen ist das verblüffend, aber leicht einzusehen: Die Frage „Terminiert die DTM M bei Eingabe w?“ beantworten wir, indem wir M mit w starten. Terminiert M, akzeptieren wir das Paar (M, w); sonst tun wir gar nichts. Nach der Churchschen These (Kap. 10) gibt es eine DTM A_{HP}, die diese Berechnung ausführt. A_{HP} akzeptiert eine Eingabe genau dann, wenn M auf w terminiert. Also akzeptiert A_{HP} das Halteproblem HP. Fertig.

Für alle, denen die Churchsche These (noch) nicht einleuchtet, skizzieren wir, wie man A_{HP} konstruieren könnte. Dazu geben wir erst ein Beispiel für eine Kodierung von Turingmaschinen und Eingaben über einem festen Alphabet Γ_{HP}. (In HP sind nur DTMn über dem Eingabealphabet Γ_{HP} berücksichtigt; da wir aber die Bandsymbole trotzdem kodieren müssen, kodieren wir alle Symbole einheitlich, können also mit A_{HP} DTMn über beliebigen Eingabealphabeten simulieren.) Wir machen es uns dabei ganz einfach und kodieren Buchstaben, Zustände und Bandanzahlen durch Folgen von Einsen mit vorgestelltem e bzw. q bzw. k. Das Alphabet $\{a, b, c\}$ wird durch die endliche Sprache $\{e, e1, e11\}$ dargestellt, allgemein ein Alphabet $\{a_0, a_1, \dots, a_n\}$ durch $\{e, e1, \dots, e1^n\}$; für Zustände sieht es entsprechend aus, und eine Zahl n wird durch $k1^n$ kodiert. Die Beispieleingabe $w = aaabbbccc$ aus 7.4 können wir dann als $\underline{w} = eeee1e1e1e11e11e11$ darstellen und die Turingmaschine

$$M = (\{q_A, q_1, q_2, q_3, q_4, q_5, q_6\}, \{a, b, c\}, \{a, b, c, \Box, \$\}, \Box, 1, \delta, q_A, \{q_5\})$$

aus 7.4 mit $\delta(q_A, \Box) = (q_5, \Box, N)$, $\delta(q_A, a) = (q_1, \$, R)$ ... erhält den Kode

$$\begin{aligned}\underline{M} =& (\{q, q1, q11, q111, q1111, q11111, q111111\}, \{e, e1, e11\}, \{e, e1, e11, e111, e1111\},\\ & e111, k1, \{((q, e111), (q11111, e111, N)), ((q, e), (q1, e1111, R)), \dots\}, q, \{q11111\}).\end{aligned}$$

Den Kode eines Zeichens z, eines Wortes w oder einer Turingmaschine M bezeichnen wir also mit $\underline{z}$, $\underline{w}$ bzw. $\underline{M}$. Hier ist $\{1, e, q, k, L, N, R, (,), ,, \{, \}\}$ das zur Darstellung verwendete Alphabet. Diese Darstellung ist ziemlich umständlich; es geht nur darum, daß es *prinzipiell* möglich ist, die Eingaben für das Halteproblem über einem festen endlichen Alphabet darzustellen.

Mit einer solchen (oder anderen) Kodierung kann die obige DTM A_{HP} folgendermaßen arbeiten. Sie hat zwei (oder mehr) Arbeitsbänder. Bei Eingabe $(\underline{M}, \underline{w})$ schreibt sie die Anfangskonfiguration von M zu w auf Band 1 und simuliert dann die Berechnung von M, indem sie Schritt für Schritt die Konfiguration ändert. In jedem Schritt kopiert sie aus Band 1 den Zustand $\underline{q}$ und die Symbole $\underline{a_1}, \dots, \underline{a_n}$, die M gerade sieht, nach Band 2, sucht dann in $\underline{M}$ (in dem Teil, der δ kodiert) den richtigen Übergang dazu, kopiert die Anweisungen auch nach Band 2 und führt sie in der Konfiguration auf Band 1 aus. (Wer will, kann dabei den neuen Zustand und die neuen gelesenen Symbole gleich nach Band 2 kopieren. Das Kopieren ist nötig, weil A_{HP} sich das nicht alles in den Zuständen merken kann – warum nicht?) Wenn es in $\underline{M}$ keinen Übergang zu $\underline{q}, \underline{a_1}, \dots, \underline{a_n}$ gibt, terminiert A_{HP} und akzeptiert. Also akzeptiert sie die Eingabe $(\underline{M}, \underline{w})$ genau dann, wenn M bei Eingabe w terminiert.

Satz 11.3.1: *Das Halteproblem ist akzeptierbar.*

Dieselbe Konstruktion für NTMn liefert eine NTM A_{NHP}, die das Halteproblem für NTMn akzeptiert; man muß nur die nichtdeterministische Suche richtig machen. Eine minimale Änderung (wo?) läßt A_{HP} die Eingabe $(\underline{M}, \underline{w})$ nur akzeptieren, wenn M die Eingabe w akzeptiert; entsprechend für A_{NHP}. Die erhaltene DTM bzw. NTM heißt U für *Universelle Turingmaschine*, weil sie beliebige Turingmaschinen simulieren kann. Schließlich kann man mit denselben Ideen eine NTM konstruieren, die beliebige Grammatiken simuliert. (Wer will, kann diese NTM mit Satz 9.3.2 in eine Universelle Grammatik verwandeln, die beliebige Grammatiken und daraus ableitbare Wörter erzeugt.) Zusammen erhalten wir:

Folgerung 11.3.2: *Ebenfalls akzeptierbar sind:*

1) Das Halteproblem für NTMn.

2) Das Akzeptierungsproblem für DTMn oder NTMn.

3) Das Wortproblem für allgemeine erzeugbare Sprachen.

Schließlich folgt daraus und aus Satz 11.1.2 und Satz 11.2.1 bzw. Folgerung 11.2.2:

Satz 11.3.3: *Das Komplement des Halteproblems ist nicht akzeptierbar; ebenso für die anderen Sprachen aus Folgerung 11.3.2.*

Wir haben also die Chomsky-Hierarchie aus Abschnitt 9.2 um zwei Stufen „erhöht“: durch die entscheidbaren Sprachen zwischen den kontextsensitiven und den akzeptierbaren und durch die nicht akzeptierbaren ganz oben. Daß es entscheidbare, nicht kontextsensitive Sprachen gibt, können wir hier nicht beweisen; daß die beiden anderen Stufen echt sind, besagen gerade die Sätze 11.2.1 und 11.3.1 sowie 11.3.3.

11.4 Aufgaben

Aufgabe 11.1: Beweist, daß die folgenden Sprachen entscheidbar sind:

$$L_1 = \{vv^{-1} \mid v \in \{a,b\}^*\}, \qquad L_2 = \{w \in \{a,b\}^* \mid w = w^{-1}\},$$
$$L_3 = \{a^n b^n c^n \mid n \in \mathbb{N}\}, \qquad L_4 = \{a^n b^m c^k \mid n,m,k \in \mathbb{N},\ n = m \text{ oder } n = k\}.$$

Aufgabe 11.2: a) Beweist, ohne Satz 11.1.1 zu benutzen, daß alle Durchschnitte $L_i \cap L_j$, $i,j = 1,\ldots,4$ der Sprachen aus Aufgabe 11.1 entscheidbar sind. Setzt voraus, daß $L_1,\ldots,L_4$ entscheidbar sind, aber konstruiert keine neuen Turingmaschinen.

b) Dasselbe für die Vereinigungen $L_i \cup L_j$. Hier braucht ihr neue Turingmaschinen.

Aufgabe 11.3: a) Beweist, daß jede kontextfreie Sprache entscheidbar ist. Wenn Ihr Euch auf Grammatiken ohne „Löschregeln" beschränkt, bekommt Ihr einen einfachen Beweis mit Abschnitt 6.6 und Kap. 10: Ihr müßt nur das Verfahren in 6.6 genauer ausarbeiten, darauf achten, daß es deterministisch ist (was heißt das bei einem Verfahren?) und immer terminiert, und dann die Churchsche These anwenden. Alternativ könnt Ihr Satz 9.3.1 anwenden und die dort konstruierte NTM total und deterministisch machen (Satz 8.4). Der zweite Beweis ist komplizierter, aber liefert explizit eine Turingmaschine.

b) Dasselbe für reguläre Sprachen. Was wird einfacher?

c) Dasselbe für kontextsensitive Sprachen. Was wird schwieriger?

d) Wo scheitern die beiden Beweise für allgemeine erzeugbare Sprachen? Könnt Ihr die Fehler beheben?

Aufgabe 11.4: (Schwierig) a) Benutzt Aufgabe 11.3, um zu zeigen, daß das Wortproblem für kontextfreie Sprachen (Abschnitt 6.6) entscheidbar ist. Was ist der Unterschied zu 3a)? Unterscheiden sich die beiden Beweise aus 3a) jetzt noch? Welche Eingaben erwartet der konstruierte Entscheider?

b) Ebenso für reguläre und kontextsensitive Sprachen.

c) Warum ist das Wortproblem für allgemeine erzeugbare Sprachen nicht entscheidbar?

Aufgabe 11.5: Beweist Satz 11.1.1.

Aufgabe 11.6: Beweist: Wenn L_1 und L_2 entscheidbar sind, so auch $L_1 \times L_2$. Was geht schief bei der Potenzmenge $\mathcal{P}(L_1)$?

Aufgabe 11.7: Beweist Folgerung 11.2.2.

Aufgabe 11.8: Welche der folgenden Probleme sind entscheidbar?

a) Die DTM M terminiert bei Eingabe w nach höchstens $|w|$ Schritten.

b) Die DTM M terminiert bei Eingabe λ.

c) Die DTM M ist total.

Um zu beweisen, daß ein Problem unentscheidbar ist, müßt Ihr es auf das Halte- oder ein anderes unentscheidbares Problem zurückführen: Annehmen, das Problem sei entscheidbar, und aus dem angenommenen Entscheider einen Entscheider fürs Halteproblem basteln – Widerspruch. Das Basteln ist nicht schwierig, aber die Konstruktion ist arg abstrakt.

Aufgabe 11.9: Konstruiert eine NTM M, die beliebige Grammatiken simuliert. Bevor Ihr anfangt zu konstruieren, macht Euch klar, was Ihr konstruieren wollt.

Aufgabe 11.10: Gebt eine *universelle Grammatik* an, die beliebige Grammatiken und daraus ableitbare Wörter erzeugt. Mahnung wie in Aufgabe 9.

Aufgabe 11.11: Konkretisiert die aufgestockte Chomsky-Hierarchie, indem Ihr Sprachen mit folgenden Eigenschaften angebt:

a) regulär, nicht endlich; b) kontextfrei, nicht regulär; c) kontextsensitiv, nicht kontextfrei; d) entscheidbar, nicht kontextsensitiv; e) erzeugbar, nicht entscheidbar; f) nicht erzeugbar. Beachtet die Hinweise am Schluß von Abschnitt 11.3.

Aufgabe 11.12: (Schwierig und langwierig) Schreibt in einer Programmiersprache Eurer Wahl ein Programm, das Kodierungen von DTMn auf (syntaktische) Korrektheit überprüft. Programmiert dann rekursiv Berechnungen von DTMn, indem Ihr Folgen von Konfigurationen simuliert. Könnt Ihr Euch das Überprüfen der Korrektheit zu Beginn auch sparen? Was passiert, wenn die simulierte DTM auf der Eingabe nicht terminiert? Programmiert damit einen Interpreter für DTMn. Wie könnt Ihr daraus das Halteproblem programmieren, obwohl es nicht entscheidbar ist?

Aufgabe 11.13: (Langwierig) Konstruiert eine DTM, die Programme einer Programmiersprache Eurer Wahl interpretiert.

12 Komplexitätstheorie: Das P-NP-Problem

Versetzen wir uns (gedanklich) etwa 30 Jahre zurück: Damals waren die Rechenzeit eines Programms und der Speicherplatzbedarf die wichtigsten Kostenfaktoren bei der Verwendung von Computern. Viele Unternehmen und wissenschaftliche Institutionen besaßen keinen eigenen Computer, sie ließen ihre Programme in großen Rechenzentren laufen. Diese Rechenzentren verkauften „Rechenzeit". Für die Kunden dieser Rechenzentren war die Anzahl der Rechenschritte proportional zu den Kosten eines Programmlaufs. Betriebswirtschaftlich betrachtet ist Aufwand *eine betriebliche Eigenkapitalsminderung, die durch Ausgaben hervorgerufen wird.* Vielleicht war dies der Grund dafür, daß die Programmlaufzeit als Zeit*aufwand* betrachtet wurde. Die Rechenzeit war damals etwas, was *verbraucht* werden konnte. Das Ziel wirtschaftlichen Handelns war es, den *Zeitverbrauch* (also die Kosten) zu minimieren. Auch heute spielt der Zeitbedarf (und teilweise der Speicherverbrauch) bei bestimmten Anwendungen eine entscheidende Rolle, etwa bei häufig verwendeten Teilprogrammen wie den Suchverfahren in Textsystemen, in Datenbanken oder im Netz. Dafür werden ständig neue extrem „schnelle" Algorithmen und dafür geeignete Datenstrukturen entwickelt.

Damals waren in der Mathematik die Turingmaschinen aus der Berechenbarkeitstheorie bekannt. Mit Turingmaschinen konnten die Mathematiker beweisen, daß es Probleme gibt, die nicht berechenbar sind. Die entsprechenden Theoreme und Beweise wurden aber nicht mit Computern in Zusammenhang gebracht! Erst später wurden die Turingmaschinen als mathematisches Modell von Computern betrachtet. Eine Turingmaschinen-„Berechnung" kann als eine schrittweise Ausführung einer Folge von Elementaroperationen betrachtet werden, ganz so wie die Programmausführung in einem Computer. Das Interesse an einer Verringerung des Zeit- und Platzverbrauchs legt also nahe, diese Aspekte mit Hilfe einer solchen Analogie mathematisch zu modellieren.

Wenn wir einen Zustandsübergang als Rechenschritt ansehen, dann können wir die Anzahl der Konfigurationen einer terminierenden Berechnung als „Programmlaufzeit" betrachten. Darauf aufbauend können wir den „Zeitaufwand" zur Lösung eines Problems mathematisch definieren. Daß die Programmlaufzeit heute meist kein Kostenfaktor mehr ist, liegt wohl kaum an den Ergebnissen dieser „mathematischen Theorie" über Zeitaufwände. Aber nachdem sie sich etabliert hatte, begann sie ein Eigenleben zu führen. Dies wird an der Definition des Aufwands nichtdeterministischer Turingmaschinen deutlich. Seit damals hat sich eine umfangreiche und weitverzweigte mathematische Disziplin mit vielen interessanten Fragestellungen, Methoden und Ergebnissen entwickelt, die *Komplexitätstheorie.*

In den folgenden Abschnitten werden wir den Zeit- und Platzbedarf von Turingmaschinen definieren und für konkrete Beispiele bestimmen. Wir beschränken uns dabei auf Akzeptoren bzw. Entscheider. Dann werden wir umgekehrt Turingmaschinen mit vorgegebenem Zeit- und Speicherbedarf betrachten und untersuchen, welche Probleme damit noch akzeptierbar bzw. entscheidbar sind. Wir fragen also, wie stark die Beschränkung von Zeit- und Platzbedarf das Modell einschränkt. Wie bisher werden wir die Frage nach der Äquivalenz

deterministischer und nichtdeterministischer Maschinen stellen.

Eine spezielle solche Frage ist als P-NP-Problem weit über die Komplexitätstheorie hinaus bekannt geworden. Es wurde Anfang der 70er Jahre formuliert, und zwar interessanterweise unabhängig sowohl in den USA als auch in der UdSSR. Vielleicht lag das daran, daß durch die technische Weiterentwicklung der Computer die „symbolische Berechnung" in den Mittelpunkt des Interesses rückte. Wenn Leute versuchen, Schachprogramme zu schreiben oder für Handelsreisende die kürzesten Routen zu berechnen, dann merken sie sehr schnell, daß die Programme für realistische Eingaben zu lange (etwa einige Jahre) laufen würden. All diese Probleme haben die Eigenschaft, daß eine mögliche Lösung schnell überprüfbar ist. Aber diese Lösung zu finden, dauert meist zu lange. Diese Probleme können also nur von nichtdeterministischen Turingmaschinen „schnell" gelöst werden.

Das führt zu der Frage, ob die Probleme, die nichtdeterministisch effizient lösbar sind, sich auch deterministisch effizient lösen lassen. Anders formuliert: Sind solche Probleme aus den Bereichen „Künstliche Intelligenz" oder „Operations Research" (irgendwann einmal) auch praktisch schnell lösbar?

12.1 Zeit- und Platzaufwand deterministischer Maschinen

Wie in der Einleitung gesagt, ist es naheliegend, die Anzahl der Konfigurationen einer terminierenden Berechnung als „Rechenzeit" zu betrachten. Ebenso kann man die Zahl der benutzten Felder als „Platzbedarf" ansehen. Für nichtterminierende Berechnungen sind Zeit und Platz nicht definiert, daher sind die Aufwandsfunktionen i.a. partiell. Wir fangen mit deterministischen Maschinen an; in praktischen Beispielen terminieren die immer, dann sind die Aufwandsfunktionen total. In Abschnitt 12.3 verallgemeinern wir die Definition auf nichtdeterministische Maschinen; dann zeigen wir auch, daß man mit totalen Maschinen auskommt.

Für den Rest des Abschnitts sei M eine DTM mit k Bändern und Eingabealphabet Σ. Die *Länge* einer terminierenden Berechnung von M ist die Anzahl von Schritten in der Berechnung. Die *Rechenzeit* (oder der *Zeitbedarf*) von M zur Eingabe w, $t_M(w)$, ist die Länge der Berechnung von M bei Eingabe w, falls sie terminiert; sonst ist $t_M(w)$ undefiniert. Das ergibt also eine partielle Funktion $t_M : \Sigma^* \to \mathbb{N}$.

Die *Größe* einer Konfiguration $k = (q, w_1, \dots, w_k)$ ist die Zahl der von ihr belegten Bandfelder, also $(|w_1|-1)+\cdots+(|w_k|-1)$. (Das Symbol $\triangleright$ zur Angabe der Kopfposition zählen wir nicht mit.) Der *Platzbedarf* einer terminierenden Berechnung von M ist die maximale Größe einer Konfiguration während der Berechnung. Der *Platzbedarf* von M zur Eingabe w, $s_M(w)$, ist der Platzbedarf der Berechnung von M bei Eingabe w, falls sie terminiert; sonst ist $s_M(w)$ undefiniert. Das ergibt also eine partielle Funktion $s_M : \Sigma^* \to \mathbb{N}$. ($s$ steht für *space*, t oben für *time*.)

Deterministische endliche Automaten, als DTMn aufgefaßt, lesen in jedem Schritt ein Symbol der Eingabe und terminieren, wenn die Eingabe fertig gelesen ist (falls die Überführungsfunktion vollständig ist, sonst eventuell früher); für eine Eingabe der Länge n ist der Platzbedarf also n, der Zeitbedarf höchstens n. Die Entscheider in Kap. 7 hatten meist höheren Bedarf. Die DTM zum Entscheiden der „Spiegelwörter" $v\$v^{-1}$ z.B., die ein Arbeitsband zur Verfügung hat, kopiert den Teil der Eingabe bis zum ersten \$ aufs Band und vergleicht dann; dazu braucht sie maximal $2n$ Bandfelder und n Schritte für Eingaben der Länge n. So viele Felder braucht sie nur, wenn die Eingabe kein \$ enthält (wieso?); dann braucht sie auch wirklich n Schritte (wann sonst noch?).

Allgemein bestimmen wir den Zeit- und Platzbedarf nicht für alle Eingaben, sondern suchen für jede Eingabelänge nur noch den *schlimmsten Fall* und schätzen die übrigen Fälle damit ab. Diese *worst case analysis* vereinfacht das Ausrechnen und Angeben des Aufwands und reicht in den meisten Fällen aus. Deswegen definieren wir den Zeit- und Platzaufwand jetzt in Abhängigkeit von der Länge der Eingabe, also als Funktionen von $\mathbb{N}$ nach $\mathbb{N}$. Auch wenn wir später umgekehrt Zeit- und Platzschranken vorgeben, die die Turingmaschinen nicht überschreiten dürfen, ist es einfacher, dies nur für Eingabelängen zu tun.

Der *Zeitaufwand* von M zur Eingabelänge n ist

$$T_M(n) = \max\{t_M(w) \mid w \in \Sigma^n \text{ und } t_M(w) \text{ definiert}\},$$

falls M für ein $w \in \Sigma^n$ terminiert, sonst ist $T_M(n)$ undefiniert. Der *Platzaufwand* von M zur Eingabelänge n ist

$$S_M(n) = \max\{s_M(w) \mid w \in \Sigma^n \text{ und } s_M(w) \text{ definiert}\},$$

falls M für ein $w \in \Sigma^n$ terminiert, sonst ist $S_M(n)$ undefiniert. Das ergibt also partielle Funktionen $T_M : \mathbb{N} \to \mathbb{N}$ und $S_M : \mathbb{N} \to \mathbb{N}$.

Bemerkungen: (1) Da die Eingabe in der Anfangskonfiguration enthalten ist, gilt $S_M(n) \geq n$ falls $S_M(n)$ definiert ist. (2) Im allgemeinen verbrauchen Turingmaschinen für längere Eingaben mehr Zeit und Platz; dann sind die Aufwände *monotone* Funktionen, d.h. für $m \leq n$ gilt $T_M(m) \leq T_M(n)$ und $S_M(m) \leq S_M(n)$. Manchmal aber sind Probleme so durch Wörter repräsentiert, daß gewisse Wortlängen – z.B. ungerade – gar nicht vorkommen. Eine gut programmierte Turingmaschine würde das beim Lesen merken und solche Eingaben gleich verwerfen, also nur wenig Zeit und Platz brauchen; ihre Aufwände sind also nicht monoton. Deswegen werden die Aufwände zur Länge n manchmal als der maximale Verbrauch für Eingaben der Länge $\leq n$ (statt $= n$) definiert.

Bisher haben wir den Aufwand von DTMn analysiert, die ein vorgegebenes Problem lösen. Umgekehrt können wir von einer totalen Funktion $f : \mathbb{N} \to \mathbb{N}$ als Aufwandsschranke ausgehen und nach den DTMn fragen, die mit dem dadurch vorgegebenen Zeit- und Platzverbrauch auskommen. Eine DTM M nennen wir *zeit-* bzw. *platzbeschränkt* durch eine totale Funktion $f : \mathbb{N} \to \mathbb{N}$, wenn für alle $n \in \mathbb{N}$ gilt:

$$T_M(n) \leq f(n) \quad \text{bzw.} \quad S_M(n) \leq f(n),$$

falls die jeweilige linke Seite definiert ist. Als Aufwandsschranken sind einfache Funktionen $f(n)$ von Interesse wie n, $n \cdot \log(n)$, n^2, n^3, 2^n; mehr dazu in den Abschnitten 12.2 und 12.4.

Schließlich übertragen wir diese Sichtweise auf Sprachen. Wird die Sprache L von einer durch f zeit- bzw. platzbeschränkten DTM akzeptiert, so sagen wir: f ist eine *obere Schranke* für den Zeit- bzw. Platzbedarf von L, oder: L hat Zeit- bzw. Platz*komplexität* höchstens f. Die Klasse der Sprachen der Zeit- bzw. Platzkomplexität höchstens f wird mit DTIME(f) bzw. DSPACE(f) bezeichnet.

12.2 Polynome als Zeit- oder Platzschranken

Polynome haben als zeit- oder platzbeschränkende Funktionen in der Komplexitätstheorie eine besondere Bedeutung erlangt. Deshalb betrachten wir im folgenden nur Polynome als Aufwandsschranken. Ein *Polynom* (über $\mathbb{N}$) ist eine Funktion $p : \mathbb{N} \to \mathbb{N}$, für die es Koeffizienten $a_0, \ldots, a_k \in \mathbb{N}$ gibt, so daß

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$$

für $n \in \mathbb{N}$ gilt; dabei sei $a_k \neq 0$, für $k = 0$ ist auch $a_k = 0$ erlaubt. k heißt *Grad* des Polynoms. Da wir negative Koeffizienten nicht zulassen, sind alle Polynome monoton.

Wir definieren nun eine der beiden berühmten Komplexitätsklassen, die uns in diesem Kapitel beschäftigen sollen. Eine DTM ist *polynomiell zeit-* bzw. *platzbeschränkt*, falls sie durch ein Polynom zeit- bzw. platzbeschränkt ist. Die *Zeitkomplexitätsklasse* P besteht aus allen Sprachen, die von polynomiell zeitbeschränkten DTMn akzeptiert werden:

$$\mathrm{P} = \bigcup_{p \text{ Polynom}} \mathrm{DTIME}(p)$$

Die Klasse P enthält die Probleme, die als „praktisch lösbar" angesehen werden. Insbesondere sind alle Sprachen, die wir bisher in Beispielen oder Aufgaben betrachtet haben, in P; sie sind sogar von *linearer*, *quadratischer* oder *kubischer* Komplexität, d.h. durch Polynome vom Grad 1, 2 oder 3 zeitbeschränkt. Reguläre Sprachen z.B. sind von linearer, kontextfreie von höchstens kubischer Zeitkomplexität. Kontextsensitive Sprachen sind von linearer Platzkomplexität, polynomielle Zeitschranken gibt es dazu i.a. nicht. (Aber $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ z.B. ist in linearer Zeit entscheidbar.) Die Zeitklasse P ist vor allem von *praktischem Interesse:* Allgemein werden die und nur die Probleme als praktisch lösbar angesehen, für die es polynomiell zeitbeschränkte Verfahren gibt.

Außerdem ist die Klasse P *theoretisch interessant*, weil sie von den sie definierenden Polynomen einige schöne Eigenschaften „erbt":

(a) Wenn wir polynomiell zeitbeschränkte DTMn „beliebig" zusammenbauen, entsteht dabei immer wieder eine polynomiell zeitbeschränkte DTM. Das brauchen wir für (b) und (c).

(b) Wir können polynomiell zeitbeschränkte DTMn „totalisieren“: Zu jeder polynomiell zeitbeschränkten DTM gibt es eine andere, die die gleiche Sprache erkennt und die für alle Eingaben terminiert. Bei polynomiell zeitbeschränkten DTMn brauchen wir uns also um die Termination nicht zu kümmern.

(c) P ist unter den Booleschen Mengenoperationen abgeschlossen.

Zu (a): Die Klasse der Polynome ist unter Addition, Multiplikation und Komposition abgeschlossen. D.h. wenn p_1, p_2 Polynome vom Grad k_1, k_2 sind, dann sind $q(n) = p_1(n) + p_2(n)$, $q'(n) = p_1(n) \cdot p_2(n)$ und $q''(n) = p_1(p_2(n))$ Polynome vom Grad $\max(k_1, k_2)$, $k_1 + k_2$ bzw. $k_1 \cdot k_2$. (Bitte nachrechnen!) Seien jetzt M_1 und M_2 durch p_1 bzw. p_2 zeitbeschränkte DTMn; wir wollen sie auf verschiedene Weise zu einer neuen DTM M zusammensetzen.

Als erstes „schalten“ wir M_1 und M_2 „hintereinander“, d.h. wir lassen erst M_1 und dann M_2 laufen. (Diese Konstruktion sollte schon geläufig sein; sie wird in den folgenden Beweisen noch einmal genau beschrieben. Was passiert, wenn M_1 oder M_2 nicht total sind?) Lassen wir M_2 mit derselben Eingabe v wie M_1 starten (oder mit einer Eingabe derselben Länge), dann ist M durch q zeitbeschränkt; warum? Geben wir dagegen M_2 die Ausgabe w von M_1 als Eingabe, wissen wir nicht, wie lang w ist. M_1 kann aber in jedem Schritt höchstens ein Ausgabefeld bedrucken, also ist $|w| \le p_1(|v|)$. Daher ist M durch $p_2(|w|) \le p_2(p_1(|v|))$, also durch q'' zeitbeschränkt.

Schalten wir M_1 und M_2 „nebeneinander“, d.h. lassen sie parallel laufen (siehe Beweis von Satz 12.2.2 unten und 7.5), ist M augenscheinlich durch $\max(p_1, p_2)$ zeitbeschränkt; das ist i.a. kein Polynom (warum nicht?), läßt sich aber leicht durch ein Polynom abschätzen (wie?).

Starten wir schließlich in jedem Schritt von M_1 die Maschine M_2, immer mit einer Eingabe der Länge n, und lassen dann M_1 warten, bis M_2 fertig ist, so rechnet M_2 höchstens $p_1(n)$-mal; also ist M durch q' zeitbeschränkt. Noch etwas anders ist die Situation bei den Simulationen in Kap. 9. Da simuliert M_2 jeweils einen Schritt von M_1, z.B. in linearer Zeit; die Eingabe für M_2 ändert sich dabei aber. Wie läßt sich die Rechenzeit von M abschätzen? Wir lassen die Frage als Aufgabe stehen.

Zu (b): Das liegt daran, daß man (bei unärer Ein- und Ausgabe) ein Polynom p in Zeit p berechnen kann, d.h. bei Eingabe von n Einsen $p(n)$ Einsen in Zeit $p(n)$ ausgeben kann. Die Ausgabe kann dann als „Wecker“ verwendet werden: Bei jedem Schritt wird eine Eins gelöscht; wenn alle Einsen weg sind, wird die Rechnung abgebrochen.

Lemma: *Jedes Polynom p ist unär in Zeit p berechenbar.*

Beweis: Übung (fummelig). □

Satz 12.2.1: Für jede polynomiell zeitbeschränkte DTM M gibt es eine äquivalente totale, polynomiell zeitbeschränkte DTM M': (i) $L(M) = L(M')$. (ii) M' terminiert für alle Eingaben. (iii) Ist p eine Zeitschranke für M, so ist $2p(n) + n$ eine für M'.

Beweis: Es sei M eine DTM, die durch ein Polynom p zeitbeschränkt ist. Wir konstruieren eine DTM M', die aus M und dem „Wecker“ für p nach dem Lemma besteht.

Wenn M' mit einer Eingabe der Länge n gestartet wird, schreibt M' das Wort $1^{p(n)}$ auf ein zusätzliches Arbeitsband, stellt also den Wecker. Danach geht M' in den Startzustand von M. In jedem Schritt wird neben dem Schritt von M der Schreib-Lesekopf auf dem zusätzlichen Arbeitsband nach links bewegt; wird dort ein $\Box$ gelesen, dann hat M' $p(n)$ viele Rechenschritte von M durchgeführt. Wenn M die Eingabe akzeptiert hätte, dann hätten diese Schritte ausreichen müssen. Wenn M' also $p(n)$ viele Schritte von M ausgeführt hat, ohne daß M akzeptiert hat, akzeptiert M' die Eingabe nicht. M' macht $p(n)$ Schritte, wenn sie den Wecker stellt, danach höchstens n fürs Starten von M (Eingabekopf zurück), und noch einmal höchstens $p(n)$ Schritte, während sie M simuliert, zusammen also höchstens $2p(n) + n$ Schritte. □

Satz 12.2.2: P *ist abgeschlossen unter Booleschen Operationen; d.h. wenn L, L_1, L_2 Sprachen in* P *sind, so auch $L_1 \cap L_2$, $L_1 \cup L_2$ und $\overline{L}$.*

Beweis: Für $i = 1, 2$ sei $M_i = (Q_i, \Sigma, \Gamma_i, \Box, k_i, \delta_i, q_{Ai}, E_i)$ eine DTM, die L_i erkennt und durch das Polynom p_i zeitbeschränkt ist; Q_1 und Q_2 seien disjunkt.

(i) Wir konstruieren eine DTM M, die $L_1 \cap L_2$ erkennt. Dazu „schalten“ wir M_1 und M_2 „hintereinander“: M kopiert zu Beginn der Berechnung die Eingabe auf ein zusätzliches Arbeitsband. Danach wird M_1 gestartet. Wenn M_1 die Eingabe akzeptiert, dann initialisiert M alle verwendeten Bänder und kopiert die Eingabe zurück. (M_1 kann sie ja gelöscht haben.) Danach wird M_2 gestartet. Wenn M_2 akzeptiert, dann war die Eingabe sowohl aus L_1 als auch aus L_2; M akzeptiert dann auch. Andernfalls verwirft M. M akzeptiert also $L_1 \cap L_2$ und ist durch $2(p_1(n) + n) + p_2(n)$ zeitbeschränkt, daher ist $L_1 \cap L_2$ in P.

(ii) Wir konstruieren eine DTM M, die $L_1 \cup L_2$ erkennt. M hat $Q_1 \times Q_2$ als Zustandsmenge und $k_1 + k_2$ Bänder. Zu Beginn der Berechnung kopiert M die Eingabe auf das $(k_1 + 1)$-te Band. Dann werden M_1 und M_2 folgendermaßen „parallel“ simuliert. Wir definieren δ von M durch

$$\begin{aligned}&\delta((q_1, q_2), a_1, \ldots, a_{k_1}, c_1, \ldots, c_{k_2}) = \\ &((q_1', q_2'), b_1, \ldots, b_{k_1}, d_1, \ldots, d_{k_2}, X_1, \ldots, X_{k_1}, Y_1, \ldots, Y_{k_2})\end{aligned}$$

für

$$\begin{aligned}\delta_1(q_1, a_1, \ldots, a_{k_1}) &= (q_1', b_1, \ldots, b_{k_1}, X_1, \ldots, X_{k_1}) \quad \text{und} \\ \delta_2(q_2, c_1, \ldots, c_{k_2}) &= (q_2', d_1, \ldots, d_{k_2}, Y_1, \ldots, Y_{k_2})\end{aligned}$$

mit $a_i, b_i, c_j, d_j \in \Gamma_1 \cup \Gamma_2$ und $X_i, Y_j \in \{L, N, R\}$. Damit M weiterrechnen kann, wenn M_1 oder M_2 nicht akzeptierend terminiert, setzen wir dabei für $q_1 \notin E_1$, $q_2 \notin E_2$:

$$\begin{aligned}\delta_1(q_1, a_1, \ldots, a_{k_1}) &= (q_1, a_1, \ldots, a_{k_1}, N, \ldots, N), \\ \delta_2(q_2, c_1, \ldots, c_{k_2}) &= (q_2, c_1, \ldots, c_{k_2}, N, \ldots, N).\end{aligned}$$

Die Menge der akzeptierenden Zustände von M ist

$$E = \{(q_1, q_2) \mid q_1 \in E_1 \text{ oder } q_2 \in E_2\}.$$

M erkennt die Sprache $L_1 \cup L_2$ und ist durch $n + \max(p_1(n), p_2(n))$ zeitbeschränkt.

(iii) Um zu beweisen, daß P unter Komplement abgeschlossen ist, benutzen wir Satz 12.2.1. Sei also $M = (Q, \Sigma, \Gamma, \Box, k, \delta, q_A, E)$ eine DTM, die L erkennt und für alle Eingaben terminiert. Wir konstruieren aus M eine DTM $\overline{M}$, indem wir die akzeptierenden Zustände gegen die anderen Zustände austauschen: $\overline{M} = (Q, \Sigma, \Gamma, \Box, k, \delta, q_A, Q \setminus E)$. Da M immer terminiert, erkennt $\overline{M}$ das Komplement von L, also $L(\overline{M}) = \overline{L}$. Die Rechenzeiten von $\overline{M}$ und M sind durch das gleiche Polynom beschränkt. □

Augenscheinlich kann man Teil (i) auch durch Nebeneinanderschalten beweisen; Teil (ii) durch Hintereinanderschalten nur, wenn M_1 total ist (warum?).

Die Eigenschaften von P, die wir jetzt bewiesen haben, gelten allgemeiner, wenn sie entsprechend formuliert sind. So kann man jede zeitbeschränkte DTM totalisieren (Satz 12.2.1), wenn die Zeitschranke f berechenbar ist: Für Eingaben der Länge n ist der Zeitaufwand der totalisierten DTM höchstens $g(n)+f(n)$, wenn g eine Zeitschranke für die Berechnung von f ist. Unverzichtbar ist dabei, daß die Zeitschranke f total ist, wie wir es in 12.1 verlangt haben; sonst könnten wir den Wecker nicht immer stellen, also auch nicht immer klingeln lassen, und die DTM würde nicht total. Tatsächlich kann man nicht jede DTM totalisieren; sonst wären alle akzeptierbaren Sprachen entscheidbar (warum?), das stimmt aber nicht fürs Halteproblem (Kap. 11). Allgemein sind die universellen Simulationsmaschinen aus Kap. 9 nicht totalisierbar. Andererseits ist trivialerweise der Zeitaufwand jeder DTM M durch die (i.a. partielle!) Funktion T_M beschränkt. Daraus folgt, daß man die Zeitaufwandsfunktionen T_M nicht zu totalen berechenbaren Funktionen erweitern kann, wenn $L(M)$ nicht entscheidbar ist. Anders gesagt: Definiert man T_M an den undefinierten Stellen irgendwie, ist die entstehende totale Funktion nicht berechenbar.

Auch Satz 12.2.2 gilt allgemeiner: Sind L_1 und L_2 mit Zeitaufwand f_1 bzw. f_2 akzeptierbar, so auch $L_1 \cap L_2$ und $L_1 \cup L_2$ mit Zeitaufwand $\max(f_1, f_2)$. Fürs Komplement muß man i.a. totalisieren, der Zeitaufwand für den Wecker kommt dann dazu.

12.3 Zeit- und Platzaufwand nichtdeterministischer Maschinen

Jetzt wollen wir die Definitionen aus 12.1 auf nichtdeterministische Turingmaschinen übertragen. Eine NTM akzeptiert ihre Eingabe, wenn es mindestens eine akzeptierende Berechnung dazu gibt. Die Existenz anderer, auch nichtterminierender Berechnungen bleibt dabei unberücksichtigt. Deswegen betrachten wir in der Definition des Aufwandes nur den Aufwand akzeptierender Berechnungen. Ob die NTM total ist oder nicht, spielt dabei keine Rolle. Weiter soll eine zeit- oder platzbeschränkte NTM ihre Eingabe akzeptieren, wenn es eine akzeptierende Berechnung gibt, deren Aufwand innerhalb der Beschränkung liegt.

Deswegen betrachten wir in der Definition des Aufwands nur die akzeptierende Berechnung mit dem geringsten Aufwand (d.h. kürzeste Rechenzeit oder geringster Platzverbrauch). Denn wenn der „kleinste Aufwand“ schon über der Beschränkung liegt, dann gibt es überhaupt keine akzeptierende Berechnung, deren Aufwand innerhalb der Beschränkung liegt.

Im folgenden sei M eine NTM mit dem Eingabealphabet Σ. Die *Rechenzeit* (oder der *Zeitbedarf*) von M zur Eingabe w, $t_M(w)$, ist die Länge der kürzesten akzeptierenden Berechnung von M bei Eingabe w, falls M die Eingabe akzeptiert; sonst ist $t_M(w)$ undefiniert. Der *Platzbedarf* von M zur Eingabe w, $s_M(w)$, ist der minimale Platzbedarf einer akzeptierenden Berechnung von M bei Eingabe w, falls M die Eingabe akzeptiert; sonst ist $s_M(w)$ undefiniert. Das definiert partielle Funktionen $t_M : \Sigma^* \to \mathbb{N}$ und $s_M : \Sigma^* \to \mathbb{N}$, die also auch dann undefiniert sind, wenn die NTM zwar terminiert, aber die Eingabe nicht akzeptiert.

Der *Zeitaufwand* von M zur Eingabelänge n ist

$$T_M(n) = \max\{t_M(w) \mid w \in \Sigma^n \text{ und } t_M(w) \text{ ist definiert}\},$$

falls M ein $w \in \Sigma^n$ akzeptiert, sonst undefiniert. Der *Platzaufwand* von M zur Eingabelänge n ist

$$S_M(n) = \max\{s_M(w) \mid w \in \Sigma^n \text{ und } s_M(w) \text{ ist definiert}\},$$

falls M ein $w \in \Sigma^n$ akzeptiert, sonst undefiniert. Das definiert partielle Funktionen $T_M : \mathbb{N} \to \mathbb{N}$ und $S_M : \mathbb{N} \to \mathbb{N}$.

Die Definitionen der Zeit- und Platzbeschränktheit durch (auch hier totale!) Funktionen $f : \mathbb{N} \to \mathbb{N}$ und der zugehörigen Komplexitätsklassen NTIME(f) und NSPACE(f) sind dieselben wie in Abschnitt 12.1.

Die NTMn aus Kap. 8 haben alle linearen Zeit- und Platzaufwand; auf polynomialen Aufwand kommen wir im nächsten Abschnitt. Beliebig hohe Aufwände haben die Simulationsmaschinen, wenn sie überhaupt total abschätzbar sind. Sich den Aufwand der Simulation von Ableitungen mit Grammatiken zu überlegen, ist eine gute Übung. Für die Simulation von NTMn durch DTMn (Satz 8.4) gilt:

Satz: *Ist die NTM M durch f und g zeit- bzw. platzbeschränkt, so gibt es eine äquivalente DTM M', die durch $c^{f(n)}$ für eine reelle Zahl $c > 1$ zeitbeschränkt und durch $f(n) + g(n)$ platzbeschränkt ist.*

Beweis: Die DTM M' aus dem Beweis von Satz 8.4 simuliert alle Berechnungen von M zu einer Eingabe w der Länge n, indem sie auf einem zusätzlichen Band die Pfade im Berechnungsbaum von M für w in lexikographischer Reihenfolge aufzählt. Wenn M die Eingabe w akzeptiert, gibt es eine akzeptierende Berechnung der Länge $\leq f(n)$; sobald M' die findet, akzeptiert sie. Also simuliert M' weniger als $k^{f(n)}$ Berechnungen von M, alle von der Länge $\leq f(n)$, wobei k die maximale Verzweigung der Übergangsrelation von M ist. (Wieso? Dazu muß man wissen, wieviel Knoten ein Baum der Tiefe h höchstens hat, wenn jeder Knoten maximal k Nachfolger besitzt.) Nach jeder Simulation wird der Pfad um eins hochgesetzt, das sind noch einmal $\leq f(n)$ Schritte. Also ist die ganze Simulation durch $2 \cdot f(n) \cdot k^{f(n)}$ zeitbeschränkt. Die Schranke kann man durch $c^{f(n)}$ für ein geeignetes $c > k$ ersetzen (Aufgabe). – Alle Berechnungen können auf dem selben Platz simuliert werden, dazu kommt der Platzaufwand $f(n)$ für den Pfadzähler. □

12.4 Das P-NP Problem

Jetzt betrachten wir Polynome als Zeitschranken für NTMn und versuchen, die Ergebnisse aus 12.2 zu übertragen.

Die *Zeitkomplexitätsklasse* NP besteht aus allen Sprachen, die von polynomiell (d.h. durch ein Polynom) zeitbeschränkten NTMn akzeptiert werden:

$$\mathrm{NP} = \bigcup_{p \text{ Polynom}} \mathrm{NTIME}(p)$$

Da wir jede DTM als NTM auffassen können, ist jede Sprache aus P in NP: $\mathrm{P} \subseteq \mathrm{NP}$. Ist die Inklusion echt, $\mathrm{P} \neq \mathrm{NP}$, oder gilt $\mathrm{P} = \mathrm{NP}$? Merkwürdigerweise ist die Frage seit vielen Jahren offen. Es gibt eine Menge interessanter Sprachen in NP, für die man keine polynomiell zeitbeschränkten Algorithmen kennt; man kann aber auch nicht beweisen, daß sie nicht in P sind. Da viele dieser Sprachen Probleme von großer praktischer Bedeutung kodieren, ist die Frage „$\mathrm{P} = \mathrm{NP}$?“ zum zentralen offenen Problem der Komplexitätstheorie, wenn nicht der Theoretischen Informatik geworden, das ständig zu neuen Fragestellungen und Theorien führt. Bevor wir typische Beispiele für Sprachen in NP bringen, untersuchen wir, welche Ergebnisse aus 12.2 sich auf NTMn übertragen lassen.

Wie DTMn können wir NTMn zusammenschalten, ohne die polynomielle Zeitbeschränkung zu verlieren; die Zeitabschätzungen sind dieselben. Aufpassen müssen wir nur wieder beim Hintereinanderschalten. Wenn die erste NTM für keine Berechnung zu einer Eingabe terminiert, kommt die zweite nie zum Zuge. – Wie DTMn können wir NTMn totalisieren, d.h. bei allen Berechnungen terminieren lassen; der Wecker und die Konstruktion sind dieselben. – Wie P ist NP unter Durchschnitt und Vereinigung abgeschlossen; das können wir bei Beweisen bzw. bei NTM-Konstruktionen ausnutzen. Beim Komplement stoßen wir auf den Kern des P-NP-Problems:

Um zu beweisen, daß mit L auch das Komplement $\overline{L}$ zu P gehört, haben wir bei der DTM M, die L in Polynomzeit entscheidet, die akzeptierenden mit den übrigen Zuständen vertauscht; die „komplementäre“ DTM $\overline{M}$ akzeptiert gerade $\overline{L}$. Ist M eine NTM, geht das nicht: Wenn M zu einer Eingabe w eine akzeptierende und eine nichtakzeptierende Berechnung hat, wird w von M und von $\overline{M}$ akzeptiert, also $L(\overline{M}) \neq \overline{L(M)}$. Also müßten wir M deterministisch machen. Aber nach Satz 12.3 hat die simulierende DTM M' exponentiellen und nicht polynomiellen Aufwand, nützt uns also nichts. Könnten wir polynomielle NTMn durch polynomielle DTMn simulieren, wäre $\mathrm{P} = \mathrm{NP}$.

Als erstes Beispiel für eine Sprache in NP, die nicht in P zu sein scheint, betrachten wir das Problem der *Hamiltonkreise* in Graphen. Ein *(gerichteter) Graph* besteht aus „Knoten“ und „Kanten“. Formal sind die Kanten durch eine zweistellige Relation auf den Knoten repräsentiert, anschaulich durch Pfeile zwischen den Knoten. So haben wir endliche Automaten und Turingmaschinen durch Graphen dargestellt, deren Knoten Zustände und deren Kanten Übergänge sind. Bei diesen Graphen sind die Kanten mit Buchstaben oder anderen Angaben versehen, man nennt sie „(kanten)markiert“. Andere Beispiele sind Verkehrsnetze: Städte, Kreuzungen oder sonstige „Knotenpunkte“ als Knoten; (Einbahn-)Straßen oder andere „Verkehrswege“ als Kanten; Entfernungen, Preise oder andere Charakteristika als Markierungen. Ein praktisch wichtiges Problem ist es, in Graphen (insbesondere in markierten) Hamiltonkreise zu finden, das sind Rundwege, die jeden Knoten genau einmal durchlaufen (und deren Markierung maxi- oder mini- oder sonstwie optimal ist). Das Problem heißt traditionell „Problem der Handlungsreisenden“; man kann damit eine Vielzahl von Optimierungsproblemen darstellen. Ein naheliegender Algorithmus ist es, Wege systematisch zu Rundwegen auszubauen und zu prüfen, ob sie Hamiltonkreise (mit optimaler Markierung) sind. Ob ein Weg ein Hamiltonkreis ist, ist einfach zu prüfen, d.h. es geht in polynomieller Zeit. Aber Rundwege gibt es viele: bis zu $n!$ in Graphen mit n Knoten; das ist mehr als polynomiell. Also können wir schon die Existenz von Hamiltonkreisen so nicht in polynomieller Zeit prüfen.

Wenn wir Graphen und Wege durch Wörter über einem geeigneten Alphabet Γ darstellen, erhalten wir als erstes Beispiel für ein „schwieriges“ Problem in NP

HAMILTONKREIS_FINDEN:

Gegeben:	Ein endlicher gerichteter Graph G.
Frage:	Existiert ein Hamiltonkreis in G?

Um zu zeigen, daß dieses Problem in NP ist, betrachten wir das einfachere Problem

HAMILTONKREIS_PRÜFEN:

Gegeben:	Ein endl. gerichteter Graph G und ein Weg W.
Frage:	Ist W ein Hamiltonkreis in G?

Dieses Problem ist in P. Es ist eine gute Übung, eine polynomiell zeitbeschränkte DTM M_P aufzuschreiben, die das Problem löst. Sie muß z.B. die folgenden Teilprobleme nacheinander in polynomieller Zeit prüfen:

- ob in der Eingabe G einen Graphen und W einen Weg repräsentiert,
- ob jede Kante des Weges in G existiert,
- ob alle Knoten von G in diesem Weg enthalten sind,
- ob der Startknoten und der Endknoten gleich sind und
- ob alle anderen Knoten nur einmal in dem Weg enthalten sind.

Jetzt können wir eine NTM M_F beschreiben, die das Problem HAMILTONKREIS_FINDEN löst; sie benutzt die DTM M_P als „Unterprogramm“. In einer ersten Phase produziert M_F nichtdeterministisch ein Wort aus Γ^* bis zu einer vorgegebenen Länge. Danach startet sie M_P, wobei die Eingabe als Graph G und das produzierten Wort als Weg W interpretiert werden. M_P prüft deterministisch in polynomieller Zeit, ob W einen Hamiltonkreis in G repräsentiert. Wenn M_P akzeptiert, dann akzeptiert M_F ebenfalls.

Die „allermeisten“ Wörter aus Γ^* werden noch nicht einmal einen Weg in G repräsentieren. Aber wenn G einen Hamiltonkreis besitzt, dann gibt es ein Wort aus Γ^*, welches den Hamiltonkreis repräsentiert. Da nichtdeterministisch *alle* Wörter produziert werden, wird auch dieses Wort in der ersten Phase irgendeiner Berechnung „geraten“. Also akzeptiert M_F genau dann, wenn G eine Hamiltonkreis besitzt.

Eine DTM könnte nacheinander in lexikographischer Reihenfolge alle Wörter aus Γ^* bis zu einer vorgegebenen Länge erzeugen und prüfen. Sie wäre aber, wie oben argumentiert, nicht mehr polynomiell zeitbeschränkt.

Zum Abschluß zwei weitere Beispiele:

RUCKSACK_PACKEN:

Gegeben:	Ein „Rucksack“ vom „Fassungsvermögen“ $f \in \mathbb{N}$ und k „Pakete“ der „Größe“ $g_1, \ldots, g_k$.
Frage:	Kann man l Pakete mit den Nummern $i_1, \ldots, i_l$ so auswählen, daß sie den Rucksack genau füllen, d.h. $g_{i_1} + \cdots + g_{i_l} = f$?

In dem harmlos aussehenden Problem versteckt sich eine Fülle von z.B. „Stundenplanproblemen“ für Schulen, Fabriken oder Verwaltungen. Wie bei den Hamiltonkreisen ist das Nachprüfen einfach, aber es gibt zu viele mögliche Kombinationen, die man nachprüfen müßte; daher kennt man keine effizienten Verfahren.

ERFÜLLBARKEIT:

Gegeben:	Eine aussagenlogische Formel.
Frage:	Ist sie erfüllbar, d.h. gibt es eine Belegung, die sie wahr macht?

Aussagenlogische Formeln sind Terme (korrekt geklammerte Ausdrücke) mit Verknüpfungen für ‘und’, ‘oder’ und ‘nicht’. Bei einer *Belegung* werden die Variablen durch ‘wahr’ oder ‘falsch’ ersetzt. Wieder ist auswerten einfach, aber für eine Formel mit n Variablen gibt es 2^n Belegungen (warum?).

Alle drei Beispiele haben dieselbe Struktur:

- es ist „leicht“ (in polynomieller Zeit), potentielle Lösungen nachzuprüfen;
- es ist „schwierig“ (bis heute nicht in polynomieller Zeit möglich) Lösungen zu finden, weil es zu viele (mehr als polynomiell viele) Kandidaten gibt.

Alle Probleme in NP, die nicht schon in P sind, haben diese Form.

Man kennt inzwischen über 1000 Probleme, die wie die drei obigen „typisch“ für NP im folgenden Sinn sind: Wenn eins von ihnen in polynomieller Zeit lösbar wäre, dann auch alle anderen. Man nennt solche Probleme *NP-vollständig*. Wer mit Aussagenlogik vertraut ist, kann das für ERFÜLLBARKEIT nachvollziehen: Die Eingaben für die anderen Probleme in aussagenlogische Formeln übersetzen, die erfüllbar sind genau dann, wenn die Antwort ‘ja’ ist. In dem Buch von Schöning finden sich im letzten Abschnitt solche Übersetzungen. Über 300 NP-vollständige Probleme behandeln Garey & Johnson schon 1979 in ihrem Buch „Computers and Intractability“.

Es wird allgemein angenommen, daß mit den heute bekannten Beweistechniken das P-NP-Problem nicht gelöst werden kann. Die Menschen, die praktische Lösungen für „schwierige“ Probleme aus NP finden müssen, haben sich deshalb nach Alternativen umgesehen:

- Meist genügt es, Näherungslösungen zu finden; die lassen sich oft schneller berechnen (z.B. Evolutionsstrategien, Heuristiken).
- Manchmal werden mit Erfolg Algorithmen verwendet, die nicht korrekt sind! Mit einer äußerst geringen Wahrscheinlichkeit liefern sie falsche Ergebnisse. Diese Wahrscheinlichkeit ist sehr viel kleiner als ein Hauptgewinn beim Zahlenlotto.
- Eine andere Möglichkeit ist, Algorithmen zu verwenden, die nur mit hoher Wahrscheinlichkeit eine polynomielle Laufzeit haben. Sie liefern immer das richtige Ergebnis, es kann aber vorkommen, daß sie für einige Eingaben (von denen man hofft, daß sie in der Praxis nicht vorkommen), länger laufen.

12.5 Aufgaben

Aufgabe 12.1: Beweist: Wenn p_1, p_2 Polynome vom Grad k_1, k_2 sind, dann sind $q(n) = p_1(n) + p_2(n)$, $q'(n) = p_1(n) \cdot p_2(n)$ und $q''(n) = p_1(p_2(n))$ Polynome vom Grad $\max(k_1, k_2)$, $k_1 + k_2$ bzw. $k_1 \cdot k_2$. Weiter: $q'''(n) = \max(p_1(n), p_2(n))$ ist i.a. kein Polynom, aber durch ein Polynom abschätzbar. Welchen Grad hat es?

Aufgabe 12.2: Zeigt an Beispielen und dann durch einen allgemeinen Beweis, daß man ein Polynom p in Zeit p in unärer Darstellung berechnen kann.

Aufgabe 12.3: Beweist, daß P unter den Kleeneschen Operationen Verkettung, Potenz und Stern (Abschnitt 1.4) abgeschlossen ist. Beschreibt dazu die nötigen Konstruktionen von DTMn und schätzt deren Rechenzeit ab.

Für den Stern ist der Beweis schwierig. Wir verraten die Idee: Es sei $L \in P$. Ein Wort w ist in L^* genau dann, wenn man es in Teilwörter aus L zerlegen kann: $w = v_1 \dots v_k$ mit $v_1, \dots, v_k \in L$. Also testet man alle Teilwörter von w, ob sie in L liegen; das geht in polynomieller Zeit (warum?). Dann versucht man, aus den Teilwörtern in L eine Kette zu bilden, die ganz w ergibt. Dazu trägt man die Teilwörter in L als Kanten in einen Graphen mit den Positionen $0, 1, \dots, n$ von w ein und sucht in dem Graphen einen Weg von 0 nach n. Dafür gibt es schnelle Algorithmen.

Aufgabe 12.4: Beweist, daß NP unter den Kleeneschen Operationen Verkettung, Potenz und Stern abgeschlossen ist. Die Konstruktionen und Abschätzungen sollten viel einfacher als die für P sein.

Aufgabe 12.5: Definiert den *Zeitaufwand fürs Erzeugen von Sprachen mit Grammatiken* auf geeignete Weise und vergleicht ihn mit dem Zeitaufwand fürs Akzeptieren durch Automaten bzw. Maschinen im regulären, kontextfreien und allgemeinen Fall.

Falls sich ein Wort w aus einer Grammatik G ableiten läßt, so liegt es nahe, als Zeitaufwand des Wortes die Anzahl der Ableitungsschritte zählen. Allerdings kann es sein, daß sich ein Wort auf verschiedene Arten ableiten läßt. Für diesen Fall müßt Ihr definieren, ob die maximale oder die minimale Anzahl an Schritten gezählt werden soll.

Wie geht Ihr mit Determinismus/Nichtdeterminismus um? An dieser Aufgabe merkt Ihr, wie gut Ihr mit den unterschiedlichen Konzepten von TheGI 1 umgehen könnt. Neben den Aufgaben zu den einzelnen Kapiteln ist sie daher das beste Mittel, sich auf das mündliche Prüfungsgespräch vorzubereiten.

13 Literaturhinweise

13.1 Zur Theoretischen Informatik

Zum Abschluß stellen wir noch einige Bücher vor, die als Einführung in die Gebiete Formale Sprachen, Komplexitätstheorie und Algorithmentheorie dienen können. Die Bücher eignen sich natürlich auch für diejenigen, die sich in diesen Richtungen spezialisieren wollen.

Allgemeines:

- Schöning, U.: *Theoretische Informatik kurz gefasst.* BI-Wissenschaftsverlag (2. Aufl.) 1995.

  Eine übersichtliche und knappe Einführung, eignet sich auch zum Nachschlagen. Gut neben dem Skript zu benutzen, da ähnlich aufgebaut.

- Hopcroft, J.E., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley 1979.

  Eine gute Einführung in die Theoretische Informatik, leider ein wenig veraltet und nicht sehr übersichtlich geschrieben. Der Vorteil dieses Buches ist, daß sehr viele Exemplare in der Lehrbuchsammlung stehen und daß es eine (allerdings schlechte) deutsche Übersetzung gibt.

- Wagner, K.: *Einführung in die Theoretische Informatik.* Springer 1994.

  Das neueste Buch zum Thema, sehr ausführlich und sorgfältig. Der Zugang ist anders als bei uns oder Schöning.

- Lewis, H.R., Papadimitriou, C.H.: *Elements of the Theory of Computation.* Prentice-Hall 1981.

  Ausführlicher und übersichtlicher als Hopcroft-Ullman.

- Kozen, D.: *Automata and Computability.* Springer 1997.

  Ein empfehlenswertes Buch, präzis in den Details und dennoch gut lesbar.

- Beutelspacher, A.: *„Das ist o.B.d.A. trivial!“.* Vieweg (2. Aufl.) 1992.

  Die Geheimnisse und Wunder mathematischer Texte werden ausführlich und mit Übungen behandelt. Das ist kein Mathematiklehrbuch, sondern ein Buch über das Verfassen mathematischer Texte.

Formale Sprachen

- Salomaa, A. *Formale Sprachen.* Springer 1978.

  Ebenfalls ein Klassiker. Insbesondere die ersten Kapitel ergänzen gut das Skript.

Algorithmentheorie:

- Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *The Design and Analysis of Computer Algorithms.* Addison–Wesley (2. Aufl.) 1975.
- Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms.* MIT-Press 1990.

  Zwar dick und teuer, aber eine aktuelle und vollständige Einführung in die Theorie effizienter Algorithmen. Dieses Buch ist für alle interessant, denen zur Lösung bestimmter Probleme die üblichen Algorithmen zu langsam sind.

Komplexitätstheorie:

- Balcázar, J. L., Diaz, J., Gabarró, J.: *Structural Complexity 1, 2.* EATCS Monographie 11 und 22, Springer (2. Aufl.) 1995 und 1990.

  "Standardwerke" zur Komplexitätstheorie, sie eignen sich zur Einführung und als Lehrbücher. Wie alles, so sind auch diese Bücher natürlich nicht fehlerlos. Aber da sie relativ aktuell und vollständig sind, sind sie ein "Muß" für alle, die sich vertieft mit der Strukturellen Komplexitätstheorie auseinandersetzen wollen. Die Bücher sind leider recht teuer, aber es gibt einige Exemplare in der Informatikbibliothek.
- Garey, M.R.; Johnson, D.S.: *Computers and Intractability.* Freeman 1979.

  Das Buch für die Theorie der NP-Vollständigkeit. Es enthält eine Liste mit ca. 300 NP-vollständigen Problemen. Außerdem werden die Vollständigkeitsbeweise klassifiziert. Eine gut geschriebene Einführung in die Anfänge der Komplexitätstheorie.

13.2 Zu einer nicht-mathematischen Theorie der Informatik

- Coy, Wolfgang: *Industrieroboter – Zur Archäologie der zweiten Schöpfung.* Rotbuch, 1985.
- Coy, Wolfgang et al.: *Sichtweisen der Informatik.* Vieweg, 1992.
- Dreyfus, Hubert L.: *Die Grenzen künstlicher Intelligenz. Was Computer nicht können.* Athenäum, 1985.
  Engl. Original: *What Computers Can't Do — The Limits of Artificial Intelligence.* Harper (rev. Ed.) 1979.
- Floyd, Christiane et al.: *Software Development and Reality Construction.* Springer, 1992.
- Friedrich, Jürgen, et al. (Hg.): *Informatik und Gesellschaft.* Spektrum, 1995.

- Heintz, Bettina: *Die Herrschaft der Regel – Zur Grundlagengeschichte des Computers.* Campus, 1993.
- Hoffmann, Ute: *Computerfrauen.* Rainer Hampp, 1987.
- Hofstadter, Douglas R.: *Gödel, Escher, Bach – ein endloses geflochtenes Band.* Klett-Kotta, 1984; dtv 1991.
  Engl. Original: *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books 1979.
- Keller, Evelyn F.: *Liebe, Macht und Erkenntnis. Männliche oder weibliche Wissenschaft?* Carl Hauser, 1986.
- Pirsig, Robert M.: *Zen und die Kunst ein Motorrad zu warten – Ein Versuch über Werte.* Fischer, 1978.
  Engl. Original: *Zen and the Art of Motorcycle Maintenance: an Inquiry into Values.* Bantam Books 1974.
- Siefkes, Dirk: *Formalisieren und Beweisen. Logik für Informatiker.* Vieweg, 2. Auflage 1992.
- Siefkes, Dirk: *Formale Methoden und kleine Systeme – Lernen, leben und arbeiten in formalen Umgebungen.* Vieweg, 1992.
- Siefkes, Dirk, et al. (Hg.): *Sozialgeschichte der Informatik — Soziale Praktiken und Orientierungen.* Deutscher Universitätsverlag 1998.
- Siefkes, Dirk, et al. (Hg.): *Pioniere der Informatik. Interviews mit E.L. Bauer, C. Floyd, J. Weizenbaum, N. Wirth, H. Zemanek.* Springer 1999.
- Turing, Alan: *Intelligence Service – Schriften.* B. Dotzler, F. Kittler (Hg.), Brinkmann & Bose, 1987.
- Weizenbaum, Joseph: *Die Macht der Computer und die Ohnmacht der Vernunft.* Suhrkamp, 1982.
  Engl. Original: *Computer Power and Human Reason. From Judgement to Calculation.* W.H. Freeman 1976.

14 Zum Skript

Am Skript haben vor allem Hans-Jörg Burtschick, Dieter Hofbauer und Dirk Siefkes gearbeitet, Ergänzungen stammen von Lars With, Arfst Nickelsen, Birgit Schelm und Till Tantau. Das Tippen besorgten Moni Schmid, Ralf Klaus, Hannelore Pribbenow und zuletzt Dieter Hofbauer, Birgit Schelm und Till Tantau.

A Projektaufgaben

Die folgenden Aufgaben bearbeitet Ihr jeweils über einen größeren Zeitraum. Deshalb ist es nötig, daß Ihr Eurer Tutorin oder Eurem Tutor jede Woche vorstellt, was Ihr schon erarbeitet habt. Nur so können wir Euch bei den Lösungen sinnvoll unterstützen, indem wir Euch von Irrwegen abhalten, auf Fehler und Lücken hinweisen oder auch nur im richtigen Vorgehen bestätigen. Vor allem lernt Ihr nur so, die mündliche und schriftliche Darstellung aufeinander abzustimmen. Dies ist nötig, weil als Ergebnis Eurer Arbeit Texte entstehen sollen, die für sich allein lesbar und verstehbar sind. Damit meinen wir, daß diese Texte auch für die Leute verständlich sein sollen, die den Diskussionsprozeß zwischen Euch und den TutorInnen nicht mitgemacht haben.

Am Ende jeder Projektaufgabe findet Ihr Terminempfehlungen. Wir raten Euch *dringend* davon ab, die Aufgaben „erst auf den letzten Drücker“ zu machen. Typischerweise verschätzt man sich sehr leicht, was den Aufwand und den Umfang der Lösung der Projektaufgaben angeht. Die Terminempfehlungen helfen Euch, die Arbeit in machbare Päckchen zu schnüren.

1. Projektaufgabe (Abgabe 23. – 24. 11. 2000)

Aufgabe

Beweist, daß die Klasse der regulären Sprachen unter den Operationen $\circ, \cup$, Komplement und $\star$ abgeschlossen ist, d.h. wenn zwei Sprachen L_1 und L_2 mit $L_1, L_2 \subseteq \Sigma^\star$ regulär sind, so auch $L_1 \circ L_2, L_1 \cup L_2, \overline{L_1}$ und $L_1^\star$.

Hinweise:

Ihr lernt gerade, wie reguläre Sprachen durch reguläre Grammatiken erzeugt und durch endliche Automaten akzeptiert werden; also müßt Ihr aus gegebenen („gedachten“) Grammatiken oder Automaten neue konstruieren. Für das Komplement müßt Ihr die Ergebnisse im Skript über deterministische Automaten benutzen. Beweist, daß eure Konstruktionen korrekt sind.

Terminempfehlungen:

Bis 9. 11. Konstruktion und Korrektheitsbeweis für $\circ$ und $\cup$.

Bis 16. 11. Konstruktion und Korrektheitsbeweis für das Komplement.

Bis 23. 11. Konstruktion und Korrektheitsbeweis für $\star$.

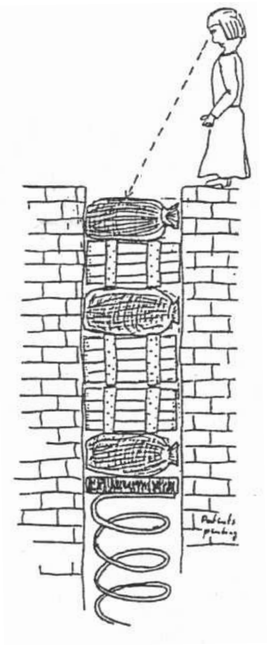
2. Projektaufgabe (Abgabe 21. – 22. 12. 2000)

Ihr lernt in dieser LV, wie man Sprachklassen durch Maschinentypen charakterisiert:

- Eine Sprache ist regulär genau dann, wenn es einen endlichen Automaten gibt, der sie akzeptiert (Skript Kapitel 2 – 5).
- Eine Sprache ist erzeugbar genau dann, wenn es eine Turingmaschine gibt, die sie akzeptiert (Skript Kapitel 7 – 9).

Da jede kontextfreie Sprache erzeugbar ist, wird sie auch von einer Turingmaschine akzeptiert. Die muß jedoch nicht in beliebiger Weise auf Daten zugreifen, die auf das Arbeitsband geschrieben wurden. Es reicht aus, die Eingabe einmal zu lesen und zum Speichern einen *Keller* zu verwenden.

Ein Keller ist eine spezielle Datenstruktur, in der man Daten speichern kann. Der Zugriff auf gespeicherte Daten ist immer nur auf das Element möglich, das zuletzt gespeichert wurde. Als gutes Bild dafür kann man sich einen Kellerschacht vorstellen, der so schmal ist, daß nicht zwei Sachen nebeneinander passen. Die Sachen, die in den Kellerschacht hineingetan werden, liegen dann übereinander. Jetzt können zwar in beliebiger Abfolge Sachen hineingetan und wieder herausgeholt werden, aber nur das oberste Teil ist jeweils sichtbar. Man bekommt also in umgekehrter Reihenfolge wieder heraus, was man hineingetan hat, und auf das herausgeholte kann man nicht mehr zurückgreifen.



Nichtdeterministische Turingmaschinen mit einem Arbeitsband, die die Eingabe nur von links nach rechts lesen und das Arbeitsband nur als Keller benutzen können, heißen **Keller-Turingmaschinen** mit **Einwegeingabe** oder kurz **Kellermaschinen**.

Aufgabe

1. Definiert Kellermaschinen als spezielle nichtdeterministische Turingmaschinen.

2. Konstruiert für die kontextfreie Sprache $L = \{a^n b^m c^k \mid n, m, k \in \mathbb{N}^+ \text{ und } (n = m \text{ oder } n = k)\}$ eine Kellermaschine M, die L akzeptiert. Gebt *alle* Berechnungen von M für die Eingaben *aabcc* und *aabc* an. Beschreibt M durch einen Zustandsgraphen, erläutert die Arbeitsweise von M und beweist, daß $L = L(M)$. Wo braucht Ihr den Nichtdeterminismus?

3. Konstruiert zu einer beliebigen kontextfreien Grammatik G eine Kellermaschine M_G, die $L(G)$ akzeptiert. Veranschaulicht wieder Eure Konstruktion und beweist, daß $L(G) = L(M_G)$. Überprüft Eure Maschine an Beispielen (z.B. der Sprache aus Teil 2) und Spezialfällen (z.B. dem leeren Wort).

Hinweise:

Die Teile 1 und 2 behandelt Ihr am besten abwechselnd: Macht Euch zuerst Kellermaschinen grob klar; erarbeitet dazu einen Algorithmus für L; präzisiert damit die Definition von Kellermaschinen; verfeinert damit die Konstruktion in Teil 2 und führt den Beweis.

Zu 1. Ihr müßt Entwurfsentscheidungen treffen. Schreibt die Maschine im Keller nach links oder nach rechts? Was passiert etwa, wenn der Keller leer wird? Wenn Ihr Kellermaschinen aus anderen Zusammenhängen kennt, könnt Ihr sie „nachbauen" (pop, push, top „simulieren"). Es muß aber eine Turingmaschine dabei herauskommen. Für die Darstellung bitte nicht die Definitionen im Skript wiederholen, sondern nur die notwendigen Einschränkungen angeben.

Zu 2. Ihr sollt keine Grammatik für L verwenden – das könnt Ihr in Teil 3 als Beispiel machen –, sondern M direkt konstruieren.

Zu 3. G ist beliebig. Ihr müßt also M_G konstruieren, ohne G zu kennen – wie in den Beweisen von Abschnitt 9.3 im Skript. Vereinfacht die Beweisidee des ersten Satzes dort für kontextfreie Grammatiken: Während M_G eine Linksableitung (Abschnitt 6.3) in den Keller „rät", kann sie jeweils die terminalen Symbole am linken Ende der geratenen Satzform mit dem linken Ende der Eingabe vergleichen und so die Eingabe von links nach rechts abarbeiten. Macht das Euch (und uns) klar, indem Ihr zunächst eine Kellermaschine konstruiert, mit der Ihr Ableitungen für eine kontextfreie Grammatik, die Ihr selbst auswählt, so nichtdeterministisch erzeugt und überprüft. Konstruiert dann M_G für eine allgemeine kontextfreie Grammatik.

Terminempfehlungen:

Bis 7. 12. Teile 1 und 2.

Bis 21. 12. Teil 3.

3. Projektaufgabe (Abgabe 23. – 24. 1. 2001)

Die Menge aller Wörter über einem Alphabet Σ ist abzählbar. Damit ist auch jede Sprache $L \subseteq \Sigma^\star$ abzählbar. Ein Sprache $L \subseteq \Sigma^\star$ heißt (*effektiv*) *aufzählbar*, wenn man ihre Elemente auf berechenbare Weise aufzählen kann. Das heißt, man kann L schreiben als $L = \{w_0, w_1, w_2, \dots\}$, wobei man die w_i in dieser Reihenfolge berechnen kann. Die Reihenfolge ist beliebig, Wiederholungen sind erlaubt. Die Aufzählung ist *ohne Wiederholungen*, wenn $w_i \neq w_j$ für $i \neq j$. Sie ist *geordnet*, wenn die w_i in Standardreihenfolge kommen, d.h. der Länge nach geordnet, bei gleicher Länge lexikographisch.

Aufgabe

Entwickelt eine geeignete Form, Sprachen durch Turingmaschinen aufzuzählen. Dazu definiert Ihr *Turing-Aufzähler.* Das sind spezielle DTMs M_{auf} ohne Eingabe, die $w_0, w_1, w_2, \dots$ nacheinander auf ein Ausgabeband schreiben, also im allgemeinen nicht stoppen. Bezeichnet die von solchen Turingmaschinen aufgezählten Sprachen mit $L(M_{\text{auf}})$ und nennt solche Sprachen *Turing-aufzählbar.*

Beweist für unendliche Sprachen:

1. Wenn L Turing-aufzählbar ist, dann ist L akzeptierbar.

2. Ist L Turing-aufzählbar, so gibt es auch einen Turing-Aufzähler für L, der ohne Wiederholungen auskommt.

3. L ist geordnet Turing-aufzählbar gdw L ist entscheidbar.

4. Was passiert bei Eurer Definition, Euren Konstruktionen und Beweisen mit endlichen Sprachen?

Hinweise:

Wie ein Turing-Aufzähler genau arbeiten soll, wird Euch am besten klar, wenn Ihr an den Beweisen bastelt. Eure schriftliche Darstellung muß dann aber mit einer Definition von Turing-Aufzählern anfangen.

Zu 1. Wenn L Turing-aufzählbar ist, gibt es eine DTM M_{auf}, die L aufzählt. Daraus könnt Ihr eine DTM M konstruieren, die L akzeptiert. Dazu benutzt Ihr M_{auf} als Teilmaschine. Macht Euch zuerst die unterschiedlichen Maschinentypen klar: Welche Bänder wofür? Stellt dann in einem Diagramm dar, wie M arbeitet, und kommentiert die Konstruktion. Beweist schließlich $L(M_{\text{auf}}) = L(M)$.

Zu 2. Konstruiert aus einem beliebigen Turing-Aufzähler einen Turing-Aufzähler, der kein Wort mehrfach ausgibt.

Zu 3. Wenn M_{auf} die Sprache L geordnet Turing-aufzählt, könnt Ihr aus dem Akzeptierer M für L aus Teil 1 einen Entscheider machen: Da M_{auf} die Wörter aus L der Größe nach aufzählt, „weiß“ M irgendwann, daß das gesuchte Wort nicht mehr kommen kann. Für die umgekehrte Richtung konstruiert Ihr aus einem Entscheider M für L einen Aufzähler, der der Größe nach aufzählt. Die Idee ist, zuerst einen Aufzähler N_{auf} für $\Sigma^\star$ zu konstruieren und dessen Ausgaben der Reihe nach von M testen zu lassen. M_{auf} gibt nur die Wörter aus, die M akzeptiert. Beschreibt die Funktionsweise von N_{auf}.

Terminempfehlungen:

Bis 11. 1. Definition von Turing-Aufzähler und Teile 1 und 2.

Bis 18. 1. Teile 3 und 4.

4. Projektaufgabe (Abgabe 6. – 7. 2. 2001)

In dieser Aufgabe sollt Ihr zeigen, daß das Auswerten von *Schaltkreisen* in Polynomialzeit möglich ist.

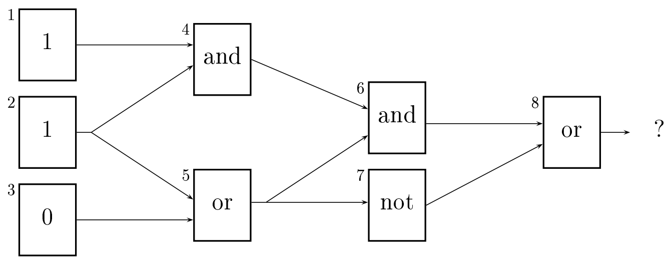
Ein Schaltkreis besteht aus einer Reihe von durchnumerierten *Gattern*. Es gibt fünf Arten von Gattern: Null-Gatter, Eins-Gatter, Nicht-Gatter, Und-Gatter und Oder-Gatter. Gatter haben Eingänge und Ausgänge und sind untereinander durch Stromleitungen verbunden. Eine Leitung führt immer von einem der Ausgänge eines Gatters zum Eingang eines anderen Gatters.

Alle Gatter haben einen Ausgang, aber je nach Art des Gatters null, ein oder zwei Eingänge. Von einem Ausgang können beliebig viele Stromleitungen ausgehen. Natürlich verhalten sich die Gatter recht unterschiedlich:

1. Ein Null-Gatter hat keine Eingänge. Aus diesem Gatter fließt nie Strom.
2. Ein Eins-Gatter hat auch keine Eingänge, aber von diesem Gatter fließt Strom zu allen Gattern, die mit dem Eins-Gatter verbunden sind.
3. Ein Nicht-Gatter hat genau einen Eingang. Aus dem Gatter fließt genau dann Strom, wenn kein Strom am Eingang hereinkommt.
4. Ein Oder-Gatter hat zwei Eingänge, und aus dem Gatter fließt Strom, falls an einem der Eingänge Strom hereinkommt.
5. Ein Und-Gatter hat auch zwei Eingänge, und aus dem Gatter fließt Strom, falls an beiden Eingängen Strom hereinkommt.

Schaltkreise haben noch eine besondere Eigenschaft: Alle Eingänge aller Gatter sind immer mit dem Ausgang eines Gatters mit kleinerer Nummer verbunden. Aus diesem Grund enthält ein Schaltkreis keine Zyklen. Weiter folgt auch, daß kein Ausgang des Gatters mit der größten Nummer mit irgendwelchen anderen Gattern verbunden ist.

Uns interessiert nun, ob aus dem Gatter mit der höchsten Nummer Strom fließt, ob also ein Lämpchen anginge, wenn wir es an dieses Gatter anschließen würden.



Aufgabe

Das Schaltkreisauswertungsproblem (CVP für *Circuit Value Problem*) ist eine Sprache, die Kodes von Schaltkreisen enthält. Sie enthält genau die Kodes derjenigen Schaltkreise, bei denen aus dem Gatter mit der größten Nummer Strom fließt.

1. Überlegt Euch eine geeignete Kodierung für Schaltkreise, um sie als Eingabe für eine Turingmaschine verwenden zu können. Wie lang wird Euer Kode eines Schaltkreises höchstens, wenn der Schaltkreis aus n Gattern besteht?
2. Gebt eine deterministische Turingmaschine an, die CVP entscheidet.
3. Beweist CVP $\in$ P, indem ihr zeigt, daß der Zeitaufwand Eurer Turingmaschine polynomiell ist.

Hinweise:

Zu 1. Es muß sich leicht überprüfen lassen, ob ein beliebiges Wort überhaupt einen Schaltkreis kodiert. Es muß sich also überprüfen lassen, ob ein Wort ein syntaktisch korrekter Kode ist. Überlegt, wie aufwendig dieser Test ist.

Zu 2. Ihr müßt keinen Zustandsgraphen einer DTM angeben, sondern die Funktionsweise der DTM beschreiben.

Zu 3. Ihr müßt ein Polynom p finden, so daß Eure DTM bei Eingabe eines Wortes w höchstens $p(|w|)$ viele Schritte macht. Wichtig ist, daß ihr nicht genau die Rechenzeit der Maschine bestimmen, sondern lediglich eine obere Schranke finden müßt. Falls Eure Maschine beispielsweise höchstens $|w|$ mal etwas tut, was jeweils höchstens $|w|$ lange dauert, so würde die Maschine höchstens $|w|^2$ viele Schritte machen und das gesuchte Polynom wäre $p(n) = n^2$.

Terminempfehlungen:

Bis 25. 1. Teile 1 und 2.

Bis 1. 2. Teile 3.

B Griechische Buchstaben

Buchstabe	Name	Buchstabe	Name
A, α	Alpha	N, ν	Ny
B, β	Beta	Ξ, ξ	Xi
Γ, γ	Gamma	O,o	Omikron
Δ, δ	Delta	Π, π	Pi
E, ϵ, ε	Epsilon	P, ρ	Rho
Z, ζ	Zeta	Σ, σ	Sigma
H, η	Eta	T, τ	Tau
Θ, θ	Theta	Υ, υ	Ypsilon
I, ι	Jota	Φ, ϕ, φ	Phi
K, κ	Kappa	X, χ	Chi
Λ, λ	Lambda	Ψ, ψ	Psi
M, μ	My	Ω, ω	Omega