Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie

John E. Hopcroft
Rajeev Motwani
Jeffrey D. Ullman

Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie

2., überarbeitete Auflage

ein Imprint der Pearson Education Deutschland GmbH

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist
bei der Deutschen Bibliothek erhältlich.

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1
05 04 03 02

ISBN 3-8273-7020-5

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Str. 10-12, D- 81829 München/Germany

www.pearson-studium.de
Übersetzung: Sigrid Richter und Ingrid Tokar für transit, München
Lektorat: Helge Sturmfels, hsturmfels@pearson.de
Fachlektorat: Prof. Dr. Manfred Paul, Technische Universität München
Korrektorat: Margret Neuhoff, München
Einbandgestaltung: Julia Graff, DYADEsign, Düsseldorf
Titelabbildung: Image Bank, Düsseldorf
Herstellung: Anna Plenk, aplenk@pearson.de
Satz: text&form GbR, Fürstenfeldbruck
Druck und Verarbeitung: Kösel, Kempten (www.koeselbuch.de)
Printed in Germany

Inhaltsverzeichnis

Vorwort

Im Vorwort des 1979 erschienenen Vorgängers dieses Buches wunderten sich Hopcroft und Ullman über die Tatsache, dass das Interesse an dem Thema Automaten gegenüber dem Zeitpunkt, zu dem sie ihr erstes Buch verfassten (1969), förmlich explodiert war. In der Tat enthielt das Buch von 1979 viele Themen, die in dem früheren Werk nicht vertreten waren, und es war etwa doppelt so dick. Wenn man das vorliegende Buch mit dem 1979 erschienenen vergleicht, stellt man fest, dass dieses Buch wie die Autos in den Siebzigern »außen größer, aber innen kleiner« ist. Das klingt nach einem Rückschritt, aber wir sind aus verschiedenen Gründen mit diesen Änderungen zufrieden.

Erstens war die Automaten- und Sprachtheorie 1979 noch ein Bereich aktiver Forschung. Dieses Buch sollte unter anderem dazu dienen, an der Mathematik interessierte Studenten dazu zu ermutigen, zu diesem Feld beizutragen. Heute gibt es kaum Forschungsprojekte, die sich direkt mit der Automatentheorie befassen (im Gegensatz zu deren Anwendungen), und daher besteht für uns kaum Veranlassung, den knappen, sehr mathematischen Stil des 1979 erschienenen Buches beizubehalten.

Zweitens hat sich die Rolle der Automaten- und Sprachtheorie in den letzten beiden Jahrzehnten geändert. 1979 war die Automatentheorie größtenteils ein fortgeschrittenes Thema für Studenten höherer Semester, und wir nahmen an, unsere Leser seien Studenten fortgeschrittener Semester, insbesondere die Leser der hinteren Kapitel des Buches. Heute gehört dieses Thema zum Curriculum der Anfangssemester. Daher darf der Inhalt dieses Buches weniger Vorkenntnisse seitens der Studenten voraussetzen und muss mehr Hintergrundinformationen und Argumentationsdetails bieten als das frühere Werk.

Eine dritte Änderung besteht darin, dass die Informatik in den letzten beiden Jahrzehnten in einem fast unvorstellbaren Maß gewachsen ist. Während es 1979 häufig eine Herausforderung darstellte, das Curriculum mit Material zu füllen, das unserer Einschätzung nach die nächste technologische Welle überdauern würde, streiten sich heute viele Teildisziplinen um einen Platz in dem begrenzten Raum des Curriculums zum Vordiplom.

Viertens wurde die Informatik zu einem stärker berufsorientierten Bereich, und bei vielen Studenten ist ein starker Pragmatismus feststellbar. Wir glauben weiterhin, dass bestimmte Aspekte der Automatentheorie grundlegende Werkzeuge für verschiedene neue Disziplinen sind und dass die in typischen Kursen zum Thema Automatentheorie enthaltenen theoretischen, bewusstseinserweiternden Übungen nach wie vor ihren Wert haben, ganz gleich, wie sehr die Studenten es auch bevorzugen

mögen, sich mit sofort monetär umsetzbarer Technologie zu befassen. Um dem Thema allerdings einen dauerhaften Platz auf der Themenliste von Informatikstudenten sicherzustellen, sind wir der Meinung, dass es notwendig ist, die Anwendungen neben der Mathematik stärker hervorzuheben. Daher haben wir eine Reihe der eher abstrakten Themen des früheren Buches durch Beispiele ersetzt, die zeigen, wie diese Ideen heute Anwendung finden. Wenngleich die Anwendungen der Automaten- und Sprachtheorie im Compilerbau jetzt so bekannt sind, dass sie normalerweise in Kursen zum Thema Compiler behandelt werden, gibt es verschiedene neuere Anwendungen wie Algorithmen zur Überprüfung von Modellen. Diese werden zur Verifizierung von Protokollen und Dokumentbeschreibungssprachen, die auf dem Konzept kontextfreier Grammatiken basieren, eingesetzt.

Die letzte Erklärung für die simultane Vergrößerung und Verkleinerung dieses Buches besteht darin, dass wir heute die Vorteile der von Don Knuth und Les Lamport entwickelten Satzsysteme TEX und LATEX nutzen können. Insbesondere LATEX ermutigt zu einem »offenen« Satzstil, der Bücher zwar umfangreicher, aber einfacher lesbar macht. Wir schätzen die Arbeit beider Herren.

Verwendung des Buches

Dieses Buch eignet sich für einen einsemestrigen Kurs für Anfangssemester oder später. An der Universität von Stanford haben wir dieses Material in dem Kurs CS154, einem Kurs zum Thema Automaten- und Sprachtheorie, verwendet. Es handelt sich um einen halbsemestrigen Kurs, der sowohl von Rajeev Motwani als auch von Jeff Ullman abgehalten wurde. Auf Grund der zeitlichen Beschränkung wird in diesem Kurs Kapitel 11 nicht behandelt und einige Materialien späterer Kapitel, wie z. B. die schwierigere Polynom-Zeit-Reduktion aus Abschnitt 10.4, werden ebenfalls weggelassen. Die Website zu diesem Buch (siehe unten) enthält Aufzeichnungen und Studienpläne für verschiedene Angebote des Kurses CS154.

Vor einigen Jahren stellten wir fest, dass viele Studenten höherer Semester von anderen Universitäten kamen und Kurse in Automatentheorie absolviert hatten, die auf die Theorie der Nichthandhabbarkeit (engl. *Intractability*) nicht eingingen. Da der Lehrkörper der Universität von Stanford der Auffassung ist, dass jeder Informatiker diese Vorstellungen zu einem Grad kennen muss, der über die Ebene der Aussage »NP-vollständig bedeutet, dass es zu lange dauert« hinausgeht, wird ein weiterer Kurs (CS154N) angeboten, der nur die Kapitel 8, 9 und 10 behandelt. Studenten, die diesen Kurs belegen, nehmen in etwa am letzten Drittel von CS154 teil und erfüllen damit die Anforderungen des Kurses CS154N. Auch heute noch gibt es in jedem Semester Studenten, die diese Option nutzen. Wir empfehlen dieses Vorgehen, da es wenig Zusatzaufwand erfordert.

Voraussetzungen

Um dieses Buch am besten erschließen zu können, sollten die Studenten bereits einen Kurs in diskreter Mathematik belegt haben, in dem z. B. Graphen, Bäume, Logik und Beweistechniken behandelt wurden. Wir setzen zudem voraus, dass die Studenten mehrere Programmierkurse absolviert haben und mit üblichen Datenstrukturen, Rekursion und der Rolle der wichtigsten Systemkomponenten wie Compilern vertraut sind. Diese Voraussetzungen sollten in den ersten Semestern eines typischen Informatikstudiengangs erworben werden.

Übungen

Das Buch enthält ausgiebig Übungen zu fast jedem Abschnitt. Wir zeichnen schwierigere Übungen oder Teile von Übungen durch ein Ausrufezeichen aus. Die schwierigsten Übungen sind durch zwei Ausrufezeichen gekennzeichnet.

Einige Übungen oder Teile davon sind durch Sternchen markiert. Für diese Übungen werden wir Lösungen auf der Website des englischsprachigen Orignalbuches bereitstellen. Diese Lösungen sind öffentlich zugänglich und sollten zur Überprüfung der eigenen Arbeitsergebnisse verwendet werden. Beachten Sie, dass in einigen Fällen eine Übung *B* die Modifikation oder Anpassung Ihrer Lösung zu Übung *A* erfordert. Falls zu bestimmten Teilen von *A* Lösungen existieren, dann können Sie davon ausgehen, dass es auch für die entsprechenden Teile von *B* Lösungen gibt.

Unterstützung im World Wide Web

Die Adresse der Website des englischsprachigen Originalbuches lautet:

http://www-db.stanford.edu/~ullman/ialc.html

Sie finden dort Lösungen zu den mit Sternen ausgezeichneten Übungen, Errata und unterstützende Materialien. Wir hoffen, dass wir die Skripte zum Kurs CS154, einschließlich Hausarbeiten, Lösungen und Prüfungen, auf der Website zur Verfügung stellen können.

Zusätzlich wurde für die deutschsprachige Ausgabe des Buches eine Website unter *http://www.pearson-studium.de* eingerichtet.

Danksagung

Ein Skript von Craig Silverstein mit dem Titel »how to do proofs« (wie man Beweise anlegt) hatte Einfluss auf einen Teil des in Kapitel 1 dargelegten Materials. Kommentare und Errata zu Entwürfen dieses Buches gingen ein von: Zoe Abrams, George Candea, Haowen Chen, Byong-Gun Chun, Jeffrey Shallit, Bret Taylor, Jason Townsend und Erik Uzureau. Wir danken hiermit dafür. Für noch vorhandene Fehler sind natürlich wir allein verantwortlich.

J.E.H.
R.M.
J.D.U.

Ithaca NY und Stanford CA

September, 2000

KAPITEL 1

Automaten: Die Methoden und der Wahnsinn

Mit dem Begriff »Automatentheorie« ist das Studium abstrakter Rechengeräte oder »Maschinen« gemeint. In den 30-er Jahren, als es noch keine Computer gab, studierte A. Turing eine abstrakte Maschine, die über sämtliche Fähigkeiten heutiger Computer verfügte, zumindest was deren Rechenleistung betraf. Turing hatte zum Ziel, die Grenze zwischen dem, was eine Rechenmaschine berechnen kann, und dem, was sie nicht berechnen kann, genau zu beschreiben. Seine Schlussfolgerungen treffen nicht nur auf seine abstrakten *Turing-Maschinen* zu, sondern auch auf die heutigen realen Maschinen.

In den vierziger und fünfziger Jahren wurden von einigen Forschern einfachere Maschinen untersucht, die heute als »endliche Automaten« bezeichnet werden. Diese Automaten, die ursprünglich zur Simulation von Gehirnfunktionen vorgesehen waren und auf die wir in Abschnitt 1.1 eingehen werden, haben sich für verschiedene andere Zwecke als außerordentlich nützlich erwiesen. In den späten fünfziger Jahren begann zudem der Linguist N. Chomsky, formale »Grammatiken« zu studieren. Diese Grammatiken sind zwar streng genommen keine Maschinen, weisen aber eine enge Verwandtschaft zu abstrakten Automaten auf und dienen heute als Grundlage einiger wichtiger Softwarekomponenten, zu denen auch Teile von Compilern gehören.

1969 führte S. Cook Turings Untersuchung der Frage fort, was berechnet werden kann und was nicht. Cook konnte die Probleme, die sich effizient mit Computern lösen lassen, von jenen Problemen trennen, die prinzipiell lösbar sind, deren Lösung jedoch in der Praxis so zeitaufwändig ist, dass Computer sich lediglich zur Lösung sehr kleiner Beispiele solcher Probleme eignen. Diese Klasse von Problemen wird als »nicht handhabbar« (englisch: intractable) oder »NP-hart« bezeichnet. Es ist sehr unwahrscheinlich, dass die exponentielle Steigerung der Rechengeschwindigkeit, die bei der Computerhardware erzielt worden ist (»Moores Gesetz«), sich bemerkenswert auf unsere Fähigkeit auswirken wird, umfangreiche Beispiele solcher nicht handhabbaren Probleme berechnen zu können.

Alle diese theoretischen Entwicklungen wirken sich direkt auf die Tätigkeit der Informatiker von heute aus. Einige dieser Konzepte, wie endliche Automaten und bestimmte Arten formaler Grammatiken, werden im Design und im Aufbau wichtiger Arten von Software verwendet. Andere Konzepte, wie Turing-Maschinen, helfen uns verstehen, was wir von unserer Software erwarten können. Insbesondere aus der Theorie der nicht handhabbaren Probleme können wir ableiten, ob es wahrscheinlich ist,

dass wir ein Problem direkt angehen und ein Programm zu seiner Lösung schreiben können (weil es nicht zur Klasse der nicht handhabbaren Probleme gehört), oder ob wir eine Möglichkeit finden müssen, das nicht handhabbare Problem zu umgehen: durch eine Approximation, die Verwendung einer Heuristik oder durch eine andere Methode, um die Menge an Zeit zu beschränken, die das Programm zur Lösung des Problems aufwendet.

In diesem einführenden Kapitel beginnen wir mit einer sehr abstrakten Betrachtung des Gegenstands der Automatentheorie und ihrer Anwendungen. Ein Großteil des Kapitels ist der Untersuchung von Beweistechniken und Tricks zur Entdeckung von Beweisen gewidmet. Wir gehen auf deduktive Beweise, Umformulierungen, Beweise durch Widerspruch, Beweise durch Induktion und andere wichtige Konzepte ein. Im letzten Abschnitt werden Konzepte vorgestellt, die in der Automatentheorie von zentraler Bedeutung sind: Alphabete, Zeichenreihen und Sprachen.

1.1 Wozu dient das Studium der Automatentheorie?

Es gibt verschiedene Gründe, warum das Studium von Automaten und Komplexität einen wichtigen Teil des Kerns der Informatik bildet. Dieser Abschnitt stellt eine Einführung in die grundlegende Motivation dar und skizziert überdies die Hauptthemen, die in diesem Buch behandelt werden.

1.1.1 Einführung in endliche Automaten

Endliche Automaten sind ein nützliches Modell für viele wichtige Arten von Hardware und Software. Wir werden in Kapitel 2 und nachfolgenden Kapiteln Beispiele dafür zeigen, wie die Konzepte verwendet werden. Für den Augenblick soll es genügen, einige der wichtigsten Arten aufzulisten:

1. Software zum Entwurf und zur Überprüfung des Verhaltens digitaler Schaltkreise.
2. Die »lexikalische Analysekomponente« von typischen Compilern, d. h. die Compilerkomponente, die den Eingabetext in logische Einheiten aufschlüsselt, wie z. B. Bezeichner, Schlüsselwörter und Satzzeichen.
3. Software zum Durchsuchen umfangreicher Texte, wie Sammlungen von Webseiten, um Vorkommen von Wörtern, Ausdrücken oder anderer Muster zu finden.
4. Software zur Verifizierung aller Arten von Systemen, die eine endliche Anzahl verschiedener Zustände besitzen, wie Kommunikationsprotokolle oder Protokolle zum sicheren Datenaustausch.

Obwohl wir bald die präzise Definition von Automaten unterschiedlicher Typen vorstellen, wollen wir unsere informelle Einführung mit einer kurzen Beschreibung dessen beginnen, was ein endlicher Automat ist und tut. Es gibt viele Systeme oder Komponenten, wie die oben aufgezählten, die so betrachtet werden können, dass sie sich zu jedem gegebenen Zeitpunkt in einem von einer endlichen Anzahl von »Zuständen« befinden. Zweck eines Zustands ist es, den relevanten Teil der Geschichte eines Systems festzuhalten. Da es lediglich eine endliche Anzahl von Zuständen gibt, kann im Allgemeinen nicht die gesamte Geschichte beschrieben werden. Das System muss daher sorgfältig entworfen werden, sodass das Wichtige beschrieben und gespeichert

und das Unwichtige vergessen wird. Lediglich mit einer endlichen Anzahl von Zuständen zu arbeiten, bietet den Vorteil, dass wir das System mit einer fixen Menge von Ressourcen implementieren können. Wir können es beispielsweise als Schaltkreis in Hardware oder als einfaches Programm implementieren, das Entscheidungen anhand einer begrenzten Menge von Daten oder basierend auf der Position der Anweisung im Code trifft.

Beispiel 1.1 Ein Kippschalter ist der vielleicht einfachste nicht triviale endliche Automat. Dieses Gerät weiß, wann es sich im Zustand *Ein* oder *Aus* befindet, und es ermöglicht dem Benutzer, einen Schalter zu drücken, der, abhängig vom Zustand des Kippschalters, eine unterschiedliche Wirkung hat. Wenn sich der Kippschalter im Zustand *Aus* befindet, dann wird er durch das Drücken des Schalters in den Zustand *Ein* versetzt, und wenn sich der Kippschalter im Zustand *Ein* befindet, dann wird er durch das Drücken des Schalters in den Zustand *Aus* versetzt.

Abbildung 1.1: Als endlicher Automat dargestellter Kippschalter

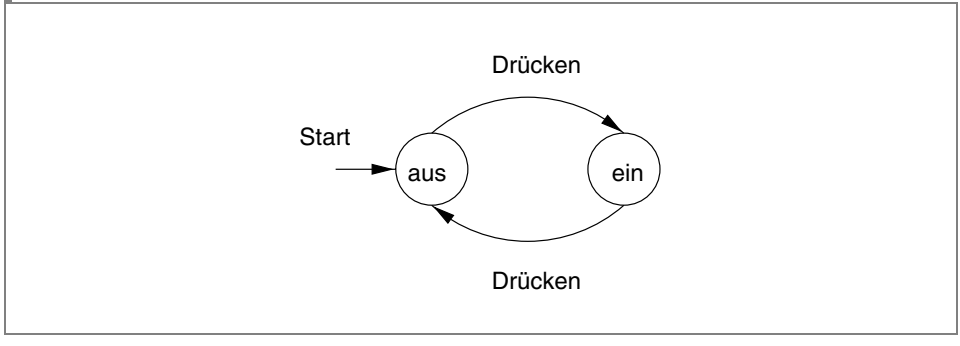


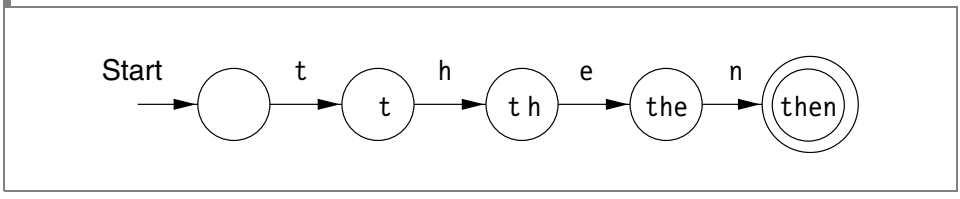
Abbildung 1.1 zeigt die Darstellung des Kippschalters als endlichen Automaten. Wie bei allen endlichen Automaten werden die Zustände durch Kreise repräsentiert. In diesem Beispiel haben wir die Zustände *Ein* und *Aus* genannt. Die Pfeile zwischen den Zuständen werden mit den »Eingaben« beschriftet, die die auf das System wirkenden externen Einflüsse darstellen. Hier sind beide Pfeile mit der Eingabe *Drücken* beschriftet, die für das Drücken des Schalters durch den Benutzer steht. Die beiden Pfeile zeigen an, dass das System beim Empfang der Eingabe *Drücken* in den jeweils anderen Zustand wechselt, unabhängig davon, in welchem Zustand es sich davor befand.

Einer der Zustände wird als »Startzustand« ausgewählt, d. h. als der Zustand, in dem sich das System ursprünglich befindet. In unserem Beispiel ist *Aus* der Startzustand, und wir bezeichnen den Startzustand konventionell durch das Wort *Start* sowie einen Pfeil, der auf den betreffenden Zustand zeigt.

Es ist oft notwendig, einen oder mehrere Zustände als »finale« oder »akzeptierende« Zustände zu kennzeichnen. Wird nach einer Reihe von Eingaben in einen dieser Zustände gewechselt, dann zeigt dies an, dass die Eingabesequenz in irgendeiner Weise gültig ist. Wir hätten zum Beispiel den Zustand *Ein* in Abbildung 1.1 als akzeptierenden Zustand betrachten können, da das Gerät, das von diesem Kippschalter gesteuert wird, in diesem Zustand eingeschaltet ist. Gemäß Konvention werden akzeptierende Zustände durch einen doppelten Kreis bezeichnet. In Abbildung 1.1 haben wir auf eine solche Bezeichnung verzichtet. ■

Beispiel 1.2 Gelegentlich kann das, was von einem Zustand beschrieben wird, sehr viel komplizierter sein als die Wahl zwischen *Ein* und *Aus*. Abbildung 1.2 zeigt einen anderen endlichen Automaten, der Teil einer lexikalischen Analysekomponente sein könnte. Dieser Automat hat die Aufgabe, das Schlüsselwort `then` zu erkennen. Daher benötigt er fünf Zustände, die jeweils eine der verschiedenen Positionen, die bislang erreicht worden sind, im Wort `then` repräsentieren. Diese Positionen entsprechen den Anfangsbuchstaben des Wortes und reichen von einer leeren Zeichenreihe (d. h. bislang wurde kein Buchstabe des Wortes erkannt) bis zum vollständigen Wort.

Abbildung 1.2: Darstellung der Erkennung des Wortes `then` als endlicher Automat



Die fünf Zustände in Abbildung 1.2 sind nach den Anfangsbuchstaben von `then` benannt, die bislang erkannt wurden. Als Eingaben werden Buchstaben empfangen. Wir können uns vorstellen, dass die lexikalische Analysekomponente die einzelnen Buchstaben des Programms, das gerade kompiliert wird, nacheinander prüft und dass der nächste Buchstabe die Eingabe des endlichen Automaten bildet. Der Startzustand ist gleich der leeren Zeichenreihe, und jeder Zustand wird mit dem nächsten Buchstaben von `then` in den Zustand überführt, der dem nächstgrößeren Präfix von `then` entspricht. Der Zustand namens `then` wird erreicht, wenn die Eingabe dem Wort `then` genau entspricht. Da dieser Automat Vorkommen des Wortes `then` erkennen soll, können wir diesen Zustand als den einzigen akzeptierenden Zustand betrachten. ■

1.1.2 Strukturelle Repräsentationen

Es gibt zwei wichtige Begriffsbildungen, die nicht zur Beschreibung von Automaten dienen, aber eine wichtige Rolle im Studium von Automaten und deren Anwendungen spielen.

1. *Grammatiken* sind nützliche Modelle zum Entwurf von Software, die Daten mit einer rekursiven Struktur verarbeitet. Das bekannteste Beispiel ist ein »Parser«, also die Komponente eines Compilers, die mit den rekursiv verschachtelten Elementen einer typischen Programmiersprache umgeht, wie arithmetische Ausdrücke, Bedingungsausdrücke usw. Beispielsweise besagt die Grammatikregel $E \Rightarrow E + E$, dass ein Ausdruck gebildet werden kann, indem zwei beliebige Ausdrücke durch ein Pluszeichen miteinander verbunden werden. Diese Grammatikregel ist typisch für die Art und Weise, wie in wirklichen Programmiersprachen Ausdrücke gebildet werden. Wir stellen diese so genannten kontextfreien Grammatiken in Kapitel 5 vor.

2. *Reguläre Ausdrücke* bezeichnen auch die Struktur von Daten, insbesondere von TextZeichenreihen. Wie wir in Kapitel 3 sehen werden, entsprechen die Muster, die durch reguläre Ausdrücke beschrieben werden, genau dem, was durch endliche Automaten beschrieben werden kann. Diese Ausdrücke sind ganz anders geartet als Grammatiken, und wir wollen uns hier mit einem einfachen

Beispiel begnügen. Der im Unix-Stil formulierte reguläre Ausdruck `'[A-Z] [a-z]*[ ] [A-Z] [A-Z]'` repräsentiert mit einem Großbuchstaben beginnende Wörter, denen ein Leerzeichen und zwei Großbuchstaben folgen. Dieser Ausdruck beschreibt Textmuster, bei denen es sich um einen amerikanischen Städtenamen samt Staatenkürzel handeln könnte, wie z. B. `Ithaca NY`. Er deckt aus zwei Wörtern bestehende Städtenamen, wie Palo Alto CA, nicht ab; diese könnten jedoch durch einen komplizierteren regulären Ausdruck erfasst werden:

```
'([A-Z] [a-z] *[ ])*[ ] [A-Z] [A-Z]'
```

Zur Interpretation solcher Ausdrücke müssen wir nur wissen, dass `[A-Z]` den Bereich von Textzeichen zwischen dem Großbuchstaben »A« und dem Großbuchstaben »Z« repräsentiert, dass `[ ]` zur Darstellung eines einzelnen Leerzeichens dient und dass das Zeichen * für »eine beliebige Anzahl« des voranstehenden Ausdrucks steht. Runde Klammern werden zur Gruppierung der Komponenten des Ausdrucks verwendet; sie repräsentieren keine Zeichen des beschriebenen Textes.

1.1.3 Automaten und Komplexität

Automaten sind für das Studium der Grenzen der Berechenbarkeit von zentraler Bedeutung. Wie wir in der Einführung zu diesem Kapitel erwähnt haben, sind zwei wichtige Fragen zu untersuchen:

1. Was können Computer überhaupt leisten? Man bezeichnet dies als die Frage der »Entscheidbarkeit«, und Probleme, die sich mithilfe eines Computers lösen lassen, werden »entscheidbar« genannt. Dieses Thema wird in Kapitel 9 behandelt.
2. Was können Computer effizient leisten? Dies wird als Frage der »Handhabbarkeit« bezeichnet, und Probleme, die von einem Computer innerhalb einer Zeitspanne gelöst werden können, die nicht stärker anwächst als eine langsam wachsende Funktion der Größe der Eingabe, werden »handhabbar« genannt. Häufig gehen wir davon aus, dass alle polynomialen Funktionen »langsam wachsen«, während Funktionen, die schneller wachsen als polynomiale Funktionen, als zu schnell wachsend betrachtet werden. Dieses Thema wird in Kapitel 10 näher untersucht.

1.2 Einführung in formale Beweise

Wer vor 1990 als High-School-Schüler die Geometrie von Flächen gelernt hat, hat wahrscheinlich detaillierte »deduktive Beweise« führen müssen, in denen die Wahrheit einer Behauptung durch eine detaillierte Folge von Schritten und Schlüssen bewiesen wurde. Die Geometrie hat selbstverständlich praktische Aspekte (z. B. müssen Sie die Formel zur Berechnung der Fläche eines Rechtecks kennen, um für ein Zimmer die richtige Menge an Teppichboden kaufen zu können), doch stellte das Studium formaler Beweismethoden einen mindestens ebenso wichtigen Grund für die Aufnahme dieses Zweigs der Mathematik in den Lehrplan von High Schools dar.
In den USA wurde es in den 90-er Jahren populär, Beweise als etwas zu unterrichten, was von der persönlichen Einstellung gegenüber der Aussage abhängt. Auch wenn es schön ist, wenn man sich der Wahrheit einer Aussage bewusst ist, die man verwenden muss, werden wichtige Beweistechniken heute in den amerikanischen High Schools

nicht mehr unterrichtet. Beweise sind jedoch etwas, was jeder Informatiker verstehen muss. Einige Informatiker vertreten sogar den extremen Standpunkt, dass ein formaler Beweis der Korrektheit eines Programms Hand in Hand mit dessen Programmierung gehen sollte. Wir bezweifeln, dass es produktiv wäre, so vorzugehen. Auf der anderen Seite gibt es jene, die behaupten, der Beweis wäre in der Disziplin der Informatik fehl am Platz. Dieses Lager proklamiert häufig das Motto »Wenn man nicht sicher ist, ob ein Programm korrekt ist, dann sollte man es starten und sehen, ob es richtig läuft«.

Unsere Position befindet sich irgendwo zwischen diesen beiden Extremen. Das Testen von Programmen ist sicherlich unabdingbar, hat jedoch seine Grenzen, da man Programme nicht mit jeder möglichen Eingabe testen kann. Wenn es sich um ein kompliziertes Programm handelt (angenommen, eine verzwickte Rekursion oder Iteration), dann ist es jedoch noch wichtiger, dass Sie verstehen, was in einer Schleife oder einer rekursiven Funktion passiert, da Sie sonst nicht in der Lage sein werden, richtig zu programmieren. Und wenn Tests ergeben, dass der Programmcode nicht korrekt ist, müssen Sie ihn auf jeden Fall korrigieren.

Um die Iteration oder Rekursion zu korrigieren, müssen Sie eine Induktionsannahme formulieren und Schlussfolgerungen ziehen können. Das Vorgehen, das zum Verständnis der Arbeitsweise eines korrekten Programms führt, entspricht im Grunde genommen genau dem Vorgehen zum Beweisen von Sätzen durch Induktion. Abgesehen davon, dass Sie damit Modelle erhalten, die für bestimmte Typen von Software nützlich sind, ist dies der Grund, warum formale Beweise zu einem traditionellen Bestandteil jedes Kurses in der Automatentheorie wurden. Die Automatentheorie eignet sich vielleicht mehr als andere Kernbereiche der Informatik für natürliche und interessante Beweise, die sowohl *deduktiver* (eine Folge von berechtigten Schritten) als auch *induktiver* (rekursive Beweise parametrisierter Aussagen, in denen die Aussage selbst mit »niedrigeren« Parameterwerten verwendet wird) Natur sein können.

1.2.1 Deduktive Beweise

Wie oben erwähnt, besteht ein deduktiver Beweis aus einer Folge von Aussagen, deren Wahrheit von einer Ausgangsaussage, der so genannten *Hypothese* oder *gegebenen Behauptung*, zu einer *Konklusion* führt. Jeder Beweisschritt muss sich nach einer akzeptierten logischen Regel aus den gegebenen Fakten oder aus vorangegangenen Aussagen dieses deduktiven Beweises ergeben.

Die Hypothese kann wahr oder falsch sein, was in der Regel von den Werten ihrer Parameter abhängt. Häufig setzt sich die Hypothese aus mehreren voneinander unabhängigen Aussagen zusammen, die durch den logischen Operator UND miteinander verknüpft sind. In diesem Fall wird jede einzelne dieser Aussagen als Hypothese oder gegebene Behauptung bezeichnet.

Der Satz, dass die Folge der Beweisschritte von einer Hypothese H zu einer Konklusion K führt, entspricht der Aussage »Wenn H, dann K«. Man sagt in diesem Fall, dass K aus H abgeleitet ist. Ein Beispielsatz der Form »wenn H, dann K«, soll diese Punkte veranschaulichen.

Satz 1.3 Wenn $x \geq 4$, dann $2^x \geq x^2$. ■

Es ist nicht allzu schwierig, sich informell davon zu überzeugen, dass Satz 1.3 wahr ist, aber der formale Beweis erfordert Induktion und wird für Beispiel 1.17 aufgespart.

Erstens ist zu beachten, dass »$x \geq 4$« die Hypothese H darstellt. Diese Hypothese hat einen Parameter x und ist daher weder wahr noch falsch. Die Wahrheit der Hypothese hängt dagegen vom Wert des Parameters x ab; z. B. ist H wahr für $x = 6$ und falsch für $x = 2$.

Analog stellt »$2^x \geq x^2$« die Konklusion K dar. Auch diese Aussage enthält den Parameter x und ist für bestimmte Werte von x wahr und für andere nicht. Beispielsweise ist K für $x = 3$ falsch, da $2^3 = 8$ und 8 kleiner ist als $3^2 = 9$. Andererseits ist K für $x = 4$ wahr, da $2^4 = 4^2 = 16$. Für $x = 5$ ist die Aussage auch wahr, da $2^5 = 32$ und 32 größer oder gleich $5^2 = 25$ ist.

Vielleicht erkennen Sie das intuitive Argument, aus dem wir schließen können, dass die Konklusion $2^x \geq x^2$ immer dann wahr ist, wenn $x \geq 4$. Wir haben bereits gesehen, dass die Konklusion für $x = 4$ wahr ist. Sobald x größer als 4 wird, verdoppelt sich die linke Seite 2^x jedes Mal, wenn x um 1 erhöht wird. Die rechte Seite x^2 wächst allerdings nur im Verhältnis $\left(\frac{x+1}{x}\right)^2$. Wenn $x \geq 4$, dann kann $\frac{x+1}{x}$ nicht größer als 1,25 sein und folglich kann $\left(\frac{x+1}{x}\right)^2$ nicht größer als 1,5625 sein. Da $1{,}5625 < 2$, wächst die linke Seite 2^x bei jedem Anstieg von x über 4 stärker an als die rechte Seite x^2. Folglich gilt, dass x um einen beliebigen Betrag erhöht werden kann, solange wir mit einem Wert von $x = 4$ beginnen, mit dem die Ungleichung bereits erfüllt ist.

Wir haben hiermit einen zwar informellen, aber akkuraten Beweis des Satzes 1.3 zu Ende geführt. Wir werden zu diesem Beweis zurückkehren und ihn in Beispiel 1.17 präzisieren, nachdem wir »induktive« Beweise vorgestellt haben.

Satz 1.3 umfasst wie alle interessanten Sätze eine unendliche Anzahl von miteinander in Beziehung stehenden Fakten, hier die Aussage »wenn $x \geq 4$, dann $2^x \geq x^2$« für alle ganzen Zahlen x. In der Tat ist die Annahme, dass x eine ganze Zahl ist, gar nicht erforderlich. Da im Beweis aber davon die Rede war, dass x beginnend mit $x = 4$ jeweils um 1 erhöht wird, wurde hier eigentlich nur der Fall berücksichtigt, dass x eine ganze Zahl ist.

Satz 1.3 kann zur Ableitung anderer Sätze herangezogen werden. Im nächsten Beispiel betrachten wir einen vollständigen deduktiven Beweis eines einfachen Satzes, in dem Satz 1.3 zur Anwendung kommt.

Satz 1.4 Wenn x die Summe der Quadrate von vier positiven ganzen Zahlen ist, dann gilt $2^x \geq x^2$.

Beweis: Intuitiv denken wir uns den Beweis so, dass x mindestens gleich 4 sein muss, wenn die Hypothese – also dass x die Summe der Quadrate von vier positiven ganzen Zahlen ist – wahr ist. Daher gilt hier die Hypothese von Satz 1.3, und weil wir an den Wahrheitsgehalt dieses Satzes glauben, können wir behaupten, auch dessen Konklusion sei für x wahr. Diese Überlegung lässt sich in einer Folge von Schritten ausdrücken. Jeder Schritt ist entweder die Hypothese des zu beweisenden Satzes, Teil dieser Hypothese oder eine Aussage, die aus einer oder mehreren vorangehenden Aussagen folgt.

Mit »folgt« ist hier gemeint, dass wir unterstellen: Wenn es sich bei einer vorangehenden Aussage um die Hypothese eines Satzes handelt, dann ist die Konklusion dieses Satzes wahr und kann als Aussage unseres Beweises niedergeschrieben werden. Diese logische Regel wird häufig als *Modus ponens* bezeichnet, d. h. wenn wir wissen, dass H wahr ist, und wenn wir wissen, dass »wenn H, dann K« wahr ist, dann können wir daraus schließen, dass K wahr ist. Wir lassen zudem bestimmte logische Schritte zur Bildung einer Aussage zu, die aus einer oder mehreren vorangehenden Aussagen folgt. Wenn beispielsweise A und B zwei vorangegangene Aussagen sind, dann können wir daraus die Aussage »A und B« ableiten.

Tabelle 1.1 zeigt die Folge der Aussagen, die zum Beweis von Satz 1.4 erforderlich ist. Obwohl wir Sätze im Allgemeinen nicht in einer so stilisierten Form beweisen werden, erleichtert es diese Darstellung, Beweise als explizite Liste von Aussagen zu begreifen, die jeweils eine präzise Begründung haben. In Schritt (1) haben wir eine der gegebenen Behauptungen des Satzes wiederholt, nämlich dass x die Summe der Quadrate von vier ganzen Zahlen ist. Es ist in Beweisen häufig hilfreich, wenn wir Objekte benennen, auf die zwar Bezug genommen wird, die aber im Satz nicht benannt sind. Wir haben dies hier getan und den vier ganzen Zahlen die Namen a, b, c und d gegeben.

Tabelle 1.1: Formaler Beweis des Satzes 1.4

Aussage		Begründung
1.	$x = a^2 + b^2 + c^2 + d^2$	Gegeben
2.	$a \geq 1$; $b \geq 1$; $c \geq 1$; $d \geq 1$	Gegeben
3.	$a^2 \geq 1$; $b^2 \geq 1$; $c^2 \geq 1$; $d^2 \geq 1$	(2) und Gesetze der Arithmetik
4.	$x \geq 4$	(1), (3) und Gesetze der Arithmetik
5.	$2^x \geq x^2$	(4) und Satz 1.3

In Schritt (2) halten wir den anderen Teil der Hypothese des Satzes fest: dass die zu potenzierenden ganzen Zahlen größer oder gleich 1 sind. Diese Aussage repräsentiert technisch vier Aussagen, nämlich jeweils eine Aussage für jede der vier implizit gegebenen ganzen Zahlen. In Schritt (3) stellen wir dann fest, dass das Quadrat einer Zahl, die größer oder gleich 1 ist, auch größer oder gleich 1 ist. Wir begründen dies mit der Tatsache, dass Aussage (2) wahr ist, und »Gesetzen der Arithmetik«. Das heißt, wir setzen voraus, dass der Leser einfache Aussagen zu Ungleichungen kennt und/oder beweisen kann, wie z. B. die Behauptung »wenn $y \geq 1$, dann $y^2 \geq 1$«.

Schritt (4) setzt die Aussagen (1) und (3) ein. Die erste Aussage besagt, dass x die Summe von vier Quadraten ist, und Aussage (3) besagt, dass jedes dieser Quadrate größer oder gleich 1 ist. Wir schließen wieder anhand wohl bekannter Gesetze der Arithmetik, dass x größer oder gleich $1 + 1 + 1 + 1$ bzw. 4 ist.

Im letzten Schritt (5) verwenden wir Aussage (4), d. h. die Hypothese von Satz 1.3. Der Satz selbst stellt die Begründung für die Konklusion dar, da dessen Hypothese die vorangegangene Aussage ist. Da Aussage (5), die der Konklusion von Satz 1.3 entspricht, die Konklusion von Satz 1.4 bildet, wurde Satz 1.4 hiermit bewiesen. Wir sind von der Hypothese dieses Satzes ausgegangen, und es ist uns gelungen, per Deduktion seine Konklusion abzuleiten. ■

1.2.2 Reduktion auf Definitionen

In den vorigen beiden Sätzen wurden Begriffe verwendet, mit denen Sie vertraut sein sollten: z. B. ganze Zahlen, Addition und Multiplikation. In vielen anderen Sätzen, einschließlich vielen der Automatentheorie, haben die in der Behauptung verwendeten Begriffe Implikationen, die weniger offensichtlich sind. Folgendes Vorgehen erweist sich in vielen Beweisen als nützlich:

- Wenn Sie sich nicht sicher sind, wie Sie einen Beweis beginnen sollen, wandeln Sie alle in der Hypothese enthaltenen Begriffe in ihre Definitionen um.

Hier ist ein Beispiel für einen Satz, der einfach zu beweisen ist, nachdem seine Aussage in Grundbegriffen ausgedrückt wurde. In diesem Satz werden die folgenden Definitionen verwendet:

1. Eine Menge S sei endlich, wenn es eine ganze Zahl n gibt, sodass S genau n Elemente enthält. Wir schreiben $\|S\| = n$ wobei $\|S\|$ für die Anzahl der Elemente der Menge S steht. Ist die Menge S nicht endlich, dann sagen wir, S sei *unendlich*. Informell ausgedrückt ist eine unendliche Menge eine Menge, die mehr als eine durch ganze Zahlen auszudrückende Anzahl von Elementen enthält.
2. Wenn S und T Teilmengen einer Menge U sind, dann ist T die Komplementärmenge von S (in Bezug auf U), wenn $S \cup T = U$ und $S \cap T = \emptyset$. Das heißt, jedes Element von U ist entweder in S oder in T enthalten, aber nicht in S und in T, oder anders ausgedrückt: T besteht genau aus den Elementen von U, die nicht in S enthalten sind.

Satz 1.5 S sei eine endliche Teilmenge einer unendlichen Menge U. T sei die Komplementärmenge von S in Bezug auf U. Dann ist T unendlich.

Beweis: Dieser Satz besagt informell ausgedrückt Folgendes: Wenn es einen unendlichen Vorrat von etwas (U) gibt und man einen endlichen Teil (S) davon abzieht, dann bleibt ein unendlicher Vorrat übrig. Zuerst wollen wir die Fakten des Satzes wie in Tabelle 1.2 dargestellt umformulieren.

Tabelle 1.2: Umformulierung der gegebenen Hypothesen des Satzes 1.5

Ursprüngliche Aussage	Neue Aussage
S ist endlich.	Es gibt eine ganze Zahl n derart, dass $\|S\| = n$
U ist unendlich.	Für keine ganze Zahl p gilt $\|U\| = p$
T ist die Komplementärmenge von S.	$S \cup T = U$ und $S \cap T = \emptyset$

Wir können noch immer keine weiteren Aussagen treffen und verwenden daher eine gängige Beweistechnik namens »Beweis durch Widerspruch«. Bei dieser Beweismethode, die in Abschnitt 1.3.3 näher erläutert wird, unterstellen wir, dass die Konklusion falsch ist. Unter Verwendung dieser Annahme und Teilen der Hypothese beweisen wir dann das Gegenteil einer der gegebenen Aussagen der Hypothese. Damit haben wir dann gezeigt, dass die Hypothese unmöglich in allen Teilen wahr und die Konklusion gleichzeitig falsch sein kann. Es bleibt lediglich die Möglichkeit übrig, dass die Konklusion wahr ist, wenn die Hypothese wahr ist. Dies bedeutet, dass der Satz wahr ist.

Im Fall von Satz 1.5 lautet das Gegenteil zur Konklusion »T ist endlich«. Angenommen, T sei endlich und die Aussage der Hypothese, die besagt, S ist endlich (also $\|S\| = n$ für eine ganze Zahl n), sei wahr. Wir können die Annahme, dass T endlich ist, in ähnlicher Weise umformulieren zu $\|T\| = m$ für eine ganze Zahl m.

Eine der gegebenen Aussagen besagt, dass $S \cup T = U$ und $S \cap T = \emptyset$. Das heißt, dass die Elemente von U genau den Elementen von S und T entsprechen. Folglich muss U genau $n + m$ Elemente enthalten. Da $n + m$ eine ganze Zahl ist, haben wir $\|U\| = n + m$ bewiesen, woraus folgt, dass U endlich ist. Genauer gesagt, haben wir bewiesen, dass die Anzahl der in U enthaltenen Elemente eine ganze Zahl ist, was der Definition von »endlich« entspricht. Die Aussage, dass U endlich ist, widerspricht jedoch der gegebenen Aussage, dass U unendlich ist. Folglich haben wir aus der Annahme, die Konklusion sei falsch, einen Widerspruch zu einer Hypothese abgeleitet, und nach dem Prinzip »Beweis durch Widerspruch« können wir damit folgern, dass der Satz wahr ist. ■

Beweise müssen nicht so wortreich sein. Nachdem wir die Gedankengänge, auf denen der Beweis beruht, dargestellt haben, wollen wir diesen Satz in einigen wenigen Zeilen erneut beweisen.

Beweis: (von Satz 1.5) Wir wissen, dass $S \cup T = U$ und dass S und T disjunkt sind, sodass $\|S\| + \|T\| = \|U\|$. Da S endlich ist, gilt $\|S\| = n$ für eine ganze Zahl n, und da U unendlich ist, gibt es keine ganze Zahl p derart, dass $\|U\| = p$. Angenommen, T sei endlich, sodass es eine ganze Zahl m gibt, für die gilt: $\|T\| = m$; dann ist $\|U\| = \|S\| + \|T\| = n + m$, was der gegebenen Behauptung widerspricht, dass es keine ganze Zahl p gibt, die gleich $\|U\|$ ist. ■

1.2.3 Andere Formen von Sätzen

Die »Wenn-dann«-Form von Sätzen ist in klassischen Gebieten der Mathematik am gebräuchlichsten. Es werden jedoch auch andere Behauptungen als Sätze bewiesen. In diesem Abschnitt untersuchen wir die gebräuchlichsten Formen von Aussagen und was in der Regel zu deren Beweis vonnöten ist.

Formen von »Wenn-dann«

Es gibt verschiedene Arten von Satzaussagen, die sich von der einfachen Aussageform »Wenn H, dann K« zwar unterscheiden, im Grunde genommen aber dasselbe aussagen: Wenn die Hypothese H für einen gegebenen Wert des/r Parameter/s wahr ist, dann ist die Konklusion K für denselben Wert wahr. Im Folgenden werden einige andere Möglichkeiten, »wenn H, dann K« auszudrücken, aufgeführt:

1. H impliziert K.
2. H nur dann, wenn K.
3. K, wenn H.
4. Wenn H gilt, folgt daraus K.

Es gibt zudem viele Varianten der Form (4), wie z. B. »wenn H gilt, dann gilt K«.

Beispiel 1.6 Die Behauptung von Satz 1.3 würde in diesen vier Aussageformen wie folgt erscheinen:

1. $x \geq 4$ impliziert $2^x \geq x^2$.
2. $x \geq 4$ nur dann, wenn $2^x \geq x^2$.

3. $2^x \geq x^2$, wenn $x \geq 4$.
4. Wenn $x \geq 4$ gilt, folgt daraus $2^x \geq x^2$.

Aussagen mit Quantoren

Viele Sätze beinhalten Aussagen, in denen der Allquantor (»für alle«) und der Existenzquantor (»es gibt«) oder Varianten dieser Quantoren, wie z. B. »für jedes« statt »für alle«, verwendet werden. Die Reihenfolge, in der diese Quantoren stehen, wirkt sich auf die Bedeutung der Aussage aus. Häufig ist es hilfreich, sich Aussagen mit mehreren Quantoren als Spiel zwischen zwei Spielern – dem Allquantor und dem Existenzquantor – vorzustellen, die abwechselnd Werte für die Parameter des Satzes festlegen. Der Allquantor muss alle möglichen Optionen berücksichtigen, und daher werden diese Optionen im Allgemeinen durch Variablen beschrieben und nicht durch Werte ersetzt. Der Existenzquantor muss dagegen lediglich einen Wert auswählen, der von den zuvor von den Spielern gewählten Werten abhängen kann. Die Reihenfolge, in der die Quantoren in der Aussage stehen, bestimmt, wer zuerst eine Auswahl treffen kann. Wenn der Spieler, der zuletzt eine Wahl treffen soll, stets einen zulässigen Wert findet, dann ist die Aussage wahr.

Betrachten Sie beispielsweise eine alternative Definition von »unendliche Menge«: Die Menge S ist *unendlich*, und zwar genau dann, wenn für alle ganzen Zahlen n gilt, dass es eine Teilmenge T von S gibt, die genau n Elemente enthält. Da hier »für alle« dem Quantor »es gibt« voransteht, müssen wir eine beliebige ganze Zahl n betrachten. Nun ist »es gibt« an der Reihe, eine Teilmenge von S auszuwählen, und kann dazu die Kenntnis der ganzen Zahl n nutzen. Wenn es sich bei S beispielsweise um die Menge der ganzen Zahlen handelt, dann könnte »es gibt« die Teilmenge $T = \{1, 2, \ldots, n\}$ auswählen und somit unabhängig davon, welchen Wert n hat, eine wahre Aussage bilden. Dies ist der Beweis, dass die Menge der ganzen Zahlen unendlich ist.

Die folgende Aussage sieht wie die Definition von »unendlich« aus, aber sie ist inkorrekt, weil darin die Reihenfolge der Quantoren vertauscht wurde: »Es gibt eine Teilmenge T der Menge S derart, dass für alle n gilt, dass die Teilmenge T genau n Elemente enthält.« Wenn mit S eine Menge wie die Menge der ganzen Zahlen gegeben ist, kann hier der Spieler »es gibt« eine beliebige Menge T auswählen. Angenommen, für T wird $\{1, 2, 5\}$ gewählt. Der Spieler »für alle« muss für diese Auswahl zeigen, dass T für jedes mögliche n auch n Elemente enthält. Dies ist allerdings nicht möglich. Beispielsweise ist diese Aussage falsch für $n = 4$. Tatsächlich ist sie falsch für alle $n \neq 3$.

Zudem wird in der formalen Logik häufig der Operator $\rightarrow$ an Stelle des Ausdrucks »wenn, dann« verwendet. Die Aussage »wenn H, dann K« kann daher in der mathematischen Fachliteratur auch in der Form $H \rightarrow K$ vorkommen. Wir werden diese Notation hier nicht verwenden.

Aussagen der Form »Genau dann, wenn«

Gelegentlich finden wir Aussagen der Form »A genau dann, wenn B«. Andere Varianten dieser Form sind »A ist äquivalent mit B« und »A dann – und nur dann –, wenn B«. Diese Aussage stellt eigentlich zwei Wenn-dann-Aussagen dar: »wenn A, dann B« und »wenn B, dann A«. Wir beweisen »A genau dann, wenn B«, indem wir diese beiden Behauptungen beweisen:

1. Den *Wenn*-Teil: »Wenn B, dann A« und

2. den *Nur-wenn*-Teil: »Wenn A, dann B«, der häufig in der äquivalenten Form »A nur dann, wenn B« ausgedrückt wird.

Es ist beliebig, in welcher Reihenfolge die Beweise erbracht werden. In vielen Sätzen ist ein Teil entschieden einfacher als der andere, und es ist üblich, den leichteren Teil zuerst zu beweisen und damit aus dem Weg zu räumen.

In der formalen Logik wird gelegentlich der Operator $\leftrightarrow$ oder $\equiv$ für Aussagen vom Typ »Genau dann, wenn ...« verwendet. Das heißt, $A \equiv B$ und $A \leftrightarrow B$ bedeuten dasselbe wie »A genau dann, wenn B«.

Wie formal müssen Beweise sein?

Es ist nicht einfach, diese Frage zu beantworten. Beweise dienen im Grunde genommen nur dem Zweck, jemanden (und hierbei kann es sich um denjenigen handeln, der Ihre Hausarbeit benotet, oder um Sie selbst) von der Richtigkeit der Strategie zu überzeugen, die Sie in Ihrem Programm einsetzen. Wenn der Beweis überzeugend ist, dann ist er ausreichend. Wenn er den Leser des Beweises nicht überzeugen kann, dann wurde im Beweis zu viel weggelassen.

Ein Teil der Unsicherheit in Bezug auf Beweise geht auf die unterschiedlichen Kenntnisstände zurück, die die Leser des Beweises haben können. In Satz 1.4 haben wir daher unterstellt, dass Sie die arithmetischen Grundgesetze kennen und eine Aussage wie »wenn $y \geq 1$, dann $y^2 \geq 1$« für wahr halten. Wären Ihnen die arithmetischen Gesetze nicht geläufig, dann müssten wir diese Aussage durch weitere Schritte in unserem deduktiven Beweis verifizieren.

Bestimmte Dinge sind in Beweisen jedoch erforderlich und ein Beweis wird inadäquat, wenn man sie weglässt. Jeder deduktive Beweis, in dem Aussagen verwendet werden, die nicht durch die gegebene Behauptung oder durch vorangegangene Aussagen gerechtfertigt sind, kann nicht adäquat sein. Der Beweis einer Aussage vom Typ »Genau dann, wenn ...« muss einen Beweis für den »Genau dann«-Teil und einen Beweis für den »Wenn«-Teil umfassen. Als weiteres Beispiel sei angeführt, dass induktive Beweise (die in Abschnitt 1.4 erörtert werden) Beweise der Basis und des Induktionsschritts erfordern.

Beim Beweis einer Aussage vom Typ »Genau dann, wenn ...« dürfen Sie nicht vergessen, dass beide Teile (also »genau dann« und »wenn«) bewiesen werden müssen. Gelegentlich ist es hilfreich, eine Aussage dieses Typs in eine Folge von Äquivalenzen aufzuspalten. Das heißt, um »A genau dann, wenn B« zu beweisen, können Sie zuerst »A genau dann, wenn C« und anschließend »C genau dann, wenn B« beweisen. Diese Methode funktioniert, solange Sie sich vor Augen halten, dass jeder Schritt in beiden Richtungen bewiesen werden muss. Wird ein Schritt lediglich in einer Richtung bewiesen, dann wird der gesamte Beweis dadurch ungültig.

Es folgt ein Beispiel für einen einfachen Beweis einer Aussage vom Typ »Genau dann, wenn ...«. In diesem Beweis werden die folgenden Notationen verwendet:

1. $\lfloor x \rfloor$, die auf eine ganze Zahl abgerundete reelle Zahl x, ist die größte ganze Zahl kleiner oder gleich x.

2. $\lceil x \rceil$, die auf eine ganze Zahl aufgerundete reelle Zahl x, ist die kleinste ganze Zahl größer oder gleich x.

Satz 1.7 x sei eine reelle Zahl. Dann gilt $\lfloor x \rfloor = \lceil x \rceil$ genau dann, wenn x eine ganze Zahl ist.

Beweis: (Genau-dann-Teil) In diesem Teil unterstellen wir $\lfloor x \rfloor = \lceil x \rceil$ und versuchen zu beweisen, dass x eine ganze Zahl ist. Mithilfe der Definitionen von Abrundung und Aufrundung stellen wir fest, dass $\lfloor x \rfloor \leq x$ und $\lceil x \rceil \geq x$. Gegeben ist jedoch $\lfloor x \rfloor = \lceil x \rceil$. Daher können wir in der ersten Ungleichung die Abrundung durch die Aufrundung ersetzen und zu dem Schluss kommen, dass $\lceil x \rceil \leq x$. Da sowohl $\lceil x \rceil \leq x$ als auch $\lceil x \rceil \geq x$ gilt, können wir auf Grund der Gesetze der Arithmetik schlussfolgern, dass $\lceil x \rceil = x$. Da $\lceil x \rceil$ stets eine ganze Zahl ist, muss auch x eine ganze Zahl sein.

(Wenn-Teil) Wir unterstellen nun, dass x eine ganze Zahl ist, und versuchen zu beweisen, dass $\lfloor x \rfloor = \lceil x \rceil$. Dieser Teil ist einfach. Gemäß den Definitionen von Abrundung und Aufrundung ist sowohl $\lfloor x \rfloor$ als auch $\lceil x \rceil$ gleich x, wenn x eine ganze Zahl ist, und folglich sind diese beiden Ausdrücke gleich. ■

1.2.4 Sätze, die keine Wenn-dann-Aussagen zu sein scheinen

Gelegentlich treffen wir auf einen Satz, der keine Hypothese zu haben scheint. Ein Beispiel hierfür ist folgende wohl bekannte Tatsache aus der Trigonometrie:

Satz 1.8 $\sin^2 \theta + \cos^2 \theta = 1$. ■

Diese Aussage besitzt tatsächlich eine Hypothese und diese Hypothese besteht aus allen Aussagen, die Sie kennen müssen, um diese Aussage interpretieren zu können. Beispielsweise umfasst die verborgene Hypothese hier die Aussage, dass θ ein Winkel ist und die Funktionen sinus und cosinus daher ihre übliche, für Winkel gültige Bedeutung haben. Ausgehend von diesen Definitionen und dem Satz des Pythagoras (in einem rechtwinkligen Dreieck ist das Quadrat der Hypotenuse gleich der Summe der Quadrate der beiden anderen Seiten) könnte man diesen Satz beweisen. Im Grunde genommen lautet die Wenn-dann-Form des Satzes eigentlich: »Wenn θ ein Winkel ist, dann $\sin^2 \theta + \cos^2 \theta = 1$.«

1.3 Weitere Formen von Beweisen

In diesem Abschnitt greifen wir verschiedene zusätzliche Themen auf, die mit der Bildung von Beweisen in Zusammenhang stehen:

1. Beweise in Bezug auf Mengen
2. Beweise durch Widerspruch
3. Beweise durch Gegenbeispiel

1.3.1 Beweise der Äquivalenz von Mengen

In der Automatentheorie müssen wir häufig einen Satz beweisen, der besagt, dass auf unterschiedliche Weise erstellte Mengen gleich sind. Häufig handelt es sich bei diesen Mengen um Mengen von Zeichenreihen, die als »Sprachen« bezeichnet werden, aber in diesem Abschnitt ist das Wesen der Mengen unwichtig. Wenn E und F zwei Ausdrücke sind, die Mengen repräsentieren, dann bedeutet die Aussage $E = F$, dass die beiden bezeichneten Mengen gleich sind. Genauer gesagt, jedes Element der durch E

angegebenen Menge ist auch in der durch F angegebenen Menge enthalten, und jedes Element der durch F angegebenen Menge ist auch in der durch E angegebenen Menge enthalten.

Beispiel 1.9 Das Kommutativgesetz der Vereinigung von Mengen besagt, dass die Vereinigung der beiden Mengen R und S in beliebiger Reihenfolge erfolgen kann. Das heißt, $R \cup S = S \cup R$. In diesem Fall ist E der Ausdruck $R \cup S$ und F der Ausdruck $S \cup R$. Das Kommutativgesetz der Vereinigung von Mengen besagt, dass $E = F$. ■

Die Mengengleichheit $E = F$ kann auch als Genau-dann-wenn-Aussage formuliert werden: Ein Element x ist genau dann in E enthalten, wenn x in F enthalten ist. Infolgedessen skizzieren wir einen Beweis für jede Aussage, die die Gleichheit der beiden Mengen $E = F$ unterstellt. Dieser Beweis hat die Form jedes Genau-dann-wenn-Beweises:

1. Beweis, dass gilt, wenn x in E enthalten ist, dann ist x in F enthalten.
2. Beweis, dass gilt, wenn x in F enthalten ist, dann ist x in E enthalten.

Als Beispiel für dieses Beweisverfahren wollen wir das *Distributivgesetz der Vereinigung von Schnittmengen* beweisen:

Satz 1.10 $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

Beweis: Die beiden Mengenausdrücke lauten $E = R \cup (S \cap T)$ und

$F = (R \cup S) \cap (R \cup T)$.

Wir werden die beiden Teile des Satzes nacheinander beweisen. Im »Wenn«-Teil unterstellen wir, dass x Element von E ist, und zeigen, dass x auch in F enthalten ist. In diesem Teil, der in Tabelle 1.3 zusammengefasst wird, werden die Definitionen von Vereinigung und Schnittmenge herangezogen, die wir als bekannt voraussetzen.

Wir müssen dann den »Genau-dann«-Teil des Satzes beweisen. Hier nehmen wir an, dass x Element von F ist, und zeigen, dass x in E enthalten ist. Die Beweisschritte sind in Tabelle 1.4 zusammengefasst. Da wir nun beide Teile der Genau-dann-wenn-Aussage bewiesen haben, gilt das Distributivgesetz von Vereinigungen von Schnittmengen als bewiesen. ■

Tabelle 1.3: Beweisschritte für den »Wenn«-Teil von Satz 1.10

	Aussage	Begründung
1.	x ist in $R \cup (S \cap T)$ enthalten.	gegeben
2.	x ist in R oder x ist in $(S \cap T)$ enthalten.	(1) und Definition von Vereinigungsmenge
3.	x ist in R oder x ist sowohl in S als auch T enthalten.	(2) und Definition von Schnittmenge
4.	x ist in $R \cup S$ enthalten.	(3) und Definition von Vereinigungsmenge

Tabelle 1.3: Beweisschritte für den »Wenn«-Teil von Satz 1.10 (Forts.)

Aussage		Begründung
5.	x ist in $R \cup T$ enthalten.	(3) und Definition von Vereinigungsmenge
6.	x ist in $(R \cup S) \cap (R \cup T)$ enthalten.	(4), (5) und Definition von Schnittmenge

Tabelle 1.4: Beweisschritte für den »Genau dann«-Teil von Satz 1.10

Aussage		Begründung
1.	x ist in $(R \cup S) \cap (R \cup T)$ enthalten.	gegeben
2.	x ist in $R \cup S$ enthalten.	(1) und Definition von Schnittmenge
3.	x ist in $R \cup T$ enthalten.	(1) und Definition von Schnittmenge
4.	x ist in R oder x ist sowohl in S als auch T enthalten.	(2), (3) und Schlussfolgerung über Vereinigungsmengen
5.	x ist in R oder x ist in $(S \cap T)$ enthalten.	(4) und Definition von Schnittmenge
6.	x ist in $R \cup (S \cap T)$ enthalten.	(5) und Definition von Vereinigungsmenge

1.3.2 Die Umkehrung

Jede Wenn-dann-Aussage verfügt über eine äquivalente Form, die unter bestimmten Bedingungen einfacher zu beweisen ist. Die Umkehrung (oder Kontraposition) der Aussage »wenn H, dann K« und »wenn nicht K, dann nicht H«. Eine Aussage und ihre Umkehrung sind entweder beide wahr oder beide falsch, daher kann eine Aussage durch ihre Umkehrung bewiesen werden und umgekehrt.

Warum die Aussagen »wenn H, dann K« und »wenn nicht K, dann nicht H« logisch äquivalent sind, wird deutlich, wenn man feststellt, dass vier Fälle zu betrachten sind:

1. H und K sind beide wahr.
2. H ist wahr und K ist falsch.
3. K ist wahr und H ist falsch.
4. H und K sind beide falsch.

Es gibt nur eine Möglichkeit, die Wenn-dann-Aussage falsch werden zu lassen. Die Hypothese muss wahr und die Konklusion falsch sein, wie in Fall (2). In den anderen drei Fällen, einschließlich Fall (4), in dem die Konklusion falsch ist, ist die Wenn-dann-Aussage selbst wahr.

Überlegen Sie jetzt, in welchen Fällen die Umkehrung »wenn nicht K, dann nicht H« falsch ist. Damit diese Aussage falsch ist, muss ihre Hypothese (die »nicht K« lautet) wahr sein und ihre Konklusion (die »nicht H« lautet) muss falsch sein. Aber »nicht K« ist genau dann wahr, wenn K falsch ist, und »nicht H« ist genau dann falsch, wenn H wahr ist. Diese beiden Bedingungen repräsentieren wiederum Fall (2), womit gezeigt wurde, dass in jedem dieser vier Fälle die ursprüngliche Aussage und ihre Umkehrung entweder beide wahr oder beide falsch sind, dass sie also logisch äquivalent sind.

Genau-dann-wenn-Aussagen mit Mengen

Wie bereits erwähnt, sind Sätze, die Äquivalenzen von Mengen zum Inhalt haben, Genau-dann-wenn-Aussagen. Daher hätte Satz 1.10 auch wie folgt formuliert werden können: Ein Element x ist in $R \cup (S \cap T)$ enthalten genau dann, wenn x in $(R \cap S) \cup (R \cap T)$ enthalten ist.

Ein anderer gebräuchlicher Ausdruck einer Mengenäquivalenz beinhaltet die Wendung: »Alle Elemente von X sind auch Elemente von Y und keiner anderen Menge.« Satz 1.10 hätte beispielsweise auch so ausgedrückt werden können: »Alle Elemente von $R \cup (S \cap T)$ sind auch Elemente von $(R \cap S) \cup (R \cap T)$ und keiner anderen Menge.«

Die Konversion

Die Begriffe »Umkehrung« (oder »Kontraposition«) und »Konversion« dürfen nicht miteinander verwechselt werden. Die *Konversion* einer Wenn-dann-Aussage ist die umgekehrte Implikation oder die »Gegenrichtung«; das heißt, die Konversion von »wenn H, dann K« lautet »wenn K, dann H«. Im Gegensatz zur Umkehrung, die mit der ursprünglichen Aussage logisch äquivalent ist, ist die Konversion *nicht* mit der ursprünglichen Aussage äquivalent. Bei den beiden Teilen eines Genau-dann-wenn-Beweises handelt es sich stets um eine Behauptung und deren Konversion.

Beispiel 1.11 Rufen Sie sich Satz 1.3 in Erinnerung, dessen Aussage lautete: »Wenn $x \geq 4$, dann $2^x \geq x^2$«. Die Umkehrung dieser Aussage lautet: »Wenn nicht $2^x \geq x^2$, dann nicht $x \geq 4$«. Weniger formal ausgedrückt und unter Verwendung der Tatsache, dass »nicht $a \geq b$« dasselbe ist wie $a < b$, lautet die Umkehrung: »Wenn $2^x < x^2$, dann $x < 4$.« ■

Wenn wir einen Genau-dann-wenn-Satz beweisen müssen, dann eröffnet uns die Verwendung der Umkehrung in einem der Teile verschiedene Möglichkeiten. Angenommen, wir wollen die Mengenäquivalenz von $E = F$ beweisen. Statt zu beweisen, dass »wenn x Element von E ist, dann ist x auch Element von F, und wenn x Element von F ist, dann ist x auch Element von E«, könnten wir eine Richtung umkehren. Eine äquivalente Beweisform ist:

- Wenn x Element von E ist, dann ist x auch Element von F, und wenn x nicht Element von E ist, dann ist x nicht Element von F.

Wir könnten E und F in der obigen Aussage auch vertauschen.

1.3.3 Beweis durch Widerspruch

Eine andere Möglichkeit, eine Aussage der Form »wenn H, dann K« zu beweisen, besteht im Beweis der Aussage:

- »H und nicht K impliziert Falschheit.«

Sie beginnen also mit der Annahme der Hypothese H und der Negation C der Konklusion K. Sie vervollständigen den Beweis, indem Sie zeigen, dass aus H und C etwas als falsch Bekanntes logisch folgt. Diese Form des Beweises wird *Beweis durch Widerspruch* genannt.

Beispiel 1.12 Erinnern Sie sich an Satz 1.5, in dem wir die Wenn-dann-Aussage mit der Hypothese H = »U ist eine unendliche Menge, S ist eine endliche Teilmenge von U und T ist die Komplementärmenge von S in Bezug auf U« und der Konklusion K = »T ist unendlich« bewiesen haben. Wir werden diesen Satz durch Widerspruch beweisen. Wir unterstellten »nicht K«; wir nahmen also an, dass T endlich ist.

Unser Beweis bestand darin, die Falschheit von H und nicht K abzuleiten. Wir haben zuerst gezeigt, dass auf Grund der Annahme, dass sowohl S als auch T endlich sind, auch U endlich sein muss. Da U aber gemäß der Hypothese H unendlich sein soll und eine Menge nicht gleichzeitig endlich und unendlich sein kann, haben wir die logische Aussage als »falsch« bewiesen. In Begriffen der Logik ausgedrückt, liegen eine Behauptung p (U ist endlich) und ihre Negation nicht p (U ist unendlich) vor. Wir nutzen dann die Tatsache, dass »p und nicht p« logisch äquivalent mit »falsch« ist. ■

Um zu verstehen, warum Beweise durch Widerspruch logisch korrekt sind, rufen Sie sich in Erinnerung, dass es vier Kombinationen von Wahrheitswerten für H und K gibt. Nur im zweiten Fall (H ist wahr und K ist falsch) ist die Aussage »wenn H, dann K« falsch. Indem gezeigt wurde, dass H und nicht K zu einer falschen Aussage führt, wurde gezeigt, dass Fall 2 nicht vorkommen kann. Folglich sind als Wahrheitswertkombinationen für H und K nur die drei Kombinationen möglich, bei denen »wenn H, dann K« wahr sind.

1.3.4 Gegenbeispiele

Im wahren Leben müssen wir keine Sätze beweisen. Stattdessen werden wir mit etwas konfrontiert, das wahr zu sein scheint (z. B. eine Strategie zur Implementierung eines Programms), und müssen entscheiden, ob dieser »Satz« wahr oder falsch ist. Um dieses Problem zu lösen, versuchen wir abwechselnd, den Satz zu beweisen, und falls dies nicht möglich ist, zu beweisen, dass dessen Behauptung falsch ist.

Bei Sätzen handelt es sich im Allgemeinen um Aussagen über eine unendliche Anzahl von Fällen, die möglicherweise alle Werte von Parametern eines Satzes sind. Nach streng mathematischer Konvention wird nur dann eine Aussage als »Satz« (oder Theorem) bezeichnet, wenn sie für eine unendliche Anzahl von Fällen gilt. Aussagen ohne Parameter oder Aussagen, die nur für eine endliche Anzahl von Werten ihrer Parameter gelten, werden *Beobachtungen* genannt. Um zu zeigen, dass ein angeblicher Satz kein Satz ist, genügt es zu zeigen, dass er in einem beliebigen Fall nicht gilt. Dies ist auf Programme übertragbar, da ein Programm im Allgemeinen als fehlerhaft gilt, wenn es mit einer Eingabe nicht korrekt funktioniert, mit der es korrekt arbeiten sollte.

Der Beweis, dass eine Aussage kein Satz ist, ist oft einfacher als der Beweis, dass eine Aussage ein Satz ist. Wenn S eine beliebige Aussage ist, dann ist die Aussage »S ist kein Satz« selbst eine Aussage ohne Parameter und kann daher als Beobachtung und nicht als Satz betrachtet werden. Es folgen zwei Beispiele. Das Erste zeigt eine Aussage, die offensichtlich kein Satz ist, und das Zweite eine Aussage, die ein Satz sein könnte und daher näher untersucht werden muss, bevor die Frage entschieden werden kann, ob es sich um einen Satz handelt.

Angeblicher Satz 1.13 Alle Primzahlen sind ungerade. (Formeller ausgedrückt: Wenn die ganze Zahl x eine Primzahl ist, dann ist x ungerade.)

Gegenbeispiel: Die ganze Zahl 2 ist eine Primzahl, aber 2 ist gerade. ■

Wir wollen nun einen »Satz« zur Modul-Arithmetik diskutieren. Eine grundlegende Definition muss hierzu zuerst eingeführt werden. Wenn a und b positive ganze Zahlen sind, dann ist $a \bmod b$ der Divisionsrest von a geteilt durch b, d. h. eine eindeutige ganze Zahl r im Bereich zwischen 0 und $b - 1$, und zwar derart, dass für eine bestimmte ganze Zahl q gilt: $a = qb + r$. Zum Beispiel: $8 \bmod 3 = 2$ und $9 \bmod 3 = 0$. Der erste angebliche Satz, dessen Falschheit wir beweisen werden, lautet:

Angeblicher Satz 1.14 Es gibt kein Paar von ganzen Zahlen a und b, derart dass

$$a \bmod b = b \bmod a.$$ ■

Wenn man eine Operation an Paaren von Objekten ausführen soll, wie hier a und b, lässt sich die Beziehung zwischen den beiden häufig vereinfachen, indem man die Symmetrie nutzt. In diesem Beispiel können wir uns auf den Fall konzentrieren, in dem $a < b$, da wir a und b vertauschen können, wenn $b < a$, und damit die gleiche Gleichung erhalten wie im angeblichen Satz 1.14. Wir dürfen allerdings nicht den dritten Fall vergessen, in dem $a = b$ und der sich in unseren Beweisversuchen hier als der kritische erweist.

Angenommen $a < b$. Dann ist $a \bmod b = a$, weil in der Definition von $a \bmod b$ festgelegt ist, dass $q = 0$ und $r = a$. Das heißt, wenn $a < b$, gilt also $a = 0 * b + a$. Aber $b \bmod a < a$, weil eine beliebige Zahl mod a stets einen Wert zwischen 0 und $a - 1$ hat. Daraus folgt, wenn $a < b$, ist $b \bmod a < a \bmod b$. Infolgedessen ist es unmöglich, dass $a \bmod b = b \bmod a$. Unter Verwendung des oben genannten Arguments der Symmetrie können wir zudem folgern, dass $a \bmod b \neq b \bmod a$, wenn $b < a$.

Betrachten wir jedoch den dritten Fall: $a = b$. Da für jede ganze Zahl x gilt, $x \bmod x = 0$, *gibt* es einen Fall, in dem gilt: $a \bmod b = b \bmod a$, wenn $a = b$. Damit verfügen wir über einen Gegenbeweis für den angeblichen Satz:

Gegenbeispiel: (des angeblichen Satzes 1.14) Sei a = b = 2. Dann gilt

$$a \bmod b = b \bmod a = 0.$$ ■

Während der Suche nach einem Gegenbeispiel haben wir sogar die exakten Bedingungen entdeckt, unter denen der angebliche Satz gilt. Es folgt die korrekte Version des Satzes und dessen Beweis.

Satz 1.15 $a \bmod b = b \bmod a$ genau dann, wenn $a = b$.

Beweis: (Wenn-Teil) Angenommen $a = b$. Wie wir oben sehen konnten, gilt für alle ganzen Zahlen x, dass $x \bmod x = 0$. Folglich ist $a \bmod b = b \bmod a$, wenn $a = b$.

(Nur-wenn-Teil) Nun nehmen wir an, dass $a \bmod b = b \bmod a$. Die beste Technik ist in diesem Fall ein Beweis durch Widerspruch, und daher nehmen wir zudem die Negation der Konklusion an, d. h. angenommen, $a \neq b$. Da $a = b$ hiermit eliminiert wurde, müssen wir lediglich die Fälle $a < b$ und $b < a$ betrachten.

Wir haben bereits festgestellt, dass $a \bmod b = a$ und $b \bmod a < a$, wenn $a < b$. Auf Grund dieser Aussagen und der Hypothese $a \bmod b = b \bmod a$ können wir einen Widerspruch ableiten.

Gemäß der Symmetrie gilt, wenn $b < a$, dann $b \bmod a = b$ und $a \bmod b < b$. Wir leiten wieder einen Widerspruch zur Hypothese ab und folgern, dass auch der Genau-dann-Teil wahr ist. Damit haben wir nun beide Richtungen bewiesen und kommen zu dem Schluss, dass der Satz wahr ist. ■

1.4 Induktive Beweise

Es gibt eine spezielle Form von Beweisen, die so genannten »induktiven« Beweise (auch Induktionsbeweise genannt), die beim Umgang mit rekursiv definierten Objekten unabdingbar sind. Viele der bekanntesten induktiven Beweise beinhalten ganze Zahlen, aber in der Automatentheorie lernen wir auch induktive Beweise für rekursiv definierte Konzepte wie Bäume und unterschiedliche Typen von Ausdrücken kennen, wie die regulären Ausdrücke, die in Abschnitt 1.1.2 kurz gestreift wurden. In diesem Abschnitt werden wir induktive Beweise zuerst anhand »einfacher« Induktion mit ganzen Zahlen vorstellen. Dann zeigen wir, wie »strukturelle« Induktionsbeweise für jedes beliebige rekursiv definierte Konzept geführt werden.

1.4.1 Induktive Beweise mit ganzen Zahlen

Angenommen, wir sollen die Aussage $S(n)$ für eine ganze Zahl n beweisen. Ein üblicher Ansatz besteht im Beweis von zwei Dingen:

1. Induktionsbeginn, wobei zu zeigen ist, dass $S(i)$ für eine bestimmte ganze Zahl i gilt. Gewöhnlich wird $i = 0$ oder $i = 1$ gewählt, aber es gibt Beispiele, in denen mit einem höheren Wert für i begonnen werden sollte, da die Aussage S möglicherweise für einige kleine ganze Zahlen falsch ist.

2. Induktionsschritt, wobei angenommen wird, dass $n \geq i$, wobei i den Wert der Induktionsbasis darstellt, und gezeigt wird, dass »wenn $S(n)$, dann $S(n + 1)$«.

Diese beiden Teile sollten uns davon überzeugen, dass $S(n)$ für jede ganze Zahl n wahr ist, die größer oder gleich der Basiszahl i ist. Wir können wie folgt argumentieren: Angenommen $S(n)$ wäre für eine oder mehrere dieser ganzen Zahlen falsch. Dann müsste es einen kleinsten Wert von n (z. B. j) geben, für den gilt, dass $S(j)$ falsch ist und gleichzeitig gilt $j \geq i$. Die ganze Zahl j kann nicht gleich i sein, da wir im Basisteil beweisen, dass $S(i)$ wahr ist. Folglich muss j größer als i sein. Daraus folgt, dass $j - 1 \geq i$ und $S(j - 1)$ wahr ist.

Im Induktionsschritt haben wir jedoch bewiesen, dass $S(n)$ impliziert $S(n + 1)$, wenn $n \geq i$. Angenommen, es sei $n = j - 1$. Wir wissen aus dem Induktionsschritt, dass $S(j - 1)$ die Aussage $S(j)$ impliziert. Da wir auch $S(j - 1)$ kennen, können wir auf $S(j)$ schließen.

Wir haben die Negation dessen angenommen, was wir beweisen wollten: Das heißt, wir nahmen an, dass $S(j)$ für eine ganze Zahl $j \geq i$ falsch sei. In beiden Fällen haben wir einen Widerspruch abgeleitet, und damit liegt ein »Beweis durch Widerspruch« dafür vor, dass S(n) für alle $n \geq i$ wahr ist.

Leider weist die obige Schlussfolgerung einen kleinen logischen Fehler auf. Unsere Annahme, dass wir mindestens eine ganze Zahl $j \geq i$ wählen können, für die $S(j)$ falsch ist, basiert auf unserem Glauben an das Prinzip der Induktion. Das heißt, die einzige Möglichkeit zu beweisen, dass wir eine solche ganze Zahl j finden können, besteht in einer Beweismethode, die im Grunde genommen einen induktiven Beweis darstellt. Der oben erörterte »Beweis« scheint jedoch vernünftig und entspricht unserem Realitätsverständnis. Wir verstehen ihn daher als integralen Bestandteil unseres logischen Schlussfolgerungssystems:

- *Das Induktionsprinzip:* Wenn wir $S(i)$ beweisen und beweisen können, dass $S(n)$ für alle $n \geq i$ $S(n + 1)$ impliziert, dann können wir darauf schließen, dass $S(n)$ für alle $n \geq i$ gilt.

Die folgenden beiden Beispiele veranschaulichen den Einsatz des Induktionsprinzips zum Beweis von Sätzen, die für ganze Zahlen gelten.

Satz 1.16 Für alle $n \geq 0$:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

Beweis: Der Beweis gliedert sich in zwei Teile: den Induktionsbeginn und den Induktionsschritt. Wir beweisen diese Teile nacheinander.

Induktionsbeginn: Für die Basis wählen wir $n = 0$. Es mag überraschen, dass der Satz selbst für $n = 0$ gilt, da die linke Seite der Gleichung (1.1) gleich $\sum_{i=1}^{0}$ ist, wenn $n = 0$. Es gibt allerdings eine allgemeine Regel, die besagt, dass eine Summe 0 ist, wenn die Obergrenze der Summe (in diesem Fall 0) kleiner als die Untergrenze (hier 1) ist, da die Summe dann mit null Termen gebildet wird. Das heißt: $\sum_{i=1}^{0} i^2 = 0$.

Die rechte Seite der Gleichung (1.1) ist ebenfalls 0, da 0 x (0 +1) x (2 x 0 + 1)/6 = 0. Folglich ist die Gleichung (1.1) wahr, wenn $n = 0$.

Induktionsschritt: Wir nehmen nun an, dass $n \geq 0$. Wir müssen den Induktionsschritt beweisen, dass Gleichung (1.1) dieselbe Formel impliziert, wenn n durch $n + 1$ ersetzt wird. Diese Formel lautet:

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \tag{1.2}$$

Wir können die Gleichungen (1.1) und (1.2) vereinfachen, indem wir die Summen und die Produkte auf der rechten Seite erweitern. Die vereinfachten Gleichungen sehen so aus:

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6 \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.4}$$

Wir müssen (1.4) mithilfe von (1.3) beweisen, da diese Gleichungen im Induktionsprinzip den Aussagen $S(n+1)$ bzw. $S(n)$ entsprechen. Der »Trick« besteht darin, die Summe bis $n+1$ auf der rechten Seite von (1.4) in eine Summe bis n plus den $(n+1)$-ten Term aufzuspalten. Auf diese Weise können wir die Summe bis n durch die linke Seite von (1.3) ersetzen und zeigen, dass (1.4) wahr ist. Diese Schritte lauten folgendermaßen:

$$\left(\sum_{i=1}^{n} i^2\right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.5}$$

$$(2n^3 + 3n^2 + n)/6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.6}$$

Zur endgültigen Verifizierung der Wahrheit von (1.6) ist nur einfache polynome Algebra auf der linken Seite erforderlich, um zu zeigen, dass die linke Seite mit der rechten identisch ist. ■

Beispiel 1.17 Im nächsten Beispiel beweisen wir Satz 1.3 aus Abschnitt 1.2.1. Dieser Satz besagt: Wenn $x \geq 4$, dann $2^x \geq x^2$. Wir haben einen informalen Beweis beschrieben, der auf der Vorstellung basierte, dass das Verhältnis $x^2/2^x$ schrumpft, wenn x über den Wert 4 hinaus wächst. Wir können diese Vorstellung präzisieren, wenn wir die Aussage $2^x \geq x^2$ durch eine vollständige Induktion über x beweisen und mit der Basis $x = 4$ beginnen. Beachten Sie, dass die Aussage für Werte $x < 4$ tatsächlich falsch ist.

Induktionsbeginn: Wenn $x = 4$, dann ergibt sowohl 2^x als auch x^2 den Wert 16. Folglich gilt $2^x \geq x^2$.

Induktionsschritt: Angenommen, für ein beliebiges $x \geq 4$ gilt $2^x \geq x^2$. Mit dieser Aussage als Hypothese müssen wir beweisen, dass dieselbe Aussage mit $x+1$ statt x wahr ist, d. h. $2^{[x+1]} \geq [x+1]^2$. Dies entspricht den Aussagen $S(x)$ und $S(x+1)$ im Induktionsprinzip. Die Tatsache, dass wir x statt n als Parameter verwenden, sollte nicht von Belang sein; x und n sind nur lokale Variablen.

Wie beim Satz 1.16 sollten wir $S(x+1)$ umformulieren, sodass wir $S(x)$ einsetzen können. In diesem Fall können wir $2^{[x+1]}$ durch 2 x 2^x ersetzen. Da $S(x)$ besagt, dass $2^x \geq x^2$, können wir schließen, dass $2^{[x+1]} = 2 \text{ x } 2^x \geq 2x^2$.

Wir brauchen aber etwas anderes. Wir müssen zeigen, dass $2^{[x+1]} \geq (x+1)^2$. Eine Möglichkeit, diese Aussage zu beweisen, besteht darin, $2^x \geq (x+1)^2$ zu beweisen und mithilfe der Transitivität von ≥ zu zeigen, dass $2^{[x+1]} \geq 2x^2 \geq (x+1)^2$. In unserem Beweis für

$$2x^2 \geq (x+1)^2 \tag{1.7}$$

können wir annehmen, dass $x \geq 4$. Wir beginnen mit der Vereinfachung von (1.7):

$$x^2 \geq 2x + 1 \tag{1.8}$$

Wir dividieren (1.8) durch x und erhalten damit:

$$x \geq 2 + \frac{1}{x} \tag{1.9}$$

Da $x \geq 4$, wissen wir dass $1/x \leq ¼$. Folglich ist die linke Seite von (1.9) größer oder gleich 4 und die rechte Seite höchstens 2,25. Wir haben damit die Wahrheit von (1.9) bewiesen. Folglich sind auch die Gleichungen (1.8) und (1.7) wahr. Die Gleichung (1.7) ergibt wiederum $2x^2 \geq (x+1)^2$ für $x \geq 4$ und ermöglicht den Beweis der Aussage $S(x+1)$, die $2^{x+1} \geq [x+1]^2$ lautete. ■

Ganze Zahlen als rekursiv definierte Konzepte

Wir erwähnten, dass induktive Beweise nützlich sind, wenn der Beweisgegenstand rekursiv definiert ist. Unsere ersten Beispiele waren allerdings Induktionen mit ganzen Zahlen, die wir normalerweise nicht für »rekursiv definiert« halten. Es gibt jedoch eine natürliche rekursive Definition einer nichtnegativen ganzen Zahl, und diese Definition entspricht in der Tat der Art und Weise, in der Induktionsbeweise mit ganze Zahlen erbracht werden: von den Objekten, die zuerst definiert wurden, zu den nachfolgend definierten Objekten.

Induktionsbeginn: 0 ist eine ganze Zahl.

Induktionsschritt: Wenn n eine Gannzahl ist, dann ist es auch $n + 1$.

1.4.2 Allgemeinere Formen der Induktion mit ganzen Zahlen

Gelegentlich wird ein induktiver Beweis nur durch den Einsatz eines allgemeineren Schemas ermöglicht, als dem in Abschnitt 1.4.1 vorgeschlagenen, wo wir eine Aussage S für einen Basiswert bewiesen haben und dann bewiesen haben, dass »wenn $S(n)$, dann $S(n + 1)$«. Zwei wichtige Verallgemeinerungen dieses Schemas sind:

1. Wir können mit mehreren Basisfällen arbeiten. Das heißt, wir beweisen $S(i)$, $S(i + 1)$, ... , $S(j)$ für ein beliebiges $j > i$.
2. Zum Beweis von $S(n + 1)$ können wir die Wahrheit aller Aussagen

$$S(i), S(i + 1), ..., S(n)$$

statt lediglich S(n) einsetzen. Überdies können wir nach dem Beweis der Basisfälle bis $S(j)$ annehmen, dass $n \geq j$ gilt statt einfach $n \geq i$.

Aus diesem Induktionsbeginn und dem Induktionsschritt ist die Schlussfolgerung zu ziehen, dass $S(n)$ wahr ist für alle $n \geq i$.

Beispiel 1.18 Das folgende Beispiel soll das Potenzial beider Prinzipien illustrieren. Die Aussage $S(n)$, die es zu beweisen gilt, lautet: Wenn $n \geq 8$, dann kann n als Summe eines Vielfachen von 3 und eines Vielfachen von 5 ausgedrückt werden. Beachten Sie, dass 7 nicht so dargestellt werden kann.

INDUKTIONSBEGINN: Als Basisfälle wurden $S(8)$, $S(9)$ und $S(10)$ gewählt. Die Beweise lauten $8 = 3 + 5$, $9 = 3 + 3 + 3$ und $10 = 5 + 5$.

INDUKTIONSSCHRITT: Angenommen, $n \geq 10$ und $S(8)$, $S(9)$, ..., $S(n)$ seien wahr. Wir müssen $S(n + 1)$ anhand dieser gegebenen Fakten beweisen. Unsere Strategie besteht darin, 3 von $n + 1$ zu subtrahieren, dann festzustellen, dass sich diese Zahl als Summe eines Vielfachen von 3 und eines Vielfachen von 5 ausdrücken lässt, und zu der Summe 3 addieren, um $n + 1$ angeben zu können.

Formaler ausgedrückt, stellen wir fest, dass $n - 2 \geq 8$, und daher können wir annehmen, dass $S(n - 2)$. Das heißt, $n - 2 = 3a + 5b$ für zwei ganze Zahlen a und b. Daraus folgt $n + 1 = 3 + 3a + 5b$, und somit lässt sich $n + 1$ als Summe von $3(a + 1)$ und $5b$ ausdrücken. Dies beweist, dass $S(n+ 1)$ und beschließt den Induktionsschritt. ■

1.4.3 Strukturelle Induktion

In der Automatentheorie gibt es verschiedene rekursiv definierte Strukturen, über die Aussagen bewiesen werden müssen. Zu den wichtigsten Beispielen gehören die bekannten Bäume und Ausdrücke. Wie Induktionsbeweise verfügen rekursive Definitionen über einen Basisfall, in dem eine oder mehrere grundlegende Strukturen definiert werden, und einen Induktionsschritt, in dem komplexere Strukturen unter Verwendung zuvor definierter Strukturen definiert werden.

Beispiel 1.19 Es folgt die rekursive Definition eines Baums:

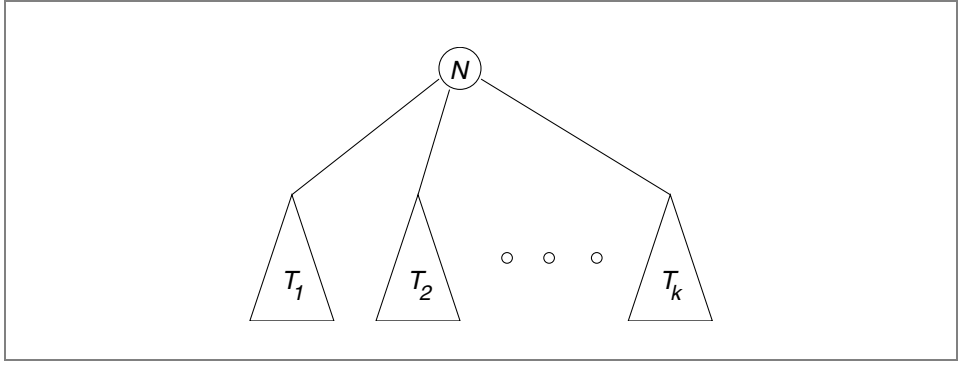
INDUKTIONSBEGINN: Ein einzelner Knoten ist ein Baum, und dieser Knoten ist die *Wurzel* des Baums.

INDUKTIONSSCHRITT: Wenn T_1, T_2, ..., T_k Bäume sind, dann kann wie folgt ein neuer Baum gebildet werden:

1. Man beginnt mit einem neuen Knoten N, der die Wurzel des Baums darstellt.
2. Dann werden Kopien aller Bäume T_1, T_2, ..., T_k hinzugefügt.
3. Anschließend fügt man Kanten vom Knoten N zu den Wurzeln der einzelnen Bäume T_1, T_2, ..., T_k hinzu.

Abbildung 1.3 zeigt den induktiven Aufbau eines Baums mit der Wurzel N aus k kleineren Bäumen. ■

Abbildung 1.3: Induktiver Aufbau eines Baums



Beispiel 1.20 Es folgt eine weitere rekursive Definition. Diesmal definieren wir Ausdrücke mithilfe der arithmetischen Operatoren + und ∗ sowie Zahlen und Variablen als Operanden.

INDUKTIONSBEGINN: Jede Zahl oder jeder Buchstabe (also eine Variable) ist ein Ausdruck.

INDUKTIONSSCHRITT: Wenn E und F Ausdrücke sind, dann seien auch $E + F$, $E * F$ und (E) Ausdrücke.

Beispielsweise sind gemäß der Induktionsbasis sowohl 2 als auch x Ausdrücke. Der Induktionsschritt besagt, dass $x + 2$, $(x + 2)$ und $2 * (x + 2)$ ebenfalls Ausdrücke sind. Beachten Sie, dass jeder dieser Ausdrücke von den zuvor definierten Ausdrücken abhängt. ■

Der der strukturellen Induktion zu Grunde liegende Gedankengang

Wir können informell erklären, warum die strukturelle Induktion eine gültige Beweismethode ist. Stellen Sie sich vor, wie durch aufeinander folgende rekursive Definitionen nacheinander festgelegt wird, dass bestimmte Strukturen X_1, X_2, ... der Definition genügen. Die Grundelemente kommen zuerst, und die Tatsache, dass X_i in der Menge der definierten Strukturen enthalten ist, kann nur von der Mengenzugehörigkeit der definierten Menge von Strukturen abhängen, die X_i in der Liste vorangehen. So gesehen ist eine strukturelle Induktion nichts anderes als ein Induktionsschluss für die ganze Zahl n der Aussage $S(X_n)$. Dieser Induktionsschluss kann die allgemeine Form haben, die in Abschnitt 1.4.2 erörtert wird, und über mehrere Basisfälle und einen Induktionsschritt verfügen, in dem alle vorherigen Instanzen der Aussage zum Einsatz kommen. Wir dürfen allerdings nicht vergessen, dass dieser Gedankengang, wie in Abschnitt 1.4.1 erläutert, kein formaler Beweis ist und wir die Gültigkeit dieses Induktionsprinzips ebenso unterstellen müssen wie die des Induktionsprinzips jenes Abschnitts.

Wenn eine rekursive Definition vorliegt, können wir mithilfe der folgenden Beweismethode, die als *strukturelle Induktion* bezeichnet wird, Sätze über diese Definition beweisen. $S(X)$ sei eine Aussage über die Strukturen X, die durch eine bestimmte rekursive Definition erklärt werden.

1. Als Basis wird $S(X)$ für die Basisstruktur(en) X bewiesen.
2. Im Induktionsschritt wird eine Struktur X gewählt, die gemäß der rekursiven Definition aus Y_1, Y_2, ..., Y_k gebildet wird. Wir nehmen an, die Aussagen $S(Y_i)$, $S(Y_2)$, ..., $S(Y_k)$ seien wahr, und verwenden sie zum Beweis von $S(X)$.

Wir kommen zu dem Schluss, dass $S(X)$ für alle X wahr ist. Die nächsten beiden Sätze sind Beispiele für Fakten über Bäume und Ausdrücke, die sich beweisen lassen.

Satz 1.21 Jeder Baum besitzt genau einen Knoten mehr als Kanten.

Beweis: Die formale Aussage $S(T)$, die durch strukturelle Induktion bewiesen werden muss, lautet: »Wenn T ein Baum ist und n Knoten und e Kanten besitzt, dann gilt $n = e + 1$.«

Induktionsbeginn: Der Basisfall ist gegeben, wenn T aus nur einem Knoten besteht. Dann ist $n = 1$ und $e = 0$, und somit gilt die Relation $n = e + 1$.

Induktionsschritt: T sei ein Baum, der durch den Induktionsschritt der Definition aus dem Wurzelknoten N und k kleineren Bäumen T_1, T_2, ..., T_k geformt wird. Wir können annehmen, dass die Aussagen $S(T_i)$ für $i = 1, 2, ..., k$ gelten. Das heißt, wenn T_i n_i Knoten und e_i Kanten besitzt, dann gilt $n_i = e_i + 1$.

Bei den Knoten von T handelt es sich um den Knoten N sowie die Knoten der Bäume T_i. T besitzt folglich $1 + n_1 + n_2 + ... + n_k$ Knoten. Die Kanten von T umfassen die k Kanten, die wir explizit im Induktionsschritt der Definition hinzugefügt haben, plus die Kanten der Bäume T_i. T besitzt folglich

$$k + e_1 + e_2 + ... + e_k \tag{1.10}$$

Kanten. Wenn wir in der Knotenanzahl von T n_i durch $e_i + 1$ ersetzen, dann ergibt sich, dass T

$$1 + [e_1 + 1] + [e_2 + 1] + ... + [e_k + 1] \tag{1.11}$$

Knoten besitzt. (1.11) lässt sich unmittelbar wie folgt umformulieren:

$$k + 1 + e_1 + e_2 + ... + e_k \tag{1.12}$$

Dieser Ausdruck ist genau um 1 größer als der Ausdruck in (1.10), der die Anzahl der Kanten von T bezeichnete. Folglich besitzt T genau eine um 1 höhere Anzahl von Knoten als Kanten. ■

Satz 1.22 Jeder Ausdruck enthält die gleiche Anzahl von linken und rechten Klammern.

BEWEIS: Formal beweisen wir die Aussage $S(G)$ für jeden Ausdruck G, der durch die rekursive Definition aus Beispiel 1.20 definiert wird: G enthält die gleiche Anzahl von linken und rechten Klammern.

INDUKTIONSBEGINN: Wenn G durch die Basis definiert wird, dann ist G eine Zahl oder Variable. Diese Ausdrücke enthalten 0 linke Klammern und 0 rechte Klammern. Folglich ist die Anzahl der Klammern gleich.

INDUKTIONSSCHRITT: Es gibt drei Regeln, nach denen der Ausdruck G gemäß dem Induktionsschritt der Definition gebildet worden sein kann:

1. $G = E + F$
2. $G = E * F$
3. $G = (E)$

Wir können annehmen, dass $S(E)$ und $S(F)$ wahr sind. Das heißt, E enthält dieselbe Anzahl von rechten und linken Klammern (sagen wir, jeweils n rechte und linke Klammern), und analog enthält F auch dieselbe Anzahl von rechten und linken Klammern (sagen wir, jeweils m rechte und linke Klammern). Wie folgt können wir dann die Anzahl der rechten und linken Klammern von G für jeden der drei Fälle berechnen:

1. Wenn $G = E + F$, dann enthält G $n + m$ linke und $n + m$ rechte Klammern, wobei jeweils n von E und m von F stammen.
2. Wenn $G = E * F$, dann enthält G aus denselben Gründen wie im Fall (1) wieder jeweils $n + m$ linke und rechte Klammern.
3. Wenn $G = (E)$, dann enthält G $n + 1$ linke Klammern (eine linke Klammer ist explizit gegeben und die anderen n sind in E enthalten). Ebenso weist G auch $n + 1$ rechte Klammern auf; eine ist explizit gegeben und die anderen sind in E enthalten.

In jedem der drei Fälle ergibt sich, dass in G dieselbe Anzahl von linken und rechten Klammern enthalten ist. Diese Beobachtung vervollständigt den Induktionsschritt und den Beweis. ■

1.4.4 Gegenseitige Induktion

Gelegentlich können wir nicht eine einzelne Aussage durch Induktion beweisen, sondern müssen eine Gruppe von Aussagen $S_1(n)$, $S_2(n)$, ..., $S_k(n)$ zusammen durch Induktion über n beweisen. In der Automatentheorie gibt es viele solcher Situationen. Mit Beispiel 1.23 geben wir ein Beispiel für eine häufig auftretende Situation, nämlich dass wir die Arbeitsweise eines Automaten erklären müssen, indem wir eine Gruppe von Aussagen beweisen, die jeweils einen Zustand beschreiben. Diese Aussagen besagen, mit welchen Eingabefolgen der Automat in die verschiedenen Zustände versetzt wird.

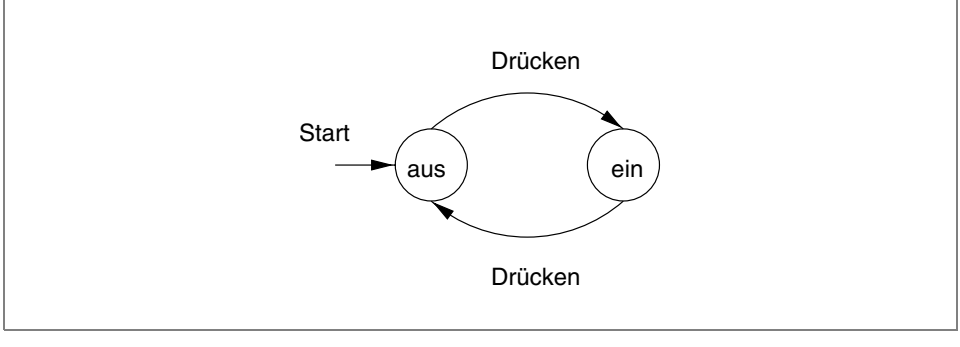
Streng genommen unterscheidet sich der Beweis einer Gruppe von Aussagen nicht vom Beweis der *Konjunktion* (logisches UND) aller Aussagen. Die Gruppe der Aussagen $S_1(n)$, $S_2(n)$, ..., $S_k(n)$ könnte beispielsweise durch die Einzelaussage $S_1(n)$ `UND` $S_2(n)$ `UND` ... `UND` $S_k(n)$ ersetzt werden. Gilt es allerdings, mehrere tatsächlich verschiedene Aussagen zu beweisen, ist es im Allgemeinen klarer, wenn man die Aussagen voneinander getrennt lässt und sie alle in eigenen Teilen der Basis und der Induktionsschritte beweist. Wir nennen diese Art von Beweis einen *gegenseitigen Induktionsbeweis*. Ein Beispiel soll die für einen gegenseitigen Induktionsbeweis notwendigen Schritte veranschaulichen.

Beispiel 1.23 Betrachten wir noch einmal den Ein-/Aus-Schalter, der in Beispiel 1.1 als Automat beschrieben wurde. Der Automat selbst wird in Abbildung 1.4 dargestellt. Da durch Drücken des Schalters der Zustand zwischen *Ein* und *Aus* hin- und hergeschaltet wird und der Schalter sich anfänglich im Zustand *Aus* befindet, erwarten wir, dass die folgenden Aussagen zusammengenommen die Arbeitsweise des Schalters erklären:

$S_1(n)$: Der Automat befindet sich nach n-maligem Drücken genau dann im Zustand *Aus*, wenn n gerade ist.

$S_2(n)$: Der Automat befindet sich nach n-maligem Drücken genau dann im Zustand *Ein*, wenn n ungerade ist.

Abbildung 1.4: Nochmals der Automat aus Abbildung 1.1



Wir können unterstellen, dass S_1 S_2 impliziert und umgekehrt, da wir wissen, dass die Zahl n nicht gleichzeitig gerade und ungerade sein kann. Allerdings gilt für einen Automaten nicht immer, dass er sich stets in genau einem Zustand befindet. Zufällig befindet sich der in Abbildung 1.5 dargestellte Automat stets in genau einem Zustand, aber diese Tatsache muss im Rahmen der gegenseitigen Induktion bewiesen werden.

Wir stellen die Basis und die Induktionsschritte der Beweise für die Aussagen $S_1(n)$ und $S_2(n)$ nachfolgend dar. Die Beweise hängen von verschiedenen Fakten zu ungeraden und geraden ganzen Zahlen ab: Wenn wir 1 von einer geraden ganzen Zahl subtrahieren oder dazuaddieren, erhalten wir eine ungerade ganze Zahl, und wenn wir 1 von einer ungeraden ganzen Zahl 1 subtrahieren oder dazuaddieren, erhalten wir eine gerade ganze Zahl.

INDUKTIONSBEGINN: Als Basis wählen wir $n = 0$. Da zwei Aussagen vorliegen, die jeweils in beiden Richtungen zu beweisen sind (weil S_1 und S_2 Aussagen vom Typ »genau dann, wenn« sind), umfasst die Basis vier Fälle, und der Induktionsschritt umfasst ebenfalls vier Fälle.

1. [S_1; Wenn]: Da 0 eine gerade ganze Zahl ist, müssen wir zeigen, dass sich der in Abbildung 1.4 gezeigte Automat nach 0-maligem Drücken im Zustand *Aus* befindet. Da dies der Anfangszustand ist, befindet sich der Automat nach 0-maligem Drücken in der Tat im Zustand *Aus*.

2. [S_1; Nur-wenn]: Der Automat befindet sich nach 0-maligem Drücken im Zustand *Aus*, und daher müssen wir zeigen, dass 0 gerade ist. 0 ist aber gemäß der Definition von »gerade« eine gerade ganze Zahl, und daher muss nichts bewiesen werden.

3. [S_2; Wenn]: Die Hypothese des Wenn-Teils von S_2 besteht darin, dass 0 eine ungerade ganze Zahl ist. Da diese Hypothese H falsch ist, ist jede Aussage der Form »wenn H, dann K« wahr, wie in Abschnitt 1.3.2 erörtert. Dieser Teil der Basis gilt daher.

4. [S_2; Nur-wenn]: Die Hypothese, dass sich der Automat nach 0-maligem Drücken im Zustand *Ein* befindet, ist auch falsch, da der Automat nur durch mindestens einmaliges Drücken in den Zustand *Ein* versetzt werden kann. Da die Hypothese falsch ist, können wir auch hier darauf schließen, dass die Nur-wenn-Aussage wahr ist.

INDUKTIONSSCHRITT: Wir nehmen nun an, dass $S_1(n)$ und $S_2(n)$ wahr sind, und versuchen, $S_1(n + 1)$ und $S_2(n + 1)$ zu beweisen. Der Beweis gliedert sich wieder in vier Teile.

1. [$S_1(n + 1)$; Wenn]: Die Hypothese dieses Teils besteht darin, dass $n + 1$ gerade ist. Folglich ist n ungerade. Der Wenn-Teil der Aussage $S_2(n)$ besagt, dass sich der Automat nach n-maligem Drücken im Zustand *Ein* befindet. Der mit *Drücken* beschriftete Bogen von *Ein* nach *Aus* zeigt an, dass das $(n + 1)$-malige Drücken den Automaten in den Zustand *Aus* versetzt. Damit ist der Beweis für den Wenn-Teil von S_1 vollständig.

2. [$S_1(n + 1)$; Nur-wenn]: Die Hypothese besteht darin, dass sich der Automat nach $(n + 1)$-maligem Drücken im Zustand *Aus* befindet. Wenn wir den Automaten in Abbildung 1.4. betrachten, wird deutlich, dass die einzige Möglichkeit, den Automaten mit wenigstens einer Aktion in den Zustand *Aus* zu versetzen, darin besteht, dass im Zustand *Ein* die Eingabe *Drücken* erfolgt. Wenn sich der Automat nach $(n + 1)$-maligem Drücken im Zustand *Aus* befindet, dann muss er sich nach n-maligem Drücken im Zustand *Ein* befunden haben. Wir können dann unter Verwendung des Nur-wenn-Teils von $S_2(n)$ darauf schließen, dass n ungerade ist. Folglich ist $(n + 1)$ gerade. Dies ist die gewünschte Schlussfolgerung für den Nur-wenn-Teil von $S_1(n + 1)$.

3. [$S_2(n + 1)$; Wenn]: Dieser Teil entspricht im Grunde genommen Teil (1), wenn man die Rollen der Aussagen S_1 und S_2 sowie »gerade« und »ungerade« vertauscht. Der Leser sollte diesen Teil des Beweises selbst formulieren können.
4. [$S_1(n + 1)$; Nur-wenn]: Dieser Teil entspricht im Grunde genommen Teil (2), wenn man die Rollen der Aussagen S_1 und S_2 sowie »gerade« und »ungerade« vertauscht. ■

Wir können Beispiel 1.23 das allgemeine Muster gegenseitiger Induktionen entnehmen:

- Jede Aussage muss im Induktionsbeginn und im Induktionsschritt getrennt bewiesen werden.
- Wenn es sich um Aussagen des Typs »genau dann, wenn« handelt, dann müssen sowohl im Induktionsbeginn als auch im Induktionsschritt beide Richtungen jeder Anweisung bewiesen werden.

1.5 Die zentralen Konzepte der Automatentheorie

In diesem Abschnitt stellen wir die wichtigsten Begriffsdefinitionen vor, die in der Automatentheorie von zentraler Bedeutung sind. Zu diesen Konzepten gehören »Alphabet« (eine Menge von Symbolen), »Zeichenreihe« (eine Liste aus Symbolen eines Alphabets) und »Sprache« (eine Menge von Zeichenreihen eines Alphabets).

1.5.1 Alphabete

Ein *Alphabet* ist eine endliche, nicht leere Menge von Symbolen. Gemäß Konvention wird ein Alphabet durch das Symbol Σ dargestellt. Zu gängigen Alphabeten gehören:

1. Σ = {0, 1}, das *binäre* Alphabet
2. Σ = {*a*, *b*, ..., *z*}, die Menge aller Kleinbuchstaben
3. Die Menge der ASCII-Zeichen oder die Menge aller druckbaren ASCII-Zeichen

1.5.2 Zeichenreihen

Eine *Zeichenreihe* (manchmal auch *Wort* oder *String* genannt) ist eine endliche Folge von Symbolen eines bestimmten Alphabets. Beispielsweise ist 01101 eine Zeichenreihe über dem binären Alphabet Σ = {0, 1). Die Zeichenreihe 111 ist ein weiteres Beispiel für eine Zeichenreihe über diesem Alphabet.

Die leere Zeichenreihe

Die leere Zeichenreihe ist eine Zeichenreihe, die keine Symbole enthält. Diese Zeichenreihe wird durch das Symbol ε dargestellt und kann aus jedem beliebigen Alphabet stammen.

Länge einer Zeichenreihe

Es ist häufig hilfreich, Zeichenreihen nach ihrer *Länge* zu klassifizieren, d. h. der Anzahl der für Symbole verfügbaren Positionen. Beispielsweise hat die Zeichenreihe

01101 die Länge 5. Es ist üblich, die Länge einer Zeichenreihe als »Anzahl der Symbole« der Zeichenreihe zu definieren. Umgangssprachlich ist diese Aussage akzeptabel, streng genommen ist sie aber nicht korrekt. Diese Zeichenreihe 01101 enthält z. B. nur zwei Symbole (0 und 1), aber fünf *Positionen* für Symbole, und daher hat sie die Länge 5. Sie können jedoch im Allgemeinen davon ausgehen, dass mit dem Ausdruck »die Anzahl von Symbolen« tatsächlich »die Anzahl von Positionen« gemeint ist.

Die Standardnotation für die Länge einer Zeichenreihe w lautet $|w|$. So ist z. B. $|011| = 3$ und $|\varepsilon| = 0$.

Potenzen eines Alphabets

Wenn Σ ein Alphabet ist, können wir die Menge aller Zeichenreihen einer bestimmten Länge über Σ durch eine Potenznotation bezeichnen. Wir definieren Σ^k als die Menge der Zeichenreihen mit der Länge k, deren Symbole aus dem Alphabet Σ stammen.

Beispiel 1.24 Beachten Sie, dass $\Sigma^0 = \{\varepsilon\}$, ungeachtet dessen, welches Alphabet Σ bezeichnet. Das heißt, ε ist die einzige Zeichenreihe mit der Länge 0.

Wenn $\Sigma = \{0, 1\}$, dann $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$,

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

usw. Beachten Sie den Unterschied zwischen Σ und Σ^1. Σ ist ein Alphabet mit den Symbolen 0 und 1 als Elementen. Σ^1 ist eine Menge von Zeichenreihen und enthält als Elemente die Zeichenreihen 0 und 1, die jeweils die Länge 1 haben. Wir werden hier nicht versuchen, diese beiden Mengen durch eine spezielle Notation zu unterscheiden, da aus dem Kontext hervorgeht, ob mit {0, 1} oder ähnlichen Mengen Alphabete oder Mengen von Zeichenreihen gemeint sind. ■

> **Typkonvention für Symbole und Zeichenreihen**
>
> Im Allgemeinen werden wir für Symbole Kleinbuchstaben vom Beginn des Alphabets (oder Ziffern) und für Zeichenreihen Kleinbuchstaben vom Ende des Alphabets, typischerweise w, x , y, z verwenden. Sie sollten versuchen, sich diese Konvention einzuprägen, um besser erkennen zu können, von welchem Typ die jeweils diskutierten Elemente sind.

Die Menge aller Zeichenreihen über einem Alphabet Σ wird gemäß Konvention durch Σ^* bezeichnet, zum Beispiel $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, ...\}$. Anders ausgedrückt:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup ...$$

Gelegentlich ist es wünschenswert, die leere Zeichenreihe aus der Menge von Zeichenreihen auszuschließen. Die Menge der nicht leeren Zeichenreihen des Alphabets Σ wird mit Σ^+ angegeben. Dementsprechend haben wir die zwei Äquivalenzen

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup ...$
- $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

Konkatenation von Zeichenreihen

Angenommen, x und y seien Zeichenreihen. Dann steht xy für die *Konkatenation* von x und y, d. h. die Zeichenreihe, die sich durch die Aneinanderreihung von x und y ergibt. Genauer gesagt, wenn x eine aus i Symbolen bestehende Zeichenreihe $x = a_1a_2$

... a_i und y eine aus j Symbolen bestehende Zeichenreihe $y = b_1b_2 ...b_j$ ist, dann ist xy die Zeichenreihe der Länge i + j: $xy = a_1a_2 ... a_ib_1b_2 ...b_j$.

Beispiel 1.25 Seien x = 01101 und y = 110. Dann ist xy = 01101110 und yx = 11001101. Für eine beliebige Zeichenreihe w gilt die Gleichung $\varepsilon w = w\varepsilon = w$. Das heißt, ε ist die Identität für die Konkatenation, da die Konkatenation der leeren Zeichenreihe mit einer beliebigen Zeichenreihe die letztere Zeichenreihe ergibt (vergleichbar mit der Addition von 0, der Identität für die Addition, zu einer beliebigen Zahl x, die als Ergebnis x liefert). ■

1.5.3 Sprachen

Eine Menge von Zeichenreihen aus Σ^*, wobei $\sum$ ein bestimmtes Alphabet darstellt, wird als *Sprache* bezeichnet. Wenn Σ ein Alphabet ist und $L \subseteq \sum^*$, dann ist L eine Sprache über dem Alphabet Σ. Beachten Sie, dass in den Zeichenreihen einer Sprache über Σ nicht alle Symbole aus Σ vorkommen müssen. Wenn also erklärt wurde, dass L eine Sprache über Σ ist, wissen wir, dass L gleichzeitig eine Sprache über jedem Alphabet ist, das eine Obermenge von Σ darstellt.

Die Wahl des Begriffs »Sprache« mag seltsam erscheinen. Die üblichen Sprachen können jedoch als Mengen von Zeichenreihen betrachtet werden. Als Beispiel lässt sich Englisch anführen, wobei die Sammlung zulässiger englischer Wörter eine Menge von Zeichenreihen über dem normalen lateinischen Alphabet ist. Ein anderes Beispiel ist C oder jede andere Programmiersprache, in der zulässige Programme eine Teilmenge der möglichen Zeichenreihen darstellen, die aus dem Alphabet der Sprache gebildet werden können. Dieses Alphabet ist eine Teilmenge des ASCII-Zeichensatzes. Das Alphabet verschiedener Programmiersprachen kann sich im Einzelnen leicht unterscheiden, aber im Allgemeinen umfasst es Groß- und Kleinbuchstaben, Ziffern, Satzzeichen und mathematische Symbole.

Es gibt allerdings zudem viele andere Sprachen, die beim Studium der Automaten in Erscheinung treten. Bei einigen handelt es sich um abstrakte Beispiele wie:

1. Die Sprache aller Zeichenreihen, die aus n Nullen gefolgt von n Einsen bestehen, wobei $n \geq 0$: {ε, 01, 0011, 000111, ...}.
2. Die Menge von Zeichenreihen aus Nullen und Einsen, die jeweils die gleiche Anzahl von Nullen und Einsen enthalten:

   {ε, 01, 10, 0011, 0101, 1001, ...}

3. Die Menge der Binärzahlen, deren Wert eine Primzahl ist:

   {10, 11, 101, 111, 1011, ...}

4. Σ^* ist eine Sprache für jedes beliebige Alphabet Σ.
5. ∅, die leere Sprache, ist eine Sprache über einem beliebigen Alphabet.
6. {ε}, die Sprache, die lediglich aus der leeren Zeichenreihe besteht, ist auch eine Sprache über einem beliebigen Alphabet. Beachten Sie, dass ∅ ≠ {ε}. Die leere Sprache ∅ enthält keine Zeichenreihen, {ε} enthält dagegen die leere Zeichenreihe.

Die einzige wichtige Einschränkung hinsichtlich der Bildung von Sprachen besteht in der Endlichkeit aller Alphabete. Sprachen können zwar eine unendliche Anzahl von

Zeichenreihen enthalten, sie sind aber auf Zeichenreihen über einem festen, endlichen Alphabet beschränkt.

1.5.4 Probleme

In der Automatentheorie besteht ein *Problem* in der Frage, ob eine gegebene Zeichenreihe Element einer bestimmten Sprache ist. Wie wir sehen werden, stellt sich heraus, dass alles, was wir umgangssprachlich als »Problem« bezeichnen, als die Frage formuliert werden kann, ob eine bestimmte Zeichenreihe in einer gegebenen Sprache enthalten ist. Genauer gesagt, wenn Σ ein Alphabet ist und L eine Sprache über Σ, dann lautet das Problem L:

- Gegeben sei eine Zeichenreihe w in Σ^*. Entscheiden Sie, ob w in L enthalten ist oder nicht.

Beispiel 1.26 Das Problem des Testens, ob eine Zahl eine Primzahl ist, kann durch die Sprache L_p ausgedrückt werden, die aus allen binären Zeichenreihen besteht, deren Wert als Binärzahl eine Primzahl darstellt. Das heißt, wenn eine aus Nullen und Einsen bestehende Zeichenreihe gegeben ist, gilt es zu entscheiden, ob die Zeichenreihe die Binärdarstellung einer Primzahl ist. Bei einigen Zeichenreihen ist diese Entscheidung einfach. Beispielsweise kann 0011101 einfach aus dem Grund nicht die Repräsentation einer Primzahl sein, weil die Binärdarstellung jeder ganzen Zahl mit Ausnahme von 0 mit einer 1 beginnt. Es ist allerdings weniger offensichtlich, dass 11101 Element von L_p ist. Daher erfordert jede Lösung für dieses Problem eine beträchtliche Menge an Rechenressourcen irgendwelcher Art, wie beispielsweise Zeit und/oder Speicherplatz. ■

Mengenvorschrift als Möglichkeit zur Definition von Sprachen

Es ist allgemein üblich, eine Sprache mithilfe einer Mengenvorschrift zu beschreiben:

$$\{w \mid \text{Aussage über } w\}$$

Dieser Ausdruck wird wie folgt gelesen: »Die Menge der Wörter w, derart dass (Aussage über w, die auf der rechten Seite des senkrechten Strichs steht)«. Beispiele hierfür sind:

1. $\{w \mid w$ enthält eine gleiche Anzahl von Nullen und Einsen$\}$.
2. $\{w \mid w$ ist eine binäre ganze Zahl, die eine Primzahl ist$\}$.
3. $\{w \mid w$ ist ein syntaktisch fehlerfreies C-Programm$\}$.

Es ist auch üblich, w durch einen Ausdruck mit Parametern zu ersetzen und die Zeichenreihe der Sprache durch für die Parameter geltende Bedingungen zu beschreiben. Es folgen einige Beispiele; das erste Beispiel mit dem Parameter n und das zweite mit den Parametern i und j:

1. $\{0^n1^n \mid n \geq 1\}$. Dieser Ausdruck wird gelesen: »Die Menge von 0 hoch n 1 hoch n, derart dass n größer oder gleich 1 ist.« Diese Sprache besteht aus den Zeichenreihen {01, 0011, 000111, ...}. Beachten Sie, dass wir ebenso wie bei einem Alphabet ein Symbol hoch n angeben können, um n Exemplare dieses Symbols darzustellen.
2. $\{0^i1^j \mid 0 \leq i \leq j\}$. Diese Sprache besteht aus Zeichenreihen, die einige (oder keine) Nullen gefolgt von mindestens ebenso vielen Einsen enthalten.

Ein potenziell unbefriedigender Aspekt unserer Definition des Begriffs »Problem« besteht darin, dass man sich Probleme gemeinhin nicht als Entscheidungsfragen (ist das Folgende wahr oder falsch?) vorstellt, sondern als Aufforderungen, eine Eingabe zu berechnen oder zu transformieren (wie lässt sich diese Aufgabe am besten lösen?). Die Aufgabe des Parsers eines C-Compilers lässt sich beispielsweise in unserem formalen Sinn als Problem beschreiben, bei dem eine ASCII-Zeichenreihe eingegeben wird und entschieden werden muss, ob diese Zeichenreihe Element von L_C – der Menge gültiger C-Programme – ist. Der Parser leistet allerdings mehr, als lediglich diese Frage zu entscheiden. Er erzeugt einen Parser-Baum, Einträge in einer Symboltabelle und möglicherweise anderes mehr. Der Compiler als Ganzes löst sogar das Problem, ein C-Programm in Objektcode für einen bestimmten Rechner umzuwandeln, was weit über die einfache Beantwortung der Frage der Gültigkeit des Programms hinausgeht.

Ist es eine Sprache oder ein Problem?

Mit den Begriffen »Sprache« und »Problem« ist eigentlich dasselbe gemeint. Welchem Begriff wir den Vorzug geben, hängt von unserem Standpunkt ab. Wenn wir uns nur mit Zeichenreihen an sich beschäftigen, z. B. in der Menge $\{0^n1^n \mid n \geq 1\}$, dann tendieren wir dazu, uns eine Menge von Zeichenreihen als Sprache vorzustellen. In den letzten Kapiteln dieses Buches werden wir eher dazu neigen, Zeichenreihen eine »Semantik« zuzuordnen, uns Zeichenreihen z. B. als Beschreibung von Graphen, logischen Ausdrücken oder sogar ganzen Zahlen vorzustellen. In diesen Fällen, in denen der durch die Zeichenreihe repräsentierte Sachverhalt wichtiger als die Zeichenreihe selbst ist, wird eine Menge von Zeichenreihen eher als Problem verstanden.

Die Definition von »Problemen« als Sprachen hat sich über die Zeit hinweg jedoch als adäquate Methode bewährt, mit den wichtigen Fragen der Komplexitätstheorie umzugehen. In dieser Theorie geht es darum, Untergrenzen der Komplexität bestimmter Probleme zu beweisen. Besonders wichtig sind Techniken zum Beweis, dass bestimmte Probleme nicht innerhalb eines Zeitraums gelöst werden können, der nicht exponentiell mit der Größe der Eingabe wächst. In dieser Hinsicht ist die sprachbasierte Version bekannter Probleme ebenso schwierig wie die aufgabenorientierte Version.

Das heißt, wenn wir beweisen, dass schwer entscheidbar ist, ob eine gegebene Zeichenreihe zur Sprache L_x gehört, die alle in einer Programmiersprache X gültigen Zeichenreihen umfasst, dann wird es zweifellos ebenso schwierig sein, Programme der Sprache X in Objektcode zu übersetzen. Wenn es einfacher wäre, Code zu generieren, dann könnten wir den Übersetzer ausführen und schließen, dass es sich bei der Eingabe um ein gültiges Element von L_x handelt, wenn der Übersetzer Objektcode erzeugen kann. Da der letzte Schritt in der Bestimmung, ob Objektcode generiert wurde, nicht schwierig sein kann, können wir den schnellen Algorithmus zur Erzeugung von Objektcode einsetzen, um die Frage der Zugehörigkeit zu L_x effizient zu entscheiden. Dies widerspricht der Annahme, dass die Zugehörigkeit zu L_x schwierig zu bestimmen ist. Hiermit verfügen wir über einen Beweis durch Widerspruch für die Aussage: »Wenn es schwierig ist, die Zugehörigkeit zu L_x bestimmen, dann ist es auch schwierig, in der Programmiersprache X geschriebene Programme zu kompilieren.«

Diese Technik, mit der die Schwierigkeit eines Problems gezeigt wird, indem mit seinem angenommenen effizienten Lösungsalgorithmus ein anderes, als schwierig bekanntes Problem effizient gelöst wird, wird als »Reduktion« des zweiten Problems auf das erste bezeichnet. Die Reduktion ist ein wichtiges Werkzeug beim Studium der Komplexität von Problemen, und sie wird durch unseren Ansatz stark erleichtert, Probleme als Fragen über die Zugehörigkeit zu einer Sprache zu betrachten statt als Fragen allgemeinerer Art.

1.6 Zusammenfassung von Kapitel 1

- *Endliche Automaten*: Endliche Automaten umfassen Zustände und Übergänge zwischen Zuständen, die in Reaktion auf Eingaben erfolgen. Sie sind beim Erstellen verschiedener Arten von Software nützlich, zu denen beispielsweise die lexikalische Analysekomponente von Compilern und Systeme zur Überprüfung der Fehlerfreiheit von Schaltkreisen oder Protokollen gehören.
- *Reguläre Ausdrücke* : Hierbei handelt es sich um eine strukturelle Notation zur Beschreibung der Muster, die durch einen endlichen Automaten repräsentiert werden können. Reguläre Ausdrücke werden in vielen gängigen Arten von Software verwendet, wie z. B. in Tools zur Suche von Mustern in Texten oder Dateinamen.
- *Kontextfreie Grammatiken*: Diese sind wichtig zur Beschreibung der Struktur einer Programmiersprache und zugehöriger Mengen von Zeichenreihen. Sie werden in der Entwicklung der Parserkomponente von Compilern eingesetzt.
- *Turing-Maschinen*: Hierbei handelt es sich um Automaten, die die Leistungsfähigkeit echter Computer modellieren. Sie ermöglichen uns die Untersuchung der Frage, was von einem Computer geleistet werden kann. Sie ermöglichen es uns zudem, handhabbare Probleme – Probleme, die mit polynomialem Zeitaufwand gelöst werden können – von nicht handhabbaren zu unterscheiden.
- *Deduktive Beweise*: Bei dieser grundlegenden Beweismethode werden Aussagen aufgelistet, die entweder als wahr gegeben sind oder logisch aus vorangegangenen Aussagen folgen.
- *Beweis von Wenn-dann-Aussagen*: Viele Sätze werden in der Form »wenn (etwas), dann (etwas anderes)« formuliert. Die Aussage oder Aussagen, die »wenn« folgen, werden als Hypothese bezeichnet, und die »dann« nachstehenden Aussagen bilden die Konklusion. Deduktive Beweise von Wenn-dann-Aussagen beginnen mit der Hypothese und fahren mit Aussagen fort, die logisch aus der Hypothese oder vorangegangenen Aussagen folgen, bis die Konklusion als eine dieser Aussagen bewiesen wurde.
- *Beweis von Genau-dann-wenn-Aussagen*: Andere Sätze haben die Syntax »(etwas) genau dann, wenn (etwas anderes)«. Zum Beweis dieser Sätze müssen die Wenn-dann-Aussagen in beiden Richtungen bewiesen werden. Einen ähnlichen Beweis fordern Sätze über die Gleichheit von Mengen, die auf zwei verschiedene Weisen beschrieben werden. Diese Sätze werden bewiesen, indem gezeigt wird, dass jede der beiden Mengen in der anderen enthalten ist.
- *Beweis der Umkehrung*: Gelegentlich ist es einfacher, eine Aussage der Form »wenn H, dann K« zu beweisen, indem die äquivalente Aussage »wenn nicht K, dann

nicht *H*« bewiesen wird. Letztere Aussage wird als Umkehrung der ersten bzw. als Kontraposition zur ersten bezeichnet.

- *Beweis durch Widerspruch*: Unter Umständen ist es günstiger, die Aussage »wenn *H*, dann *K*« durch »wenn *H* und nicht *K*, dann (etwas als falsch Bekanntes)« zu beweisen. Beweise dieser Art werden Beweise durch Widerspruch genannt.
- *Gegenbeispiele*: Manchmal müssen wir zeigen, dass eine bestimmte Aussage nicht wahr ist. Wenn die Aussage einen oder mehrere Parameter besitzt, dann können wir ihre generelle Falschheit beweisen, indem wir ein einziges Gegenbeispiel angeben, d. h. eine Wertzuweisung zu den Parametern, mit der die Aussage falsch wird.
- *Induktive Beweise*: Eine Aussage mit einem ganzzahligen Parameter n kann häufig durch Induktion über n bewiesen werden. Wir beweisen, dass die Aussage für die Basis (eine endliche Anzahl von Fällen für bestimmte Werte von n) wahr ist und beweisen dann den Induktionsschritt: Wenn die Aussage für Werte bis n wahr ist, dann ist sie auch für $n + 1$ wahr.
- *Strukturelle Induktion*: In einigen Situationen, einschließlich vieler der in diesem Buch beschriebenen, ist der Satz, den es zu beweisen gilt, ein Satz über ein rekursiv definiertes Konstrukt, wie z. B. Bäume. Wir können einen Satz über die konstruierten Objekte durch Induktion über die zu seiner Konstruktion erforderliche Anzahl von Schritten beweisen. Dieser Typ der Induktion wird als strukturelle Induktion bezeichnet.
- *Alphabete*: Ein Alphabet ist eine endliche Menge von Symbolen.
- *Zeichenreihen*: Eine Zeichenreihe (häufig auch String genannt) ist eine Folge von Symbolen, die eine endliche Länge hat.
- *Sprachen und Probleme*: Eine Sprache ist eine (möglicherweise unendliche) Menge von Zeichenreihen, deren Symbole aus ein- und demselben Alphabet gewählt wurden. Wenn die Zeichenreihen einer Sprache in irgendeiner Weise interpretiert werden sollen, wird die Frage, ob eine Zeichenreihe einer Sprache angehört, gelegentlich auch als Problem bezeichnet.

LITERATURANGABEN ZU KAPITEL 1

Zur Vertiefung des in diesem Kapitel vorgestellten Materials, einschließlich der der Informatik zu Grunde liegenden mathematischen Konzepte, empfehlen wir [1].

1. A. V. Aho und J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1994.

KAPITEL 2

Endliche Automaten

In diesem Kapitel wird die Klasse der Sprachen eingeführt, die als »reguläre Sprachen« bezeichnet werden. Bei diesen Sprachen handelt es sich um genau diejenigen, die von endlichen Automaten beschrieben werden können, welche wir in Abschnitt 1.1.1 kurz vorgestellt haben. Nach einem umfangreicheren Beispiel, das Anreize für die nachfolgende Untersuchung geben wird, definieren wir endliche Automaten formal.

Wie bereits erwähnt, besitzt ein endlicher Automat eine Menge von Zuständen, und seine »Steuerung« geht in Reaktion auf externe »Eingaben« von einem Zustand in einen anderen Zustand über. Eine der wichtigsten Unterscheidungen zwischen endlichen Automaten besteht darin, ob diese Steuerung »deterministisch« ist, was bedeutet, dass der Automat zu einem bestimmten Zeitpunkt nur einen Zustand haben kann, oder ob sie »nichtdeterministisch« ist, das heißt, dass der Automat gleichzeitig mehrere Zustände besitzen kann. Wir werden sehen, dass Nichtdeterminismus keine Sprachen zu definieren erlaubt, die nicht auch mit einem deterministischen endlichen Automaten definiert werden können, dass sich Anwendungen mithilfe nichtdeterministischer Automaten jedoch erheblich effizienter beschreiben lassen. In der Tat ermöglicht uns der Nichtdeterminismus die »Programmierung« von Problemlösungen in höheren Sprachen. Der nichtdeterministische endliche Automat wird dann nach einem Algorithmus, den wir in diesem Kapitel erläutern werden, in einen deterministischen Automaten transformiert der auf einem konventionellen Computer ausgeführt werden kann.

Wir beschließen das Kapitel mit der Studie eines erweiterten nichtdeterministischen Automaten, der die zusätzliche Option bietet, spontan von einem Zustand in einen anderen zu wechseln, d. h. nach »Eingabe« einer leeren Zeichenreihe. Dieser Automat akzeptiert auch nichts anderes als die regulären Sprachen. In Kapitel 3 beim Studium regulärer Ausdrücke und deren Äquivalenz mit Automaten werden wir jedoch feststellen, dass dieser Automat recht wichtig ist.

Die Untersuchung regulärer Sprachen wird in Kapitel 3 fortgesetzt. Wir stellen dort eine andere wichtige Möglichkeit zur Beschreibung regulärer Sprachen vor: die algebraische Notation für so genannte reguläre Ausdrücke. Nachdem wir reguläre Ausdrücke erörtert und ihre Äquivalenz mit endlichen Automaten gezeigt haben, verwenden wir in Kapitel 4 sowohl Automaten als auch reguläre Sprachen als Hilfsmittel, um bestimmte wichtige Eigenschaften regulärer Sprachen aufzuzeigen. Beispiele für solche Eigenschaften sind die »Abgeschlossenheit«, dank derer wir behaupten können, dass eine Sprache regulär ist, weil eine oder mehrere andere Sprachen

regulär sind, und »Entscheidbarkeitseigenschaften«. Hiermit sind Algorithmen gemeint, mit denen Fragen zu Automaten oder regulären Ausdrücken beantwortet werden können, z. B. ob zwei Automaten oder reguläre Ausdrücke dieselbe Sprache repräsentieren.

2.1 Eine informelle Darstellung endlicher Automaten

In diesem Abschnitt werden wir ein umfangreicheres Beispiel für ein praktisches Problem studieren, bei dessen Lösung endliche Automaten eine wichtige Rolle spielen. Wir untersuchen Protokolle, die den Gebrauch elektronischen »Geldes« ermöglichen. Mit elektronischem »Geld« sind Dateien gemeint, mit denen Kunden Waren im Internet bezahlen können und die der Verkäufer mit der Gewissheit entgegennehmen kann, dass das »Geld« echt ist. Der Verkäufer muss sichergehen können, dass die Datei nicht gefälscht ist und dass der Käufer keine Kopie der Datei zurückbehalten hat, mit der er erneut einkauft.

Die Fälschungssicherheit muss durch die Bank und durch Verschlüsselungsregeln garantiert werden. Um das Fälschungsproblem zu lösen, muss daher eine dritte Partei, die Bank, die »Gelddateien« ausgeben und verschlüsseln. Die Bank hat jedoch noch eine zweite wichtige Aufgabe: Sie muss eine Datenbank mit allen von ihr ausgestellten gültigen Gelddateien verwalten, damit sie überprüfen kann, ob die Gelddatei, die ein Geschäft erhalten hat, einen echten Geldbetrag repräsentiert und dem Konto des Geschäfts gutgeschrieben werden kann. Wir werden hier weder auf die verschlüsselungstechnische Seite dieses Problems eingehen noch darauf, wie die Bank diese »elektronischen Geldscheine«, von denen potenziell Milliarden im Umlauf sein können, speichert und abruft. Diese Probleme stellen langfristig gesehen wahrscheinlich kein Hindernis für das Konzept des elektronischen Geldes dar und Beispiele für dessen Einsatz gibt es seit den späten Neunzigern.

Voraussetzung für die Verwendung elektronischen Geldes ist allerdings die Festlegung von Protokollen, damit das Geld in der vom Benutzer gewünschten Weise gehandhabt werden kann. Da monetäre Systeme zum Betrug verleiten, müssen sämtliche Richtlinien verifiziert werden, die für die Verwendung des Geldes gelten. Das heißt, wir müssen beweisen, dass nur vorgesehene Transaktionen auftreten können, also nur solche, die es skrupellosen Benutzern nicht erlauben, anderer Leute Geld zu stehlen oder selbst Geld zu »produzieren«. Im weiteren Verlauf dieses Abschnitts stellen wir ein sehr einfaches Beispiel eines (schlechten) Protokolls für die Verwendung elektronischen Geldes vor, modellieren es mit endlichen Automaten und zeigen, wie die Automaten zur Verifizierung der Protokolle (bzw. in diesem Fall zur Erkennung eines Protokollfehlers) eingesetzt werden können.

2.1.1 Die Grundregeln

Es gibt drei Parteien: den Kunden, das Geschäft und die Bank. Wir nehmen der Einfachheit halber an, dass es nur eine »Gelddatei« gibt. Der Kunde kann diese Gelddatei an das Geschäft übertragen, das die Datei dann bei der Bank einlöst (d. h. das Geschäft veranlasst die Bank dazu, eine neue Gelddatei auszustellen, die nunmehr dem Geschäft statt dem Kunden gehört) und dem Kunden Waren liefert. Zudem hat der Kunde die Möglichkeit, die Gelddatei zu löschen. Das heißt, der Kunde kann die Bank bitten, das Geld wieder seinem Konto gutzuschreiben, sodass es nicht mehr ausgege-

ben werden kann. Die Interaktion zwischen diesen drei Parteien ist daher auf die folgenden fünf Ereignisse beschränkt:

1. Der Kunde kann sich dazu entschließen zu *zahlen*. Das heißt, der Kunde sendet Geld an das Geschäft.
2. Der Kunde kann sich dazu entschließen, das Geld zu *löschen* . Das Geld wird zusammen mit der Nachricht an die Bank geschickt, dass der Wert des Geldes dem Bankkonto des Kunden hinzugefügt werden soll.
3. Das Geschäft kann dem Kunden Waren *zusenden*.
4. Das Geschäft kann Geld *einlösen* . Das heißt, das Geld wird mit der Aufforderung an die Bank gesendet, den Gegenwert dem Geschäft auszuhändigen.
5. Die Bank kann das Geld *überweisen* , indem sie eine neue, entsprechend verschlüsselte Gelddatei erstellt und dem Geschäft zusendet.

2.1.2 Das Protokoll

Die drei Parteien müssen ihr Vorgehen sorgfältig planen, damit nicht das Falsche passiert. In unserem Beispiel gehen wir realistischerweise davon aus, dass man sich nicht darauf verlassen kann, dass sich der Kunde vernünftig verhält. Insbesondere nehmen wir an, dass der Kunde möglicherweise versucht, die Gelddatei zu kopieren, um sie mehrfach für Einkäufe zu verwenden, oder dass er mit der Gelddatei bezahlt und sie gleichzeitig löscht, um die Waren »umsonst« zu bekommen.

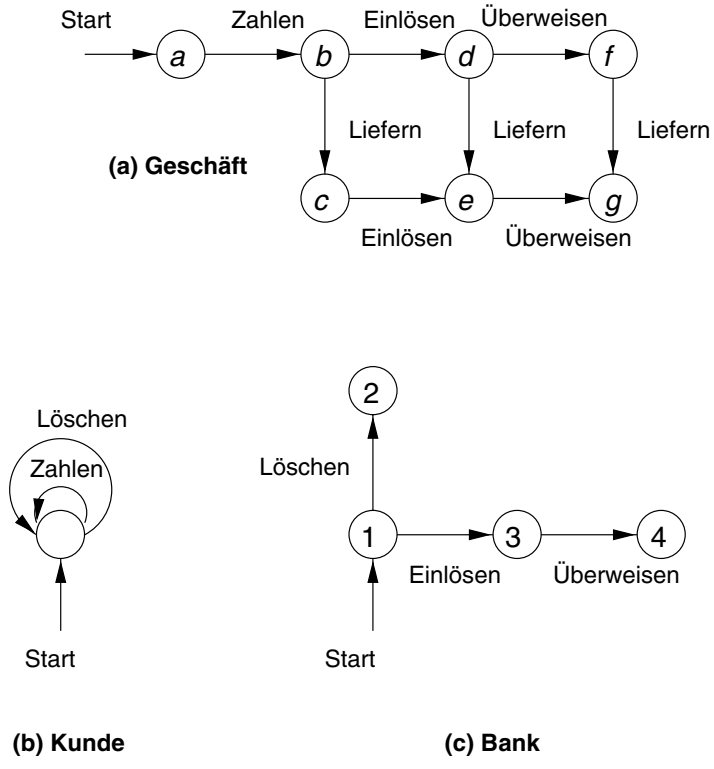
Die Bank muss sich verantwortlich verhalten, da sie sonst keine Bank ist. Genauer gesagt, muss die Bank Gewähr leisten, dass eine Gelddatei nicht von zwei Geschäften eingelöst werden kann, und sie darf nicht zulassen, dass dieselbe Gelddatei sowohl eingelöst als auch gelöscht wird. Auch das Geschäft muss vorsichtig sein. Insbesondere darf es erst dann Waren liefern, wenn es sicher sein kann, dass es dafür gültiges Geld erhalten hat.

Protokolle dieser Art lassen sich durch endliche Automaten darstellen. Jeder Zustand repräsentiert eine Situation, in der sich eine der Parteien befinden kann. Das heißt, der Zustand »merkt sich«, dass bestimmte Ereignisse vorgefallen und andere noch nicht vorgefallen sind. Zustandsänderungen treten auf, wenn eines der oben beschriebenen fünf Ereignisse eintritt. Wir stellen uns diese Ereignisse als etwas vor, das »außerhalb« der Automaten liegt, die die drei Parteien repräsentieren, obwohl jede Partei dafür verantwortlich ist, eines oder mehrere dieser Ereignisse zu initiieren. Es stellt sich heraus, dass die Reihenfolge, in der die Ereignisse eintreten, für das Problem von Bedeutung ist und nicht, wer zur Initiierung eines Ereignisses berechtigt ist.

Abbildung 2.1 stellt die drei Parteien als Automaten dar. In diesem Diagramm zeigen wir nur diejenigen Ereignisse, die eine Partei betreffen. Der Vorgang *Zahlen* betrifft beispielsweise lediglich den Kunden und das Geschäft. Die Bank weiß nicht, dass das Geld vom Kunden an das Geschäft gesandt wurde. Sie erfährt davon erst dann, wenn das Geschäft den Vorgang *Einlösen* ausführt.

Wir wollen uns zuerst den Automaten (c) ansehen, der die Bank repräsentiert. Zustand 1 ist der Startzustand, der die Situation widerspiegelt, in der die Bank die fragliche Gelddatei ausgestellt hat, aber noch nicht dazu aufgefordert wurde, sie einzulösen oder zu löschen. Wenn ein Kunde die Aufforderung zum *Löschen* sendet, dann schreibt die Bank den Geldbetrag wieder dem Konto des Kunden gut und

Abbildung 2.1: Endliche Automaten, die einen Kunden, ein Geschäft und eine Bank repräsentieren



nimmt den Zustand 2 an. Dieser Zustand repräsentiert die Situation, in der das Geld gelöscht wurde. Da die Bank verantwortlich handelt, behält sie Zustand 2 bei, wenn sie ihn einmal angenommen hat, da die Bank nicht zulassen darf, dass der Kunde das Geld erneut löscht oder ausgibt.[1]

Die Bank kann im Zustand 1 auch vom Geschäft eine Aufforderung zum *Einlösen* des Geldes erhalten. Wenn dieser Fall eintritt, dann wechselt die Bank in den Zustand 3 und sendet dem Geschäft eine Mitteilung über eine *Überweisung* zusammen mit einer neuen Gelddatei, die dann dem Geschäft gehört. Nach dem Senden der Überweisungsnachricht wechselt die Bank in den Zustand 4. In diesem Zustand akzeptiert die Bank weder Aufforderungen zum *Löschen* oder *Einlösen* noch führt sie andere Operationen mit der betreffenden Gelddatei aus.

Sehen wir uns nun Abbildung 2.1 (a) an, den Automaten, der die Eingaben des Geschäfts repräsentiert. Während die Bank immer das Richtige tut, weist das System

1. Sie dürfen nicht vergessen, dass Gegenstand dieser Diskussion eine einzige Gelddatei ist. Tatsächlich verwaltet die Bank viele Gelddateien und arbeitet dabei für jede dieser Dateien mit demselben Protokoll, dessen Funktionsweise für jede Gelddatei dieselbe ist. Deshalb können wir das Problem so erörtern, als gäbe es nur eine einzige elektronische Gelddatei.

des Geschäfts einige Mängel auf. Angenommen, die Liefer- und Finanzoperationen werden von unterschiedlichen Prozessen ausgeführt, sodass es möglich ist, dass die Eingabe *Liefern* vor, nach oder während des Einlösens des elektronischen Geldes erfolgen kann. Auf Grund dieser Regel kann das Geschäft in die Lage geraten, dass es die Waren bereits ausgeliefert hat und anschließend herausfindet, dass das Geld nichts wert war.

Das Geschäft beginnt in Zustand *a*. Wenn der Kunde Waren bestellt, indem er die Eingabe *Zahlen* ausführt, geht das Geschäft in den Zustand *b* über. In diesem Zustand startet das Geschäft sowohl den Liefer- als auch den Einlöseprozess. Falls die Waren zuerst geliefert werden, wechselt das Geschäft in den Zustand *c*, in dem es das Geld noch bei der Bank einlösen und die *Überweisung* einer entsprechenden Gelddatei von der Bank erhalten muss. Das Geschäft kann auch zuerst die Aufforderung zum *Einlösen* senden und damit in Zustand *d* übergehen. Ausgehend von Zustand *d* kann das Geschäft als Nächstes liefern und damit in Zustand *e* wechseln oder es kann als Nächstes die Geldüberweisung von der Bank erhalten und damit den Zustand *f* annehmen. Wir erwarten, dass das Geschäft ausgehend vom Zustand *f* schließlich die Waren liefert, woraufhin das Geschäft in den Zustand *g* übergeht, in dem die Transaktion abgeschlossen ist und nichts anderes mehr passiert. Im Zustand *e* wartet das Geschäft auf die *Überweisung* der Bank. Leider wurden die Waren bereits ausgeliefert und das Geschäft hat Pech gehabt, wenn die Überweisung ausbleibt.

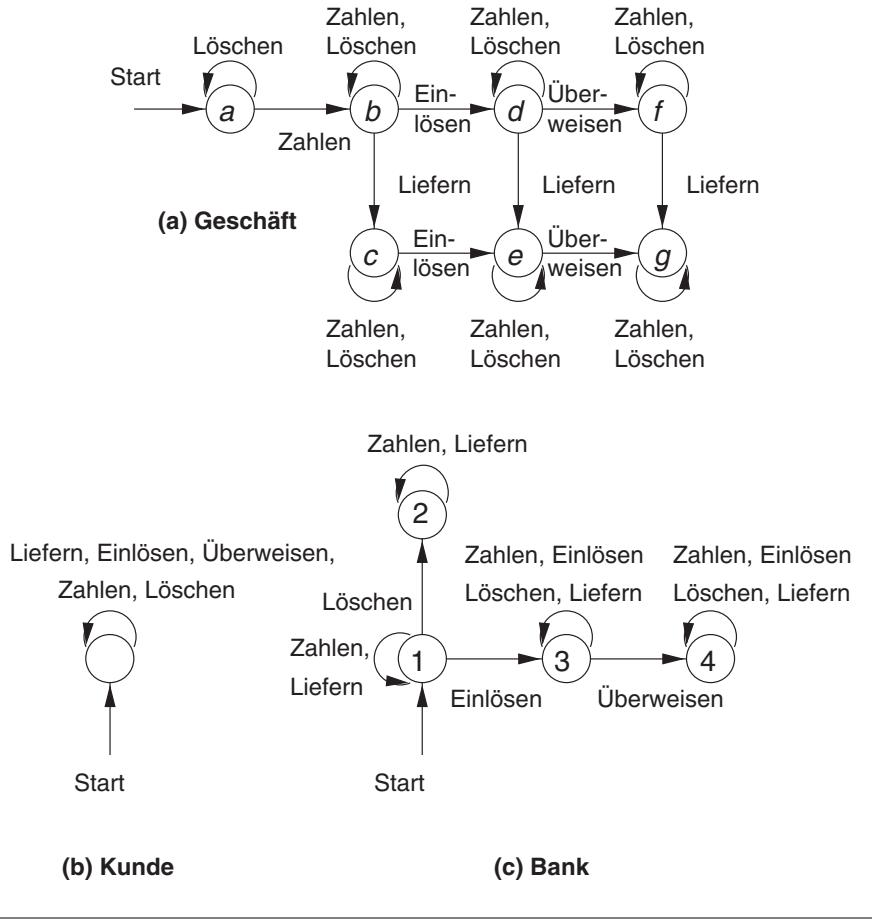
Zuletzt betrachten wir den Automaten in Abbildung 2.1: (b), der den Kunden repräsentiert. Dieser Automat kennt nur einen Zustand, was die Tatsache widerspiegelt, dass der Kunde »zu allem fähig ist«. Der Kunde kann die Eingaben *Zahlen* und *Löschen* beliebig oft in jeder beliebigen Reihenfolge ausführen und bleibt immer im gleichen Zustand.

2.1.3 Den Automaten dazu befähigen, Eingaben zu ignorieren

Obgleich die drei in Abbildung 2.1 gezeigten Automaten das Verhalten der drei Parteien unabhängig voneinander darstellen, gibt es bestimmte Zustandsübergänge, die hier fehlen. Beispielsweise betrifft die Nachricht über das *Löschen* das Geschäft nicht, und daher sollte das Geschäft seinen Zustand nicht ändern, wenn der Kunde *Löschen* eingibt. Nach der formalen Definition eines endlichen Automaten, mit der wir uns in Abschnitt 2.2 beschäftigen werden, muss der Automat, sobald er eine Eingabe *X* erhält, einem mit *X* benannten Pfeil von seinem aktuellen Zustand zu irgendeinem anderen Zustand folgen. Daher benötigt der Automat für das Geschäft einen Pfeil namens *Löschen* , der von jedem Zustand aus jeweils zurück zu diesem Zustand führt. Ohne diese zusätzlichen Pfeile befände sich der Automat »Geschäft« in überhaupt keinem Zustand, sobald *Löschen* eingegeben würde.

Ein weiteres potenzielles Problem besteht darin, dass eine der Parteien beabsichtigt oder unbeabsichtigt eine nicht vorhergesehene Nachricht senden kann und die Automaten dadurch nicht in einen undefinierten Zustand geraten sollen. Nehmen wir beispielsweise an, der Kunde gibt *Zahlen* ein zweites Mal ein, während sich das Geschäft in dem Zustand *c* befindet. Da von diesem Zustand kein Pfeil mit der Beschriftung *Zahlen* ausgeht, würde der Automat »Geschäft« in einen undefinierten Zustand geraten, bevor die Überweisung von der Bank eintrifft. Kurzum, wir müssen bei den Automaten aus Abbildung 2.1 an bestimmten Zuständen Schleifen hinzufügen und sie mit Beschriftungen für all jene Eingaben versehen, die der Automat in dem betreffenden Zustand ignorieren muss. Die vollständigen Automaten sind in Abbildung 2.2 darge-

Abbildung 2.2: Vollständige Mengen der Zustandsänderungen, die die drei Automaten erfahren können



stellt. Um Platz zu sparen, kombinieren wir die Beschriftungen an einem Pfeil, statt mehrere gleichgerichtete Pfeile mit unterschiedlichen Beschriftungen zu zeichnen. Die folgenden beiden Arten von Eingaben müssen ignoriert werden:

1. *Eingaben, die für eine Partei irrelevant sind* . Wie wir gesehen haben, ist für das Geschäft lediglich die Eingabe *Löschen* irrelevant und daher verfügen alle sieben Zustandsknoten über eine Schleife mit der Beschriftung *Löschen* . Für die Bank sind die Eingaben *Zahlen* und *Liefern* irrelevant und deshalb wurden sämtlichen Zustandsknoten der Bank Schleifen mit der Beschriftung *Zahlen, Liefern* hinzugefügt. Da für den Kunden die Eingaben *Liefern*, *Einlösen* und *Überweisen* nicht relevant sind, fügen wir Schleifen mit diesen Beschriftungen hinzu. Der Automat, der den Kunden repräsentiert, bleibt nach jeder Eingabesequenz im selben Zustand, und somit hat er im Endeffekt keinen Einfluss auf die Arbeitsweise des gesamten Systems. Natürlich ist der Kunde nach wie vor

Teil dieses Systems, denn er ist es schließlich, der die Aktionen *Zahlen* und *Löschen* initiiert. Wie bereits erwähnt, ist es für das Verhalten der Automaten allerdings nicht von Belang, wer welche Aktionen initiiert.

2. *Eingaben, die ignoriert werden müssen, da mit die Automaten nicht in einen undefinierten Zustand geraten*. Wie erwähnt, dürfen wir nicht zulassen, dass der Kunde den Automaten des Geschäfts in einen undefinierten Zustand bringt, indem er die Eingabe *Zahlen* zweimal ausführt. Wir haben daher allen Zustandsknoten mit Ausnahme von *a* (in dem die Eingabe *Zahlen* erwartet wird und relevant ist) Schleifen mit der Beschriftung *Zahlen* hinzugefügt. Zudem haben wir Schleifen mit der Beschriftung *Löschen* den Zuständen *3* und *4* der Bank hinzugefügt, damit der Kunde diesen Automaten nicht dadurch in einen undefinierten Zustand bringen kann, dass er versucht, Geld zu löschen, das bereits eingelöst wurde. Die Bank ignoriert daraufhin richtigerweise eine solche Anforderung. Ebenso verfügen die Zustände 3 und 4 über Schleifen mit der Beschriftung *Einlösen* . Das Geschäft darf nicht versuchen, Geld zweimal einzulösen, und falls dies doch geschieht, ignoriert die Bank die zweite Anforderung.

2.1.4 Das gesamte System aus Automaten darstellen

Wir haben nun zwar Modelle für das Verhalten der drei beteiligten Parteien, aber keine Repräsentation der Interaktion dieser Parteien. Da für das Verhalten des Kunden keine Beschränkungen festgelegt wurden, verfügt der betreffende Automat nur über einen einzigen Zustand, d. h. es ist unmöglich, dass das System als Ganzes »aus den Fugen gerät«, weil der Automat »Kunde« auf eine Eingabe nicht zu reagieren weiß. Allerdings zeigen das Geschäft und die Bank ein kompliziertes Verhalten und es ist nicht sofort erkennbar, welche Kombination von Zuständen diese beiden Automaten aufweisen.

Normalerweise erforscht man die Interaktion von solchen Automaten, indem man einen *Produktautomaten* konstruiert. Die Zustände dieses Automaten repräsentieren ein Zustandspaar, das sich jeweils aus dem Zustand der Bank und dem Zustand des Geschäfts zusammensetzt. Beispielsweise bezeichnet der Zustand (3, *d*) des Produktautomaten die Situation, in der sich die Bank im Zustand 3 und das Geschäft im Zustand *d* befindet. Da die Bank vier Zustände hat und das Geschäft sieben Zustände kennt, hat der Produktautomat 4 x 7 = 28 Zustände.

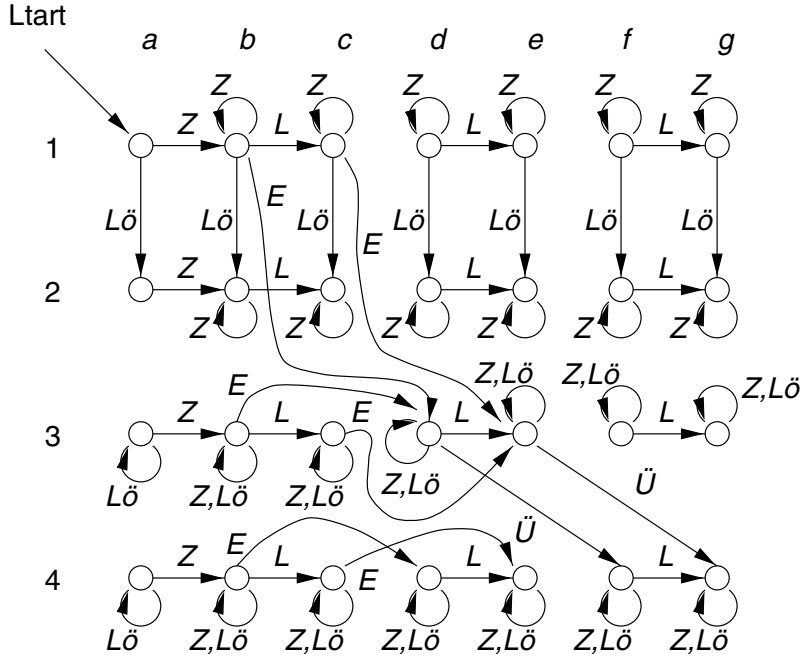
Der Produktautomat ist in Abbildung 2.3 dargestellt. Der Übersichtlichkeit halber haben wir die 28 Zustände in einem Array angeordnet. Die Zeilen entsprechen den Zuständen der Bank und die Spalten denen des Geschäfts. Um Platz zu sparen, wurden die Beschriftungen der Pfeile abgekürzt, wobei *Z* für *Zahlen*, *L* für *Liefern*, *Lö* für *Löschen* , *E* für *Einlösen* und *Ü* für *Überweisen* steht.

Damit wir die Pfeile zwischen den Zustandsknoten des Produktautomaten erhalten, müssen wir die Automaten »Bank« und »Geschäft« quasi parallel ausführen. Die beiden Komponenten des Produktautomaten gehen auf die verschiedenen Eingaben hin jeweils unabhängig voneinander in andere Zustände über. Es ist allerdings unbedingt zu beachten, dass der Produktautomat in einen undefinierten Zustand gerät, wenn einer der beiden Automaten eine Eingabe erhält und nicht weiß, in welchen Zustand er auf diese Eingabe hin wechseln soll.

Wir wollen diese Regel für Zustandsänderungen präzisieren und dazu annehmen, der Produktautomat befände sich im Zustand (*i*, *x*). Dieser Zustand entspricht der

Situation, in der sich die Bank im Zustand *i* und das Geschäft im Zustand *x* befindet. Nun sei *Z* eine der Eingaben. Wir betrachten den Automaten »Bank« und prüfen, ob es eine Zustandsänderung vom Zustand *i* ausgehend mit der Beschriftung *Z* gibt. Angenommen, es gäbe sie und sie führt zum Zustand *j* (der mit *i* identisch sein kann, wenn die Bank auf die Eingabe *Z* hin den aktuellen Zustand beibehält). Dann sehen wir uns den Automaten »Geschäft« an und stellen fest, dass es einen Pfeil mit der Beschriftung *Z* gibt, der zum Zustand *y* führt. Sind *j* und *y* vorhanden, dann verfügt der Produktautomat über einen Pfeil, der vom Zustand (*i*, *x*) zum Zustand (*j*, *y*) führt und die Beschriftung *Z* trägt. Ist einer der beiden Zustände *j* oder *y* nicht vorhanden (weil die Bank oder das Geschäft für die Eingabe *Z* keinen Pfeil zum Zustand *j* bzw. *y* aufweist), dann gibt es keinen vom Zustand (*i*, *x*) ausgehenden Pfeil mit der Beschriftung *Z*.

Abbildung 2.3: Produktautomat für Bank und Geschäft



Nun ist verständlich, wie die in Abbildung 2.3 dargestellten Pfeile gewählt wurden. Auf die Eingabe *Zahlen* hin wechselt das Geschäft beispielsweise vom Zustand *a* in den Zustand *b*, ändert seinen Zustand jedoch nicht, falls es sich in irgendeinem anderen Zustand als *a* befindet. Die Bank behält ihren aktuellen Zustand bei, wenn die Eingabe *Zahlen* lautet, da diese Eingabe für die Bank irrelevant ist. Diese Beobachtung erklärt die vier Pfeile mit der Beschriftung *Z* am linken Rand der vier Zeilen von Abbildung 2.3 und die Schleifen mit der Beschriftung *Z*, die sich an den übrigen Zustandsknoten befinden.

Wir wollen die Auswahl der Pfeile an einem anderen Beispiel betrachten, nämlich anhand der Eingabe *Einlösen* . Erhält die Bank eine Aufforderung zum *Einlösen* , wenn sie sich im Zustand 1 befindet, dann wechselt sie in den Zustand 3. Befindet sie sich im Zustand 3 oder 4, dann ändert sie ihren Zustand nicht. Sie gerät jedoch in einen undefinierten Zustand, wenn sie sich bei der Eingabe *Einlösen* im Zustand 2 befindet, d. h. hier sind keine Folgezustände verfügbar. Das Geschäft kann dagegen Zustandswechsel von *b* nach *d* oder von *c* nach *e* durchführen, wenn es die Eingabe *Einlösen* erhält. Wir finden sechs Pfeile mit der Beschriftung *Einlösen* in Abbildung 2.3, die den sechs Kombinationen entsprechen, die sich aus den drei Zuständen der Bank und den beiden Zuständen des Geschäfts ergeben, die drei nach außen gerichtete Pfeile mit der Beschriftung *E* aufweisen. Vom Zustand (1, *b*) ausgehend führt der Pfeil mit der Beschriftung *E* beispielsweise zum Zustand (3, *d*), da die Eingabe *Einlösen* bei der Bank eine Zustandsänderung von 1 nach 3 und beim Geschäft von *b* nach *d* bewirkt. Ein anderes Beispiel ist der Pfeil mit der Beschriftung *E*, der von (4, *c*) nach (4, *e*) führt, da die Eingabe *Einlösen* die Bank vom Zustand 4 wieder in den Zustand 4 und das Geschäft vom Zustand *c* in den Zustand *e* versetzt.

2.1.5 Mithilfe des Produktautomaten die Gültigkeit des Protokolls überprüfen

Abbildung 2.3 können wir einige interessante Dinge entnehmen. Beispielsweise sind von den 28 Zuständen nur zehn vom Startzustand (1, *a*) aus erreichbar, der sich aus der Kombination der Startzustände des Bank- und des Geschäftsautomaten ergibt. Beachten Sie, dass Zustände wie (2, *e*) und (4, *d*) nicht *erreichbar* sind, d. h. es führt kein Pfad vom Startzustand zu diesen Zuständen. Nicht erreichbare Zustände müssen nicht in den Automaten aufgenommen werden. Wir haben es in diesem Beispiel nur deswegen getan, um systematisch zu sein.

Eigentlich soll die Analyse eines Protokolls wie diesem mithilfe eines Automaten jedoch dem Zweck dienen, Fragen zu stellen und zu beantworten, wie z.B.: »Kann die folgende Art von Fehler auftreten?« Im vorliegenden Beispiel können wir fragen, ob es möglich ist, dass das Geschäft Waren liefert und diese nie bezahlt werden. Das heißt, kann der Produktautomat in einen Zustand geraten, in dem das Geschäft Waren geliefert hat (d. h. der Automat befindet sich in einem Zustand aus Spalte *c*, *e* oder *g*) und kein Übergang nach der Eingabe *Ü* erfolgt ist oder erfolgen wird?

Im Zustand (3, *e*) zum Beispiel sind die Waren geliefert worden, aber es wird auf Grund der Eingabe *Ü* schließlich einen Übergang in den Zustand (4, *g*) geben. Wenn sich die Bank im Zustand 3 befindet, dann hat sie die Aufforderung zum *Einlösen* erhalten und diese Operation bereits ausgeführt. Das bedeutet, dass sie sich im Zustand 1 befunden haben muss, bevor sie die Aufforderung zum *Einlösen* empfangen hat, und folglich hatte sie keine Aufforderung zum *Löschen* empfangen und wird diese ignorieren, wenn sie sie in Zukunft empfängt. Infolgedessen wird die Bank dem Geschäft das Geld schließlich überweisen.

Der Zustand (2, *c*) ist allerdings problematisch. Dieser Zustand ist erreichbar, aber der einzige Pfeil führt zurück zu diesem Zustand. Dieser Zustand repräsentiert die Situation, in der die Bank eine Aufforderung zum *Löschen* vor der Aufforderung zum *Einlösen* erhielt. Das Geschäft empfing jedoch eine Aufforderung zum *Zahlen*; d. h. der Kunde hat betrogen und dasselbe Geld sowohl ausgegeben als auch gelöscht. Das Geschäft lieferte die Waren dummerweise, bevor es versuchte, das Geld einzulösen,

und wenn es die Operation *Einlösen* auszuführen versucht, wird die Bank die Aufforderung nicht einmal wahrnehmen, weil sie sich im Zustand 2 befindet, in dem das Geld gelöscht wurde und die Bank die Aufforderung zum *Einlösen* nicht ausführt.

2.2 Deterministische endliche Automaten

Nun ist es an der Zeit, das formale Konzept eines Automaten vorzustellen, damit wir einige der informellen Argumente und Beschreibungen aus den Abschnitten 1.1.1 und 2.1. weiter präzisieren können. Wir beginnen mit der Vorstellung des Formalismus eines deterministischen endlichen Automaten, der sich nach dem Lesen einer Folge von Eingaben in einem einzigen Zustand befindet. Der Begriff »deterministisch« bezieht sich auf die Tatsache, dass der Automat nach jeder Eingabe von seinem aktuellen Zustand aus in genau einen Zustand übergehen kann. Im Gegensatz dazu können sich »nichtdeterministische« Automaten, die in Abschnitt 2.3 behandelt werden, gleichzeitig in mehreren Zuständen befinden. Der Begriff »endlicher Automat« wird für die deterministische Variante verwendet, obwohl wir normalerweise explizit den Begriff »deterministisch« oder die Abkürzung *DEA* verwenden werden, um dem Leser zu verdeutlichen, von welcher Art von Automaten die Rede ist.

2.2.1 Definition eines deterministischen endlichen Automaten

Ein *deterministischer endlicher Automat* besteht aus:

1. einer endlichen Menge von *Zuständen* , die meist durch Q bezeichnet wird.
2. einer endlichen Menge von *Eingabesymbolen*, die meist durch Σ repräsentiert wird.
3. einer *Übergangsfunktion* , der ein Zustand und ein Eingabesymbol als Argumente übergeben werden und die einen Zustand zurückgibt. Die Übergangsfunktion wird in der Regel mit δ angegeben. In unserer informellen grafischen Darstellung der Automaten wurde δ durch die gerichteten Pfeile zwischen Zuständen und die Beschriftungen dieser Pfeile repräsentiert. Wenn q ein Zustand und a ein Eingabesymbol ist, dann ist $\delta(q, a)$ der Zustand p, derart dass es einen mit a beschrifteten Pfeil von q nach p gibt.[2]
4. einem *Startzustand*, bei dem es sich um einen der in Q enthaltenen Zustände handelt.
5. einer Menge F finaler oder akzeptierender Zustände. Die Menge F ist eine Teilmenge von Q.

Für deterministische oder endliche Automaten wird häufig das Akronym *DEA* verwendet. Die prägnanteste Repräsentation eines DEA besteht in der Auflistung der fünf oben genannten Komponenten. In Beweisen werden wir einen DEA häufig in der »Tupel«-Notation angeben:

$$A = (Q, \Sigma, \delta, q_0, F)$$

2. Genauer gesagt, der Graph ist die bildliche Darstellung einer Übergangsfunktion δ und die Pfeile des Graphen sind so aufgebaut, dass sie die durch δ angegebenen Übergänge widerspiegeln.

wobei A für den Namen des DEA, Q für die Menge der Zustände, Σ für die Menge der Eingabesymbole, δ für die Übergangsfunktion, q_0 für den Startzustand und F für die Menge der akzeptierenden Zustände steht.

2.2.2 Wie ein DEA Zeichenreihen verarbeitet

Eines der ersten Dinge, die wir über einen DEA wissen müssen, ist, wie der DEA entscheidet, ob er eine Folge von Eingabesymbolen »akzeptiert«. Die »Sprache« des DEA besteht aus der Menge aller Zeichenreihen, die der DEA akzeptiert. Angenommen, $a_1a_2 \ldots a_n$ sei eine Folge von Eingabesymbolen. Wir gehen davon aus, dass sich der DEA A in seinem Startzustand q_0 befindet. Wir ermitteln mithilfe der Übergangsfunktion δ den Zustand $\delta(q_0, a_1) = q_1$, in den der DEA A nach der Verarbeitung des Eingabesymbols a_1 wechselt. Wir verarbeiten das nächste Eingabesymbol a_2, indem wir $\delta(q_1, a_2)$ berechnen, und nehmen an, diese Auswertung ergibt den Zustand q_2. Wir fahren auf diese Weise fort und ermitteln die Zustände $q_3, q_4 \ldots q_n$, wobei für jedes i gilt $\delta(q_{i-1}, a_i) = q_i$. Wenn q_n ein Element von F ist, dann wird die Eingabe $a_1a_2 \ldots a_n$ akzeptiert, andernfalls wird sie »zurückgewiesen«.

Beispiel 2.1 Wir wollen formal einen DEA definieren, der alle aus Nullen und Einsen bestehenden Zeichenreihen akzeptiert, die die Folge 01 enthalten, und keine anderen Zeichenreihen. Wir können diese Sprache L wie folgt formal beschreiben:

$$\{w \mid w \text{ hat die Gestalt } x01y \text{ und } x \text{ und } y \text{ bestehen ausschließlich aus Nullen und Einsen}\}$$

Eine äquivalente Beschreibung, in der die Parameter x und y links vom Längsstrich stehen, lautet:

$$\{x01y \mid x \text{ und } y \text{ sind beliebige Zeichenreihen aus Nullen und Einsen}\}$$

Beispiele für Zeichenreihen dieser Sprache sind unter anderem 01, 11010 und 100011. Beispiele für nicht in dieser Sprache enthaltene Zeichenreihen sind ε, 0 und 111000.

Was wissen wir über den Automaten A, der diese Sprache akzeptieren kann? Erstens lautet sein Eingabealphabet $\Sigma = \{0, 1\}$. Er verfügt über eine Zustandsmenge Q, die einen Startzustand q_0 enthält. Dieser Automat muss sich die wichtigen Fakten über die bislang gelesenen Eingabesymbole merken können. Um entscheiden zu können, ob 01 eine Teilzeichenreihe der Eingabe ist, muss A die folgenden Fragen beantworten und die Antworten im Gedächtnis behalten können:

1. Hat er die Zeichenreihe 01 bereits gelesen? Falls ja, dann akzeptiert er jede weitere Eingabefolge; d. h. er befindet sich ab diesem Zeitpunkt nur in akzeptierenden Zuständen.
2. Hat er die Folge 01 noch nicht gelesen, aber als letzte Eingabe 0 erhalten, sodass er ab dem jetzigen Zeitpunkt alles akzeptieren muss, wenn er als nächste Eingabe eine Eins liest?
3. Hat er die Folge 01 noch nicht gelesen, aber als letzte Eingabe nichts (er befindet sich im Startzustand) oder eine Eins gelesen? In diesem Fall kann A erst dann Eingaben akzeptieren, wenn er eine Null und unmittelbar nachfolgend eine Eins liest.

Diese drei Bedingungen lassen sich jeweils durch einen Zustand repräsentieren. Bedingung (3) wird durch den Zustand q_0 beschrieben. Wenn sich der Automat im Startzustand befindet, muss er zuerst eine Null und dann eine Eins lesen, um in den akzeptierenden Zustand zu kommen. Wenn der Automat im Startzustand q_0 eine Eins liest, muss er also im Zustand q_0 verbleiben. Das heißt, $\delta(q_0, 1) = q_0$.

Wenn sich der Automat jedoch im Zustand q_0 befindet und als Nächstes eine Null liest, dann ist die Bedingung (2) erfüllt. Das heißt, es wurde zwar noch keine Eins gelesen, aber die Null liegt bereits vor. Folglich können wir mit q_2 die Bedingung (2) beschreiben. Vom Zustand q_0 und der Eingabe 0 ausgehend, wird der Übergang beschrieben durch $\delta(q_0, 0) = q_2$.

Sehen wir uns nun die Übergänge von q_2 an. Wenn eine Null gelesen wird, bleibt die Situation unverändert bestehen: Die Folge 01 lag zwar noch nicht vor, aber 0 ist das zuletzt gelesene Eingabesymbol, und der Automat wartet weiterhin auf eine Eins. Der Zustand q_2 beschreibt diese Situation genau, sodass gelten soll: $\delta(q_2, 0) = q_2$. Wenn sich der Automat im Zustand q_2 befindet und die Eingabe 1 liest, dann weiß er, dass die Eingabefolge 01 vorliegt. Daraufhin kann er in einen akzeptierenden Zustand wechseln, den wir hier q_1 nennen und der der oben beschriebenen Bedingung (1) entspricht. Das heißt: $\delta(q_2, 1) = q_1$.

Schließlich müssen wir die Übergänge für den Zustand q_1 entwerfen. Wenn sich der Automat in diesem Zustand befindet, hat er bereits die Eingabefolge 01 gelesen. Der Automat muss, ungeachtet aller weiteren Eingaben, in diesem Zustand verbleiben, in dem die Eingabefolge 01 als gelesen gilt. Das heißt: $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.

Daraus ergibt sich $Q = \{q_0, q_1, q_2\}$. Wie angegeben ist q_0 der Startzustand, und q_1 ist der einzige akzeptierende Zustand; folglich ist $F = \{q_1\}$. Die vollständige Spezifikation des Automaten, der die Zeichenreihen der Sprache L akzeptiert, die die Teilzeichenreihe 01 enthalten, lautet:

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

wobei δ für die oben beschriebene Übergangsfunktion steht. ■

2.2.3 Einfachere Notationen für DEAs

Die Spezifikation eines DEA als Tupel mit einer detaillierten Beschreibung der Übergangsfunktion δ ist sowohl mühsam als auch schwer lesbar. Es gibt zwei andere Notationen zur Beschreibung von Automaten, die der Tupel-Notation vorgezogen werden:

1. *Übergangsdiagramme*, wobei es sich um Graphen handelt, wie die in Abschnitt 2.1 gezeigten.
2. *Übergangstabellen*, d. h. tabellarische Auflistungen der Übergangsfunktion δ, die implizit Aufschluss über die Zustandsmenge und das Eingabealphabet geben.

Übergangsdiagramme

Ein *Übergangsdiagramm* für einen DEA $A = (Q, \Sigma, \delta, q_0, F)$ ist ein Graph, der wie folgt definiert ist:

a) Jeder in Q enthaltene Zustand wird durch einen Knoten dargestellt.

b) Für jeden Zustand q aus Q und für jedes Eingabesymbol a aus Σ sei $\delta(q, a) = p$. Das Übergangsdiagramm enthält dann einen Pfeil, der von Knoten q zu Knoten p

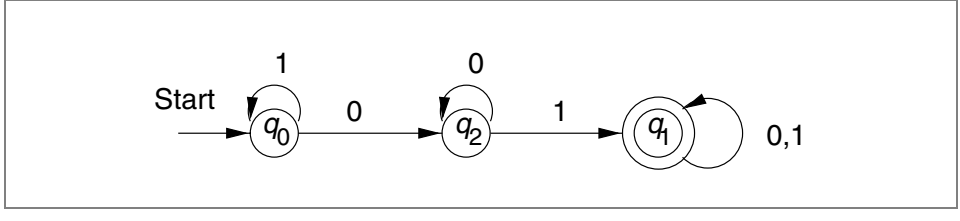
führt und mit a beschriftet ist. Falls es mehrere Eingabesymbole gibt, die einen Übergang von q nach p bewirken, kann dies im Übergangsdiagramm durch einen mit der Liste dieser Eingabesymbole beschrifteten Pfeil dargestellt werden.

c) Ein mit *Start* beschrifteter Pfeil zeigt auf den Startzustand q_0. Dieser Pfeil beginnt in keinem Knoten.

d) Knoten, die akzeptierenden Zuständen entsprechen (die in F enthalten sind), werden durch eine doppelte Kreislinie gekennzeichnet. Nicht in F enthaltene Zustände werden durch einen einfachen Kreis dargestellt.

Beispiel 2.2 Abbildung 2.4 zeigt das Übergangsdiagramm für den DEA, der im Beispiel 2.1 entworfen wurde. Wir sehen in diesem Diagramm die drei Knoten für die drei Zustände. Es gibt einen Pfeil mit der Beschriftung *Start*, der auf den Startzustand q_0 weist, und der einzige akzeptierende Zustand q_1 wird durch eine doppelte Kreislinie dargestellt. Von jedem Zustand geht ein Pfeil mit der Beschriftung 0 und ein Pfeil mit der Beschriftung 1 aus. Beim Zustand q_1 sind diese Pfeile in einem Pfeil mit der Beschriftung 0, 1 zusammengefasst. Diese Pfeile entsprechen den in Beispiel 2.1 beschriebenen δ-Fakten. ■

Abbildung 2.4: Übergangsdiagramm für den DEA, der alle Zeichenreihen akzeptiert, die die Teilzeichenreihe 01 enthalten



Übergangstabellen

Eine *Übergangstabelle* ist eine konventionelle tabellarische Darstellung einer Funktion wie δ, die zwei Argumente verarbeitet und einen Wert zurückgibt. Die Zeilen der Tabelle entsprechen den Zuständen und die Spalten den Eingaben. Der Eintrag in der Zeile für den Zustand q und der Spalte für die Eingabe a entspricht dem Zustand $\delta(q, a)$.

Beispiel 2.3 Die Übergangstabelle für die Funktion δ aus dem Beispiel 2.1 ist in Tabelle 2.1 dargestellt. Dort werden zudem zwei weitere Merkmale einer Übergangstabelle gezeigt. Der Startzustand ist durch einen Pfeil und der akzeptierende Zustand durch einen Stern gekennzeichnet. Da wir aus den Spalten- und Zeilentiteln die Zustandsmenge und die Menge der Eingabesymbole ableiten können, können wir der Übergangstabelle alle Informationen entnehmen, die wir zur eindeutigen Spezifikation eines endlichen Automaten benötigen. ■

Tabelle 2.1: Übergangstabelle für den DEA aus Beispiel 2.1

	0	1
$\rightarrow q_0$	q_2	q_0
$* q_1$	q_1	q_1
q_2	q_2	q_1

2.2.4 Die Übergangsfunktion auf Zeichenreihen erweitern

Wir haben informell erklärt, dass der DEA eine Sprache definiert: die Menge aller Zeichenreihen, die in einer Folge von Zustandsänderungen vom Startzustand zu einem akzeptierenden Zustand resultieren. In Übergangsdiagrammen wird die Sprache eines DEA durch die Menge der Beschriftungen aller Pfade dargestellt, die vom Startzustand zu einem akzeptierenden Zustand führen.

Wir müssen diese Vorstellung von der Sprache eines DEA nun präzisieren. Zu diesem Zweck definieren wir eine *erweiterte Übergangsfunktion*, die beschreibt, was vorgeht, wenn wir von einem beliebigen Knoten ausgehen und einer beliebigen Folge von Eingaben folgen. Wenn δ die Übergangsfunktion ist, dann wird die aus δ gebildete erweiterte Übergangsfunktion $\hat{\delta}$ genannt. Die erweiterte Funktion ist eine Funktion, der ein Zustand q und eine Zeichenreihe w als Argumente übergeben und die einen Zustand q zurückgibt, d. h. den Zustand, den der Automat erreicht, wenn er ausgehend von Zustand q die Eingabefolge w verarbeitet. Wir definieren $\hat{\delta}$ wie folgt durch Induktion über die Länge der Eingabezeichenreihe:

INDUKTIONSBEGINN: $\hat{\delta}(q, \varepsilon) = q$. Das heißt, wenn sich der Automat im Zustand q befindet und keine Eingabe liest, dann verbleibt er im Zustand q.

INDUKTIONSSCHRITT: Angenommen, w ist eine Zeichenreihe der Form xa, wobei a das letzte Symbol von w und x die Zeichenreihe ist, die aus allen Eingabesymbolen mit Ausnahme des letzten Symbols besteht.[3] Beispielsweise wird $w = 1101$ aufgeteilt in $x = 110$ und $a = 1$. Dann gilt:

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a) \tag{2.1}$$

(2.1) mag kompliziert aussehen, der zu Grunde liegende Gedanke ist jedoch einfach. Zur Berechnung von $\hat{\delta}(q, w)$ berechnen wir zuerst $\hat{\delta}(q, x)$, den Zustand, in dem sich der Automat befindet, nachdem er alle Eingabesymbole, mit Ausnahme des letzten, verarbeitet hat. Angenommen, dieser Zustand hieße p; dann gilt $\hat{\delta}(q, x) = p$. Folglich stellt $\hat{\delta}(q, w)$ den Zustand dar, der durch den Übergang von Zustand p auf die Eingabe a (dem letzten Symbol von w) hin erreicht wird. Das heißt: $\hat{\delta}(q, w) = \delta(p, a)$.

3. Sie erinnern sich an unsere Konvention, dass Buchstaben vom Anfang des Alphabets für Symbole und Buchstaben vom Ende des Alphabets für Zeichenreihen stehen. Wir brauchen diese Konvention, damit der Ausdruck »der Form xa« sinnvoll interpretiert werden kann.

Beispiel 2.4 Wir wollen einen DEA entwerfen, der die Sprache L akzeptiert, wobei gilt:

$$L = \{ w \mid w \text{ enthält eine gerade Anzahl von Nullen und eine gerade Anzahl von Einsen } \}$$

Es sollte nicht überraschen, dass die Zustände dieses DEA dazu dienen sollen, die Anzahl der Nullen und Einsen modulo 2 zu zählen. Das heißt, im Zustand wird gespeichert, ob bislang eine gerade oder ungerade Anzahl von Nullen oder Einsen gelesen wurde. Folglich gibt es vier Zustände, die wie folgt interpretiert werden können:

q_0: Es wurde eine gerade Anzahl von Nullen und eine gerade Anzahl von Einsen gelesen.

q_1: Es wurde eine gerade Anzahl von Nullen, aber eine ungerade Anzahl von Einsen gelesen.

q_2: Es wurde eine ungerade Anzahl von Nullen, aber eine gerade Anzahl von Einsen gelesen.

q_3: Sowohl die Anzahl der bislang gelesenen Nullen als auch die Anzahl der bislang gelesenen Einsen ist ungerade.

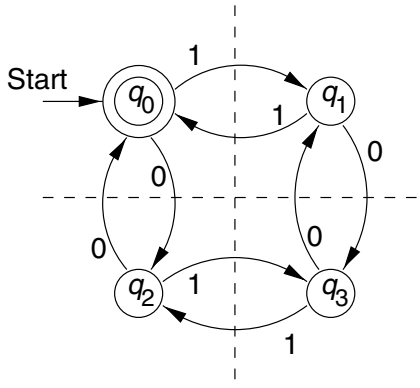
Zustand q_0 ist der Startzustand und gleichzeitig der einzige akzeptierende Zustand. Es handelt sich um den Startzustand, weil vor dem Lesen von Eingabesymbolen die Anzahl der bislang gelesenen Nullen und Einsen null ist und null eine gerade Zahl ist. Zustand q_0 ist der einzige akzeptierende Zustand, weil er genau die Bedingung beschreibt, die eine Folge von Nullen und Einsen erfüllen muss, um der Sprache L anzugehören.

Wir wissen jetzt beinahe, wie sich der DEA für die Sprache L definieren lässt. Er wird angegeben mit:

$$A = \{(q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

wobei die Übergangsfunktion δ durch das in Abbildung 2.5 dargestellte Übergangsdiagramm beschrieben wird. Beachten Sie, dass jede Eingabe einer Null bewirkt, dass ein Zustandswechsel über die gestrichelte horizontale Linie hinweg erfolgt. Nachdem eine gerade Anzahl von Nullen gelesen wurde, wird stets ein Zustand oberhalb der horizontalen Linie – q_0 oder q_1 – erreicht, während der Zustand q_2 oder q_3, die unterhalb der horizontalen Linie dargestellt sind, erreicht wird, nachdem eine ungerade Anzahl von Nullen gelesen wurde. Ebenso bewirkt jede Eingabe einer Eins, dass ein Zustandswechsel über die gestrichelte vertikale Linie hinweg erfolgt. Nachdem eine gerade Anzahl von Einsen gelesen wurde, befindet sich der Automat im Zustand q_0 oder q_2, die links von der vertikalen Linie dargestellt sind, während er sich nach der Eingabe einer ungeraden Anzahl von Einsen im Zustand q_1 oder q_3 befindet, die rechts von der vertikalen Linie dargestellt sind. Diese Beobachtungen stellen einen informellen Beweis dafür dar, dass für die vier Zustände die ihnen zugeordneten Interpretationen gelten. Man könnte die Richtigkeit der vier Behauptungen über die Zustände jedoch auch formal durch gegenseitige Induktion im Sinn von Beispiel 1.23 beweisen. Wir können diesen DEA auch durch eine Übergangstabelle repräsentieren. Tabelle 2.2 zeigt diese Tabelle. Uns geht es jedoch nicht nur um den Entwurf dieses DEA, sondern wir möchten diesen DEA dazu einsetzen, die Bildung der Funktion $\hat{\delta}$ aus der Übergangsfunktion δ zu veranschaulichen. Angenommen, die Eingabe lautet 110101. Da

Abbildung 2.5: Übergangsdiagramm für den DEA aus Beispiel 2.4



diese Zeichenreihe über eine gerade Anzahl von Nullen und Einsen verfügt, erwarten wir, dass sie in der Sprache enthalten ist. Daher erwarten wir, dass $\hat{\delta}(q_0, 110101) = q_0$, denn q_0 ist der einzige akzeptierende Zustand. Wir wollen nun die Gültigkeit dieser Behauptung überprüfen.

Tabelle 2.2: Übergangstabelle für den DEA aus Beispiel 2.4

	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Hierzu müssen wir $\hat{\delta}(q_0, w)$ für jedes Präfix w von 110101 berechnen, wobei wir mit ε beginnen und das Präfix stetig vergrößern. Diese Berechnung lässt sich wie folgt zusammenfassen:

- $\hat{\delta}(q_0, \varepsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \varepsilon), 1) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_{310}$
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = q_1$
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$ ■

Standardnotation und lokale Variablen

Nachdem Sie diesen Abschnitt gelesen haben, nehmen Sie vielleicht an, die hier verwendete Notation sei zwingend vorgeschrieben, also dass δ für die Übergangsfunktion, A für den Namen des DEA verwendet werden muss etc. Wir verwenden in unseren Beispielen für gewöhnlich die gleichen Variablen zur Darstellung der gleichen Dinge, da man sich dann leichter an den Typ der Variablen erinnern kann, ähnlich wie in Programmiersprachen die Variable *i* meist für den Typ *Integer* verwendet wird. Es steht uns allerdings frei, die Komponenten eines Automaten und alles Übrige beliebig zu benennen. Folglich steht es auch Ihnen frei, einen DEA M und seine Übergangsfunktion $\ddot{U}$ zu nennen, wenn Sie möchten.

Sie sollten überdies nicht überrascht sein, dass eine Variable in verschiedenen Kontexten verschiedene Bedeutungen hat. Beispielsweise wurde in den Beispielen 2.1 und 2.4 die Übergangsfunktion des DEA jeweils δ genannt. Die beiden Übergangsfunktionen sind jedoch lokale Variablen, die lediglich in dem jeweiligen Beispiel gültig sind. Diese beiden Übergangsfunktionen sind sehr verschieden und stehen in keiner Beziehung zueinander.

2.2.5 Die Sprache eines DEA

Wir können jetzt die Sprache eines DEA $A = \{Q, \Sigma, \delta, q_0, F)$ definieren. Diese Sprache wird als $L(A)$ bezeichnet und sie wird definiert durch:

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ ist in } F \text{ enthalten}\}$$

Das heißt, die Sprache von A ist die Menge der Zeichenreihen w, die vom Startzustand q_0 in einen akzeptierenden Zustand führen. Wenn L für einen DEA A $L(A)$ ist, dann bezeichnen wir L als reguläre Sprache.

Beispiel 2.5 Wie an früherer Stelle erwähnt, gilt, wenn A der DEA aus Beispiel 2.1 ist, dann ist $L(A)$ die Menge aller aus Nullen und Einsen bestehenden Zeichenreihen, die die Teilzeichenreihe 01 enthalten. Ist A dagegen der DEA aus dem Beispiel 2.4, dann ist $L(A)$ die Menge aller aus Einsen und Nullen bestehenden Zeichenreihen, die jeweils eine gerade Anzahl von Einsen und Nullen enthalten. ■

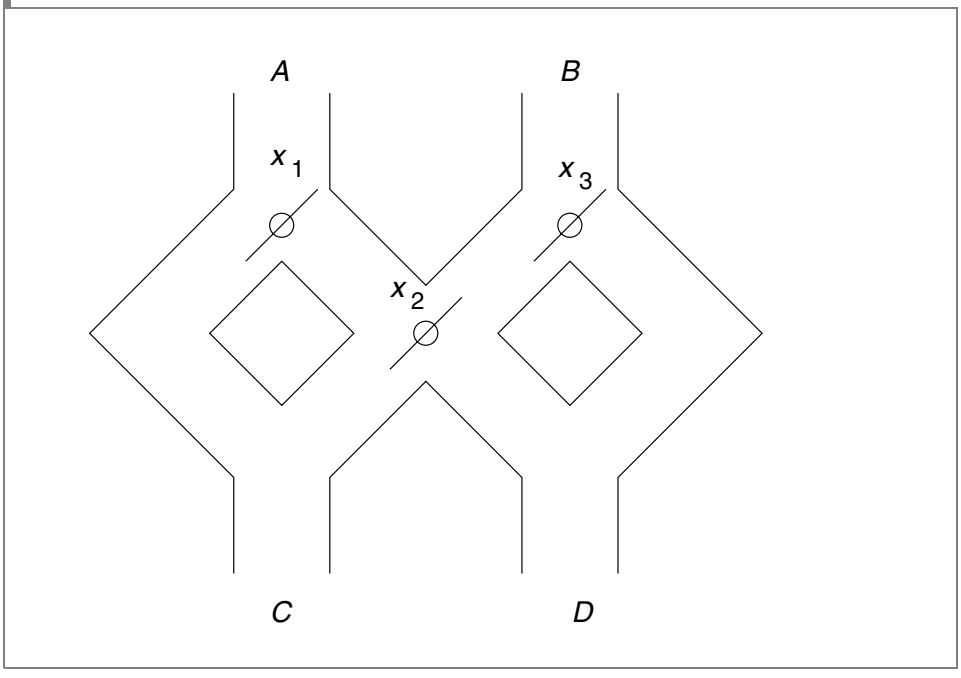
2.2.6 Übungen zum Abschnitt 2.2

Übung 2.2.1 Abbildung 2.6 zeigt ein Spiel, in dem Murmeln in Bahnen rollen. Eine Murmel wird bei A oder B in die Spielbahn fallen gelassen. Die Hebel x_1, x_2 und x_3 an den Verzweigungen bewirken, dass die Murmel nach links oder rechts rollt. Sobald eine Murmel auf einen dieser Hebel trifft, wird der Hebel nach dem Passieren der Murmel umgestellt, sodass die nächste Murmel in die andere Richtung rollt.

* a) Modellieren Sie dieses Spiel als endlichen Automaten. Die Eingaben A und B sollen die Öffnungen repräsentieren, in die man die Murmel fallen lässt. Wenn die Murmel durch Öffnung D aus dem Spiel rollt, dann soll dies als akzeptierender Zustand gelten. Als nicht akzeptierender Zustand soll gelten, wenn die Murmel durch C aus dem Spiel rollt.

! b) Beschreiben Sie die Sprache des Automaten informell.

Abbildung 2.6: Murmelspiel



c) Angenommen, die Hebel werden umgestellt, *bevor* die Murmel passieren kann. Wie würde sich diese Änderung auf Ihre Antworten zu den Teilen a) und b) auswirken?

***!** **Übung 2.2.2:** Wir definierten $\hat{\delta}$, indem wir die Eingabezeichenreihe in eine beliebige Zeichenreihe, der ein einzelnes Symbol folgt, aufgeteilt haben (im Induktionsteil, Gleichung 2.1). Wir stellen uns informell jedoch vor, dass $\hat{\delta}$ beschreibt, was entlang eines Pfads mit einer bestimmten Zeichenreihe passiert, und wenn diese Vorstellung korrekt ist, dann sollte es gleichgültig sein, wie wir die Eingabezeichenreihe in der Definition von $\hat{\delta}$ unterteilen. Zeigen Sie, dass in der Tat für jeden Zustand q und für alle Zeichenreihen x und y gilt: $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q,x),y)$. *Hinweis:* Führen Sie einen Induktionsbeweis über $|y|$.

! Übung 2.2.3: Zeigen Sie, dass für jeden Zustand q, für jede Zeichenreihe x und für jedes Eingabesymbol a gilt: $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$. *Hinweis:* Verwenden Sie Übung 2.2.2.

Übung 2.2.4: Definieren Sie DEAs, die die folgenden Sprachen über dem Alphabet {0, 1} akzeptieren:

* a) Die Menge aller Zeichenreihen, die auf 00 enden

b) Die Menge aller Zeichenreihen, die drei aufeinander folgende Nullen (nicht notwendigerweise am Ende) enthalten

c) Die Menge aller Zeichenreihen, die die TeilZeichenreihe 011 enthalten

! **Übung 2.2.5:** Definieren Sie DEAs, die die folgenden Sprachen über dem Alphabet {0, 1}akzeptieren:

a) Die Menge aller Zeichenreihen, die so geformt sind, dass jeder Block aus fünf aufeinander folgenden Zeichen mindestens zwei Nullen enthält

b) Die Menge aller Zeichenreihen, deren zehntes Symbol links vom Ende eine Eins ist

c) Die Menge aller Zeichenreihen, an deren Anfang und/oder Ende die Zeichenreihe 01 steht

d) Die Menge aller Zeichenreihen, die so geformt sind, dass sie eine durch 5 teilbare Anzahl von Nullen und eine durch 3 teilbare Anzahl von Einsen enthalten

!! **Übung 2.2.6:** Definieren Sie einen DEA, der die folgenden Sprachen über dem Alphabet {0, 1} akzeptiert:

* a) Die Menge aller mit 1 beginnenden Zeichenreihen, die ein Vielfaches von 5 bilden, wenn sie als Binärdarstellung einer ganzen Zahl interpretiert werden. Beispielsweise sind die Zeichenreihen 101, 1010 und 1111 in dieser Sprache enthalten, 0, 100 und 111 dagegen nicht.

b) Die Menge aller Zeichenreihen, deren Binärdarstellung durch 5 teilbar ist, wenn sie in umgekehrter Reihenfolge gelesen als Binärdarstellung einer ganzen Zahl interpretiert werden.

Übung 2.2.7: Sei A ein DEA und q sei ein bestimmter Zustand von A, derart dass für alle Eingabesymbole a gilt: $\delta(q, a) = q$. Zeigen Sie durch einen Induktionsbeweis über die Länge der Eingabe, dass für alle Eingabezeichenreihen w gilt: $\hat{\delta}(q, w) = q$.

Übung 2.2.8: Sei A ein DEA und a ein bestimmtes Eingabesymbol von A, derart dass für alle Zustände q von A gilt: $\delta(q, a) = q$.

a) Zeigen Sie durch einen Induktionsbeweis über n, dass für alle $n = 0$ gilt: $\hat{\delta}(q, a^n) = q$, wobei a^n eine aus n Exemplaren des Symbols a bestehende Zeichenreihe ist.

b) Zeigen Sie, dass entweder $\{a\}^* \subseteq L(A)$ oder $\{a\}^* \cap L(A) = 0$.

*! **Übung 2.2.9:** Sei $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ ein DEA, und nehmen wir an, dass für alle a in Σ gilt: $\delta(q_0, a) = \delta(q_f, a)$.

a) Zeigen Sie, dass für alle $w \neq \varepsilon$ gilt: $\hat{\delta}(q_0, w) = \hat{\delta}(q_f, w)$.

b) Zeigen Sie, dass gilt: Wenn x eine nicht leere Zeichenreihe der Sprache $L(A)$ ist, dann ist x^k für alle $k > 0$ (d. h. die k-fache Wiederholung von x) auch in der Sprache $L(A)$ enthalten.

***! Übung 2.2.10:** Betrachten Sie den DEA mit der folgenden Übergangstabelle:

	0	1
→ *A*	*A*	*B*
* *B*	*B*	*A*

Beschreiben Sie informell die Sprache, die dieser DEA akzeptiert, und beweisen Sie durch einen Induktionsbeweis über die Länge der Eingabefolge, dass Ihre Beschreibung korrekt ist. *Hinweis:* Bei der Formulierung der Induktionshypothese ist es empfehlenswert, Aussagen darüber zu machen, welche Eingaben den Übergang in die verschiedenen Zustände bewirken, und sich hierbei nicht auf die Eingaben zu beschränken, die den Übergang in den akzeptierenden Zustand verursachen.

! Übung 2.2.11: Wiederholen Sie Übung 2.2.10 für die folgende Übergangstabelle:

	0	1
→ **A*	*B*	*A*
* *B*	*C*	*A*
C	*C*	*C*

2.3 Nichtdeterministische endliche Automaten

Ein »nichtdeterministischer« endlicher Automat (NEA) ist in der Lage, gleichzeitig über mehrere Zustände zu verfügen. Diese Fähigkeit wird häufig als die Fähigkeit ausgedrückt, über die Eingabe »Vermutungen anzustellen«. Wenn der Automat beispielsweise eingesetzt wird, um nach einer bestimmten Folge von Zeichen (z. B. Schlüsselwörter) in einer langen Textzeichenreihe zu suchen, dann ist es hilfreich, wenn man vermutet, dass man sich am Anfang einer dieser Zeichenreihen befindet, und eine Reihe von Zuständen dafür aufwendet, den Text Zeichen für Zeichen auf ein Vorkommen dieser Zeichenreihe zu überprüfen. Wir werden in Abschnitt 2.4 ein Beispiel für diese Art von Anwendungen kennen lernen.

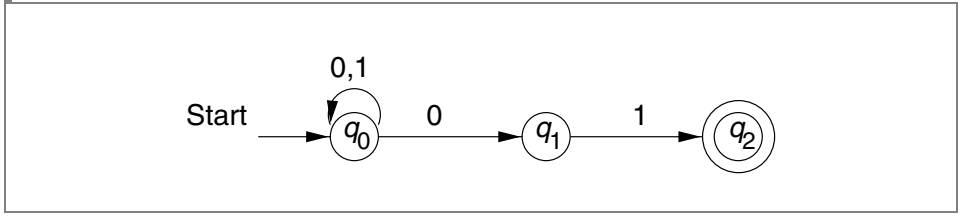
Bevor wir Anwendungen untersuchen, müssen wir nichtdeterministische endliche Automaten definieren und zeigen, dass jeder dieser Automaten eine Sprache akzeptiert, die auch von einem DEA akzeptiert wird. Das heißt, NEAs akzeptieren ebenso wie DEAs reguläre Sprachen. Es gibt jedoch Gründe, näher über NEAs nachzudenken. Sie sind häufig prägnanter und einfacher zu entwerfen als DEAs. Während wir einen NEA immer in einen DEA umwandeln können, kann der DEA dagegen eine exponentiell höhere Anzahl von Zuständen besitzen als der NEA. Glücklicherweise sind diese Art von Fällen recht selten.

2.3.1 Eine informelle Darstellung nichtdeterministischer endlicher Automaten

Ebenso wie ein DEA verfügt auch ein NEA über eine endliche Menge von Zuständen, eine endliche Menge von Eingabesymbolen, einen Startzustand und eine Menge von akzeptierenden Zuständen. Er besitzt auch eine Übergangsfunktion, die wir üblicherweise δ nennen werden. Beim NEA ist δ eine Funktion, der (ebenso wie der Übergangsfunktion eines NEA) ein Zustand und ein Eingabesymbol als Argumente übergeben werden, die jedoch eine Menge von null, einem oder mehreren Zuständen zurückgibt (statt genau einen Zustand wie der DEA). Wir werden mit einem Beispiel für einen NEA beginnen und die Definitionen dann präzisieren.

Beispiel 2.6 Abbildung 2.7 zeigt einen nichtdeterministischen Automaten, der ausschließlich alle aus Nullen und Einsen bestehenden Zeichenreihen akzeptieren soll, die mit 01 enden. Zustand q_0 ist der Startzustand, und wir können annehmen, der Automat befindet sich immer dann im Zustand q_0 (und möglicherweise anderen Zuständen), wenn er noch nicht »vermutet«, die abschließende 01-Sequenz habe begonnen. Es ist zudem möglich, dass das nächste Symbol nicht das erste Zeichen der Abschlusssequenz 01 darstellt, auch wenn dieses Symbol eine Null ist. Folglich kann der Zustand q_0 sowohl nach der Eingabe 0 als auch nach der Eingabe 1 zu sich selbst zurückführen.

Abbildung 2.7: Ein NEA, der alle mit 01 endenden Zeichenreihen akzeptiert

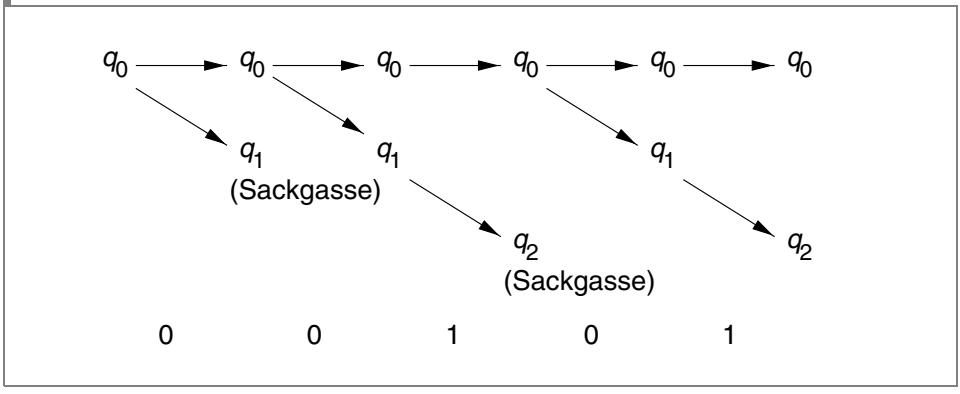


Handelt es sich bei dem nächsten Symbol jedoch um eine Null, dann vermutet dieser NEA überdies, dass die Abschlusssequenz 01 begonnen hat. Daher führt ein mit 0 beschrifteter Pfeil vom Zustand q_0 zum Zustand q_1. Beachten Sie, dass zwei Pfeile mit der Beschriftung 0 vom Zustand q_0 ausgehen. Der NEA hat die Möglichkeit, in den Zustand q_0 oder den Zustand q_1 zu wechseln, und er wird sogar in beide Zustände überführt, wie wir sehen werden, wenn wir die Definitionen präzisieren. Im Zustand q_1 prüft der NEA, ob das nächste Symbol eine Eins ist, und wenn dem so ist, geht er in den Zustand q_2 über und akzeptiert.

Beachten Sie, dass von q_1 kein Pfeil mit der Beschriftung 0 ausgeht und dass von q_2 überhaupt kein Pfeil ausgeht. In diesen Situationen führt der Pfad, der diesen Zuständen entspricht, einfach in eine Sackgasse, andere Pfade können aber durchaus noch vorhanden sein. Während bei einem DEA für jedes Eingabesymbol genau ein Pfeil von jedem Zustand ausgeht, gelten für den NEA keine Beschränkungen dieser Art. Wir haben in Abbildung 2.7 beispielsweise Fälle gezeigt, in denen die Anzahl von Pfeilen null, eins und zwei ist.

Abbildung 2.8 illustriert, wie ein NEA Eingaben verarbeitet. Wir haben dargestellt, was passiert, wenn der Automat aus Abbildung 2.7 die Eingabesequenz 00101 erhält.

Abbildung 2.8: Die Zustände, in denen sich ein NEA während der Verarbeitung der Eingabesequenz 00101 befindet



Am Beginn befindet er sich lediglich in seinem Startzustand q_0. Nachdem die erste Null gelesen wurde, kann der NEA in den Zustand q_0 oder q_1 wechseln, und daher tut er beides. Diese beiden Pfade werden durch die zweite Spalte in Abbildung 2.8 veranschaulicht.

Dann wird die zweite Null gelesen. Aus dem Zustand q_0 kann wiederum sowohl in den Zustand q_0 und q_1 gewechselt werden. Da q_1 keinen Übergang für 0 besitzt, endet dieser Pfad. Wenn das dritte Eingabesymbol (eine Eins) gelesen wird, müssen wir die Übergänge von q_0 und q_1 betrachten. Wir stellen fest, dass der Automat von q_0 bei der Eingabe 1 nur zu q_0 übergeht und dass er von q_1 nur zu q_2 übergeht. Folglich verbleibt der NEA in den Zuständen q_0 und q_2, nachdem er die Eingabe 001 gelesen hat. Da q_2 ein akzeptierender Zustand ist, akzeptiert der NEA 001.

Die Eingabe ist allerdings noch nicht abgeschlossen. Das vierte Eingabesymbol (eine Null) bewirkt, dass der Pfad von q_2 endet, während von q_0 sowohl zu q_0 als auch zu q_1 überführt wird. Das letzte Eingabesymbol (eine Eins) bewirkt den Übergang von q_0 zu q_0 und von q_1 zu q_2. Da dies wiederum ein akzeptierender Zustand ist, wird die Eingabe 00101 akzeptiert. ■

2.3.2 Definition nichtdeterministischer endlicher Automaten

Wir wollen nun die formalen Definitionen im Zusammenhang mit nichtdeterministischen endlichen Automaten (NEA) einführen. Die Unterschiede zwischen DEAs und NEAs werden an den entsprechenden Stellen aufgezeigt. Ein NEA wird im Grunde genommen wie ein DEA beschrieben:

$$A = (Q, \Sigma, \delta, q_0, F)$$

wobei

1. Q eine endliche Menge von *Zuständen* ist.
2. Σ eine endliche Menge von *Eingabesymbolen* ist.
3. q_0, ein Element von Q, der *Startzustand* ist.

4. F, eine Teilmenge von Q, die Menge der *finalen* (oder *akzeptierenden*) Zustände ist.
5. δ, die *Übergangsfunktion*, eine Funktion ist, der ein Zustand aus Q und ein Eingabesymbol aus Σ als Argumente übergeben werden und die eine Teilmenge von Q zurückgibt. Beachten Sie, dass sich ein NEA und ein DEA lediglich durch den Typ des Rückgabewerts von δ unterscheiden: Im Fall eines NEA handelt es sich um eine Menge von Zuständen und im Fall eines DEA um einen einzigen Zustand.

Beispiel 2.7 Der NEA aus Abbildung 2.7 lässt sich wie folgt formal spezifizieren:

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

Tabelle 2.3: Übergangstabelle für einen NEA, der alle auf 01 endenden Zeichenreihen akzeptiert

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_1\}$
q_1	$\varnothing$	$\{q_2\}$
$* q_2$	$\varnothing$	$\varnothing$

wobei die Übergangsfunktion δ durch die Übergangstabelle in Tabelle 2.3 angegeben wird. ■

Beachten Sie, dass Übergangstabellen zur Spezifikation der Übergangsfunktion eines NEA und eines DEA eingesetzt werden können. Unterschiedlich ist hier nur, dass die einzelnen Tabelleneinträge bei einem NEA jeweils Mengen angeben, auch wenn es sich um eine einelementige Menge handelt. Zudem ist zu beachten, dass in den Fällen, in denen bei einem bestimmten Zustand für eine gegebene Eingabe kein Übergang definiert ist, die Tabelle korrekterweise die leere Menge als Eintrag enthalten muss.

2.3.3 Die erweiterte Übergangsfunktion

Wie für DEAs müssen wir die Übergangsfunktion δ eines NEA zu einer Funktion $\hat{\delta}$ erweitern, die einen Zustand q und eine Zeichenreihe w von Eingabesymbolen verarbeitet und die Menge der Zustände zurückgibt, die der NEA annimmt, wenn er ausgehend vom Zustand q die Zeichenreihe w verarbeitet. Dieses Konzept wurde in Abbildung 2.8 illustriert. Im Grunde genommen ist $\hat{\delta}(q, w)$ die Spalte der Zustände, die nach dem Lesen von w vorkommen, wenn q der einzige Zustand in der ersten Spalte ist. Beispielsweise legt Abbildung 2.8 nahe, dass $\hat{\delta}(q_0, 001) = \{q_0, q_2\}$. Formal definieren wir $\hat{\delta}$ für die Übergangsfunktion δ eines NEA wie folgt:

Induktionsbeginn: $\hat{\delta}(q, \varepsilon) = \{q\}$. Das heißt, solange noch keine Eingabesymbole gelesen wurden, befindet sich der Automat in seinem Anfangszustand.

INDUKTIONSSCHRITT: Angenommen, w hat die Form xa, wobei a das letzte Symbol von w repräsentiert und x die Anfangszeichenreihe von w bis auf a. Nehmen wir weiter an, dass $\hat{\delta}(q, x) = \{p_1, p_2, ..., p_k\}$. Sei

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$$

Dann gilt: $\hat{\delta}(q, w) = \{r_1, r_2, ..., r_m\}$. Weniger formal ausgedrückt, wir berechnen $\hat{\delta}(q, w)$, indem wir zuerst $\hat{\delta}(q, x)$ berechnen und dann jedem Übergang von einem dieser Zustände folgen, der die Beschriftung a trägt.

Beispiel 2.8 Wir wollen mithilfe von $\hat{\delta}$ beschreiben, wie der NEA aus Abbildung 2.7 die Eingabe 00101 verarbeitet. Die einzelnen Schritte lassen sich wie folgt zusammenfassen:

1. $\hat{\delta}(q_0, \varepsilon) = \{q_0\}$.
2. $\hat{\delta}(q_0, 0) = \delta\{q_0, 0\} = \{q_0, q_1\}$.
3. $\hat{\delta}(q_0, 00) = \delta\{q_0, 0\} \cup \delta\{q_1, 0\} = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$.
4. $\hat{\delta}(q_0, 001) = \delta\{q_0, 1\} \cup \delta\{q_1, 1\} = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.
5. $\hat{\delta}(q_0, 0010) = \delta\{q_0, 0\} \cup \delta\{q_2, 0\} = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$.
6. $\hat{\delta}(q_0, 00101) = \delta\{q_0, 1\} \cup \delta\{q_1, 1\} = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

Zeile (1) enthält die Grundregel. Wir erhalten Zeile (2), indem wir δ auf den einzigen Zustand der vorherigen Menge – q_0 – anwenden, woraus sich $\{q_0, q_1\}$ ergibt. Zeile (3) gewinnen wir, indem wir δ mit der Eingabe 0 auf die beiden in der vorherigen Menge enthaltenen Zustände anwenden und die Ergebnisse vereinigen. Das heißt, $\delta\{q_0, 0\} = \{q_0, q_1\}$ und $\delta\{q_1, 0\} = \varnothing$. Um das Ergebnis in Zeile (4) zu erhalten, vereinigen wir $\delta\{q_0, 1\} = \{q_0\}$ und $\delta\{q_1, 1\} = \{q_2\}$. Wir erhalten die Zeilen (5) und (6) in ähnlicher Weise wie die Zeilen (3) und (4). ■

2.3.4 Die Sprache eines NEA

Wir haben behauptet, ein NEA akzeptiert eine Zeichenreihe w, wenn der Automat während des Lesens der in w enthaltenen Zeichen von einem Startzustand in einen akzeptierenden Zustand übergehen kann, wobei andere Zustandsübergänge nicht ausgeschlossen sind. Die Tatsache, dass die Eingabesymbole von w andere Zustandsübergänge bewirken können, die zu keinem akzeptierenden Zustand oder zu gar keinem Zustand (der Pfad der Zustandsänderungen »endet«) führen, hat nicht zur Folge, dass der NEA w als Ganzes nicht akzeptiert. Formal ausgedrückt, wenn $A = (Q, \Sigma, \delta, q_0, F)$ ein NEA ist, dann gilt:

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$

Das heißt, $L(A)$ ist die Menge aller Zeichenreihen w aus Σ^*, derart dass $\hat{\delta}(q_0, w)$ mindestens einen akzeptierenden Zustand enthält.

Beispiel 2.9 Wir wollen als Beispiel formal beweisen, dass der NEA aus Abbildung 2.7 die Sprache $L = \{w \mid w \text{ endet mit } 01\}$ akzeptiert. Der Beweis ist eine gegenseitige Induktion der folgenden drei Aussagen, die die drei Zustände charakterisieren:

1. $\hat{\delta}(q_0, w)$ enthält für jede Zeichenreihe w den Zustand q_0.
2. $\hat{\delta}(q_0, w)$ enthält genau dann den Zustand q_1, wenn die Zeichenreihe w mit 0 endet.
3. $\hat{\delta}(q_0, w)$ enthält genau dann den Zustand q_2, wenn die Zeichenreihe w mit 01 endet.

Zum Beweis dieser Aussagen müssen wir betrachten, wie A die einzelnen Zustände erreichen kann, d. h. welches Eingabesymbol zuletzt gelesen wurde und in welchem Zustand sich A befand, bevor dieses Symbol gelesen wurde.

Da die Sprache dieses Automaten die Menge an Zeichenreihen w umfasst, die so geformt sind, dass $\hat{\delta}(q_0, w)$ q_2 enthält (weil q_2 der einzige akzeptierende Zustand ist), garantiert der Beweis dieser drei Aussagen (insbesondere der Beweis der Aussage (3)), dass es sich bei der Sprache dieses NEA um die Menge aller mit 01 endenden Zeichenreihen handelt. Der Beweis dieses Satzes besteht aus einem Induktionsbeweis über die Länge von w ($|w|$), wobei mit der Länge 0 begonnen wird.

Induktionsbeginn: Wenn $|w| = 0$, dann $w = \varepsilon$. Aussage (1) besagt, dass $\hat{\delta}(q_0, w)$ den Zustand q_0 enthält, was auf Grund der Definition von $\hat{\delta}$ auch der Fall ist. Was Aussage (2) betrifft, so wissen wir, dass ε nicht mit 0 endet, und wir wissen zudem auf Grund der Definition von $\hat{\delta}$, dass $\hat{\delta}(q_0, \varepsilon)$ q_1 nicht enthält. Folglich sind die Hypothesen beider Richtungen der Genau-dann-wenn-Aussage falsch und daher sind beide Richtungen der Aussage wahr. Der Beweis der Aussage (3) für $w = \varepsilon$ entspricht im Grunde genommen dem oben geschilderten Beweis von Aussage (2).

Induktionsschritt: Angenommen, w sei xa, wobei a ein Symbol mit dem Wert 0 oder 1 sei. Wir können annehmen, die Aussagen (1) bis (3) gelten für x, und wir müssen beweisen, dass sie für w gelten. Das heißt, wir nehmen an $|w| = n + 1$, daher sei $|x| = n$. Wir unterstellen, dass die Induktionsannahme für n gilt und beweisen sie für $n + 1$.

1. Wir wissen, dass $\hat{\delta}(q_0, x)$ q_0 enthält. Da sowohl mit der Eingabe 0 als auch mit 1 ein Übergang von q_0 zu q_0 erfolgt, folgt daraus, dass auch $\hat{\delta}(q_0, w)$ q_0 enthält, und damit ist Aussage (1) für w bewiesen.
2. (Wenn-Teil) Angenommen, w endet mit 0, d. h. $a = 0$. Aus der Anwendung von Aussage (1) auf x wissen wir, dass $\hat{\delta}(q_0, x)$ q_0 enthält. Da die Eingabe 0 einen Übergang von q_0 nach q_1 bewirkt, können wir darauf schließen, dass $\hat{\delta}(q_0, w)$ q_1 enthält.

   (Nur-dann-Teil) Angenommen, $\hat{\delta}(q_0, w)$ enthält q_1. Wenn wir das Diagramm in Abbildung 2.7 betrachten, sehen wir, dass der Automat nur dann den Zustand q_1 annehmen kann, wenn die Eingabefolge w die Form $x0$ hat. Die ist für den Beweis des Genau-dann-Teils der Aussage ausreichend.
3. (Wenn-Teil) Angenommen, w endet mit 01. Wenn $w = xa$, dann wissen wir, dass $a = 1$ und dass x mit 0 endet. Die Anwendung von Aussage (2) auf x ergibt, dass $\hat{\delta}(q_0, x)$ q_1 enthält. Da für die Eingabe 1 ein Übergang von q_1 nach q_2 definiert ist, können wir schließen, dass $\hat{\delta}(q_0, w)$ q_2 enthält.

(Nur-dann-Teil) Angenommen, $\hat{\delta}(q_0, w)$ enthält q_2. Aus dem Diagramm in Abbildung 2.7 geht hervor, dass der Übergang in den Zustand q_2 nur dann möglich ist, wenn w die Form $x1$ hat, wobei $\hat{\delta}(q_0, x)$ q_1 enthält. Die Anwendung von Aussage (2) auf x ergibt, dass x mit 0 endet. Folglich endet w mit 01, und damit haben wir Aussage (3) bewiesen. ■

2.3.5 Äquivalenz deterministischer und nichtdeterministischer endlicher Automaten

Obwohl es viele Sprachen gibt, für die ein NEA einfacher zu konstruieren ist als ein DEA, wie z. B. die Sprache der mit 01 endenden Zeichenreihen aus Beispiel 2.6, ist es eine überraschende Tatsache, dass jede Sprache, die sich durch einen NEA beschreiben lässt, auch durch einen DEA beschrieben werden kann. Zudem zeigt sich in der Praxis, dass ein DEA in etwa genauso viele Zustände besitzt wie ein NEA, doch häufig mehr Zustandsänderungen erfährt als dieser. Im schlimmsten Fall jedoch kann der kleinste DEA über 2^n Zustände verfügen, während der kleinste NEA für dieselbe Sprache nur n Zustände aufweist.

Der Beweis, dass das Leistungsvermögen von DEAs dem von NEAs entspricht, erfordert eine wichtige »Konstruktion«, die *Teilmengenkonstruktion* genannt wird, weil hierbei alle Teilmengen der Zustandsmenge eines NEA konstruiert werden. Im Allgemeinen wird bei vielen Beweisen im Zusammenhang mit Automaten ein Automat aus einem anderen gebildet. Die Teilmengenkonstruktion ist wichtig als Beispiel dafür, wie man einen Automaten durch einen anderen formal beschreiben kann, ohne die Spezifikation des anderen zu kennen.

Die Teilmengenkonstruktion beginnt mit einem NEA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Ziel ist es, einen DEA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ zu beschreiben, derart dass $L(D) = L(N)$. Beachten Sie, dass die beiden Automaten über dieselben Eingabealphabete verfügen und dass es sich bei dem Startzustand von D um die Menge handelt, die lediglich den Startzustand von N enthält. Die übrigen Komponenten von D werden wie folgt konstruiert:

- Q_D ist die Menge der Teilmengen von Q_N; d. h. Q_D ist die Potenzmenge von Q_N. Wenn Q_N also n Zustände enthält, dann umfasst Q_D 2^n Zustände. Häufig sind nicht alle diese Zustände vom Startzustand von Q_D aus erreichbar. Nicht erreichbare Zustände können eliminiert werden, und daher kann die tatsächliche Anzahl von Zuständen von D sehr viel kleiner sein als 2^n.
- F_D ist die Menge S der Teilmengen von Q_N, derart dass $S \cap F_N \neq \emptyset$. Das heißt, F_D umfasst alle Teilmengen von Q_N, die mindestens einen akzeptierenden Zustand von N enthalten.
- Für jede Menge $S \subseteq Q_N$ und für jedes Eingabesymbol a aus Σ gilt:

$$\delta_D(S,a) = \bigcup_{p \in S} \delta_N(p,a)$$

Zur Berechnung von $\delta_D(S, a)$ betrachten wir alle in S enthaltenen Zustände p, untersuchen, zu welchen Zuständen N auf die Eingabe a hin von p aus übergeht, und nehmen die Vereinigung aller dieser Zustände.

Tabelle 2.4: Die vollständige Teilmengenkonstruktion basierend auf Abbildung 2.7

	0	1
ý	ý	ý
→ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	ý	$\{q_2\}$
* $\{q_2\}$	ý	ý
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
* $\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
* $\{q_1, q_2\}$	ý	$\{q_2\}$
* $\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Beispiel 2.10 Sei N der Automat aus Abbildung 2.7, der alle mit 01 endenden Zeichenreihen akzeptiert. Da die Zustandsmenge von N gleich $\{q_0, q_1, q_2\}$ ist, ergibt die Teilmengenkonstruktion einen DEA mit $2^3 = 8$ Zuständen, die allen Teilmengen aus diesen drei Zuständen entsprechen. Tabelle 2.4 zeigt die Übergangstabelle für diese acht Zustände. Wir werden nun kurz zeigen, wie einige dieser Tabelleneinträge berechnet wurden.

Beachten Sie, dass diese Übergangstabelle zu einem deterministischen endlichen Automaten gehört. Da es sich bei den Tabelleneinträgen um Mengen handelt, sind die Zustände des konstruierten DEA Mengen. Um diesen Punkt zu verdeutlichen, können wir den Zuständen neue Namen zuordnen, z. B. A für $\emptyset$, B für $\{q_0\}$ usw. Die DEA-Übergangstabelle in Tabelle 2.5 definiert den gleichen Automaten wie Tabelle 2.4, verdeutlicht jedoch die Tatsache, dass es sich bei den Tabelleneinträgen um einzelne Zustände des DEA handelt.

Tabelle 2.5: Umbenennung der Zustände aus Tabelle 2.4

	0	1
A	*A*	*A*
→ *B*	*E*	*B*
C	*A*	*D*
* *D*	*A*	*A*
E	*E*	*F*
* *F*	*E*	*B*
* *G*	*A*	*D*
* *H*	*E*	*F*

Von den acht in Tabelle 2.5 dargestellten Zuständen sind vom Zustand B aus nur die Zustände B, E und F erreichbar. Die anderen fünf Zustände sind vom Startzustand aus nicht erreichbar und können genauso gut weggelassen werden. Wir können häufig den exponentiellen Zeitaufwand der Konstruktion von Übergangstabellen vermeiden, wenn wir die Teilmengen durch »abwartende Auswertung«[4] wie folgt berechnen.

Induktionsbeginn: Wir wissen genau, dass die einelementige Menge, die nur den Startzustand von N enthält, erreichbar ist.

Induktionsschritt: Angenommen, wir haben ermittelt, dass die Menge S von Zuständen erreichbar ist. Wenn wir dann für jedes Eingabesymbol a die Zustandsmenge $\delta_D(S, a)$ berechnen, wissen wir, dass auch diese Mengen von Zuständen erreichbar sind.

In dem vorliegenden Beispiel wissen wir, dass $\{q_0\}$ ein Zustand des DEA D ist. Wir ermitteln, dass $\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ und $\delta_D(\{q_0\}, 1) = \{q_0\}$. Diese beiden Fakten gehen aus dem Übergangsdiagramm in Abbildung 2.7 hervor, das zeigt, dass bei der Eingabe 0 zwei Pfeile von q_0 ausgehen und zu q_0 sowie q_1 führen, während bei der Eingabe 1 nur ein Pfeil zu q_0 führt. Wir haben damit eine Zeile für die Übergangstabelle des DEA: die zweite Zeile in Tabelle 2.5.

Eine der beiden berechneten Mengen ist »alt«; $\{q_0\}$ wurde bereits betrachtet. Allerdings ist die zweite Menge $\{q_0, q_1\}$ neu, sodass der Zustandsübergang hier berechnet werden muss. Wir erhalten $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ und $\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$. Die Berechnung des zweiten Falls lässt sich beispielsweise wie folgt nachvollziehen: Wir wissen, dass

$$\delta_D(\{q_0, q_1\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$$

Wir haben nun die fünfte Zeile aus Tabelle 2.4 und einen neuen Zustand von D entdeckt, der $\{q_0, q_2\}$ lautet. Aus einer ähnlichen Berechnung geht hervor:

$$\delta_D(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0, q_1\} \cup \{\emptyset\} = \{q_0, q_1\}$$
$$\delta_D(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \{\emptyset\} = \{q_0\}$$

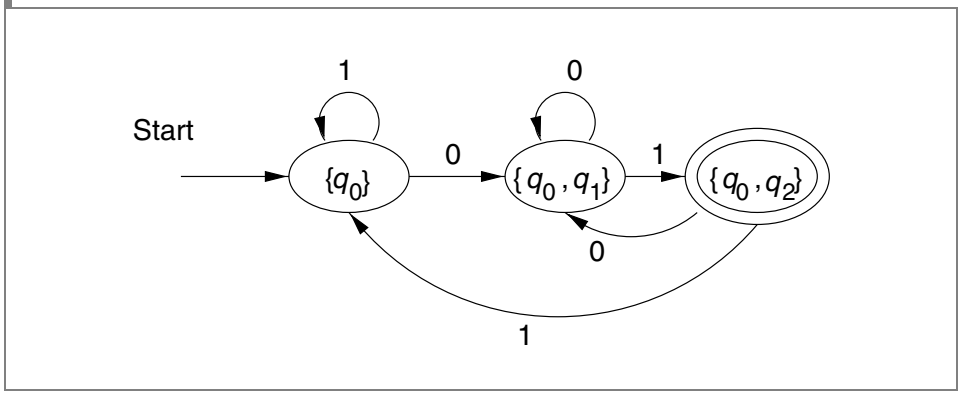
Durch diese Berechnungen erhalten wir die sechste Zeile von Tabelle 2.4. Wir erhalten jedoch nur Mengen von Zuständen, die bereits vorkamen.

Folglich hat die Teilmengenkonstruktion eine Grenze erreicht. Wir kennen alle erreichbaren Zustände und ihre Übergänge. Abbildung 2.9 zeigt den gesamten DEA. Beachten Sie, dass er nur über drei Zustände verfügt, was zufälligerweise die gleiche Anzahl von Zuständen ist wie bei dem NEA aus Abbildung 2.7, aus dem dieser DEA konstruiert wurde. Der DEA in Abbildung 2.9 besitzt allerdings sechs Übergänge, während der NEA aus Abbildung 2.7 nur vier aufweist. ■

Wir müssen formal zeigen, dass die Teilmengenkonstruktion korrekt ist, auch wenn die Beispiele das Konzept schon veranschaulichen. Nachdem eine Folge von Eingabesymbolen w gelesen wurde, befindet sich der konstruierte DEA in einem Zustand, der der Menge der NEA-Zustände entspricht, in der sich der NEA nach dem Lesen von w befinden würde. Nachdem es sich bei den akzeptierenden Zuständen des DEA um diejenigen Mengen handelt, die mindestens einen akzeptierenden Zustand des NEA

4. Im Englischen »lazy evaluation«

Abbildung 2.9: Der DEA, der aus dem NEA aus Abbildung 2.7 konstruiert wurde



enthalten, und der NEA auch akzeptiert, wenn er in mindestens einen seiner akzeptierenden Zustände übergeht, können wir darauf schließen, dass der DEA und der NEA genau dieselben Zeichenreihen akzeptieren und demnach genau dieselbe Sprache akzeptieren.

Satz 2.11 Wenn $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ der DEA ist, der mithilfe der Teilmengenkonstruktion aus dem NEA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ konstruiert wird, dann gilt $L(D) = L(N)$.
Beweis: Wir beweisen zuerst durch Induktion über $|w|$, dass

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Beachten Sie, dass beide $\hat{\delta}$-Funktionen eine Zustandsmenge aus Q_N zurückgeben, die Funktion $\hat{\delta}_D$ diese Menge jedoch als einen der Zustände aus Q_D (Q_D ist die Potenzmenge von Q_N) interpretiert, während die Funktion $\hat{\delta}_N$ diese Menge als Teilmenge von Q_N interpretiert.

Induktionsbeginn: Sei $|w| = 0$, d. h. $w = \varepsilon$. Gemäß der Definition der Übergangsfunktion δ für DEAs und NEAs gilt, dass sowohl $\hat{\delta}_D(\{q_0\}, \varepsilon)$ als auch $\hat{\delta}_N(q_0, \varepsilon)$ gleich $\{q_0\}$ sind.

Induktionsschritt: Sei w von der Länge $n + 1$, und nehmen wir an, die Aussage sei für die Länge n wahr. Wir teilen w auf in $w = xa$, wobei a das letzte Symbol von w ist. Gemäß der Induktionsannahme gilt $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Angenommen, beide Zustandsmengen von N seien $\{p_1, p_2, ..., p_k\}$.

Aus dem induktiven Teil der Definition von $\hat{\delta}$ geht hervor, dass

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

Die Teilmengenkonstruktion ergibt andererseits

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

Nun setzen wir (2.3) und die Tatsache, dass $\hat{\delta}_D(\{q_0,\}, x) = \{p_1, p_2, ..., p_k\}$ in den induktiven Teil der Definition der Übergangsfunktion $\hat{\delta}$ für DEAs ein:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D(\hat{\delta}_D(\{q_0\}, x), a) = \delta_D(p_1, p_2, \ldots, p_k), a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.4}$$

Die Gleichungen (2.2) und (2.4) zeigen, dass $\hat{\delta}_D(\{q_0,\}, w) = \hat{\delta}_N(\{q_0,\}, w)$. Wenn wir berücksichtigen, dass sowohl D als auch N w ausschließlich dann akzeptieren, wenn $\hat{\delta}_D(\{q_0,\}, w)$ bzw. $\hat{\delta}_N(\{q_0,\}, w)$ einen in F_N enthaltenen Zustand enthalten, dann liegt der vollständige Beweis für $L(D) = L(N)$ vor. ■

Satz 2.12 Eine Sprache L wird von einem DEA genau dann akzeptiert, wenn L von einem NEA akzeptiert wird.

Beweis: (Wenn-Teil) Der »Wenn«-Teil besteht aus der Teilmengenkonstruktion und Satz 2.11.

(Nur-dann-Teil) Dieser Teil ist einfach. Wir müssen den DEA nur in einen identischen NEA umwandeln. Umgangssprachlich gesagt, wenn wir das Übergangsdiagramm für den DEA haben, dann können wir es ebenso als Übergangsdiagramm eines NEA interpretieren, der zufälligerweise in jeder Situation genau eine Übergangsmöglichkeit besitzt. Formaler ausgedrückt, sei $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F)$ ein DEA. Zu definieren ist ein äquivalenter NEA N = $(Q_N, \Sigma, \delta_N, \{q_0\}, F)$, wobei δ_N durch folgende Regel definiert wird:

- Wenn $\delta_D(q, a) = p$, dann $\delta_N(q, a) = \{p\}$.

Dann lässt sich durch Induktion über $|w|$ einfach zeigen, dass gilt: wenn $\hat{\delta}_D(q, w) = p$, dann

$$\hat{\delta}_N(q, \mathrm{w}) = \{p\}.$$

Wir überlassen diesen Beweis dem Leser. Folglich wird die Zeichenreihe w von D genau dann akzeptiert, wenn sie von N akzeptiert wird, d. h. $L(D) = L(N)$. ■

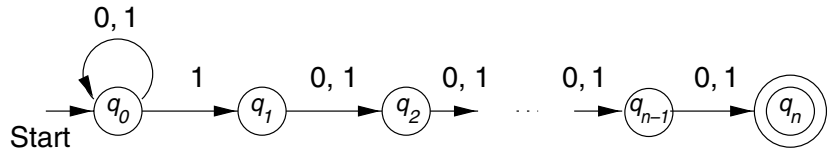
2.3.6 Ein ungünstiger Fall für die Teilmengenkonstruktion

In Beispiel 2.10 stellten wir fest, dass der DEA mehr Zustände aufwies als der NEA. Wie erwähnt, ist es nicht ungewöhnlich, dass der DEA in der Praxis die gleiche Anzahl von Zuständen besitzt wie der NEA, aus dem er konstruiert wird. Es ist allerdings möglich, dass die Anzahl der Zustände exponentiell zunimmt; alle 2^n DEA-Zustände, die aus einem NEA mit n Zuständen konstruiert werden können, könnten in der Tat erreichbar sein. Das folgende Beispiel erreicht diesen Grenzwert nicht ganz, zeigt aber einen verständlichen Weg, 2^n Zustände bei dem kleinsten DEA zu erreichen, der äquivalent mit einem NEA mit $n + 1$ Zuständen ist.

Beispiel 2.13 Betrachten Sie den NEA N in Abbildung 2.10. $L(N)$ ist die Menge aller aus Einsen und Nullen bestehenden Zeichenreihen, derart dass das n-te Zeichen vor dem Ende eine Eins ist. Ein DEA D, der diese Sprache akzeptiert, muss sich an die letzten n Zeichen, die er gelesen hat, erinnern. Wenn nun D weniger als 2^n Zustände hätte, dann gäbe es einen Zustand q und zwei verschiedene Zeichenreihen $a_1a_2\ldots a_n$ und $b_1b_2\ldots b_n$, sodass sich D nach dem Einlesen sowohl von $a_1a_2...a_n$ als auch von $b_1b_2...b_n$ im Zustand q befände.

Da die Zeichenreihen verschieden sind, müssen sie sich an einer bestimmten Position unterscheiden; sei $a_i \neq b_i$. Angenommen (auf Grund der Symmetrie o.B.d.A.[5]), $a_i = 1$ und $b_i = 0$. Wenn $i = 1$, dann muss q sowohl ein akzeptierender Zustand als auch ein nicht akzeptierender Zustand sein, da $a_1a_2...a_n$ akzeptiert wird (das n-te Zeichen vor dem Ende ist 1) und $b_1b_2...b_n$ nicht. Ist i > 1, dann betrachten wir den Zustand p, in den D nach dem Einlesen sowohl von $a_1a_2...a_n0^{i-1}$ als auch von $b_1b_2...b_n0^{i-1}$ käme. Da $a_i = 1$ und $b_i = 0$, müsste $a_1a_2...a_n0^{i-1}$ akzeptiert werden (das n-te Zeichen vor dem Ende ist $a_i = 1$) und $b_1b_2...b_n0^{i-1}$ nicht, d. h. p müsste ein akzeptierender und zugleich ein nicht akzeptierender Zustand sein.

Abbildung 2.10: Es gibt keinen mit diesem NEA äquivalenten DEA mit weniger als 2^n Zuständen



Wir wollen uns nun ansehen, wie der NEA N aus Abbildung 2.10 funktioniert. Es gibt einen Zustand q_0, in dem sich der NEA, unabhängig von der gelesenen Eingabe, immer befindet. Handelt es sich bei der nächsten Eingabe um 1, dann »vermutet« N zudem, dass diese 1 das n-te Symbol vor dem Ende ist, und nimmt neben dem Zustand q_0 zusätzlich den Zustand q_1 an. Wenn sich N im Zustand q_1 befindet, dann bewirkt jede Eingabe einen Zustandsübergang in q_2, die nächste Eingabe bewirkt den Übergang in Zustand q_2 und so weiter, bis $n - 1$ Eingaben später der akzeptierende Zustand q_n erreicht ist. Die formale Beschreibung der Zustandsübergänge von N lautet:

1. Nach dem Lesen einer beliebigen Folge von Eingaben w befindet sich N im Zustand q_0.
2. N befindet sich nach dem Lesen der Eingabefolge w genau dann im Zustand q_i, für $i = 1, 2, ..., n$, wenn das i-te Zeichen vor dem Ende von w eine Eins ist, d. h. wenn w die Form $x1a_1a_2 ...a_{i-1}$ hat, wobei die a_j Eingabezeichen sind.

Wir werden diese Aussagen nicht formal beweisen. Der Beweis ist eine einfache Induktion über $|w|$, ähnlich wie in Beispiel 2.9. Um den Beweis zu vervollständigen, dass der Automat genau die Zeichenreihen akzeptiert, die an der n-ten Position vor dem Ende eine 1 enthalten, betrachten wir die Aussage (2) für $i = n$. Diese Aussage besagt, N befindet sich genau dann im Zustand q_n, wenn das n-te Zeichen vor dem Ende eine 1 ist. Da aber q_n auch der einzige akzeptierende Zustand ist, beschreibt diese Bedingung zudem genau die Menge von Zeichenreihen, die von N akzeptiert werden.

5. ohne Beschränkung der Allgemeinheit

Das Schubfachprinzip

In Beispiel 2.13 haben wir eine wichtige Schlussfolgerungstechnik verwendet, die als *Schubfachprinzip* bezeichnet wird. Umgangssprachlich ausgedrückt, besagt das Schubfachprinzip Folgendes: Befinden sich $m > n$ Dinge in n Schubfächern, dann enthält mindestens ein Schubfach zwei Dinge. In unserem Beispiel stellen die Folgen von n Eingabezeichen die »Dinge« und die Zustände die »Schubfächer« dar. Da es weniger Zustände als Eingabefolgen gibt, muss ein Zustand zwei Eingabefolgen zugeordnet werden.

Das Schubfachprinzip scheint auf der Hand zu liegen, es beruht aber wesentlich auf der Annahme, dass die Anzahl der Schubfächer endlich ist. Folglich lässt es sich auf endliche Automaten übertragen, indem die Zustände als Schubfächer interpretiert werden, aber es ist nicht auf andere Typen von Automaten anwendbar, die über eine unendliche Anzahl von Zuständen verfügen.

Weshalb die Endlichkeit der Anzahl von Schubfächern entscheidend ist, wird klar, wenn man den unendlichen Fall betrachtet, in dem die Schubfächer ganze Zahlen 1, 2, ... repräsentieren. Wir nummerieren die Dinge nun beginnend mit Null 0, 1, 2, ... , sodass es scheinbar ein Ding mehr als Schubfächer gibt. Nun können wir in diesem Fall für alle $i \geq 0$ ein Ding i in ein Schubfach $i + 1$ legen. Folglich steht für jedes der unendlichen Anzahl von Dingen ein Schubfach zur Verfügung, und keine zwei Dinge müssen sich ein Schubfach teilen.

2.3.7 Übungen zum Abschnitt 2.3

* **Übung 2.3.1** Wandeln Sie den folgenden NEA in einen DEA um:

	0	1
→ p	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
* s	$\{s\}$	$\{s\}$

Übung 2.3.2 Wandeln Sie den folgenden NEA in einen DEA um:

	0	1
→ p	$\{q, s\}$	$\{q\}$
* q	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
* s	$\emptyset$	$\{p\}$

Zustände, die Sackgassen darstellen, und DEAs, die Übergänge auslassen

Nach der formalen Definition verfügt ein DEA für jedes Eingabesymbol von jedem Zustand aus über genau einen Übergang. Gelegentlich ist es jedoch komfortabler, den DEA so zu entwerfen, dass er in den Zuständen endet, in denen wir wissen, dass keine Erweiterung einer Eingabefolge, die in einen dieser Zustände führt, akzeptiert werden kann. Sehen Sie sich beispielsweise den Automat aus Abbildung 1.2 an, der lediglich dem Zweck dient, ein einziges Schlüsselwort (`then`) zu erkennen. Technisch gesehen, ist dieser Automat kein DEA, weil aus den meisten Zuständen heraus für die meisten Eingabesymbole keine Übergänge erfolgen.

Ein solcher Automat ist demnach ein NEA. Wenn wir diesen mithilfe der Teilmengenkonstruktion in einen DEA umwandeln, dann sieht dieser fast genauso aus, besitzt jedoch einen Zustand, der eine Sackgasse darstellt, d. h. einen nicht akzeptierenden Zustand, der auf jede Eingabe hin zu sich selbst zurückführt. Dieser Zustand entspricht der leeren Menge $\emptyset$ als Teilmenge der Zustände des Automaten aus Abbildung 1.2.

Im Allgemeinen können wir jedem Automaten, der für jeden Zustand und jedes Eingabesymbol über *nicht mehr* als einen Übergang verfügt, einen »Sackgassenzustand« f hinzufügen. Anschließend können wir zu jedem Zustand $q \neq f$ für alle Eingabesymbole, für die q keinen Übergang definiert, einen Übergang zu diesem Sackgassenzustand f hinzufügen. Daraus ergibt sich ein DEA, der die Definition genau erfüllt. Daher werden wir gelegentlich einen Automaten als DEA bezeichnen, wenn er von jedem Zustand ausgehend für jedes Eingabesymbol über *höchstens* einen Übergang statt *genau einen* Übergang verfügt.

! Übung 2.3.3 Wandeln Sie den folgenden NEA in einen DEA um, und beschreiben Sie dessen Sprache informell:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r, s\}$	$\{t\}$
r	$\{p, r\}$	$\{t\}$
$*\ s$	$\emptyset$	$\emptyset$
$*\ t$	$\emptyset$	$\emptyset$

! Übung 2.3.4 Definieren Sie nichtdeterministische endliche Automaten, die die folgenden Sprachen akzeptieren. Nutzen Sie den Nichtdeterminismus so weit wie möglich.

***** a) Die Menge der Zeichenreihen aus dem Alphabet {0, 1, ..., 9}, derart dass die letzte Ziffer schon vorher vorgekommen ist.

b) Die Menge der Zeichenreihen aus dem Alphabet {0, 1, ..., 9}, derart dass die letzte Ziffer vorher noch nicht vorgekommen ist.

c) Die Menge aller aus Einsen und Nullen bestehenden Zeichenreihen, derart dass zwei Nullen durch eine durch vier teilbare Anzahl von Stellen voneinander getrennt sind. Beachten Sie, dass 0 ein zulässiges Vielfaches von 4 ist.

Übung 2.3.5 Im Nur-dann-Teil von Satz 2.12 ließen wir den Beweis durch Induktion über $|w|$ aus, dass gilt: Wenn $\hat{\delta}_D(q, w) = p$, dann $\hat{\delta}_N(q, \mathrm{w}) = \{p\}$. Liefern Sie diesen Beweis.

! **Übung 2.3.6** Im Exkurs »Zustände, die Sackgassen darstellen, und DEAs, die Übergänge auslassen« behaupten wir Folgendes: Wenn N ein NEA ist, der höchstens einen Übergang für jeden Zustand und jedes Eingabesymbol besitzt (d. h. $\delta|q, a|$ ist nie größer als 1), dann verfügt der durch Teilmengenkonstruktion aus N erstellte DEA D über dieselben Übergänge und Zustände wie N sowie zusätzlich in jenen Zuständen s Übergänge zu einem neuen »Sackgassenzustand« für jedes Eingabezeichen a, für das N keinen Übergang von s für a besitzt. Beweisen Sie diese Behauptung.

Übung 2.3.7 Im Beispiel 2.13 behaupten wir, dass sich der NEA N genau dann nach dem Lesen der Eingabefolge w im Zustand q_i mit $i = 1, 2, ..., n$ befindet, wenn das i-te Zeichen vor dem Ende von w gleich 1 ist. Beweisen Sie diese Behauptung.

2.4 Eine Anwendung: Textsuche

In diesem Abschnitt werden wir zeigen, dass die abstrakte Untersuchung des vorherigen Abschnitts, in dem wir das »Problem« betrachteten, wie man entscheidet, ob eine Zeichenreihe auf 01 endet, tatsächlich ein ausgezeichnetes Modell für einige praktische Probleme ist, die sich in Anwendungen wie der Internetsuche und der Extraktion von Daten aus Texten stellen.

2.4.1 Zeichenreihen in Texten finden

Im Zeitalter des Internet und anderen Online-Textsammlungen stellt sich häufig folgendes Problem: Gegeben ist eine Menge von Wörtern, und es sollen alle Dokumente gefunden werden, die eines (oder alle) dieser Wörter enthalten. Suchmaschinen sind ein gängiges Beispiel für diesen Vorgang. Die Suchmaschine verwendet eine spezielle Technik, die so genannten *invertierten Indexe*, wobei für jedes im Internet vorkommende Wort (ca. 100.000.000 verschiedene Wörter) eine Liste all jener Stellen, an denen dieses Wort vorkommt, gespeichert wird. Rechner mit einem sehr großen Arbeitsspeicher halten die gebräuchlichsten dieser Listen abrufbereit im Speicher und ermöglichen es damit, dass viele Personen gleichzeitig diese Listen durchsuchen.

Bei den auf invertierten Indexen basierenden Techniken werden zwar keine endlichen Automaten verwendet, aber es gibt eine Reihe verwandter Anwendungen, die sich nicht für invertierte Indexe eignen, jedoch gute Anwendungen für auf Automaten basierende Techniken darstellen. Anwendungen, die für eine Suche unter Verwendung von Automaten geeignet sind, zeichnen sich durch folgende Charakteristika aus:

1. Die Sammlung, die durchsucht werden soll, ändert sich rasch. Beispiele:
   a) Jeden Tag möchten Nachrichtenanalysten die Online-Nachrichtenartikel des aktuellen Tages nach relevanten Themen durchsuchen. Ein Finanz-

analyst kann beispielsweise nach bestimmten Aktiensymbolen oder Firmennamen suchen wollen.

b) Ein »Einkaufsroboter« möchte nach den aktuellen Preisen der Artikel suchen, die von seinen Kunden angefordert werden. Der Roboter ruft die aktuellen Katalogseiten aus dem Internet ab und durchsucht diese Seiten dann nach Wörtern, aus denen der Preis eines bestimmten Artikels hervorgeht.

2. Die zu durchsuchenden Dokumente lassen sich nicht katalogisieren. Amazon.com macht es den so genannten Crawlern z. B. nicht leicht, alle Seiten zu allen von diesem Unternehmen vertriebenen Büchern zu finden. Diese Seiten werden stattdessen erst *»on the fly«* auf konkrete Anfragen hin erstellt. Es ist allerdings möglich, eine Anfrage nach Büchern zu einem bestimmten Thema, wie z. B. »endliche Automaten«, abzuschicken und die gefundenen Seiten nach bestimmten Wörtern zu durchsuchen, z. B. nach dem Wort »hervorragend« in einem Rezensionsteil.

2.4.2 Nichtdeterministische endliche Automaten für die Textsuche

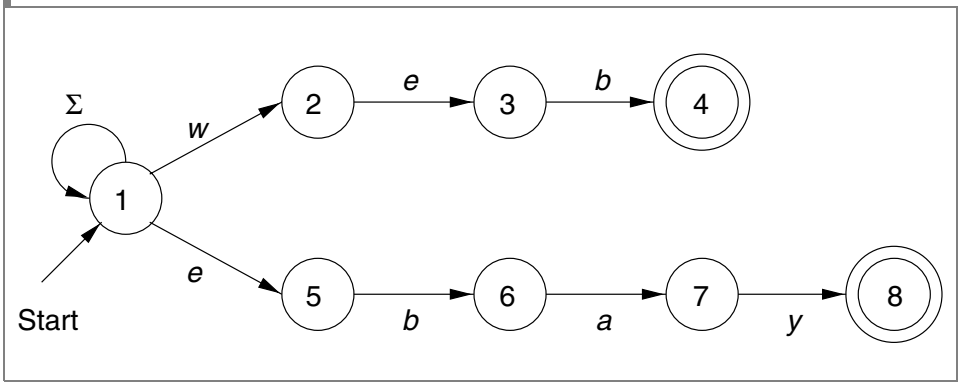
Angenommen, wir erhalten eine Menge von Wörtern, die wir *Schlüsselwörter* nennen wollen, und möchten sämtliche Vorkommen dieser Wörter finden. In Anwendungen wie dieser empfiehlt es sich, einen nichtdeterministischen endlichen Automaten zu entwerfen, der durch den Übergang in einen akzeptierenden Zustand signalisiert, dass er ein Schlüsselwort gelesen hat. Der Text eines Dokuments wird dem NEA zeichenweise eingegeben, der dann das Vorkommen von Schlüsselwörtern in diesem Text erkennt. Ein einfacher NEA, der eine Menge von Schlüsselwörtern erkennt, hat folgende Gestalt:

1. Es gibt einen Startzustand mit einem Übergang zu sich selbst für jedes Eingabezeichen, z. B. jedes druckbare ASCII-Zeichen, wenn wir Text untersuchen. Der Startzustand repräsentiert die »Vermutung«, dass bislang noch keine Zeichen vom Anfang eines Schlüsselworts gelesen wurden, auch wenn der Automat möglicherweise bereits einige Buchstaben aus einem dieser Schlüsselwörter gelesen hat.

2. Für jedes Schlüsselwort $a_1a_2 \ldots a_k$ sind k Zustände definiert, die $q_1, q_2, \ldots, q_k$ heißen sollen. Auf die Eingabe des Symbols a_1 hin erfolgt ein Übergang vom Startzustand in den Zustand q_1, auf die Eingabe des Symbols a_2 im Zustand q_1 ein Übergang in den Zustand q_2 und so weiter. Der Zustand q_k ist ein akzeptierender Zustand und zeigt an, dass das Schlüsselwort $a_1a_2 \ldots a_k$ gefunden wurde.

Beispiel 2.14 Angenommen, wir möchten einen NEA entwerfen, der Vorkommen der Wörter `web` und `ebay` erkennt. Das Übergangsdiagramm für den NEA, der unter Verwendung der oben genannten Regeln entworfen wird, ist in Abbildung 2.11 dargestellt. Zustand 1 ist der Startzustand, und wir verwenden Σ als Bezeichnung für die Menge aller ASCII-Zeichen. Die Zustände 2 bis 4 dienen zur Erkennung des Wortes `web`, während die Zustände 5 bis 8 zur Erkennung des Wortes `ebay` dienen. ■

Abbildung 2.11: Ein NEA, der nach den Wörtern web und ebay sucht



Natürlich ist der NEA kein Programm. Uns stehen zwei wichtige Möglichkeiten zur Implementierung dieses NEA zur Auswahl.

1. Wir können ein Programm schreiben, das diesen NEA simuliert, indem es die Menge der Zustände berechnet, in denen sich der NEA nach dem Lesen der einzelnen Eingabesymbole befindet (vgl. Abb. 2.8).
2. Wir können den NEA mithilfe der Teilmengenkonstruktion in einen äquivalenten DEA umwandeln und diesen DEA dann direkt simulieren.

Einige Textverarbeitungsprogramme, wie z. B. erweiterte Formen des Unix-Befehls `grep` (`egrep` und `fgrep`), setzen tatsächlich eine Mischung dieser beiden Ansätze ein. Für unsere Zwecke ist es jedoch einfacher und ausreichend, den NEA in einen DEA umzuwandeln. Dieses Vorgehen ist leicht und garantiert, dass sich die Anzahl der Zustände nicht erhöht.

2.4.3 Ein DEA, der eine Menge von Schlüsselwörtern erkennt

Wir können die Teilmengenkonstruktion auf jeden beliebigen NEA anwenden. Wenn wir dieses Verfahren jedoch auf einen NEA anwenden, der entsprechend der in Abschnitt 2.4.2 beschriebenen Strategie für eine Menge von Schlüsselwörtern konzipiert ist, dann stellt sich heraus, dass die Anzahl der Zustände des DEA niemals größer ist als die Anzahl der Zustände des NEA. Da sich die Anzahl der Zustände bei der Umwandlung in einen DEA im schlimmsten Fall exponentiell erhöht, erklärt diese erfreuliche Beobachtung, warum die Methode, einen NEA für Schlüsselwörter zu entwerfen und dann einen DEA daraus zu konstruieren, so häufig verwendet wird. Die Konstruktion der Zustandsmenge des DEA erfolgt nach folgenden Regeln:

a) Wenn q_0 der Startzustand des NEA ist, dann ist $\{q_0\}$ der Startzustand des DEA.

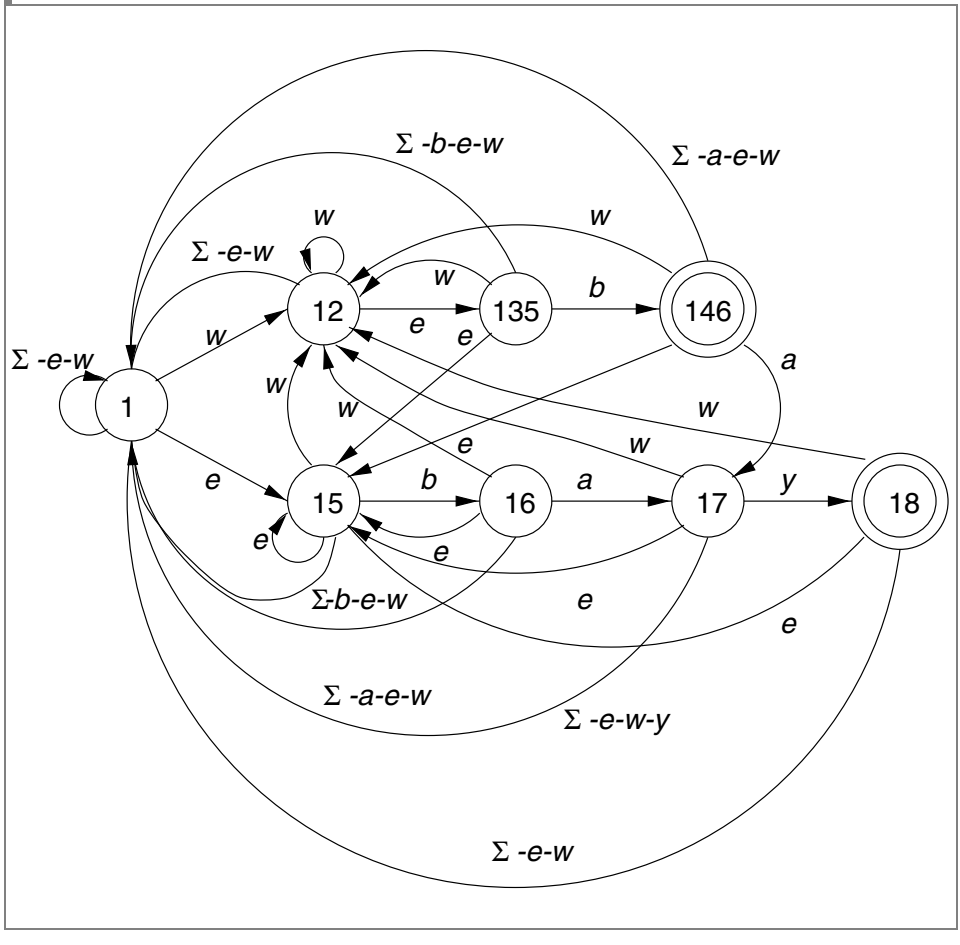
b) Angenommen, p sei ein Zustand des NEA, der vom Startzustand aus über einen Pfad zur Zeichenreihe $a_1a_2 \ldots a_m$ erreicht wird. Dann ist einer der DEA-Zustände die Menge der NEA-Zustände, die sich zusammensetzt aus:

   1. q_0
   2. p

3. Jedem anderen Zustand des NEA, der von q_0 aus über einen Pfad zu einer Zeichenreihe $a_j a_{j+1} \ldots a_m$, d. h. zu einem Suffix von $a_1 a_2 \ldots a_m$, erreicht werden kann

Beachten Sie, dass es im Allgemeinen einen DEA-Zustand für jeden NEA-Zustand p geben wird. In Schritt (b) können allerdings zwei Zustände dieselbe Menge von NFA-Zuständen ergeben und folglich zu einem DEA-Zustand werden. Wenn zwei der Schlüsselwörter beispielsweise mit dem gleichen Buchstaben beginnen, angenommen a, dann ergeben zwei NEA-Zustände, die von q_0 aus über einen Pfeil mit der Beschriftung a erreichbar sind, dieselbe Menge von DEA-Zuständen und werden im DEA folglich zusammengefasst.

Abbildung 2.12: Umwandlung des NEA aus Abbildung 2.11 in einen DEA



Beispiel 2.15 Abbildung 2.12 zeigt die Konstruktion eines DEA aus dem in Abbildung 2.11 dargestellten NEA. Jeder Zustand des DEA befindet sich an derselben Position wie der Zustand p, aus dem er mithilfe der oben beschriebenen Regel b) abgeleitet wurde. Betrachten Sie z. B. den Zustand 135, dessen Name unsere Kurzschreibweise für {1, 3, 5} ist. Dieser Zustand wurde aus Zustand 3 konstruiert. Er enthält den Start-

zustand 1, weil jede Menge von DEA-Zuständen den Startzustand enthält. Er enthält zudem den Zustand 5, weil dieser Zustand von Zustand 1 aus über das Suffix `e` der Zeichenreihe `we` erreicht wird, die in Abbildung 2.11 von 1 nach 3 führt.

Die Übergänge für die einzelnen DEA-Zustände können gemäß der Teilmengenkonstruktion berechnet werden. Die Regel ist jedoch einfach. Ermittle für jeden DEA-Zustand q, der den Startzustand q_0 und einige andere Zustände $\{p_1, p_2, ..., p_n\}$ enthält, für jedes Eingabesymbol x, wohin q_0 und die p_i im NEA führen, und füge diesem DEA-Zustand q einen Übergang mit der Beschriftung x zu dem DEA-Zustand hinzu, der aus q_0 und allen Zuständen besteht, in die q_0 und die p_i im NEA durch x überführt werden.

Betrachten Sie beispielsweise den Zustand 135 in Abbildung 2.12. Der NEA aus Abbildung 2.11 weist an den Zuständen 3 und 5 für das Eingabesymbol b Übergänge zu den Zuständen 4 bzw. 6 auf. Folglich wird 135 bei der Eingabe b in 146 überführt. Für das Symbol e gibt es im NEA keine Übergänge von den Zuständen 3 oder 5 aus, aber es gibt einen Übergang von Zustand 1 zum Zustand 5. Folglich wird der Zustand 135 im DEA bei der Eingabe e in 15 überführt. Analog wird der Zustand 135 bei der Eingabe w in 12 überführt.

Für jedes andere Symbol x gibt es im NEA keinen Übergang aus 3 und 5 heraus, und Zustand 1 führt lediglich zu sich selbst zurück. Folglich gibt es auf die Eingabe jedes Symbols aus Σ hin, das nicht w, e oder b lautet, Übergänge von 135 nach 1. Wir verwenden die Notation $\Sigma - b - e - w$ zur Darstellung dieser Menge und ähnliche Darstellungen anderer Mengen, in denen einige Symbole aus Σ fehlen. ■

2.4.4 Übungen zum Abschnitt 2.4

Übung 2.4.1 Entwerfen Sie NEAs, die die folgenden Mengen von Zeichenreihen erkennen:

* a) `abc`, `abd` und `aacd`. Angenommen, das Alphabet sei $\{a, b, c, d\}$.

b) `0101`, `101` und `011`.

c) `ab`, `bc` und `ca`. Angenommen, das Alphabet sei $\{a, b, c\}$.

Übung 2.4.2 Wandeln Sie jeden ihrer NEAs aus Übung 2.4.1 in einen DEA um.

2.5 Endliche Automaten mit Epsilon-Übergängen

Wir werden nun noch eine andere Erweiterung des endlichen Automaten vorstellen. Das neue Leistungsmerkmal besteht darin, dass wir Übergänge für die leere Zeichenreihe ε zulassen. Dies bedeutet im Endeffekt, dass ein NEA spontan in einen anderen Zustand übergehen kann, ohne ein Eingabesymbol empfangen zu haben. Wie der Nichtdeterminismus, der in Abschnitt 2.3 eingeführt wurde, stellt dieses neue Leistungsmerkmal keine Erweiterung der Klasse der Sprachen dar, die von einem endlichen Automaten akzeptiert werden kann, sondern sie dient uns als Hilfsmittel für die Programmierung. Wir werden zudem im Zusammenhang mit regulären Ausdrücken, die in Abschnitt 3.1 vorgestellt werden, sehen, dass NEAs mit ε-Übergängen (die wir ε-NEAs nennen werden) eng mit regulären Ausdrücken verwandt und hilfreich sind, wenn die Äquivalenz von Sprachklassen bewiesen werden soll, die von regulären Ausdrücken bzw. von endlichen Automaten akzeptiert werden.

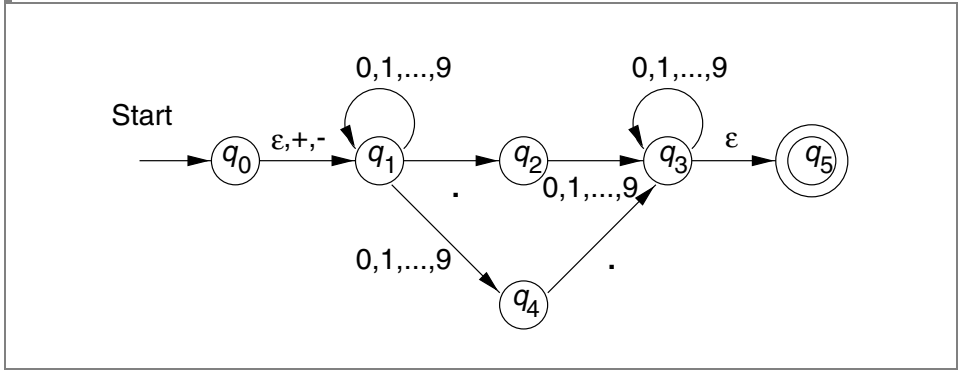
2.5.1 Verwendung von ε-Übergängen

Wir werden mit einer informellen Behandlung von ε-NEAs beginnen, wobei wir in Übergangsdiagrammen ε als Beschriftung zulassen. Stellen Sie sich die Automaten in den folgenden Beispielen so vor, dass sie diese Folgen von Beschriftungen entlang Pfaden akzeptieren, die vom Startzustand zu einem akzeptierenden Zustand führen. Allerdings ist jedes auf einem Pfad liegende Eingabesymbol ε »unsichtbar«, d. h. es trägt nichts zu der Zeichenreihe des Pfades bei.

Beispiel 2.16 Abbildung 2.13 zeigt einen ε-NEA, der Dezimalzahlen akzeptiert, die sich aus folgenden Komponenten zusammensetzen:

1. einem optionalen Plus- (+) oder Minuszeichen (-),
2. einer Zeichenreihe von Ziffern,
3. einem Dezimalpunkt und
4. einer weiteren Zeichenreihe von Ziffern. Sowohl diese Zeichenreihe von Ziffern als auch die Zeichenreihe (2) kann leer sein, jedoch muss mindestens eine der beiden Zeichenreihen Ziffern enthalten.

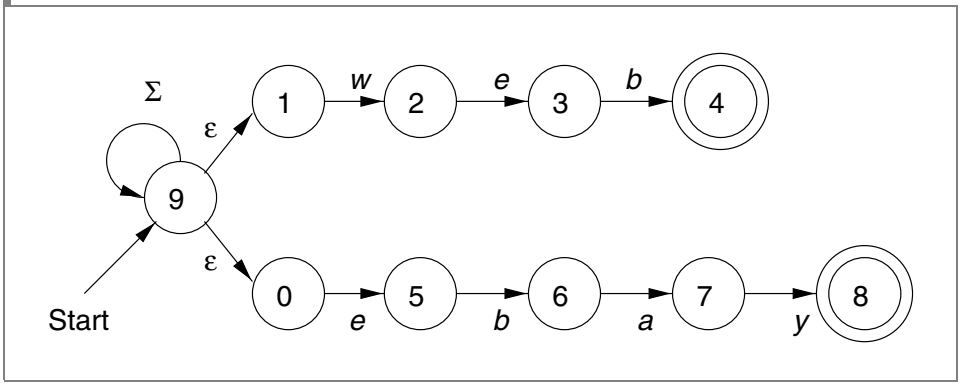
Abbildung 2.13: Ein ε-NEA, der Dezimalzahlen akzeptiert



Von besonderem Interesse ist der Übergang von q_0 nach q_1 auf die Eingabe ε, + und – hin. Der Zustand q_1 repräsentiert somit den Fall, in dem das Vorzeichen, sofern es eines gibt, und möglicherweise einige Ziffern gelesen wurden, jedoch noch nicht der Dezimalpunkt. Zustand q_2 stellt den Fall dar, in dem der Dezimalpunkt und möglicherweise davor stehende Ziffern und gegebenenfalls ein Vorzeichen gelesen wurden. In q_4 ist (abgesehen von einem möglichen Vorzeichen) mindestens eine Ziffer gelesen worden, aber der Dezimalpunkt noch nicht. Die Interpretation von q_3 ist, dass (abgesehen von einem möglichen Vorzeichen) der Dezimalpunkt und mindestens eine Ziffer, entweder vor oder nach dem Dezimalpunkt, gelesen wurden. In q_3 können weitere Ziffern gelesen werden, sofern noch welche folgen, aber der ε-NEA kann von q_3 auch spontan in den Zustand q_5 übergehen, gewissermaßen in der Vermutung, dass die abschließende Ziffernfolge vollständig gelesen wurde. ■

Beispiel 2.17 Die Strategie, die wir in Beispiel 2.14 zum Aufbau eines NEA skizziert haben, der eine Menge von Schlüsselwörtern akzeptiert, lässt sich weiter vereinfachen, wenn wir ε-Übergänge zulassen. Der in Abbildung 2.11 gezeigte NEA, der die Wörter `web` und `ebay` erkennt, kann auch mit ε-Übergängen wie in Abbildung 2.14 gezeigt implementiert werden. Allgemein gesagt, konstruieren wir hierzu eine vollständige Folge von Zuständen für jedes Schlüsselwort, so als wäre dies das einzige Wort, das der Automat erkennen müsste. Dann fügen wir einen neuen Startzustand (in Abbildung 2.14 ist dies Zustand 9) mit -Übergängen zu den Startzuständen der Automaten für die verschiedenen Schlüsselwörter hinzu. ■

Abbildung 2.14: ε-Übergänge zur Erkennung von Schlüsselwörtern nutzen



2.5.2 Die formale Notation eines ε-NEA

Jeder ε-NEA kann auf die gleiche Weise wie ein NEA beschrieben werden, wobei allerdings die Übergangsfunktion Informationen zu den Übergängen, die auf die Eingabe ε hin erfolgen, enthalten muss. Formal geben wir einen ε-NEA A mit $A = (Q, \Sigma, \delta, q_0, F)$ an, wobei alle Komponenten, mit Ausnahme der Übergangsfunktion δ, wie bei einem NEA interpretiert werden. Die Übergangsfunktion akzeptiert hier folgende Argumente:

1. einen in Q enthaltenen Zustand und
2. ein Element von $\Sigma \cup \{\varepsilon\}$, d. h. entweder ein Eingabesymbol oder das Symbol ε. Wir fordern, dass ε (das Symbol für die leere Zeichenreihe) kein Element von Σ (dem Alphabet) sein darf, damit keine Verwechslungen auftreten.

Beispiel 2.18 Der ε-NEA aus Abbildung 2.13 wird formal angegeben mit

$$E = (\{q_0, q_1, ..., q_5\}, \{., +, -, 0, 1, ..., 9\}, \delta, q_0, \{q_5\})$$

wobei δ durch Tabelle 2.6 definiert wird. ■

Tabelle 2.6: Übergangstabelle für den NEA aus Abbildung 2.13

	ε	+, –	.	0, 1, ... , 9
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_4	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
q_5	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

2.5.3 Epsilon-Hüllen

Wir fahren mit den formalen Definitionen einer erweiterten Übergangsfunktion für ε-NEAs fort, was zur Definition der von diesen Automaten akzeptierten Zeichenreihen und Sprachen sowie schließlich zu der Erklärung führt, warum ε-NEAs von DEAs simuliert werden können. Wir müssen zuerst jedoch eine zentrale Definition kennen lernen, die Definition der so genannten *ε-Hülle* eines Zustands. Informell ausgedrückt schließen wir einen Zustand q in eine ε-Hülle ein, indem wir allen von q ausgehenden Übergängen mit der Beschriftung ε folgen. Wenn wir durch die Verfolgung der mit ε beschrifteten Pfade zu anderen Zuständen gelangen, dann verfolgen wir dort die ε-Übergänge usw., sodass wir schließlich alle Zustände finden, die von q aus durch Verfolgung der mit ε beschrifteten Pfade erreicht werden können. Formal definieren wir die ε-Hülle ε-HÜLLE(q) induktiv:

INDUKTIONSBEGINN: Zustand q ist in ε-HÜLLE(q) enthalten.

INDUKTIONSSCHRITT: Wenn der Zustand p in ε-HÜLLE(q) enthalten ist und es einen Übergang vom Zustand p zum Zustand r mit der Beschriftung ε gibt, dann ist r in ε-HÜLLE(q) enthalten. Genauer gesagt, wenn δ die Übergangsfunktion des betreffenden ε-NEA und p in ε-HÜLLE(q) enthalten ist, dann umfasst ε-HÜLLE(q) auch alle in $\delta(p, \varepsilon)$ enthaltenen Zustände.

Beispiel 2.19 Bei dem Automaten aus Abbildung 2.13 stellt jeder Zustand seine eigene ε-Hülle dar, abgesehen von den folgenden beiden Ausnahmen: ε-HÜLLE(q_0) = $\{q_0, q_1\}$ und ε-HÜLLE(q_3) = $\{q_3, q_5\}$. Das ist deswegen so, weil es nur zwei ε-Übergänge gibt, nämlich den Übergang, der q_1 zu ε-HÜLLE(q_0) hinzufügt, und den Übergang, der q_5 zu ε-HÜLLE(q_3) hinzufügt.

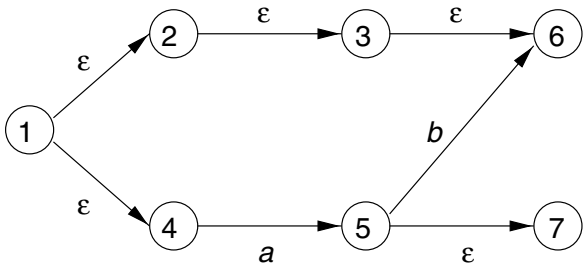
In Abbildung 2.15 ist ein komplexeres Beispiel dargestellt. Dafür gilt:

$$\varepsilon\text{-HÜLLE}(1) = \{1, 2, 3, 4, 6\}$$

Jeder dieser Zustände ist vom Zustand 1 aus über einen Pfad erreichbar, der lediglich die Beschriftungen ε trägt. Beispielsweise wird Zustand 6 über den Pfad $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ erreicht. Zustand 7 ist nicht in ε-HÜLLE(1) enthalten, da er zwar vom Zustand 1 aus erreichbar ist, aber hierzu die Strecke $4 \rightarrow 5$ im Pfad liegen muss, die nicht mit ε beschriftet ist. Die Tatsache, dass Zustand 6 auch über den Pfad $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ erreichbar ist, der nicht nur ε-Übergänge enthält, ist hier nicht relevant. Die Existenz eines

Pfades, der nur die Beschriftungen ε enthält, ist ausreichend für den Beweis, dass Zustand 6 in ε-HÜLLE(1) enthalten ist.

Abbildung 2.15: Einige Zustände und Übergänge



2.5.4 Erweiterte Übergänge und Sprachen für ε-NEAs

Mithilfe der ε-Hülle können wir einfach erklären, wie die Übergänge eines ε-NEA aussehen, wenn dieser eine Folge von Eingaben erhält, die ungleich der leeren Zeichenreihe ε sind. Davon ausgehend, können wir definieren, was bei einem ε-NEA das Akzeptieren der Eingabe bedeutet.

Angenommen, $E = (Q, \Sigma, \delta, q_0, F)$ sei ein ε-NEA. Wir definieren zuerst $\hat{\delta}$, die erweiterte Übergangsfunktion, um widerzuspiegeln, wie der ε-NEA auf eine Folge von Eingaben reagiert. Wir wollen zeigen, dass $\hat{\delta}(q, w)$ die Menge aller Zustände darstellt, die über Pfade erreicht werden, deren verkettete Beschriftungen die Zeichenreihe w ergeben. Wie immer tragen auf diesem Pfad liegende ε nicht zu w bei. Die entsprechende induktive Definition von $\hat{\delta}$ lautet:

INDUKTIONSBEGINN: $\hat{\delta}(q, \varepsilon) = \varepsilon$-HÜLLE$(q)$. Das heißt, wenn die Beschriftung des Pfads ε lautet, dann können wir nur die mit ε Beschriftungen versehenen Pfade verfolgen, die von Zustand q ausgehen. Dies entspricht genau der Definition von ε-HÜLLE.

INDUKTIONSSCHRITT: Angenommen, w habe die Form xa, wobei a das letzte Symbol von w sei. Beachten Sie, dass a ein Element von Σ ist. a kann daher nicht gleich ε sein, da ε nicht in Σ enthalten ist. Wir berechnen $\hat{\delta}(q, w)$ wie folgt:

1. Sei $\{p_1, p_2, ..., p_k\}$ gleich $\hat{\delta}(q, x)$. Das heißt, bei den Zuständen p_i handelt es sich ausschließlich um all jene Zustände, die von q aus entlang eines Pfades mit der Beschriftung x erreicht werden können. Dieser Pfad kann mit einem oder mehreren Übergängen mit der Beschriftung ε enden und auch andere ε-Übergänge enthalten.

2. Sei $\bigcup_{i=1}^{k} \delta(p_i, a)$ die Menge $\{r_1, r_2, ..., r_m\}$. Das heißt, wir verfolgen alle Übergänge mit der Beschriftung a, die von Zuständen ausgehen, die von q aus entlang eines Pfades mit der Beschriftung w erreichbar sind. Die Zustände r_j repräsentieren *einige* der Zustände, die vom Zustand q aus entlang Pfaden mit der Beschriftung w erreichbar sind. Weitere erreichbare Pfade lassen sich über die Zustände r_j ermitteln, indem wir die mit ε beschrifteten Pfade im nachfolgenden Schritt (3) verfolgen.

3. Dann gilt: $\hat{\delta}(q, w) = \bigcup_{i=1}^{m} \varepsilon\text{-HÜLLE}(r_i)$. Dieser zusätzliche Schritt umfasst alle von q ausgehenden Pfade mit der Beschriftung w unter Berücksichtigung der Möglichkeit, dass es zusätzliche mit ε beschriftete Pfade geben kann, denen wir folgen können, nachdem ein Übergang auf das letzte »echte« Symbol a hin erfolgte.

Beispiel 2.20 Wir wollen die Übergangsfunktion $\hat{\delta}(q_0, 5, 6)$ für den ε-NEA aus Abbildung 2.13 berechnen. Die erforderlichen Schritte lassen sich wie folgt zusammenfassen:

- $\hat{\delta}(q_0, \varepsilon) = \varepsilon\text{-HÜLLE}(q_0) = \{q_0, q_1\}$.
- Berechne $\hat{\delta}(q_0, 5)$ wie folgt:
  1. Berechne zuerst die Übergänge von den Zuständen q_0 und q_1, die wir aus obiger Berechnung von $\hat{\delta}(q_0, \varepsilon)$ erhielten, nach der Eingabe 5. Das heißt, wir berechnen $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.
  2. Berechne die ε-Hülle für die Elemente der Menge, die in Schritt (1) berechnet wurde. Wir erhalten $\varepsilon\text{-HÜLLE}(q_1) \cup \varepsilon\text{-HÜLLE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$. Diese Menge entspricht $\delta(q_0, 5)$. Das aus zwei Schritten bestehende Muster wiederholt sich bei den nächsten beiden Eingabesymbolen.
- Berechne $\hat{\delta}(q_0, 5.)$ wie folgt:
  1. Berechne zuerst $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$.
  2. Berechne dann $\hat{\delta}(q_0, 5.) = \varepsilon\text{-HÜLLE}(q_2) \cup \varepsilon\text{-HÜLLE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$.
- Berechne $\hat{\delta}(q_0, 5.6)$ wie folgt:
  1. Berechne zuerst $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \varnothing = \{q_3\}$.
  2. Berechne dann $\hat{\delta}(q_0, 5.6) = \varepsilon\text{-HÜLLE}(q_3) = \{q_3, q_5\}$. ■

Wir können nun die Sprache eines ε-NEA $E = (Q, \Sigma, \delta, q_0, F)$ auf die erwartete Weise definieren: $L(E) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$. Das heißt, bei der Sprache von E handelt es sich um die Menge der Zeichenreihen w, die den Startzustand in mindestens einen akzeptierenden Zustand überführen. Beispielsweise haben wir in Beispiel 2.20 gesehen, dass $\hat{\delta}(q_0, 5.6)$ den akzeptierenden Zustand q_5 enthält; folglich ist die Zeichenreihe 5.6 in der Sprache dieses ε-NEA enthalten.

2.5.5 ε-Übergänge eliminieren

Wir können für jeden beliebigen ε-NEA E einen DEA D finden, der die gleiche Sprache wie E akzeptiert. Hierzu verwenden wir eine Konstruktion, die der Teilmengenkonstruktion stark ähnelt, da die Zustände von D Teilmengen der Zustände von E sind. Der einzige Unterschied besteht darin, dass wir die ε-Übergänge von E berücksichtigen müssen, wozu wir den Mechanismus der ε-Hülle einsetzen.

Sei $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Dann wird der äquivalente DEA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

wie folgt definiert:

1. Q_D ist die Menge der Teilmengen von Q_E. Genauer gesagt, wir werden sehen, dass alle erreichbaren Zustände von D ε-abgeschlossene Teilmengen von Q_E sind, d. h. Mengen $S \subseteq Q_E$, derart dass $S = \varepsilon$-HÜLLE(S). Anders ausgedrückt, die ε-abgeschlossene Teilmengen $S \subseteq Q_E$ sind die, für die gilt, dass jeder ε-Übergang von einem in S enthaltenen Zustand zu einem ebenfalls in S enthaltenen Zustand führt. Beachten Sie, dass $\varnothing$ eine ε-abgeschlossene Menge ist.

2. $q_D = \varepsilon$-HÜLLE(q_0). Beachten Sie, dass sich diese Regel von der ursprünglichen Teilmengenkonstruktion unterscheidet, bei der der Startzustand des konstruierten Automaten einfach gleich der Menge ist, die nur den Startzustand des gegebenen NEA enthält.

3. F_D enthält die Mengen von Zuständen, die mindestens einen akzeptierenden Zustand von E enthalten. Das heißt, $F_D = \{ S \mid S$ ist Element von Q_D und $S \cap F_E \neq \varnothing$.

4. $\delta_D(S, a)$ wird für alle in Σ enthaltenen a und alle in Q_D enthaltenen S berechnet durch:

   a) Sei $S = \{p_1, p_2, ..., p_k\}$.

   b) Berechne $\bigcup_{i=1}^{k} \delta(p_i, a)$; sei diese Menge $\{r_1, r_2, ..., r_m\}$.

   c) Dann ist $\delta_D(S, a) = \bigcup_{j=1}^{m} \varepsilon$-HÜLLE (r_j).

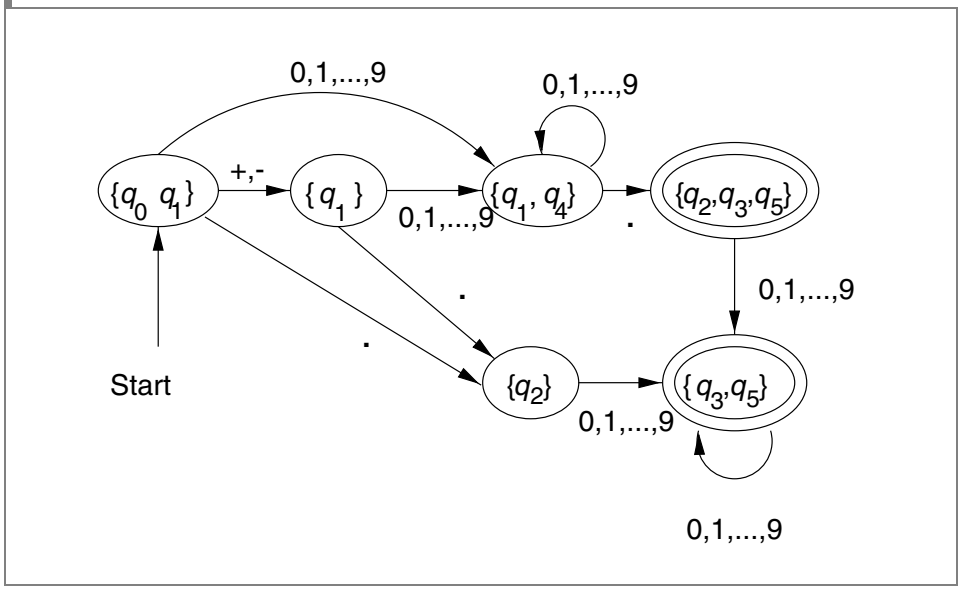
Beispiel 2.21 Wir wollen die ε-Übergänge aus dem ε-NEA von Abbildung 2.13 eliminieren, den wir im Folgenden E nennen werden. Aus E konstruieren wir den DEA D, der in Abbildung 2.16 dargestellt ist. Damit die Abbildung nicht zu unübersichtlich wird, haben wir in Abbildung 2.16 den Zustand $\varnothing$ und alle Übergänge in diesen Zustand weggelassen. Sie sollten sich vorstellen, dass es zu jedem der in Abbildung 2.16 dargestellten Zustände für jedes Eingabesymbol einen Übergang nach $\varnothing$ gibt, für das kein Übergang eingezeichnet ist. Der Zustand $\varnothing$ hat zudem für alle Eingabesymbole einen Übergang zu sich selbst.

Da q_0 der Startzustand von E ist, ist der Startzustand von D gleich ε-HÜLLE(q_0) und somit $\{q_0, q_1\}$. Zuerst müssen wir die Nachfolger von q_0 und q_1 für die verschiedenen in Σ enthaltenen Eingabesymbole finden. Beachten Sie, dass es sich bei diesen Symbolen um die Plus- und Minuszeichen, den Punkt und die Ziffern 0 bis 9 handelt. Auf die Eingabe von + und – hin, ist in Abbildung 2.13 für q_1 kein Übergang dargestellt, während q_0 dafür einen Übergang nach q_1 aufweist. Zur Berechnung von $\delta_D(\{q_0, q_1\},+)$ beginnen wir daher mit $\{q_1\}$ und berechnen ε-HÜLLE(q_1). Da keine ε-Übergänge von q_1 ausgehen, erhalten wir $\delta_D(\{q_0, q_1\}, +) = \{q_1\}$. Analog hierzu erhalten wir $\delta_D(\{q_0, q_1\}, -) = \{q_1\}$. Diese beiden Übergänge werden in Abbildung 2.16 durch einen Pfeil dargestellt.

Als Nächstes müssen wir $\delta_D(\{q_0, q_1\}, .)$ berechnen. Da in Abbildung 2.13 von q_0 auf die Eingabe Punkt (.) hin kein Übergang zu einem anderen Zustand erfolgt und q_1 in q_2 überführt wird, müssen wir ε-HÜLLE(q_2) berechnen. Daher ist $\delta_D(\{q_0, q_1\}, .) = \{q_2\}$.

Schließlich müssen wir $\delta_D(\{q_0, q_1\}, 0)$ als ein Beispiel für die Übergänge von $\{q_0, q_1\}$ auf die verschiedenen Eingabesymbole hin berechnen. Wir stellen fest, dass von q_0 auf die Eingabe von Ziffern hin kein Übergang zu einem anderen Zustand erfolgt, aber q_1

Abbildung 2.16: Der DEA *D*, in dem die ε-Übergänge des NEA von Abbildung 2.13 eliminiert sind



sowohl in q_1 als auch q_4 überführt wird. Da von diesen Zuständen keine ε-Übergänge ausgehen, schließen wir, dass $\delta_D(\{q_0, q_1\}, 0) = \{q_1, q_4\}$ und dass dies auch für die anderen Ziffern gilt.

Wir haben nun die in Abbildung 2.16 dargestellten Pfeile, die von $\{q_0, q_1\}$ ausgehen, erklärt. Die übrigen Übergänge werden auf ähnliche Weise berechnet. Wir überlassen dies dem Leser. Da q_5 der einzige akzeptierende Zustand von E ist, handelt es sich bei den akzeptierenden Zuständen von D um die erreichbaren Zustände, die q_5 enthalten. Diese beiden Mengen sind in Abbildung 2.16 durch eine doppelte Kreislinie gekennzeichnet. ■

Satz 2.22 Eine Sprache L wird genau dann von einem ε-NEA akzeptiert, wenn L von einem DEA akzeptiert wird.

Beweis: (Wenn-Teil) Der Beweis ist in dieser Richtung einfach. Angenommen, für einen DEA sei $S = L(D)$. Wir wandeln D in einen ε-DEA E um, indem wir für alle Zustände q von D die Übergänge $\delta(q, \varepsilon) = \emptyset$ hinzufügen. Technisch gesehen, müssen wir auch die Übergänge von D für Eingabesymbole umwandeln, z. B. $\delta_D(q, a) = p$ in einen NEA-Übergang zu der Menge, die nur p enthält, also $\delta_E(q, a) = \{p\}$. Folglich verfügen E und D im Wesentlichen über dieselben Übergänge, aber E gibt explizit an, dass es aus keinem Zustand Übergänge für die Eingabe ε gibt.

(Nur-dann-Teil) Sei $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ ein ε-NEA. Wir wenden die oben beschriebene modifizierte Teilmengenkonstruktion an, um den folgenden DEA zu gewinnen:

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

Wir müssen nun zeigen, dass $L(D) = L(E)$, und tun dies, indem wir zeigen, dass die erweiterten Übergangsfunktionen von D und E identisch sind. Formal zeigen wir durch Induktion über die Länge von w, dass gilt: $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$.

INDUKTIONSBEGINN: Wenn $|w| = 0$, dann $w = \varepsilon$. Wir wissen, dass gilt $\hat{\delta}_E(q_0, \varepsilon) = \varepsilon$-HÜLLE$(q_0)$. Wir wissen zudem, dass $q_D = \varepsilon$-HÜLLE(q_0), weil der Startzustand von D so definiert ist. Wir wissen schließlich, dass für einen DEA für jeden Zustand p $\hat{\delta}_D(p, \varepsilon) = p$ gilt, daher gilt auch $\hat{\delta}_D(q_D, \varepsilon) = \varepsilon$-HÜLLE$(q_0)$. Somit haben wir bewiesen, dass $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$.

INDUKTIONSSCHRITT: Angenommen, $w = xa$, wobei a das letzte Symbol von w ist, und nehmen wir weiter an, dass die Aussage für x gilt. Das heißt, $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x)$. Diese beiden Zustandsmengen seien $\{p_1, p_2, ..., p_k\}$.

Gemäß der Definition von $\hat{\delta}$ für ε-NEAs, berechnen wir $\hat{\delta}_E(q_0, w)$ wie folgt:

1. Sei $\{r_1, r_2, ... r_m\}$ gleich $\bigcup_{i=1}^{k} \delta_E(p_i, a)$.
2. Dann gilt $\hat{\delta}_E(q_0, w) = \bigcup_{j=1}^{m} \varepsilon\text{-HÜLLE}(r_j)$.

Wenn wir die Konstruktion des DEA D in der modifizierten Teilmengenkonstruktion überprüfen, dann stellen wir fest, dass auch $\delta_D(\{p_1, p_2, .. p_k\}, a)$ mit den oben beschriebenen Schritten 1) und 2) konstruiert wird. Folglich handelt es sich bei $\hat{\delta}_D(q_D, w)$, also $\delta_D(\{p_1, p_2, .. p_k\}, a)$, um dieselbe Menge wie $\hat{\delta}_E(q_0, w)$. Damit haben wir bewiesen, dass $\hat{\delta}_D(q_D, w) = \hat{\delta}_E(q_0, w)$, und den Induktionsschritt damit abgeschlossen. ■

2.5.6 Übungen zum Abschnitt 2.5

* **Übung 2.5.1** Betrachten Sie den folgenden ε-NEA:

	ε	a	b	c
$\rightarrow p$	$\varnothing$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\varnothing$
$* r$	$\{q\}$	$\{r\}$	$\varnothing$	$\{p\}$

a) Berechnen Sie die ε-HÜLLE für jeden Zustand.

b) Nennen Sie alle Zeichenreihen mit der Länge von drei oder weniger Zeichen, die von dem Automaten akzeptiert werden.

c) Wandeln Sie den Automaten in einen DEA um.

Übung 2.5.2 Führen Sie die Übung 2.5.1 für den folgenden ε-NEA aus:

	ε	a	b	c
$\rightarrow p$	$\{q, r\}$	$\varnothing$	$\{q\}$	$\{r\}$
q	$\varnothing$	$\{p\}$	$\{r\}$	$\{p, q\}$
$* r$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Übung 2.5.3 Entwerfen Sie ε-NEAs für die folgenden Sprachen. Versuchen Sie, ε-Übergänge zu verwenden, um Ihren Entwurf zu vereinfachen.

a) Die Menge aller Zeichenreihen, die aus null oder mehr Buchstaben *a*, gefolgt von null oder mehr Buchstaben *b*, gefolgt von null oder mehr Buchstaben *c* bestehen

! b) Die Menge aller Zeichenreihen, die entweder aus der ein- oder mehrmaligen Wiederholung der Zeichenreihe 01 oder aus der ein- oder mehrmaligen Wiederholung der Zeichenreihe 010 bestehen

! c) Die Menge aller aus Nullen und Einsen bestehenden Zeichenreihen, derart, dass mindestens eine der letzten zehn Stellen eine 1 ist

2.6 Zusammenfassung von Kapitel 2

- *Deterministische endliche Automaten (DEA)*: Ein DEA verfügt über eine endliche Menge von Zuständen und eine endliche Menge von Eingabesymbolen. Ein Zustand wird als Startzustand festgelegt, und null oder mehr Zustände sind akzeptierende Zustände. Eine Übergangsfunktion bestimmt, wie sich der Zustand auf die Verarbeitung eines Eingabesymbols hin verändert.
- *Übergangsdiagramme* : Automaten lassen sich durch einen Graphen darstellen, in dem die Knoten die Zustände repräsentieren und die mit Eingabesymbolen beschrifteteten Pfeile die Zustandsänderungen des Automaten kennzeichnen. Der Startzustand wird durch einen Pfeil gekennzeichnet und die akzeptierenden Zustände durch doppelte Kreislinien.
- *Sprache eines Automaten*: Der Automat akzeptiert Zeichenreihen. Eine Zeichenreihe wird akzeptiert, wenn die zeichenweise Verarbeitung der in dieser Zeichenreihe enthaltenen Eingabesymbole Übergänge bewirkt, die vom Startzustand zu einem akzeptierenden Zustand führen. Aus Übergangsdiagrammen ist erkennbar, ob eine Zeichenreihe akzeptiert wird, wenn sie die Beschriftung eines Pfades bildet, der vom Startzustand zu einem akzeptierenden Zustand führt.
- *Nichtdeterministische endliche Automaten (NEA)*: Der NEA unterscheidet sich vom DEA dadurch, dass der NEA zu einem gegebenen Zustand für jedes Eingabesymbol eine beliebige Anzahl (einschließlich Null) von Übergängen zu anderen Zuständen haben kann.
- *Teilmengenkonstruktion*: Indem die Zustandsmengen eines NEA als Zustände eines DEA behandelt werden, kann ein NEA in einen DEA umgewandelt werden, der die gleiche Sprache akzeptiert.
- *ε-Übergänge* : Wir können den NEA erweitern, indem wir Übergänge für eine leere Eingabe, d. h. ohne ein Zeichen, zulassen. Diese erweiterten NEAs können in DEAs umgewandelt werden, die dieselbe Sprache akzeptieren.
- *Textsuchanwendungen*: Nichtdeterministische endliche Automaten stellen eine hilfreiche Möglichkeit dar, einen Mustererkenner zu beschreiben, der umfangreiche Texte nach einem oder mehreren Schlüsselwörtern durchsucht. Diese Automaten werden entweder direkt durch Software simuliert oder zuerst in einen DEA umgewandelt, der dann simuliert wird.

LITERATURANGABEN ZU KAPITEL 2

Als Beginn des formalen Studiums von Systemen mit endlichen Zuständen wird im Allgemeinen [2] erachtet. Dieses Werk basierte jedoch auf dem Rechenmodell der »neuronalen Netze« statt auf den endlichen Automaten, wie wir sie heute kennen. Der konventionelle DEA wurde von [1], [3] und [4] voneinander unabhängig in ähnlichen Varianten beschrieben. Der nichtdeterministische endliche Automat und die Teilmengenkonstruktion stammen von [5].

1. D. A. Huffman [1954]. »The syntheses of sequential switching circuits«, *J. Franklin Inst.* **257**:3–4, S. 161–190 und 275–303.
2. W. S. McCulloch und W. Pitts [1943]. »A logical calculus of the ideas immanent in nervious activity«, *Bull. Math. Biophysics* **5**, S. 115–133.
3. G. H. Mealy [1955]. »A method for synthesizing sequential circuits«, *Bell System Technical Journal* **34**:5, S. 1045–1079.
4. E. F. Moore. »Gedanken experiments on sequential machines«, in [6], S. 129–153.
5. M. O. Rabin und D. Scott [1959]. »Finite automata and their decision problems«, *IBM J. Research and Development* **3**:2, S. 115–125.
6. C. E. Shannon und J. McCarthy [1956]. *Automata Studies*, Princeton Univ. Press.

KAPITEL 3

Reguläre Ausdrücke und Sprachen

Wir beginnen dieses Kapitel mit der Einführung einer Notation namens »reguläre Ausdrücke«. Bei diesen Ausdrücken handelt es sich um eine andere Art von sprachdefinierender Notation, für die wir in Abschnitt 1.1.2 ein kurzes Beispiel anführten. Man kann sich reguläre Ausdrücke auch als »Programmiersprache« vorstellen, in der einige wichtige Anwendungen wie Suchmaschinen oder Compilerkomponenten beschrieben werden. Reguläre Ausdrücke sind mit nichtdeterministischen endlichen Automaten eng verwandt und können als »benutzerfreundliche« Alternative zur NEA-Notation zur Beschreibung von Softwarekomponenten betrachtet werden.

In diesem Kapitel zeigen wir nach der Definition regulärer Ausdrücke, dass diese Ausdrücke zur Definition aller regulären Sprachen und keiner anderen Sprachen taugen. Wir erörtern die Art und Weise, in der reguläre Ausdrücke in verschiedenen Softwaresystemen eingesetzt werden. Anschließend untersuchen wir die algebraischen Gesetze, die für reguläre Ausdrücke gelten. Obwohl sie den algebraischen Gesetzen der Arithmetik stark ähneln, gibt es einige bedeutende Unterschiede zwischen der Algebra regulärer Ausdrücke und der Algebra arithmetischer Ausdrücke.

3.1 Reguläre Ausdrücke

Wir wenden unsere Aufmerksamkeit nun von maschinenähnlichen Beschreibungen von Sprachen – deterministischen und nichtdeterministischen endlichen Automaten – ab und einer algebraischen Beschreibung zu, nämlich den »regulären Ausdrücken«. Wir werden feststellen, dass reguläre Ausdrücke genau die gleichen Sprachen definieren können, die von den verschiedenen Arten von Automaten beschrieben werden: die regulären Sprachen. Im Gegensatz zu Automaten stellen reguläre Ausdrücke jedoch ein deklaratives Verfahren zur Beschreibung der Zeichenreihen zur Verfügung, die akzeptiert werden sollen. Daher dienen reguläre Ausdrücke als Eingabesprache vieler Systeme, die Zeichenreihen verarbeiten. Beispiele hierfür sind unter anderem:

- Suchbefehle wie der Unix-Befehl `grep` und äquivalente Befehle zur Suche nach Zeichenreihen, die in Webbrowsern oder Textformatiersystemen eingesetzt werden. Diese Systeme verwenden eine regulären Ausdrücken ähnliche Notation zur Beschreibung von Mustern, die der Benutzer in einer Datei sucht. Verschiedene

Suchsysteme wandeln den regulären Ausdruck in einen DEA oder NEA um und simulieren diesen Automaten beim Durchsuchen der gegebenen Datei.

- Generatoren lexikalischer Analysekomponenten, wie z. B. Lex oder Flex. Wie Sie wissen, handelt es sich bei einer lexikalischen Analysekomponente um die Compilerkomponente, die das Quellprogramm in logische Einheiten (so genannte *Token*) zerlegt. Diese logischen Einheiten können ein oder mehrere Zeichen umfassen, die eine gemeinsame Bedeutung haben. Beispiele für Token sind Schlüsselwörter (z. B. `while`), Bezeichner (z. B. ein beliebiger Buchstabe, dem null oder mehr Buchstaben und/oder Ziffern folgen) und Operatorzeichen wie +, - und <=. Ein Generator einer lexikalischen Analysekomponente akzeptiert Beschreibungen der Form der Token, die im Grunde genommen reguläre Ausdrücke sind, und erzeugt einen DEA, der erkennt, welcher Token in der Eingabe als Nächstes folgt.

3.1.1 Die Operatoren regulärer Ausdrücke

Reguläre Operatoren beschreiben Sprachen. Als einfaches Beispiel sei der reguläre Ausdruck $\mathbf{01^* + 10^*}$ angeführt, der die Sprache beschreibt, die aus allen Zeichenreihen besteht, die sich aus einer einzelnen führenden Null und einer beliebigen Anzahl von nachfolgenden Einsen oder aus einer einzelnen führenden Eins und einer beliebigen Anzahl von nachfolgenden Nullen zusammensetzen. Wir erwarten von Ihnen noch nicht, dass Sie reguläre Ausdrücke interpretieren können, und daher müssen Sie unsere Aussage über die Sprache dieses Ausdrucks jetzt einfach erst einmal glauben. Wir werden in Kürze alle Symbole definieren, die in diesem Ausdruck verwendet werden, damit Sie sich selbst davon überzeugen können, dass unsere Interpretation dieses regulären Ausdrucks korrekt ist. Bevor wir die Notation regulärer Ausdrücke beschreiben, müssen wir drei Operationen auf Sprachen vorstellen, die von den Operatoren regulärer Ausdrücke repräsentiert werden. Es handelt sich um folgende Operationen:

1. Die *Vereinigung* zweier Sprachen, L und M, angegeben durch $L \cup M$, ist die Menge aller Zeichenreihen, die entweder in L oder M oder in beiden Sprachen enthalten sind. Wenn z. B. $L = \{001, 10, 111\}$ und $M = \{\varepsilon, 001\}$, dann gilt $L \cup M = \{\varepsilon, 10, 001, 111\}$.

2. Die *Verkettung* (oder Konkatenation) der Sprachen L und M ist die Menge aller Zeichenreihen, die gebildet werden, indem eine Zeichenreihe aus L mit einer beliebigen Zeichenreihe aus M verkettet wird. Rufen Sie sich Abschnitt 1.5.2 in Erinnerung, in dem wir die Verkettung eines Zeichenreihenpaars definiert haben. Eine Zeichenreihe wird an eine andere angehängt, um das Ergebnis der Verkettung zu bilden. Wir geben die Verkettung von Sprachen entweder durch einen Punkt oder durch gar keinen Operator an, auch wenn der Konkatenations- oder Verkettungsoperator häufig »Punkt« genannt wird. Wenn beispielsweise $L = \{001, 10, 111\}$ und $M = \{\varepsilon, 001\}$, dann gilt $L.M$ oder einfach $LM = \{001, 10, 111, 001001, 10001, 111001\}$. Die ersten drei Zeichenreihen von LM sind die mit ε verketteten Zeichenreihen aus L. Da ε die Identität der Verkettung ist, entsprechen die resultierenden Zeichenreihen den Zeichenreihen aus L. Die letzten drei Zeichenreihen von LM werden jedoch gebildet, indem die einzelnen Zeichenreihen von L mit der zweiten Zeichenreihe von M, also 001, verket-

tet werden. Beispielsweise ergibt die Verkettung der Zeichenreihe 10 aus L mit der Zeichenreihe 001 aus M die Zeichenreihe 10001 in LM.

3. Die *Kleenesche Hülle* [1] (oder einfach *Hülle* oder *Stern* genannt) einer Sprache L wird durch L^* angegeben und beschreibt die Menge aller Zeichenreihen, die durch die Verkettung einer beliebigen Anzahl von Zeichenreihen aus L (wobei Wiederholungen zulässig sind, d. h. dieselbe Zeichenreihe kann mehrmals verwendet werden) gebildet werden. Wenn z. B. $L = \{0, 1\}$, dann enthält L^* alle Zeichenreihen aus Nullen und Einsen. Wenn $L = \{0, 11\}$, dann enthält L^* alle Zeichenreihen aus Nullen und Einsen, in denen die Einsen paarweise auftreten, also z. B. 011, 11110 und ε, nicht jedoch 01011 oder 101. Formal ausgedrückt ist L^* die unendliche Vereinigung $\bigcup_{i \geq 0} L^i$, wobei $L^0 = \{\varepsilon\}$, $L^1 = L$ und L^i für $i > 1$ gleich $LL ... L$ (die Verkettung von i Exemplaren von L) ist.

Beispiel 3.1 Da das Konzept der Hülle einer Sprache nicht banal ist, wollen wir einige Beispiele betrachten. Sei $L = \{0, 11\}$. Unabhängig davon, um was für eine Sprache es sich bei L handelt, gilt stets $L^0 = \{\varepsilon\}$; L hoch null repräsentiert die Auswahl von null Zeichenreihen aus L. $L^1 = L$ steht also für die Auswahl einer Zeichenreihe aus L. Die ersten beiden Terme der Erweiterung von L^* ergeben somit $\{\varepsilon, 0, 11\}$.

Als Nächstes betrachten wir L^2. Wir wählen zwei Zeichenreihen aus L aus, und da Wiederholungen zulässig sind, gibt es vier Möglichkeiten. Diese vier Kombinationen ergeben $L^2 = \{00, 011, 110, 1111\}$. Analog hierzu ist L^3 die Menge der Zeichenreihen, die sich durch Verkettung von drei aus L gewählten Zeichenreihen ergeben, also

$$\{000, 0011, 0110, 01111, 1100, 11011, 11110, 111111\}$$

Zur Berechnung von L^* müssen wir L^i für jedes i berechnen und alle diese Sprachen vereinigen. L^i enthält 2^i Elemente. Obwohl jede Sprache L^i endlich ist, ergibt die Vereinigung der unendlichen Anzahl von Termen L^i im Allgemeinen wie in unserem Beispiel eine unendliche Sprache.

Nehmen wir nun an, L sei die Menge aller aus Nullen bestehenden Zeichenreihen. Beachten Sie, dass L hier unendlich ist, im Gegensatz zum vorigen Beispiel, in dem L eine endliche Sprache war. Es ist allerdings nicht schwierig, L^* zu ermitteln. Wie immer gilt: $L^0 = \{\varepsilon\}$. $L^1 = L$. L^2 ist die Menge der Zeichenreihen, die gebildet werden, indem man eine Zeichenreihe aus Nullen mit einer anderen Zeichenreihe aus Nullen verkettet. Auch das Ergebnis ist eine Zeichenreihe aus Nullen. Jede Zeichenreihe aus Nullen kann als Verkettung zweier Zeichenreihen aus Nullen dargestellt werden. (Denken Sie daran, dass ε eine »Zeichenreihe aus Nullen« ist; in der Verkettung zweier Zeichenreihen können wir in jedem Fall diese Zeichenreihe verwenden.) Folglich gilt $L^2 = L$. Analog gilt $L^3 = L$. Folglich ist in dem speziellen Fall, in dem die Sprache L die Menge aller aus Nullen bestehenden Zeichenreihen umfasst, die unendliche Vereinigung $L^* = L^0 \cup L^1 \cup L^2 \cup ...$ gleich L.

Als letztes Beispiel wollen wir die Sprache $\varnothing^* = \{\varepsilon\}$ betrachten. Beachten Sie, dass $\varnothing^0 = \{\varepsilon\}$, während $\varnothing^i$ für jedes $i > 1$ leer ist, da wir aus einer leeren Menge keine Zeichenreihe auswählen können. In der Tat stellt $\varnothing$ eine der beiden einzigen Sprachen dar, deren Hülle *nicht* unendlich ist. ■

1. Der Begriff »Kleenesche Hülle« geht auf S. C. Kleene zurück, von dem die Notation der regulären Ausdrücke und dieser Operator stammen.

Verwendung des Sternoperators

Der Sternoperator wurde im Abschnitt 1.5.2 zum ersten Mal erwähnt, in dem wir ihn auf ein Alphabet anwendeten, z. B. Σ^*. Dieser Operator bildete sämtliche Zeichenreihen, deren Symbole aus dem Alphabet Σ stammten. Der Kleene-Operator (oder Hüllen-Operator) ist im Grunde genommen dasselbe, obwohl ein subtiler Typunterschied zu beachten ist.

Angenommen, L sei die Sprache, die aus Zeichenreihen der Länge 1 besteht, und für jedes in Σ enthaltene Zeichen a gäbe es eine Zeichenreihe a in L. Dann haben L und Σ zwar dasselbe »Aussehen«, sind jedoch unterschiedlichen Typs. L ist eine Menge von Zeichenreihen, und Σ ist eine Menge von Symbolen. Andererseits bezeichnet L^* dieselbe Sprache wie Σ^*.

3.1.2 Reguläre Ausdrücke bilden

Jegliche Art von Algebra setzt auf gewissen elementaren Ausdrücken auf, für gewöhnlich Konstanten und/oder Variablen. Durch die Anwendung bestimmter Operatoren auf diese elementaren Ausdrücke und auf bereits gebildete Ausdrücke können dann weitere Ausdrücke gebildet werden. Für gewöhnlich ist zudem eine Methode zur Gruppierung der Operatoren und ihrer Operanden erforderlich, wie z. B. das Klammern. Die bekannte arithmetische Algebra beginnt beispielsweise mit Konstanten (z. B. ganze und reelle Zahlen) und Variablen und ermöglicht die Bildung komplexerer Ausdrücke unter Verwendung der arithmetischen Operatoren wie + und x.

Die Arithmetik regulärer Ausdrücke folgt diesem Muster insofern, als dass Konstanten und Variablen zur Bezeichnung von Sprachen und Operatoren für die in Abschnitt 3.1.1. beschriebenen Operationen – Vereinigung, Punkt und Stern – verwendet werden. Wir können die regulären Ausdrücke wie folgt induktiv beschreiben. In dieser Definition legen wir nicht nur fest, was zulässige reguläre Ausdrücke sind, sondern wir beschreiben für jeden regulären Ausdruck E auch die von diesem Ausdruck repräsentierte Sprache, die wir mit $L(E)$ angeben.

INDUKTIONSBEGINN: Er besteht aus drei Teilen:

1. Die Konstanten ε und $\emptyset$ sind reguläre Ausdrücke, die die Sprache $\{\varepsilon\}$ bzw. $\emptyset$ beschreiben. Das heißt, $L(\varepsilon) = \{\varepsilon\}$ und $L(\emptyset) = \emptyset$.
2. Wenn a ein beliebiges Symbol ist, dann ist **a** ein regulärer Ausdruck. Dieser Ausdruck beschreibt die Sprache $\{a\}$. Das heißt, $L(\mathbf{a}) = \{a\}$. Beachten Sie, dass wir einen Ausdruck, der einem Symbol entspricht, durch Fettschrift angeben. Die Zuordnung, d. h. dass **a** sich auf a bezieht, sollte offensichtlich sein.
3. Variablen, die für gewöhnlich als kursive Großbuchstaben wie L angegeben werden, repräsentieren eine Sprache.

INDUKTIONSSCHRITT: Der Induktionsschritt umfasst vier Teile, jeweils einen für die drei Operatoren und einen für die Einführung von Klammern.

1. Wenn E und F reguläre Ausdrücke sind, dann ist $E + F$ ein regulärer Ausdruck, der die Vereinigung von $L(E)$ und $L(F)$ angibt. Das heißt, $L(E + F) = L(E) \cup L(F)$.
2. Wenn E und F reguläre Ausdrücke sind, dann ist EF ein regulärer Ausdruck, der die Verkettung von $L(E)$ und $L(F)$ angibt. Das heißt, $L(EF) = L(E)L(F)$.

Beachten Sie, dass der Punkt optional für den Verkettungsoperator angegeben werden kann, der für eine Operation auf Sprachen oder als Operator in regulären Ausdrücken verwendet werden kann. Beispielsweise ist **0.1** ein regulärer Ausdruck, der gleichbedeutend ist mit **01** und die Sprache {01} beschreibt. Wir werden den Punkt als Verkettungsoperator in regulären Ausdrücken vermeiden[2].

3. Wenn E ein regulärer Ausdruck ist, dann ist E^* ein regulärer Ausdruck, der die Hülle von $L(E)$ angibt. Das heißt $L(E^*) = (L(E))^*$.

4. Wenn E ein regulärer Ausdruck ist, dann ist auch (E) – der in Klammern gesetzte Bezeichner – ein regulärer Ausdruck, der die gleiche Sprache wie E beschreibt. Formal ausgedrückt: $L((E)) = L(E)$.

Ausdrücke und ihre Sprachen

Streng genommen ist ein regulärer Ausdruck E einfach ein Ausdruck und keine Sprache. Wir sollten $L(E)$ verwenden, wenn wir auf die Sprache Bezug nehmen wollen, die von E beschrieben wird. Es ist jedoch übliche Praxis, einfach »E« zu sagen, wenn eigentlich »$L(E)$« gemeint ist. Wir werden diese Konvention hier verwenden, solange klar ist, dass von einer Sprache und nicht von einem regulären Ausdruck die Rede ist.

Beispiel 3.2 Wir sollen einen regulären Ausdruck formulieren, der die Menge der Zeichenreihen beschreibt, die aus in abwechselnder Reihenfolge stehenden Nullen und Einsen bestehen. Wir wollen zuerst einen regulären Ausdruck für die Sprache entwickeln, die lediglich die Zeichenreihe 01 beinhaltet. Wir können dann mithilfe des Sternoperators einen Ausdruck bilden, der für alle Zeichenreihen der Form 0101 ... 01 steht.

Die Grundregel für reguläre Ausdrücke besagt, dass **0** und **1** Ausdrücke sind, die die Sprachen {0} bzw. {1} repräsentieren. Wenn wir die beiden Ausdrücke verketten, erhalten wir einen regulären Ausdruck für die Sprache {01}. Dieser Ausdruck lautet **01**. Als allgemeine Regel gilt, dass wir als regulären Ausdruck zur Beschreibung einer Sprache, die nur aus der Zeichenreihe w besteht, die Zeichenreihe w selbst als regulären Ausdruck einsetzen können. Beachten Sie, dass im regulären Ausdruck die in w enthaltenen Symbole normalerweise in Fettschrift dargestellt werden, aber diese Änderung der Schriftart soll es Ihnen lediglich erleichtern, Ausdrücke von Zeichenreihen zu unterscheiden, und hat keine weitere Bewandtnis.

Um alle Zeichenreihen zu erhalten, die aus null oder mehr Vorkommen von 01 bestehen, verwenden wir den regulären Ausdruck $(\mathbf{01})^*$. Beachten Sie, dass wir **01** in Klammer setzen, um Verwechslungen mit dem Ausdruck **01*** zu vermeiden, der die Sprache beschreibt, die alle Zeichenreihen umfasst, die sich aus einer führenden Null und einer beliebigen Anzahl von nachfolgenden Einsen zusammensetzen. Warum dieser Ausdruck so interpretiert wird, wird in Abschnitt 3.1.3 erklärt; hier soll der Hinweis genügen, dass der Sternoperator Vorrang vor dem Punktoperator hat und das Argument des Sternoperators daher zuerst ausgewählt wird, bevor eine Verkettung durchgeführt wird.

2. In regulären Ausdrücken unter Unix hat der Punkt sogar eine völlig andere Bedeutung: Hier repräsentiert er ein beliebiges ASCII-Zeichen.

Allerdings ist $L((\mathbf{01})^*)$ nicht genau die Sprache, die gewünscht ist. Sie umfasst nur solche Zeichenreihen aus in abwechselnder Reihenfolge stehender Nullen und Einsen, die mit null beginnen und mit 1 enden. Wir müssen zudem die Möglichkeit betrachten, dass eine Eins am Anfang und/oder eine Null am Ende steht. Ein Ansatz besteht darin, drei oder mehr reguläre Ausdrücke zu bilden, die die anderen drei Möglichkeiten abdecken. Das heißt, $(\mathbf{10})^*$ repräsentiert die Zeichenreihen, die mit einer Eins beginnen und einer Null enden, während $\mathbf{0}(\mathbf{10})^*$ für Zeichenreihen steht, die sowohl mit einer Null beginnen als auch mit einer Null enden, und $\mathbf{1}(\mathbf{01})^*$ Zeichenreihen beschreibt, die sowohl mit einer Eins beginnen als auch mit einer Eins enden. Der gesamte reguläre Ausdruck lautet:

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Beachten Sie, dass wir den Operator + verwendet haben, um die Vereinigung der vier Sprachen zu bilden, die insgesamt sämtliche Zeichenreihen umfasst, in denen Nullen und Einsen einander abwechseln.

Es gibt jedoch noch einen anderen Ansatz, der einen völlig anders aussehenden regulären Ausdruck ergibt und überdies etwas prägnanter ist. Wir beginnen wieder mit dem Ausdruck $(\mathbf{01})^*$. Wir können eine optionale Eins am Anfang hinzufügen, wenn wir Links mit dem Ausdruck $\varepsilon + \mathbf{1}$ verketten. Analog können wir mit dem Ausdruck $\varepsilon + \mathbf{0}$ eine optionale Null am Ende hinzufügen. Unter Verwendung der Definition des Operators + ergibt sich beispielsweise:

$$L(\varepsilon + \mathbf{1}) = \mathrm{L}(\varepsilon) \cup \mathrm{L}(\mathbf{1}) = \{\varepsilon\} \cup \{\mathbf{1}\} = \{\varepsilon, \mathbf{1}\}$$

Wenn wir diese Sprache mit einer beliebigen Sprache L verketten, dann erhalten wir durch die Verkettung mit ε alle in L enthaltenen Zeichenreihen, während die Verkettung von 1 mit jeder Zeichenreihe w in L die Zeichenreihe $1w$ ergibt. Folglich lautet ein anderer Ausdruck zur Beschreibung der Menge der Zeichenreihen, die abwechselnd Nullen und Einsen enthalten:

$$(\varepsilon + \mathbf{1})\ (\mathbf{01})^*(\varepsilon + \mathbf{0})$$

Beachten Sie, dass die Ausdrücke mit ε jeweils in Klammern gesetzt werden müssen, damit die Operatoren korrekt ausgewertet werden. ■

3.1.3 Auswertungsreihenfolge der Operatoren regulärer Ausdrücke

Wie in jeder anderen Algebra gelten für die Operatoren regulärer Ausdrücke festgelegte »Vorrangregeln«, das heißt, dass die Operatoren in einer bestimmten Reihenfolge ihren Operanden zugeordnet werden. Dieses Konzept der Vorrangregel ist uns von gewöhnlichen arithmetischen Ausdrücken bekannt. Wir wissen beispielsweise, dass im Ausdruck $xy + z$ das Produkt xy vor der Summe berechnet wird, und daher ist er gleichbedeutend mit dem Klammerausdruck $(xy) + z$ und nicht mit dem Ausdruck $x(y + z)$. Zwei gleiche Operatoren werden in der Arithmetik von links nach rechts ausgewertet, sodass $x - y - z$ gleichbedeutend ist mit $(x - y) - z$ und nicht mit $x - (y - z)$. Für die Operatoren regulärer Ausdrücke gilt die folgende Auswertungsreihenfolge:

1. Der Sternoperator hat die höchste Priorität. Das heißt, er wird nur auf die kleinste Sequenz von Symbolen auf seiner linken Seite angewandt, die einen wohl geformten regulären Ausdruck darstellen.

2. Der Verkettungsoperator bzw. der Operator »Punkt« folgt an zweiter Stelle in der Auswertungsreihenfolge. Nachdem alle Sterne ihren Operanden zugeordnet wurden, werden die Verkettungsoperatoren ihren Operanden zugeordnet. Das heißt, alle unmittelbar aufeinander folgenden Ausdrücke werden gruppiert. Da der Verkettungsoperator assoziativ ist, ist es gleichgültig, in welcher Reihenfolge aufeinander folgende Verkettungen gruppiert werden. Falls aber eine Gruppierung erwünscht ist, sollten die Ausdrücke von links nach rechts gruppiert werden; z. B. wird **012** gruppiert in (**01**)**2**.
3. Zuletzt werden alle Vereinigungsoperatoren (+) ihren Operanden zugeordnet. Da die Vereinigung assoziativ ist, spielt es auch hier keine Rolle, in welcher Reihenfolge aufeinander folgende Vereinigungen gruppiert werden; eine Gruppierung sollte aber auch in diesem Fall von links nach rechts erfolgen.

Natürlich kann es vorkommen, dass wir die Operanden anders gruppieren möchten, als durch die Auswertungsreihenfolge der Operatoren vorgegeben. In diesem Fall können wir die Operanden mithilfe runder Klammern nach Belieben gruppieren. Zudem schadet es nichts, die Operanden in Klammern zu setzen, die gruppiert werden sollen, auch wenn die gewünschte Gruppierung durch die Vorrangregeln bereits impliziert wird.

3.1.4 Übungen zum Abschnitt 3.1

Übung 3.1.1 Schreiben Sie reguläre Ausdrücke für die folgenden Sprachen:

* a) Die Menge der Zeichenreihen über dem Alphabet $\{a, b, c\}$, die mindestens ein a und mindestens ein b enthalten

b) Die Menge der Zeichenreihen aus Nullen und Einsen, deren zehntes Symbol von rechts eine Eins ist

c) Die Menge der Zeichenreihen aus Nullen und Einsen, die mindestens ein Paar aufeinander folgender Einsen enthalten

! Übung 3.1.2 Formulieren Sie reguläre Ausdrücke für folgende Sprachen:

* a) Die Menge aller Zeichenreihen aus Nullen und Einsen, derart dass alle Paare aufeinander folgender Nullen vor allen Paaren aufeinander folgender Einsen stehen

b) Die Menge der Zeichenreihen aus Nullen und Einsen, deren Anzahl von Nullen durch 5 teilbar ist

!! Übung 3.1.3 Formulieren Sie reguläre Ausdrücke für folgende Sprachen:

a) Die Menge aller Zeichenreihen aus Nullen und Einsen, die die Teilzeichenreihe 101 nicht enthalten

b) Die Menge aller Zeichenreihen mit der gleichen Anzahl von Nullen und Einsen, die so geformt sind, dass jedes Präfix weder zwei Nullen mehr als Einsen noch zwei Einsen mehr als Nullen enthält

c) Die Menge aller Zeichenreihen aus Nullen und Einsen, deren Anzahl von Nullen durch 5 teilbar und deren Anzahl von Einsen gerade ist

! Übung 3.1.4 Beschreiben Sie die Sprachen der folgenden regulären Ausdrücke in Worten:

* a) $(\mathbf{1} + \varepsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$

b) $(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0} + \mathbf{1})^*$

c) $(\mathbf{0} + \mathbf{10})^*\mathbf{1}^*$

***! Übung 3.1.5** Wir haben in Beispiel 3.1 darauf hingewiesen, dass $\emptyset$ eine der beiden Sprachen ist, deren Hülle endlich ist. Wie lautet die andere Sprache?

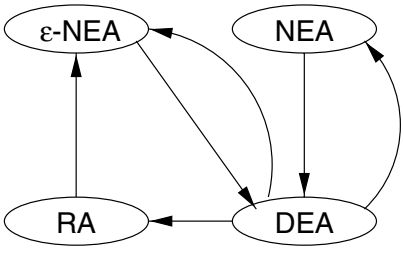
3.2 Endliche Automaten und reguläre Ausdrücke

Obwohl der in regulären Ausdrücken verwendete Ansatz zur Beschreibung von Sprachen sich grundlegend von dem endlicher Automaten unterscheidet, repräsentieren diese beiden Notationen die gleiche Menge von Sprachen, die wir als »reguläre Sprachen« bezeichnet haben. Wir haben bereits gezeigt, dass deterministische endliche Automaten und die beiden Typen nichtdeterministischer Automaten – mit und ohne ε-Übergänge(n) – die gleiche Sprachklasse akzeptieren. Um zu zeigen, dass reguläre Ausdrücke dieselbe Klasse definieren, müssen wir zeigen, dass

1. jede Sprache, die durch einen dieser Automaten definiert wird, auch durch einen regulären Ausdruck definiert wird. Für diesen Beweis können wir annehmen, dass die Sprache von einem DEA akzeptiert wird.
2. jede Sprache, die durch einen regulären Ausdruck definiert wird, auch durch einen dieser Automaten definiert wird. Für diesen Teil des Beweises ist es am einfachsten, wenn wir zeigen, dass es einen NEA mit ε-Übergängen gibt, der dieselbe Sprache akzeptiert.

Abbildung 3.1 zeigt die Äquivalenzen, die wir bewiesen haben oder beweisen werden. Ein Pfeil von Klasse X zu Klasse Y bedeutet, dass wir bewiesen haben, dass jede von Klasse X definierte Sprache auch von Klasse Y definiert wird. Da der Graph stark zusammenhängend ist (d. h. wir können von jedem der vier Knoten zu jedem anderen Knoten gelangen), sehen wir, dass alle vier Klassen tatsächlich gleich sind.

Abbildung 3.1: Plan des Beweises der Äquivalenz von vier verschiedenen Notationen für reguläre Sprachen



3.2.1 Von DEAs zu regulären Ausdrücken

Die Bildung eines regulären Ausdrucks, der die Sprache eines beliebigen DEA definiert, ist überraschenderweise nicht so einfach. In groben Zügen erklärt, müssen wir Ausdrücke formulieren, die Mengen von Zeichenreihen beschreiben, die als Beschriftungen bestimmter Pfade im Übergangsdiagramm des DEA auftreten. Allerdings dürfen diese Pfade nur eine begrenzte Teilmenge von Zuständen passieren. In einer induktiven Definition dieser Ausdrücke beginnen wir mit den einfachsten, die Pfade beschreiben, die überhaupt *keine* Zustände passieren dürfen (d. h. es sind einzelne Knoten oder einzelne Pfeile), und bilden dann die Ausdrücke, die die Pfade durch zunehmend größere Mengen von Zuständen führen. Schließlich lassen wir zu, dass die Pfade durch eine beliebige Anzahl von Zuständen führen; d. h. die Ausdrücke, die wir am Schluss erzeugen, repräsentieren alle möglichen Pfade. Der Gedankengang wird im Beweis des folgenden Satzes deutlich.

Satz 3.4 Wenn für einen beliebigen DEA A $L = L(A)$ gilt, dann gibt es einen regulären Ausdruck R, derart dass $L = L(R)$.

Beweis: Angenommen, A verfügt für eine ganze Zahl n über die Zustände $\{1, 2, ..., n\}$. Gleichgültig, wie die Zustände von A geartet sind, für eine endliche Zahl n gibt es immer n Zustände, und indem wir die Zustände umbenennen, können wir darauf so Bezug nehmen, als handele es sich um die ersten n positiven ganzen Zahlen. Unsere erste und schwierigste Aufgabe besteht darin, eine Menge von regulären Ausdrücken zu konstruieren, die die zunehmend umfangreicher werdende Menge von Pfaden im Übergangsdiagramm von A beschreiben.

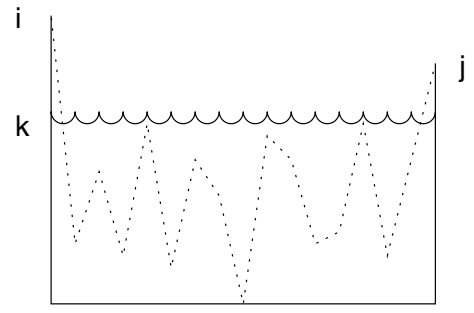
Wir wollen $R_{ij}^{(k)}$ als Namen für den regulären Ausdruck verwenden, dessen Sprache die Menge der Zeichenreihen w umfasst, sodass w die Beschriftung eines Pfades von Zustand i zum Zustand j in A ist und der Pfad keine dazwischenliegenden Knoten berührt, deren Nummer größer als k ist. Beachten Sie, dass Anfangs- und Endpunkt des Pfades nicht »dazwischenliegend« sind; es gibt also keine Beschränkung, die besagt, dass i und/oder j kleiner gleich k sein müssen.

Abbildung 3.2 veranschaulicht die Anforderungen, die für die durch $R_{ij}^{(k)}$ beschriebenen Pfade gelten. In dieser Abbildung repräsentiert die vertikale Richtung die Zustände, die von unten nach oben von 1 bis n an der vertikalen Achse angetragen sind. Die horizontale Achse repräsentiert die Bewegung entlang des Pfades. Beachten Sie, dass i und j auch kleiner oder gleich k sein könnten, obwohl wir in diesem Diagramm sowohl i als auch j größer als k dargestellt haben. Beachten Sie zudem, dass der Pfad zweimal durch Knoten k verläuft, jedoch, mit Ausnahme der Endpunkte, nie einen Knoten berührt, dessen Nummer größer als k ist.

Zur Bildung der Ausdrücke $R_{ij}^{(k)}$ verwenden wir die folgende induktive Definition, wobei wir bei $k = 0$ beginnen und schließlich $k = n$ erreichen. Beachten Sie, dass im Fall $k = n$ die dargestellten Pfade keinerlei Beschränkungen unterliegen, da es keine Zustände größer n gibt.

Induktionsbeginn: $k = 0$. Da alle Zustände beginnend mit 1 aufsteigend nummeriert sind, gilt als Beschränkung für Pfade, dass der Pfad keine dazwischenliegenden Knoten enthalten darf. Es gibt nur zwei Arten von Pfaden, die eine solche Bedingung erfüllen:

Abbildung 3.2: Ein Pfad, dessen Beschriftung in der Sprache des regulären Ausdrucks $R_{ij}^{(k)}$ enthalten ist



1. Ein Pfeil von Knoten (Zustand) i zu Knoten j
2. Ein Pfad mit der Länge 0, der nur aus dem Knoten i besteht

Wenn $i \neq j$, dann ist nur der Fall (1) möglich. Wir müssen den DEA A überprüfen und jene Eingabesymbole a finden, nach deren Eingabe ein Übergang vom Zustand i in den Zustand j erfolgt.

a) Existiert kein solches Symbol a, dann gilt $R_{ij}^{(0)} = \emptyset$.

b) Gibt es genau ein solches Symbol, dann gilt $R_{ij}^{(0)} = \mathbf{a}$.

c) Wenn es Symbole $a_1, a_2, \ldots, a_k$ gibt, die als Beschriftung der Pfeile auf dem Pfad vom Zustand i zum Zustand j dienen, dann gilt $R_{ij}^{(0)} = \mathbf{a}_1 + \mathbf{a}_2 + \ldots + \mathbf{a}_k$.

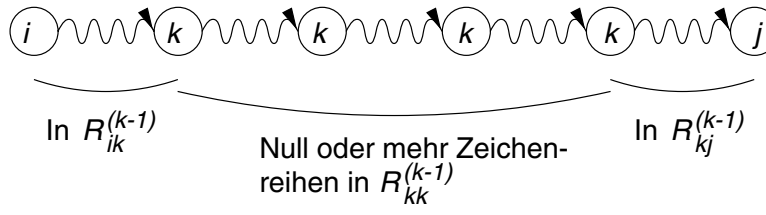
Wenn jedoch $i = j$ ist, dann sind nur der Pfad der Länge 0 und alle Schleifen im Knoten i zulässig. Der Pfad der Länge 0 wird durch den regulären Ausdruck ε repräsentiert, da auf diesem Pfad keine Symbole liegen. Folglich fügen wir ε zu den oben unter a) und c) angegebenen Ausdrücken hinzu. Das heißt, im Fall (a) [keine Symbole a] lautet der Ausdruck ε , im Fall (b) [ein Symbol a] erhalten wir nun den Ausdruck $\varepsilon + \mathbf{a}$ und im Fall (c) [mehrere Symbole] lautet der Ausdruck jetzt $\varepsilon + \mathbf{a}_1 + \mathbf{a}_2 + \ldots + \mathbf{a}_k$.

INDUKTIONSSCHRITT: $k > 0$. Angenommen, es gibt einen Pfad vom Zustand i zum Zustand j, der durch keinen Zustand mit einer höheren Nummer als k führt. Hier sind zwei mögliche Fälle zu betrachten:

1. Zustand k liegt überhaupt nicht auf dem Pfad. In diesem Fall ist die Beschriftung dieses Pfades in der Sprache von $R_{ij}^{(k-1)}$ enthalten.
2. Der Pfad führt mindestens einmal durch Zustand k. In diesem Fall können wir den Pfad in mehrere Teilstücke aufspalten, wie in Abbildung 3.3 dargestellt. Das erste Segment führt vom Zustand i zum Zustand k, ohne k zu passieren, und das letzte Segment führt vom Zustand k direkt zum Zustand j, ohne k zu passieren. Alle anderen Segmente in der Mitte führen von k nach k, ohne k zu passieren. Beachten Sie, dass es keine »mittleren« Segmente gibt, wenn der Pfad den Zustand k nur einmal durchläuft, sondern lediglich einen Pfad von i nach k und einen Pfad von k nach j. Die Menge der Beschriftungen für alle Pfade dieses Typs werden durch den regulären Ausdruck $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$

beschrieben. Das heißt, der erste Ausdruck repräsentiert den Teil des Pfades, der zum ersten Mal zum Zustand *k* führt, der zweite repräsentiert den Teil, der nullmal, einmal oder mehrmals von *k* nach *k* führt, und der dritte Ausdruck repräsentiert den Teil des Pfades, der *k* zum letzten Mal verlässt und zum Zustand *j* führt.

Abbildung 3.3: Ein Pfad von *i* nach *j* kann an jedem Punkt, an dem er durch Zustand *k* führt, in Segmente aufgespaltet werden



Wenn wir die Ausdrücke für die Pfade der oben beschriebenen beiden Typen kombinieren, erhalten wir den Ausdruck

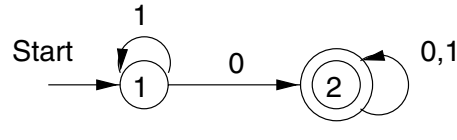
$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)}) * R_{kj}^{(k-1)}$$

für die Beschriftungen aller Pfade vom Zustand *i* zum Zustand *j*, die durch keinen Zustand mit einer höheren Nummer als *k* führen. Wenn wir diese Ausdrücke in der Reihenfolge zunehmender oberer Indizes *k* bilden, dann sind alle Ausdrücke an der richtigen Stelle verfügbar, denn jeder Ausdruck $R_{ij}^{(k)}$ hängt nur von Ausdrücken mit kleineren oberen Indizes ab.

Schließlich erhalten wir $R_{ij}^{(n)}$ für alle *i* und *j*. Wir können annehmen, dass Zustand 1 der Startzustand ist, auch wenn die akzeptierenden Zustände aus jeder beliebigen Menge von Zuständen bestehen könnten. Der reguläre Ausdruck für die Sprache des Automaten ergibt sich dann aus der Summe (Vereinigung) aller Ausdrücke $R_{1j}^{(n)}$, wobei *j* ein akzeptierender Zustand ist. ■

Beispiel 3.5 Wir wollen den in Abbildung 3.4 gezeigten DEA in einen regulären Ausdruck umwandeln. Dieser DEA akzeptiert alle Zeichenreihen, die mindestens eine Null enthalten. Dies geht daraus hervor, dass der Automat vom Startzustand 1 in den akzeptierenden Zustand 2 übergeht, sobald er die Eingabe 0 erhält. Der Automat bleibt danach für alle weiteren Eingabesequenzen im Zustand 2.

Abbildung 3.4: Ein DEA, der alle Zeichenreihen akzeptiert, die mindestens eine Null enthalten



Nachfolgend sind die Anfangsausdrücke in der Konstruktion von Satz 3.4 aufgeführt:

$R_{11}^{(0)}$	$\varepsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\varnothing$
$R_{22}^{(0)}$	$(\varepsilon + \mathbf{0} + \mathbf{1})$

$R_{11}^{(0)}$ enthält den Term ε, weil Anfangs- und Endzustand gleich sind, nämlich Zustand 1. Der Ausdruck enthält den Term 1, weil für die Eingabe 1 ein Pfeil (Schleife) vom Zustand 1 zum Zustand 1 führt. $R_{12}^{(0)}$ ist $\mathbf{0}$, weil ein Pfeil mit der Beschriftung 0 vom Zustand 1 zum Zustand 2 führt. Hier ist der Term ε nicht gegeben, weil Anfangs- und Endzustand verschieden sind. Als drittes Beispiel sei $R_{21}^{(0)} = \varnothing$ angeführt; dieser Ausdruck entspricht der leeren Menge, weil kein Pfeil vom Zustand 2 zum Zustand 1 führt.

Wir müssen jetzt den Induktionsschritt durchführen und komplexere Ausdrücke erstellen, die zuerst Pfaden Rechnung tragen, die durch Zustand 1 führen, und anschließend Pfaden, die durch die Zustände 1 und 2 führen. Damit erhalten wir alle Pfade. Die Ausdrücke $R_{ij}^{(1)}$ werden nach der Regel berechnet, die im Induktionsschritt von Satz 3.4 angegeben wurde:

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

In Tabelle 3.1 sind zuerst die Ausdrücke aufgeführt, die durch direkte Substitution von Werten in die obige Formel berechnet wurden, und in der nachfolgenden Spalte die vereinfachten Ausdrücke, die nachweislich dieselbe Sprache repräsentieren wie die komplizierteren Ausdrücke.

Tabelle 3.1: Reguläre Ausdrücke für Pfade, die nur Zustand 1 passieren dürfen

	Direkte Substitution	**Vereinfacht**
$R_{11}^{(1)}$	$\varepsilon + \mathbf{1} + (\varepsilon + \mathbf{1})(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$	$\mathbf{1}^*$
$R_{12}^{(1)}$	$\mathbf{0} + (\varepsilon + \mathbf{1})(\varepsilon + \mathbf{1})^* \mathbf{0}$	$\mathbf{1}^*\mathbf{0}$
$R_{21}^{(1)}$	$\varnothing + \varnothing(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$	$\varnothing$
$R_{22}^{(1)}$	$\varepsilon + \mathbf{0} + \mathbf{1} + \varnothing(\varepsilon + \mathbf{1})^* \mathbf{0}$	$(\varepsilon + \mathbf{0} + \mathbf{1})$

Betrachten Sie beispielsweise $R_{12}^{(1)}$. Der entsprechende Ausdruck lautet $R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}$, den wir nach (3.1) durch Substitution von $i = 1$ und $j = 2$ erhalten.

Zum Verständnis der Vereinfachung ist die allgemeine Grundregel zu beachten, die besagt: Wenn R ein beliebiger regulärer Ausdruck ist, dann gilt $(\varepsilon + R)^* = R^*$. Gerechtfertigt wird dies dadurch, dass beide Seiten der Gleichung die Sprache beschreiben, die aus beliebigen Verkettungen von null oder mehr Zeichenreihen aus

$L(R)$ bestehen. In unserem Fall liegt $(\varepsilon + \mathbf{1})^* = \mathbf{1}^*$ vor. Beachten Sie, dass beide Ausdrücke eine beliebige Anzahl von Einsen angeben. Zudem gilt: $(\varepsilon + \mathbf{1})\mathbf{1}^* = \mathbf{1}^*$. Wieder ist feststellbar, dass beide Ausdrücke »eine beliebige Anzahl von Einsen« angeben. Folglich ist der ursprüngliche Ausdruck $R_{12}^{(1)}$ äquivalent mit $\mathbf{0} + \mathbf{1}^*\mathbf{0}$. Dieser Ausdruck beschreibt die Sprache, die die Zeichenreihe 0 und alle Zeichenreihen enthält, die aus einer Null mit einer beliebigen Anzahl voranstehender Einsen bestehen. Diese Sprache wird auch durch den einfacheren Ausdruck **1*0** repräsentiert.

Die Vereinfachung von $R_{11}^{(1)}$ ähnelt der Vereinfachung von $R_{12}^{(1)}$, die wir gerade betrachtet haben. Die Vereinfachung von $R_{21}^{(1)}$ und $R_{22}^{(1)}$ hängt von zwei Regeln zur Bedeutung von $\varnothing$ ab. Für jeden regulären Ausdruck R gilt:

1. $\varnothing R = R\varnothing = \varnothing$. Das heißt, $\varnothing$ ist ein Nulloperator der Verkettung. Wenn die leere Menge mit einem beliebigen Ausdruck, der links oder rechts stehen kann, verkettet wird, dann ist das Ergebnis die leere Menge. Diese Regel leuchtet ein, weil eine Zeichenreihe nur dann Teil des Ergebnisses einer Verkettung ist, wenn sie Zeichenreihen beider Argumente der Verkettung enthält. Wenn eines der Argumente gleich $\varnothing$ ist, trägt es keine Zeichenreihe zur Verkettung bei und folglich gibt es kein Verkettungsergebnis, das aus Zeichenreihen beider Argumente besteht.

2. $\varnothing + R = R + \varnothing = R$. Das heißt, $\varnothing$ ist der Identitätsoperator der Vereinigung. Wenn die leere Menge mit einem Ausdruck vereinigt wird, ist das Ergebnis immer dieser Ausdruck.

Infolgedessen kann ein Ausdruck wie $\varnothing\,(\varepsilon + \mathbf{1})^*(\varepsilon + \mathbf{1})$ durch $\varnothing$ ersetzt werden. Damit sollten die letzten beiden Vereinfachungen klar sein.

Wir wollen nun die Ausdrücke $R_{ij}^{(2)}$ berechnen. Durch Anwendung der Induktionsregel mit $k = 2$ erhalten wir:

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}(R_{22}^{(1)})^* R_{2j}^{(1)} \tag{3.2}$$

Wenn wir die vereinfachten Ausdrücke aus Tabelle 3.1 in (3.2) einsetzen, erhalten wir die in Tabelle 3.2 aufgeführten Ausdrücke. Diese Tabelle enthält zudem Vereinfachungen, die nach den oben für Tabelle 3.1 beschriebenen Regeln gewonnen wurden.

Tabelle 3.2: Reguläre Ausdrücke für Pfade, die durch beliebige Zustände führen können

	Direkte Substitution	**Vereinfacht**
$R_{11}^{(2)}$	$\mathbf{1}^* + \mathbf{1}^*\,\mathbf{0}(\varepsilon + \mathbf{0} + \mathbf{1})^*\varnothing$	$\mathbf{1}^*$
$R_{12}^{(2)}$	$\mathbf{1}^*\mathbf{0} + \mathbf{1}^*\mathbf{0}(\varepsilon + \mathbf{0} + \mathbf{1})^*\ \ (\varepsilon + \mathbf{0} + \mathbf{1})$	$\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$
$R_{21}^{(2)}$	$\varnothing + (\varepsilon + \mathbf{0} + \mathbf{1})\,(\varepsilon + \mathbf{0} + \mathbf{1})^*\,\varnothing$	$\varnothing$
$R_{22}^{(2)}$	$\varepsilon + \mathbf{0} + \mathbf{1} + (\varepsilon + \mathbf{0} + \mathbf{1})(\varepsilon + \mathbf{0} + \mathbf{1})^*\,(\varepsilon + \mathbf{0} + \mathbf{1})$	$(\mathbf{0} + \mathbf{1})^*$

Der letzte mit dem Automaten aus Abbildung 3.4 äquivalente reguläre Ausdruck wird konstruiert, indem die Vereinigung aller Ausdrücke gebildet wird, die als ersten Zustand den Startzustand und als zweiten Zustand einen akzeptierenden Zustand angeben. In diesem Beispiel, in dem 1 der Startzustand und 2 der einzige akzeptie-

rende Zustand ist, brauchen wir lediglich den Ausdruck $R_{12}^{\{2\}}$. Dieser Ausdruck ergibt $\mathbf{1^*0(0 + 1)^*}$. Die Interpretation dieses Ausdrucks ist einfach. Er beschreibt die Sprache, die aus allen Zeichenreihen besteht, die mit null oder mehr Einsen beginnen und denen eine Null und anschließend eine beliebige Zeichenreihe aus Nullen und Einsen folgen. Anders ausgedrückt, die Sprache umfasst alle Zeichenreihen aus Nullen und Einsen, die mindestens eine Null enthalten. ■

3.2.2 DEA durch die Eliminierung von Zuständen in reguläre Ausdrücke umwandeln

Die in Abschnitt 3.2.1 beschriebene Methode zur Umwandlung eines DEA in einen regulären Ausdruck funktioniert immer. Wie Sie vielleicht bemerkt haben, hängt sie nicht davon ab, dass der Automat deterministisch ist; sie könnte genauso gut auf einen NEA oder sogar einen ε-NEA angewandt werden. Die Bildung des regulären Ausdrucks ist jedoch aufwändig. Wir müssen für einen Automaten mit n Zuständen nicht nur etwa n^3 Ausdrücke konstruieren, sondern die Länge des Ausdrucks kann in jedem der n Induktionsschritte durchschnittlich um den Faktor 4 wachsen, falls sich die Ausdrücke nicht vereinfachen lassen. Folglich können die Ausdrücke selbst eine Größenordnung von 4^n Symbolen erreichen.

Mit einem ähnlichen Ansatz lässt es sich vermeiden, an gewissen Punkten dieselben Arbeitsschritte wiederholt ausführen zu müssen. Beispielsweise wird in der Konstruktion von Theorem 3.4 in allen Ausdrücken mit dem oberen Index $(k + 1)$ derselbe Teilausdruck $R_{kk}^{(k)}$ verwendet, folglich muss dieser Ausdruck n^2-mal niedergeschrieben werden.

Der Ansatz zur Bildung regulärer Ausdrücke, den wir nun kennen lernen werden, beinhaltet das Eliminieren von Zuständen. Wenn wir einen Zustand s eliminieren, dann werden damit auch sämtliche Pfade, die durch s verliefen, aus dem Automaten entfernt. Soll sich die Sprache des Automaten nicht ändern, dann müssen wir die Beschriftungen von Pfaden, die von einem Zustand q über s zu einem Zustand p führten, Pfeilen hinzufügen, die direkt von q nach p führen. Da die Beschriftung dieses Pfeils nun unter Umständen Zeichenreihen statt einzelner Symbole sind und da es sogar eine unendliche Anzahl solcher Zeichenreihen geben kann, können wir diese Zeichenreihen nicht einfach als Beschriftung auflisten. Glücklicherweise gibt es eine einfache, endliche Möglichkeit, alle derartigen Zeichenreihen darzustellen: reguläre Ausdrücke.

Wir werden daher Automaten betrachten, die als Beschriftungen reguläre Ausdrücke verwenden. Die Sprache des Automaten ist die Vereinigung aller Pfade, die vom Startzustand zu einem akzeptierenden Zustand der Sprache führen, die gebildet wird, indem die Sprachen der auf dem Pfad liegenden regulären Ausdrücke miteinander verkettet werden. Beachten Sie, dass diese Regel konsistent ist mit der Definition der Sprache eines jeden der verschiedenen Automaten, die wir bislang betrachtet haben. Man kann sich jedes Symbol a oder ε, falls zulässig, auch als regulären Ausdruck vorstellen, dessen Sprache aus einer einzigen Zeichenreihe, nämlich $\{a\}$ oder $\{\varepsilon\}$, besteht. Wir können diese Beobachtung als Grundlage eines Verfahrens zum Eliminieren von Zuständen betrachten, das wir als Nächstes beschreiben.

Abbildung 3.5 zeigt einen generischen Zustand s, der eliminiert werden soll. Wir nehmen an, dass der Automat, zu dem der Zustand s gehört, über die s vorausgehenden Zustände $q_1, q_2, ..., q_k$ und die s nachfolgenden Zustände $p_1, p_2, ..., p_m$ verfügt. Es ist möglich, dass einige q auch p sind, aber wir unterstellen, dass s nicht zu den Zustän-

den q oder p gehört, auch wenn, wie in Abbildung 3.5 dargestellt, von s ein Pfeil zurück zu s führt. Wir zeigen zudem einen regulären Ausdruck an jedem Pfeil von einem der q-Zustände zu s. Der Ausdruck Q_i dient als Beschriftung des von q_i ausgehenden Pfeils. Analog zeigen wir den regulären Ausdruck P_j, der den Pfeil von s nach p_j für alle j beschriftet. An s befindet sich ein Pfeil mit der Beschriftung S, der zurück zu s führt. Schließlich gibt es für alle i und j den regulären Ausdruck R_{ij} an dem Pfeil, der von q_i nach p_j führt. Beachten Sie, dass einige dieser Pfeile im Automaten möglicherweise nicht vorhanden sind. Ist dies der Fall, dann nehmen wir an, dass der Ausdruck an dem Pfeil gleich ∅ ist.

Abbildung 3.5: Ein Zustand s, der eliminiert werden soll

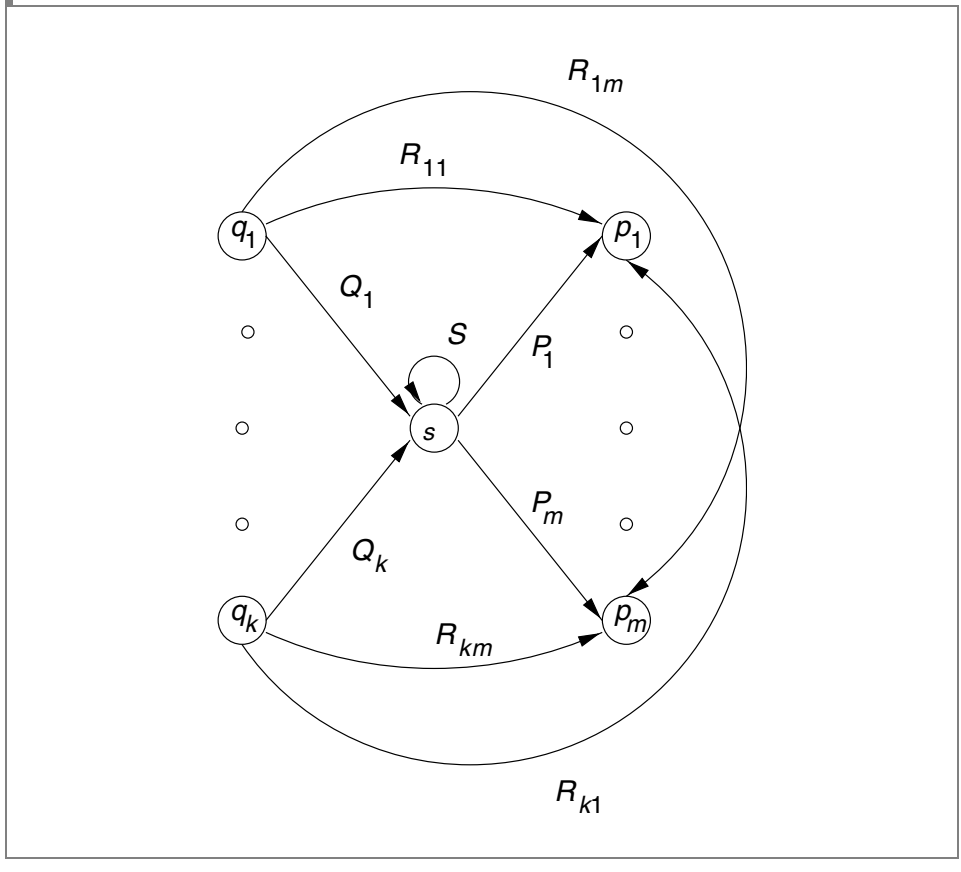
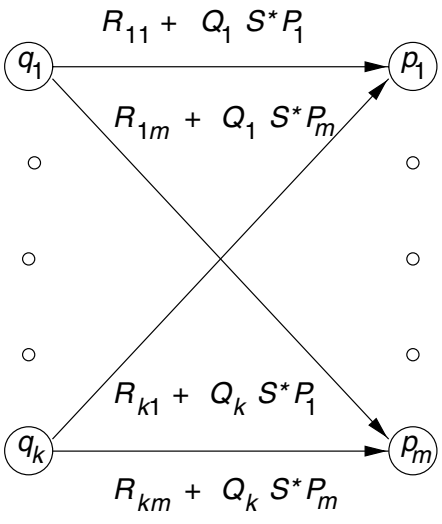


Abbildung 3.6 zeigt, was passiert, wenn wir den Zustand s eliminieren. Alle Pfeile, die vom oder zum Zustand s führen, wurden gelöscht. Zur Kompensierung führen wir für jeden Vorgängerzustand q_i von s sowie für jeden Nachfolgerzustand p_j von s einen regulären Ausdruck ein, der all jene Pfade repräsentiert, die bei q_i beginnen, zu s führen, s null oder mehrere Male in einer Schleife berühren und schließlich zu p_j führen. Der Ausdruck für diese Pfade lautet $Q_i S^* P_j$. Dieser Ausdruck wird (mit dem Mengenvereinigungsoperator) zu dem Pfeil, der von q_i nach p_j führt, hinzugefügt. Falls es keinen Pfeil $q_i \rightarrow p_j$ gibt, dann führen wir mit dem regulären Ausdruck ∅ einen ein.

Abbildung 3.6: Ergebnis der Eliminierung von Zustand *s* aus Abbildung 3.5

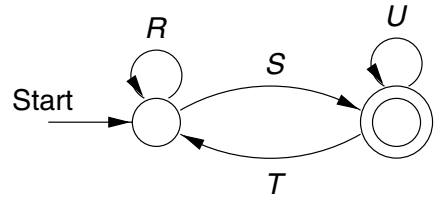


Folgende Strategie wird zur Bildung eines regulären Ausdrucks auf der Grundlage eines endlichen Automaten angewandt:

1. Wende für jeden akzeptierenden Zustand q das oben beschriebene Reduktionsverfahren an, um einen äquivalenten Automaten zu erzeugen, dessen Pfeile mit regulären Ausdrücken beschriftet sind. Eliminiere alle Zustände, mit Ausnahme von q und dem Startzustand q_0.

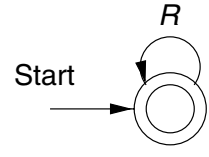
2. Wenn $q \neq q_0$, dann sollten wir einen zwei Zustände umfassenden Automaten erhalten, der wie der Automat in Abbildung 3.7 aussieht. Der reguläre Ausdruck für die akzeptierenden Zeichenreihen kann auf verschiedene Arten beschrieben werden. Eine Möglichkeit lautet $(R + SU^*T)^*SU^*$. Zur Erläuterung: Wir können vom Startzustand beliebig oft zurück zum Startzustand gehen, indem wir einer Folge von Pfaden folgen, deren Beschriftungen entweder in $L(R)$ oder in $L(SU^*T)$ enthalten sind. Der Ausdruck SU^*T repräsentiert Pfade, die über einen in $L(S)$ enthaltenen Pfad zum akzeptierenden Zustand führen, möglicherweise unter Verwendung einer Folge von Pfaden, deren Beschriftungen in $L(U)$ enthalten sind, mehrmals zum akzeptierenden Zustand zurückführen und dann über einen Pfad, dessen Beschriftungen in $L(T)$ enthalten sind, zum Startzustand zurückkehren. Wir müssen dann zum akzeptierenden Zustand gelangen, wobei wir nicht mehr zum Startzustand zurückkehren dürfen, indem wir einem Pfad folgen, dessen Beschriftungen in $L(S)$ enthalten sind. Sobald der akzeptierende Zustand erreicht ist, können wir beliebig oft zu ihm zurückkehren, indem wir einem Pfad folgen, dessen Beschriftungen in $L(U)$ enthalten sind.

Abbildung 3.7: Ein generischer, zwei Zustände umfassender Automat



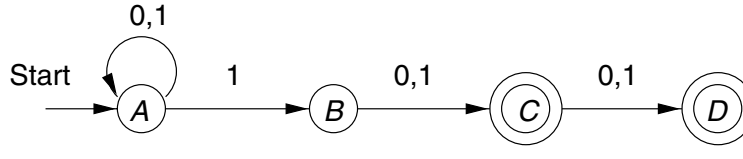
3. Falls der Startzustand zugleich ein akzeptierender Zustand ist, müssen wir zudem eine Zustandseliminierung am ursprünglichen Automaten durchführen, die jeden Zustand außer dem Startzustand beseitigt. Ergebnis dieser Eliminierung ist ein Automat, der über einen Zustand verfügt und wie der in Abbildung 3.8 dargestellte aussieht. Der reguläre Ausdruck, der die Zeichenreihen beschreibt, die dieser Automat akzeptiert, lautet R^*.

Abbildung 3.8: Ein generischer Automat mit einem Zustand



4. Der gewünschte reguläre Ausdruck ist die Summe (Vereinigung) aller Ausdrücke, die aus den reduzierten Automaten nach den Regeln (2) und (3) für jeden akzeptierenden Zustand abgeleitet wurden.

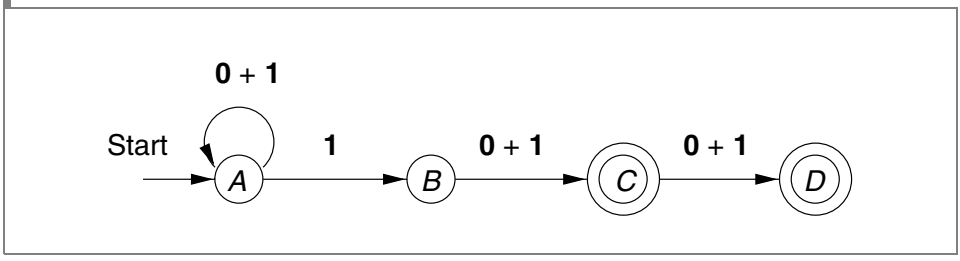
Abbildung 3.9: Ein NEA, der Zeichenreihen akzeptiert, an deren zweiter oder dritter Position vor dem Ende eine 1 steht



Beispiel 3.6 Wir wollen den NEA aus Abbildung 3.9 betrachten, der alle Zeichenreihen aus Nullen und Einsen akzeptiert, in denen entweder an der zweiten oder an der dritten Position vor dem Ende eine Eins steht. Unser erster Schritt besteht darin, diesen Automaten in einen Automaten mit regulären Ausdrücken als Beschriftungen

umzuwandeln. Da keine Zustandseliminierung durchgeführt wurde, müssen wir lediglich die Beschriftungen »0,1« durch die äquivalenten Ausdrücke **0** + **1** ersetzen. Das Ergebnis ist in Abbildung 3.10 dargestellt.

Abbildung 3.10: Der Automat aus Abbildung 3.9 mit regulären Ausdrücken als Beschriftungen



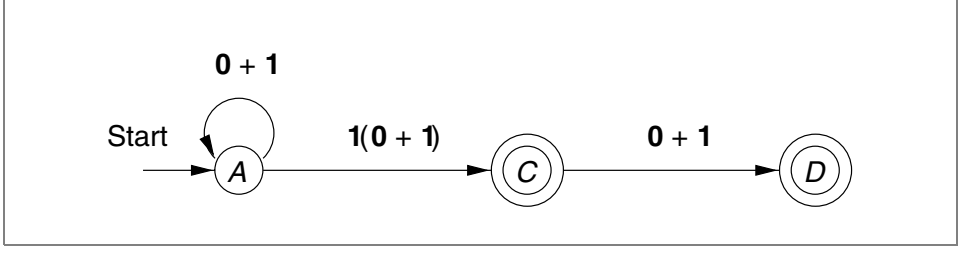
Wir wollen zuerst Zustand *B* eliminieren. Da es sich bei diesem Zustand weder um einen akzeptierenden Zustand noch um den Startzustand handelt, kommt er in den reduzierten Automaten nicht vor. Folglich sparen wir uns etwas Arbeit, wenn wir ihn zuerst eliminieren, bevor wir die beiden reduzierten Automaten entwickeln, die den beiden akzeptierenden Zuständen entsprechen.

Zustand *B* hat einen Vorgänger namens *A* und einen Nachfolger namens *C*. Diese Zustände werden durch die folgenden regulären Ausdrücke (vgl. Abbildung 3.5) beschrieben: $Q_1 = \mathbf{1}$, $P_1 = \mathbf{0} + \mathbf{1}$, $R_{11} = \varnothing$ (da kein Pfeil von *A* nach *C* führt) und $S = \varnothing$ (weil kein Pfeil von Zustand *B* zurück zu *B* führt). Daraus ergibt sich der Ausdruck, der als Beschriftung des neuen Pfeils von *A* nach *C* dient: $\varnothing + \mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$.

Um diesen Ausdruck zu vereinfachen, eliminieren wir zuerst die voranstehende $\varnothing$, da die leere Menge in einer Vereinigung ignoriert werden kann. Der Ausdruck lautet nun $\mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$. Beachten Sie, dass der reguläre Ausdruck $\varnothing^*$ gleichbedeutend ist mit dem regulären Ausdruck ε, da $L(\varnothing^*) = \{\varepsilon\} \cup L(\varnothing) \cup L(\varnothing)\,L(\varnothing) \cup \ldots$

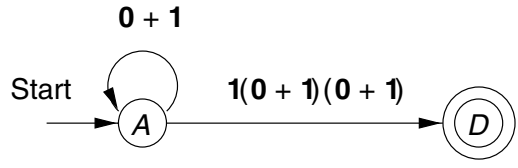
Da alle Terme außer dem ersten leer sind, erkennen wir, dass $L(\varnothing^*) = \{\varepsilon\}$, was dasselbe wie $L(\varepsilon)$ ist. Folglich ist $\mathbf{1}\varnothing^*(\mathbf{0} + \mathbf{1})$ gleichbedeutend mit $\mathbf{1}(\mathbf{0} + \mathbf{1})$, was dem Ausdruck entspricht, der in Abbildung 3.11 die Beschriftung des Pfeils $A \rightarrow C$ bildet.

Abbildung 3.11: Eliminierung von Zustand B



Nun müssen wir verzweigen und die Zustände *C* und *D* in jeweils getrennten Reduktionen eliminieren. Zur Eliminierung von Zustand *C* gehen wir ähnlich vor, wie oben zur Eliminierung von Zustand *B* beschrieben, und der resultierende Automat ist in Abbildung 3.12 dargestellt.

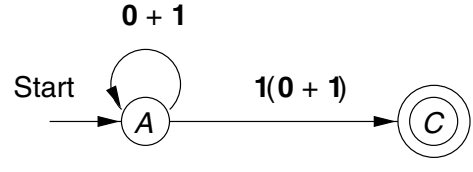
Abbildung 3.12: Ein Automat mit den Zuständen *A* und *D*



In der Notation des generischen Automaten mit zwei Zuständen aus Abbildung 3.7 lauten die regulären Ausdrücke für den Automaten in Abbildung 3.12: $R = \mathbf{0} + \mathbf{1}$, $S = \mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$, $T = \emptyset$ und U = $\emptyset$. Der Ausdruck U^* kann durch ε ersetzt werden, d. h. in einer Verkettung eliminiert werden. Begründen lässt sich dies damit, dass $\emptyset^* = \varepsilon$, wie oben erläutert. Zudem ist der Ausdruck SU^*T gleichbedeutend mit $\emptyset$, da T, einer der Terme in der Verkettung, gleich $\emptyset$ ist. Der allgemeine Ausdruck $(R + SU^*T)^*SU^*$ lässt sich in diesem Fall daher zu R^*S oder $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$ vereinfachen. Informell ausgedrückt, beschreibt dieser Automat eine Sprache, die alle Zeichenreihen aus Nullen und Einsen enthält, an deren drittletzter Stelle eine 1 steht. Diese Sprache stellt einen Teil der Zeichenreihen dar, die der Automat aus Abbildung 3.9 akzeptiert.

Wir müssen nun wieder bei Abbildung 3.11 beginnen, und den Zustand *D* statt *C* eliminieren. Da *D* keine Nachfolger hat, können wir aus Abbildung 3.5 ersehen, dass sich keine Pfadänderungen ergeben, und der Pfeil von *C* nach *D* wird zusammen mit dem Zustand *D* eliminiert. Der resultierende zwei Zustände umfassende Automat ist in Abbildung 3.13 dargestellt.

Abbildung 3.13: Zwei Zustände umfassender Automat, der aus der Elimination von *D* resultiert



Dieser Automat unterscheidet sich von dem in Abbildung 3.12 dargestellten Automaten nur durch die Beschriftung des vom Startzustand zum akzeptierenden Zustand führenden Pfeils. Wir können daher die Regel für zwei Zustände umfassende Automaten anwenden und den Ausdruck vereinfachen, womit wir $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})$ erhalten. Dieser Ausdruck repräsentiert den zweiten Typ von Zeichenreihe, den der Automat akzeptiert: Zeichenreihen mit einer Eins an der zweiten Position vor dem Ende.

Jetzt müssen wir die beiden Ausdrücke nur noch addieren, um den Ausdruck für den gesamten in Abbildung 3.9 dargestellten Automaten zu erhalten. Dieser Ausdruck lautet:

$$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$$

■

Die Reihenfolge der Eliminierung von Zuständen festlegen

Wie wir in Beispiel 3.6 gesehen haben, wird ein Zustand, der weder ein Start- noch ein Akzeptanzzustand ist, aus allen abgeleiteten Automaten eliminiert. Folglich besteht einer der Vorteile des Zustandseliminierungsverfahrens gegenüber der mechanischen Erzeugung von regulären Ausdrücken, die wir in Abschnitt 3.2.1 beschrieben haben, darin, dass wir zuerst alle Zustände, die weder Start- noch akzeptierender Zustand sind, eliminieren und damit ein für alle Mal tilgen können. Wir müssen lediglich dann erneut reduzieren, wenn wir akzeptierende Zustände eliminieren müssen.

Auch dann können wir einige Arbeitsschritte zusammenfassen. Wenn es beispielsweise drei akzeptierende Zustände p, q und r gibt, können wir p eliminieren und dann verzweigen, um entweder q oder r zu eliminieren, und somit den Automaten für den akzeptierenden Zustand r bzw. q erzeugen. Wir können dann erneut mit allen drei akzeptierenden Zuständen beginnen und sowohl q als auch r eliminieren, um den Automaten für p zu erhalten.

3.2.3 Reguläre Ausdrücke in Automaten umwandeln

Wir werden nun den in Abbildung 3.1 skizzierten Plan vervollständigen, indem wir zeigen, dass jede Sprache $L = L(R)$ für einen regulären Ausdruck R auch $L = L(E)$ für einen ε-NEA E ist. Der Beweis besteht aus einer strukturellen Induktion über den Ausdruck R. Wir beginnen, indem wir zeigen, wie der Automat für die Anfangsausdrücke ε und $\emptyset$ konstruiert wird. Wir zeigen anschließend, wie diese Automaten zu größeren Automaten zusammengefasst werden, die die Vereinigung, Verkettung und Hülle der von den kleineren Automaten akzeptierten Sprachen akzeptieren.

Alle dabei konstruierten Automaten sind ε-NEAs mit einem einzigen akzeptierenden Zustand.

Satz 3.7 Jede Sprache, die durch einen regulären Ausdruck definiert wird, wird auch durch einen endlichen Automaten definiert.

Beweis: Angenommen, $L = L(R)$ für einen regulären Ausdruck R. Wir zeigen, dass $L = L(E)$ für einen ε-NEA E mit

1. genau einem akzeptierenden Zustand
2. keinen Pfeilen zum Startzustand
3. keinen Pfeilen, die vom akzeptierenden Zustand ausgehen

Der Beweis wird durch strukturelle Induktion über R erbracht, wobei wir der induktiven Definition von regulären Ausdrücken, die in Abschnitt 3.1.2. vorgestellt wurde, folgen.

Induktionsbeginn: Er gliedert sich in drei Teile, die in Abbildung 3.14 dargestellt sind. In Teil (a) wird gezeigt, wie der Ausdruck ε zu behandeln ist. Es lässt sich einfach zeigen, dass die Sprache des Automaten $\{\varepsilon\}$ ist, da der einzige Pfad vom Startzustand zu einem akzeptierenden Zustand mit ε beschriftet ist. Teil (b) zeigt die Konstruktion für $\emptyset$. Offensichtlich gibt es keine Pfade vom Startzustand zum akzeptierenden Zustand, und daher ist $\emptyset$ die Sprache dieses Automaten. Schließlich zeigt Teil (c) den Automaten für einen regulären Ausdruck **a**. Die Sprache dieses Automaten besteht aus der einen Zeichenreihe a oder, anders ausgedrückt, $L(\mathbf{a})$. Es ist leicht nachprüfbar, dass diese Automaten die Bedingungen (1), (2) und (3) der Induktionshypothese erfüllen.

Abbildung 3.14: Beginn der Konstruktion eines Automaten aus einem regulären Ausdruck

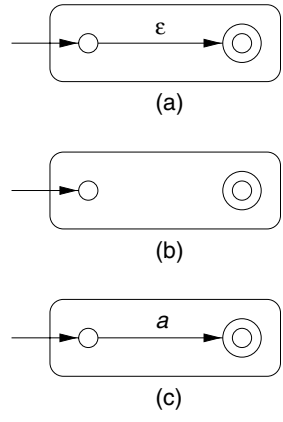
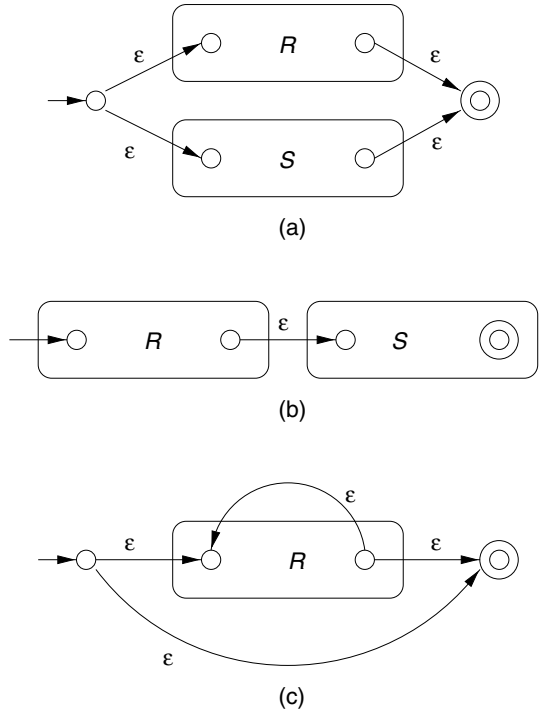


Abbildung 3.15: Der Induktionsschritt in der Konstruktion eines ε-NEA aus einem regulären Ausdruck



INDUKTIONSSCHRITT: Diese drei Teile der Induktion sind in Abbildung 3.15 dargestellt. Wir nehmen an, dass die Aussage des Satzes für die unmittelbaren Teilausdrücke eines gegebenen regulären Ausdrucks wahr ist; das heißt, dass die Sprachen dieser Teilausdrücke auch Sprachen von ε-NEAs mit einem einzigen akzeptierenden Zustand sind. Es sind vier Fälle zu unterscheiden:

1. Der Ausdruck lautet $R + S$ für beliebige kleinere Ausdrücke R und S. Auf diesen Fall trifft der Automat in Abbildung 3.15 (a) zu. Das heißt, ausgehend von einem neuen Startzustand können wir sowohl zum Startzustand des Automaten für R als auch zu dem des Automaten für S gehen. Wir erreichen den akzeptierenden Zustand des jeweiligen Automaten, indem wir einem Pfad folgen, der mit einer Zeichenreihe aus $L(R)$ bzw. $L(S)$ beschriftet ist. Sobald wir den akzeptierenden Zustand von R bzw. S erreicht haben, können wir einem der ε-Pfeile zum akzeptierenden Zustand des neuen Automaten folgen. Folglich lautet die Sprache des Automaten aus Abbildung 3.15 (a) $L(R) \cup L(S)$.

2. Der Ausdruck lautet RS für beliebige kleinere Ausdrücke R und S. Der Automat für die Verkettung ist in Abbildung 3.15 (b) dargestellt. Beachten Sie, dass der Startzustand des ersten Automaten zum Startzustand der Gesamtheit wird und dass der akzeptierende Zustand des zweiten Automaten zum akzeptierenden Zustand der Gesamtheit wird. Begründen lässt sich dies damit, dass die einzigen Pfade, die vom Start- zum akzeptierenden Zustand führen, zuerst den Automaten für R passieren, wo sie einem Pfad folgen müssen, der mit einer in $L(R)$ enthaltenen Zeichenreihe beschriftet ist, und dann den Automaten für S durchqueren, wo sie einem Pfad folgen müssen, der mit einer in $L(S)$ enthaltenen Zeichenreihe beschriftet ist. Daher hat der in Abbildung 3.15 (b) dargestellte Automat genau die Pfade, die mit Zeichenreihen aus $L(R)L(\mathrm{S})$ beschriftet sind.

3. Der Ausdruck lautet R^* für einen kleineren Ausdruck R. In diesem Fall verwenden wir den Automaten aus Abbildung 3.15 (c). Dieser Automat hat zwei mögliche Pfade zur Auswahl:

   a) Direkt vom Startzustand zum akzeptierenden Zustand entlang eines mit ε beschrifteten Pfades. Dieser Pfad ermöglicht es, ε zu akzeptieren, das für jeden beliebigen Ausdruck R in $L(R^*)$ enthalten ist.

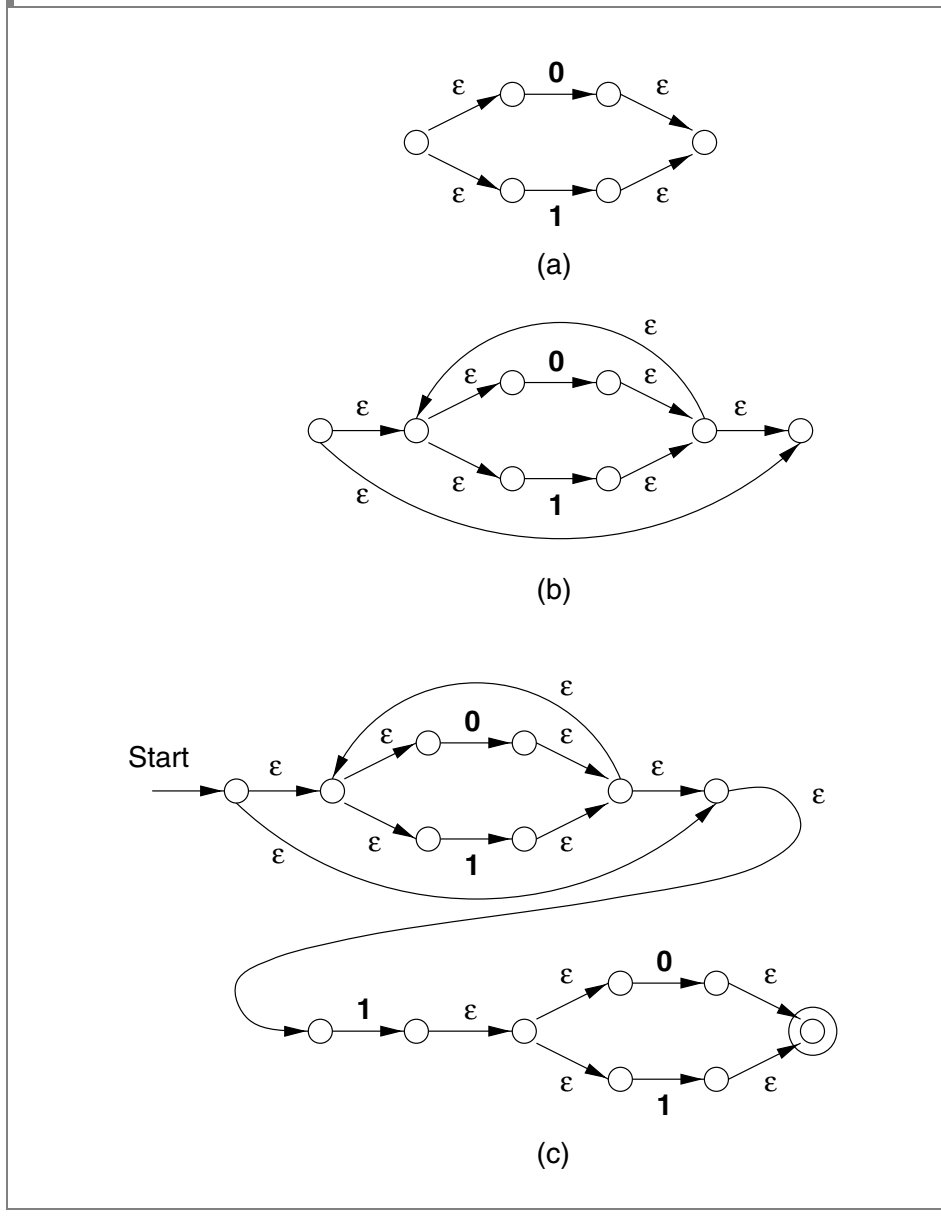
   b) Zum Startzustand des Automaten für R, einmal oder mehrmals durch diesen Automaten und dann zum akzeptierenden Zustand. Diese Menge von Pfaden erlaubt es, Zeichenreihen zu akzeptieren, die in $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$ etc. enthalten sind, womit alle Zeichenreihen in $L(R^*)$ abgedeckt sind, womöglich mit Ausnahme von ε, das durch den unter (3a) beschriebenen direkten Pfad zum akzeptierenden Zustand abgedeckt wird.

4. Der Ausdruck lautet (R) für einen kleineren Ausdruck R. Der Automat für R dient auch als Automat für (R), da Klammern die durch den Automaten definierte Sprache nicht ändern.

Es ist eine einfache Beobachtung, dass der konstruierte Automat die drei in der Induktionsannahme gegebenen Bedingungen erfüllt – ein akzeptierender Zustand, keine auf den Startzustand hinweisenden oder von dem akzeptierenden Zustand ausgehenden Pfeile. ■

Beispiel 3.8 Wir wollen nun den regulären Ausdruck $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})$ in einen ε-NEA umwandeln. Der erste Schritt besteht in der Konstruktion eines Automaten für $\mathbf{0} + \mathbf{1}$. Wir verwenden zwei Automaten, die gemäß Abbildung 3.14 (c) aufgebaut wurden, einen mit der Beschriftung **0** auf dem Pfeil und der andere mit der Beschriftung **1**. Diese beiden Automaten werden dann mit Hilfe der Vereinigungskonstruktion aus Abbildung 3.15 (a) kombiniert. Das Ergebnis ist in Abbildung 3.16 (a) zu sehen.

Abbildung 3.16: Für Beispiel 3.8 erzeugter Automat



Als Nächstes wenden wir die Sternkonstruktion aus Abbildung 3.15 (c) auf Abbildung 3.16 (a) an. Dieser Automat ist in Abbildung 3.16 (b) dargestellt. Die letzten beiden Schritte beinhalten die Anwendung der Verkettungskonstruktion aus Abbildung 3.15 (b). Zuerst verbinden wir den Automaten aus Abbildung 3.16 (b) mit einem anderen Automaten, der nur die Zeichenreihe 1 akzeptieren soll. Dieser Automat stellt eine weitere Anwendung der Grundkonstruktion aus Abbildung 3.14 (c) dar, wobei der Pfeil die Beschriftung **1** trägt. Beachten Sie, dass wir einen neuen Automaten erstellen müssen, damit 1 erkannt wird; der Automat aus Abbildung 3.16 (a), der **1** akzeptierte, darf hier nicht verwendet werden. Bei dem dritten Automaten in der Verkettung handelt es sich um einen anderen Automaten für $\mathbf{0} + \mathbf{1}$. Wieder müssen wir eine Kopie des Automaten aus Abbildung 3.16 (a) erstellen; hier darf nicht die Kopie verwendet werden, die Bestandteil von Abbildung 3.16 (b) ist. Der vollständige Automat ist in Abbildung 3.16 (c) dargestellt.

Beachten Sie, dass dieser ε-NEA, wenn die ε-Übergänge eliminiert werden, genauso aussieht wie der Automat aus Abbildung 3.13, der auch die Zeichenreihen akzeptiert, an deren zweitletzter Position eine 1 steht. ■

3.2.4 Übungen zum Abschnitt 3.2

Übung 3.2.1 Die Übergangstabelle eines DEA sehe wie folgt aus:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

* a) Formulieren Sie alle regulären Ausdrücke $R_{ij}^{(0)}$. *Hinweis*: Stellen Sie sich den Zustand q_i so vor, als handle es sich um den Zustand mit der ganzzahligen Nummer i.

* b) Formulieren Sie alle regulären Ausdrücke $R_{ij}^{(1)}$. Versuchen Sie, die Ausdrücke so weit wie möglich zu vereinfachen.

c) Formulieren Sie alle regulären Ausdrücke $R_{ij}^{(2)}$. Versuchen Sie, die Ausdrücke so weit wie möglich zu vereinfachen.

d) Formulieren Sie einen regulären Ausdruck für die Sprache des Automaten.

* e) Konstruieren Sie das Übergangsdiagramm für den DEA, und geben Sie einen regulären Ausdruck für dessen Sprache an, indem Sie Zustand q_2 eliminieren.

Übung 3.2.2 Wiederholen Sie Übung 3.2.1 für den folgenden DEA:

	0	1
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

Beachten Sie, dass Lösungen zu den Teilen (a), (b) und (e) für diese Übung (auf den Internetseiten dieses Buches) *nicht* verfügbar sind.

Übung 3.2.3 Wandeln Sie folgenden DEA in einen regulären Ausdruck um, und verwenden Sie hierbei die in Abschnitt 3.2.2 beschriebene Technik zur Zustandselimination:

	0	1
$\rightarrow * p$	s	p
q	p	s
r	r	q
s	q	r

Übung 3.2.4 Wandeln Sie die folgenden regulären Ausdrücke in NEAs mit ε-Übergängen um:

* a) $\mathbf{01}^*$.

b) $(\mathbf{0} + \mathbf{1})\mathbf{01}$.

c) $\mathbf{00}(\mathbf{0} + \mathbf{1})^*$.

Übung 3.2.5 Eliminieren Sie die ε-Übergänge aus Ihrem ε-NEA aus Übung 3.2.4. Eine Lösung zu Teil (a) erscheint auf den Webseiten zu diesem Buch.

! **Übung 3.2.6** Sei $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ ein ε-NEA, derart dass es keine zu q_0 hinführenden und keine von q_f ausgehenden Übergänge gibt. Beschreiben Sie für jede der folgenden Modifikationen von A die akzeptierte Sprache als Modifikation von $L = L(A)$:

* a) Der Automat, der aus A konstruiert wird, indem ein ε-Übergang von q_f nach q_0 hinzugefügt wird

* b) Der Automat, der aus A konstruiert wird, indem ε-Übergänge von q_0 zu jedem Zustand hinzugefügt werden, der von q_0 aus erreichbar ist (auf einem Pfad, dessen Beschriftungen Symbole aus Σ sowie ε enthalten können)

c) Der Automat, der aus A konstruiert wird, indem von jedem Zustand ε-Übergänge nach q_f hinzugefügt werden, von dem aus q_f auf irgendeinem Pfad erreichbar ist

d) Der Automat, der aus A konstruiert wird, indem die unter (b) und (c) geforderten Modifikationen ausgeführt werden

***!! Übung 3.2.7** Die Konstruktionen von Satz 3.7, mit denen wir einen regulären Ausdruck in einen ε-NEA umgewandelt haben, lassen sich auf verschiedene Weise vereinfachen. Folgende drei Vereinfachungen sind beispielsweise möglich:

1. Vereinigungsoperator: Statt einen neuen Start- und einen neuen akzeptierenden Zustand zu bilden, werden die beiden Startzustände in einen Zustand zusammengeführt, der über alle Übergänge aus beiden Startzuständen verfügt. Analog werden die beiden akzeptierenden Zustände zusammengeführt, wobei alle Übergänge in die beiden akzeptierenden Zustände dann zum zusammengeführten Zustand führen.
2. Verkettungsoperator: Der akzeptierende Zustand des ersten Automaten wird mit dem Startzustand des zweiten zusammengeführt.
3. Sternoperator: Es werden einfach ε-Übergänge vom akzeptierenden Zustand zum Startzustand und in der umgekehrten Richtung hinzugefügt.

Jede dieser Vereinfachungen ergibt für sich genommen eine korrekte Konstruktion: d. h. der für einen beliebigen Ausdruck resultierende ε-NEA akzeptiert die durch den Ausdruck definierte Sprache. Welche der Vereinfachungen (1), (2), (3) lassen sich kombinieren, sodass man aus diesen Kombinationen wiederum einen korrekten Automaten für jeden regulären Ausdruck erhält?

***!! Übung 3.2.8** Geben Sie einen Algorithmus an, der einen DEA A als Eingabe erhält und die Anzahl der Zeichenreihen mit der Länge n (für eine gegebene Zahl n, die unabhängig von der Anzahl der Zustände von A ist) berechnet, die von A akzeptiert werden. Ihr Algorithmus sollte sowohl in Bezug auf n als auch die Anzahl der Zustände von A polynomial sein. *Hinweis:* Setzen Sie die Technik ein, die von der Konstruktion im Beweis von Satz 3.4 nahe gelegt wird.

3.3 Anwendungen regulärer Ausdrücke

Reguläre Ausdrücke, die ein »Bild« des Musters darstellen, das erkannt werden soll, sind das geeignete Mittel für Anwendungen, die nach Mustern im Text suchen. Die regulären Ausdrücke werden dann im Hintergrund in deterministische oder nichtdeterministische Automaten kompiliert, die simuliert werden, um ein Programm zu erzeugen, das Muster im Text erkennt. In diesem Abschnitt betrachten wir zwei wichtige Klassen auf regulären Ausdrücken basierender Anwendungen: lexikalische Analysekomponenten und Textsuche.

3.3.1 Reguläre Ausdrücke in Unix

Bevor wir die Anwendungen betrachten, stellen wir die Unix-Notation für erweiterte reguläre Ausdrücke vor. Diese Notation stellt uns eine Reihe zusätzlicher Hilfsmittel zur Verfügung. Tatsächlich enthalten die Unix-Erweiterungen bestimmte Leistungs-

merkmale, z. B. die Fähigkeit, Zeichenreihen zu benennen und wieder aufzurufen, die mit einem Muster übereingestimmt haben, sodass sogar die Erkennung gewisser nichtregulärer Sprachen möglich ist. Wir werden diese Leistungsmerkmale hier nicht behandeln. Stattdessen stellen wir lediglich die Kürzel vor, die eine prägnante Darstellung komplexer regulärer Ausdrücke erlauben.

Die erste Erweiterung der Notation regulärer Ausdrücke betrifft die Tatsache, dass die meisten praktischen Anwendungen mit dem ASCII-Zeichensatz arbeiten. In unseren Beispielen wurden in der Regel kleine Alphabete verwendet, z. B. {0, 1}. Da es nur zwei Symbole gab, konnten wir knappe Ausdrücke wie $\mathbf{0} + \mathbf{1}$ für »beliebiges Zeichen« angeben. Wenn es allerdings 128 Zeichen gibt, dann würde derselbe Ausdruck eine Liste all dieser Zeichen beinhalten und seine Angabe wäre daher äußerst mühsam. Reguläre Ausdrücke im Unix-Stil erlauben es uns, mithilfe von *Zeichenklassen* umfangreiche Mengen von Zeichen so prägnant wie möglich darzustellen. Folgende Regeln gelten für Zeichenklassen:

- Das Symbol . (Punkt) steht für »ein beliebiges Zeichen«.
- Die Folge $[a_1 a_2 \ldots a_k]$ steht für den regulären Ausdruck

  $$a_1 + a_2 + \ldots + a_k$$

  Mit dieser Notation können wir uns die Hälfte der Zeichen sparen, da wir die Pluszeichen nicht angeben müssen. Beispielsweise können wir die vier Zeichen, die in den C-Vergleichsoperatoren vorkommen, darstellen durch `[<>=!]`.
- In eckigen Klammern können wir einen Bereich von Zeichen in der Form *x-y* angeben, um alle Zeichen von *x* bis *y* in der ASCII-Reihenfolge auszuwählen. Da die Ziffern ebenso wie die Groß- und Kleinbuchstaben einen ordnungserhaltenden numerischen Code besitzen, können wir viele Zeichenklassen, die für uns von Belang sind, mit wenigen Tastatureingaben darstellen. Beispielsweise können die Ziffern angegeben werden durch `[0-9]`, die Großbuchstaben durch `[A-Z]` und die Menge aller Buchstaben und Ziffern durch `[A-Za-z0-9]`. Wenn wir ein Minuszeichen in die Liste der Zeichen aufnehmen wollen, dann können wir es als erstes oder letztes Zeichen angeben, sodass es nicht mit seiner Bedeutung als Operator zur Definition eines Zeichenbereichs verwechselt wird. Die Menge der Ziffern plus Komma, Plus- und Minuszeichen, die zur Bildung von Dezimalzahlen dient, kann beispielsweise beschrieben werden durch `[-+.0-9]`. Eckige Klammern und andere Zeichen, die eine besondere Bedeutung in der Unix-Notation für reguläre Ausdrücke haben, können als Zeichen dargestellt werden, indem man ihnen einen umgekehrten Schrägstrich (\) voranstellt.
- Für einige der gebräuchlichsten Zeichenklassen sind spezielle Schreibweisen definiert. Zum Beispiel:

  a) `[:digit:]` steht für die Menge der zehn Ziffern und ist gleichbedeutend mit `[0-9]`.[3]

3. Die Notation `[:digit:]` hat den Vorteil, dass sie auch bei Verwendung eines anderen Zeichencodes als ASCII, in dem die Ziffern keine aufeinander folgenden numerischen Codes besitzen, die Zeichen `[0123456789]` repräsentiert, während der Ausdruck `[0-9]` in diesem Fall die Zeichen repräsentieren würde, die einen Zeichencode zwischen 0 und 9 haben.

b) `[:alpha:]` steht ebenso wie `[A-Za-z]` für die Menge der alphabetischen Zeichen.

c) `[:alnum:]` steht für die Menge aller Ziffern und Buchstaben (alphabetische und numerische Zeichen) genau wie `[A-Za-z0-9]`.

Zudem werden in der Unix-Notation für reguläre Ausdrücke einige Operatoren verwendet, die bislang noch nicht vorgekommen sind. Keiner dieser Operatoren stellt eine Erweiterung der Klasse der Sprachen dar, die sich durch reguläre Ausdrücke beschreiben lassen, aber sie erleichtern es gelegentlich, die gewünschte Sprache darzustellen:

1. Der Operator | wird an Stelle des Pluszeichens (+) zur Angabe einer Vereinigung verwendet.
2. Der Operator `?` bedeutet »null oder ein Vorkommen von«. Daher hat der Ausdruck *R?* in Unix dieselbe Bedeutung wie der Ausdruck $\varepsilon + R$ in der in diesem Buch verwendeten Notation für reguläre Ausdrücke.
3. Der Operator + bedeutet »ein oder mehr Vorkommen von«. Daher ist der Unix-Ausdruck *R*+ ein Kürzel für den Ausdruck RR^* in unserer Notation.
4. Der Operator {*n*} bedeutet »*n* Exemplare von«. Daher ist der Unix-Ausdruck *R*{5} ein Kürzel für den Ausdruck *RRRRR* in unserer Notation.

Beachten Sie, dass auch in der Unix-Notation für reguläre Ausdrücke Teilausdrücke durch runde Klammern gruppiert werden können, wie die in Abschnitt 3.1.2 beschriebenen regulären Ausdrücke, und dass dieselbe Operatorvorrangfolge gilt (wobei ?, + und {*n*}, was die Auswertungsreihenfolge betrifft, wie * behandelt werden). Der Sternoperator hat in Unix dieselbe Bedeutung wie die in diesem Buch angegebene (in Unix aber nicht als oberer Index geschrieben).

3.3.2 Lexikalische Analyse

Eine der ältesten Anwendungen regulärer Ausdrücke bestand in der Spezifikation einer Compilerkomponente, die als »lexikalische Analyse« bezeichnet wird. Diese Komponente liest das Quelltextprogramm ein und erkennt alle *Token*, d. h. jene Teilzeichenreihen aus aufeinander folgenden Zeichen, die eine logische Einheit bilden. Schlüsselwörter und Bezeichner sind übliche Beispiele für Token, stellen aber nur eine kleine Auswahl dar.

> **Umfassende Informationen zur Unix-Notation für reguläre Ausdrücke**
>
> Leser, die an einer kompletten Liste der in der Unix-Notation für reguläre Ausdrücke verfügbaren Operatoren und Kürzel interessiert sind, sollten die Manualseiten zu den Kommandos konsultieren. Die verschiedenen Unix-Versionen unterscheiden sich leicht, aber mit einem Kommando wie `man grep` werden Sie die Notation für das grundlegende Kommando `grep` angezeigt bekommen. »Grep« steht für »Global (search for) Regular Expression and Print«, übersetzt: »Globale (Suche nach) regulärem Ausdruck und Ausgabe«.

Der Unix-Befehl `lex` und dessen GNU-Version `flex` akzeptieren eine Liste regulärer Ausdrücke im Unix-Stil, denen jeweils ein in eckige Klammern gesetzter Codeabschnitt folgt, der angibt, was die Analysekomponente tun soll, wenn sie ein Vorkommen des betreffenden Token findet. Eine derartige Programmfunktion wird *lexical-analyzer generator* (übersetzt: Generator einer lexikalischen Analysekomponente) genannt, weil sie eine Beschreibung einer lexikalischen Analysekomponente als Eingabe erhält und daraus eine Funktion generiert, die wie eine lexikalische Analysekomponente arbeitet.

Kommandos wie `lex` und `flex` haben sich als äußerst nützlich erwiesen, weil die Notation der regulären Ausdrücke genau so mächtig ist, wie es zur Beschreibung von Token erforderlich ist. Diese Kommandos können mithilfe eines Verfahrens zur Umwandlung von regulären Ausdrücken in DEAs eine effiziente Funktion generieren, die Quelltextprogramme in Token aufschlüsselt. Mit ihnen wird die Implementierung einer lexikalischen Analyse zu einer in wenigen Stunden zu bewältigenden Aufgabe, während die manuelle Entwicklung der lexikalischen Analyse vor der Einführung dieser auf regulären Ausdrücken basierenden Hilfsmittel Monate dauern konnte. Wenn die lexikalische Analyse aus irgendeinem Grund modifiziert werden muss, lässt sich dies häufig durch die Änderung einiger weniger regulärer Ausdrücke erledigen, statt eine bestimmte Stelle in schwer verständlichem Code zu suchen und zu korrigieren.

Beispiel 3.9 Listing 3.17 zeigt einen Ausschnitt einer Eingabe für den Befehl `lex`, der einige Token aus der Programmiersprache C beschreibt. Die erste Zeile behandelt das Schlüsselwort `else`, und als Aktion wird festgelegt, dass eine symbolische Konstante (in diesem Beispiel `ELSE`) zur weiteren Verarbeitung an den Parser zurückgegeben wird. Die zweite Zeile enthält einen regulären Ausdruck, der Bezeichner beschreibt: ein Buchstabe, dem null oder mehr Buchstaben und/oder Ziffern folgen. Die Aktion besteht zunächst darin, den Bezeichner in die Symboltabelle einzutragen, sofern er dort noch nicht vorhanden ist, d. h. `lex` fügt den gefundenen Token in einen Puffer ein, und daher weiß das Programm genau, welcher Bezeichner gefunden wurde. Die lexikalische Analyse gibt schließlich die symbolische Konstante `ID` zurück, die in diesem Beispiel Bezeichner repräsentiert.

Listing 3.17: Beispiel für eine `lex`-Eingabe

```
else                  {return(ELSE);}
[A-Za-z][A-Za-z0-9]*  {Code, der den gefundenen Bezeichner
                       in die Symboltabelle einträgt;
                       return(ID);
                      }
>=                    [return(GE);}
=                     {return(EQ);}
...
```

Der dritte Eintrag in Listing 3.17 betrifft das Zeichen >=, einen zwei Zeichen umfassenden Operator. Als letztes Beispiel zeigen wir den Eintrag für das Zeichen =, einen aus einem Zeichen bestehenden Operator. In der Praxis wären Einträge für alle Schlüsselwörter, alle Operatorzeichen und Satzzeichen, wie Kommas und Klammern, und Familien von Konstanten wie Zahlen und Zeichenreihen vorhanden. Viele dieser Einträge sind sehr einfach; sie stellen nur eine Folge von einem oder mehreren speziellen Zeichen dar. Andere ähneln eher Bezeichnern, deren Beschreibung die gesamte Leistungsstärke der Notation regulärer Ausdrücke erfordert. Ganze Zahlen, Gleitkommazahlen, Zeichenreihen und Kommentare sind weitere Beispiele für Mengen von Zeichenreihen, die von der Verarbeitung regulärer Ausdrücke durch Befehle wie `lex` profitieren. ■

Die Umwandlung einer Sammlung von Ausdrücken, wie die in Listing 3.17 dargestellten, in einen Automaten erfolgt ungefähr so, wie wir es formal in den vorangegangenen Abschnitten beschrieben haben. Wir beginnen, indem wir einen Automaten für die Vereinigung aller Ausdrücke entwickeln. Dieser Automat kann im Grunde genommen nur aussagen, dass irgendein Token erkannt worden ist. Wenn wir allerdings die Konstruktion von Satz 3.7 für die Vereinigung der Ausdrücke nachvollziehen, dann gibt der Zustand des ε-NEA genau darüber Aufschluss, welcher Token erkannt wurde.

Problematisch ist hier nur, dass mehrere Token zugleich erkannt werden können. Beispielsweise entspricht die Zeichenreihe `else` nicht nur dem Schlüsselwort `else`, sondern auch dem Ausdruck für Bezeichner. Die Standardlösung besteht darin, dass der zuerst angegebene Ausdruck die Priorität erhält. Wenn Schlüsselwörter wie `else` also reserviert (nicht als Bezeichner verwendbar) sein sollen, dann geben wir sie einfach vor dem Ausdruck für Bezeichner an.

3.3.3 Textmuster finden

In Abschnitt 2.4.1 haben wir erstmals die Vorstellung beschrieben, dass Automaten eingesetzt werden könnten, um effizient nach einer Menge von Wörtern zu suchen, die in einer umfangreichen Textsammlung wie dem Internet enthalten sind. Obwohl die Werkzeuge und die Technologie hierfür noch nicht so weit entwickelt sind wie für die lexikalische Analyse, ist die Notation regulärer Ausdrücke eine wertvolle Hilfe zur Beschreibung der Suche nach interessanten Textmustern. Wie bei der lexikalischen Analyse kann man von der Fähigkeit, von der natürlichen, beschreibenden Notation regulärer Ausdrücke zu einer effizienten (auf Automaten basierenden) Implementierung zu gelangen, erheblich profitieren.

Ein allgemeines Problem, zu dessen Lösung sich die auf regulären Ausdrücken basierenden Techniken als nützlich erwiesen, besteht in der Beschreibung vage definierter Klassen von Textmustern. Die Vagheit der Beschreibung garantiert praktisch, dass wir das Muster anfangs nicht korrekt beschreiben – möglicherweise finden wir die ganz korrekte Beschreibung niemals. Durch den Einsatz der Notation regulärer Ausdrücke wird es einfach, das Muster mühelos abstrakt zu beschreiben und die Beschreibung rasch zu ändern, wenn etwas nicht stimmt. Ein »Compiler« für reguläre Ausdrücke ist hilfreich, um die formulierten Ausdrücke in ausführbaren Code umzuwandeln.

Wir wollen ein umfangreicheres Beispiel für diese Art von Problemen studieren, das sich in vielen Internetanwendungen stellt. Angenommen, wir möchten eine umfangreiche Menge von Webseiten nach Adressen durchsuchen. Wir möchten viel-

leicht einfach eine Mailingliste erstellen. Oder wir versuchen vielleicht, Unternehmen nach ihrem Standort zu klassifizieren, damit wir Abfragen wie »Finde das nächste Restaurant, das in einem Radius von zehn Autominuten von meinem aktuellen Aufenthaltsort liegt« beantworten können.
Wir werden uns im Folgenden auf das Erkennen amerikanischer Straßenangaben konzentrieren. Wie sieht eine solche Straßenangabe aus? Das müssen wir herausfinden, und wenn wir während des Testens der Software erkennen, dass wir einen Fall vergessen haben, dann müssen wir die Ausdrücke abändern, um dem Fehlenden Rechnung zu tragen. Wir beginnen mit der Feststellung, dass amerikanische Straßenangaben wahrscheinlich mit dem Wort »Street« oder dessen Abkürzung »St« enden. Es kommen allerdings auch die Bezeichnungen »Avenue« und »Road« sowie die entsprechenden Abkürzungen vor. Folglich können wir die Endung unseres regulären Ausdrucks etwa wie folgt formulieren:

```
Street|St\.|Avenue|Ave\.|Road|Rd\.
```

Wir haben im obigen Ausdruck die Unix-Notation verwendet und als Operator für die Mengenvereinigung daher den vertikalen Strich statt + angegeben. Beachten Sie zudem, dass den Punkten ein umgekehrter Schrägstrich vorangestellt ist, da der Punkt in Unix-Ausdrücken ein Sonderzeichen ist, das für »ein beliebiges Zeichen« steht. In diesem Fall wollen wir nur, dass der Punkt die Abkürzungen abschließt.

Einer Straßenbezeichnung wie `Street` muss der Straßenname voranstehen. In der Regel handelt es sich beim Namen um einen Großbuchstaben, dem einige Kleinbuchstaben folgen. Wir können dieses Muster mit dem Unix-Ausdruck **[A-Z][a-z]*** beschreiben. Manche Straßennamen bestehen allerdings aus mehreren Wörtern, z. B. Rhode Island Avenue in Washington DC. Nachdem wir erkannt haben, dass Angaben dieser Form nicht erkannt würden, ändern wir unsere Beschreibung von Straßennamen wie folgt ab:

```
'[A-Z][a-z]*( [A-Z][a-z]*)*'
```

Der obige Ausdruck beginnt mit einer Gruppe, die aus einem Großbuchstaben und null oder mehr Kleinbuchstaben besteht. Anschließend folgen null oder mehr Gruppen, die aus einem Leerzeichen, einem weiteren Großbuchstaben und null oder mehr Kleinbuchstaben bestehen. Das Leerzeichen ist in Unix-Ausdrücken ein gewöhnliches Zeichen. Damit der obige Ausdruck in einer Unix-Kommandozeile aber nicht so aussieht, als handele es sich um zwei durch ein Leerzeichen voneinander getrennte Ausdrücke, müssen wir den gesamten Ausdruck in Anführungszeichen setzen. Die Anführungszeichen sind nicht Teil des eigentlichen Ausdrucks.

Nun müssen wir die Hausnummer in die Adresse aufnehmen. Die meisten Hausnummern bestehen aus einer Folge von Ziffern. Manchen ist jedoch ein Buchstabe nachgestellt, wie in »123A Main St.«. Daher umfasst der Ausdruck, den wir für die Hausnummer formulieren, einen optionalen Großbuchstaben am Ende: `[0-9]+[A-Z]?`. Beachten Sie, dass wir den Unix-Operator + für »eine oder mehr« Ziffern und den Operator ? für »null oder einen« Großbuchstaben verwenden. Der gesamte Ausdruck, den wir zur Beschreibung der Straßenangabe entwickelt haben, lautet:

```
'[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd\.)'
```

Dieser Ausdruck sollte einigermaßen funktionieren. In der Praxis werden wir jedoch erkennen, dass Folgendes fehlt:

1. Straßen, die nicht als Street, Avenue oder Road bezeichnet werden. Beispielsweise fehlen hier die Bezeichnungen »Boulevard«, »Place«, »Way« und deren Abkürzungen
2. Straßennamen, die ganz oder teilweise aus Zahlen bestehen, wie »42nd Street«
3. Postfächer und andere Zustelladressen
4. Straßennamen, die nicht mit einer Bezeichnung wie »Street« enden. Ein Beispiel hierfür ist El Camino Real im Silicon Valley. Da dieser Name spanisch ist und »die königliche Straße« bedeutet, wäre »El Camino Real Street« redundant. Wir müssen also komplette Angaben wie »2000 El Camino Real« berücksichtigen.
5. Alle möglichen sonderbaren Sachen, die wir uns kaum vorstellen können. Sie etwa?

Nachdem wir einen Compiler für reguläre Ausdrücke haben, ist es viel leichter, langsam ein Modul zu entwickeln, das sämtliche Straßenangaben erkennt, als wenn wir jede Änderung direkt in einer konventionellen Programmiersprache vornehmen müssten.

3.3.4 Übungen zum Abschnitt 3.3

! **Übung 3.3.1** Geben Sie einen regulären Ausdruck an, der Telefonnummern in allen möglichen Varianten beschreibt. Berücksichtigen Sie internationale Nummern ebenso wie die Tatsache, dass die Vorwahlen und die lokalen Anschlussnummern in verschiedenen Ländern eine unterschiedliche Anzahl von Stellen besitzen.

!! **Übung 3.3.2** Geben Sie einen regulären Ausdruck an, der Gehaltsangaben repräsentiert, wie sie in Stellenanzeigen erscheinen. Beachten Sie, dass das Gehalt pro Stunde, pro Monat oder pro Jahr angegeben werden kann. Die Gehaltsangabe kann möglicherweise ein Währungssymbol oder eine andere Einheit wie »K« enthalten. In der näheren Umgebung der Gehaltsangabe befinden sich möglicherweise Wörter, die diese als solche identifizieren. Tipp: Sehen Sie sich Stellenanzeigen in der Zeitung oder Online-Joblistings an, um sich eine Vorstellung davon zu machen, welche Muster unter Umständen angebracht sind.

! **Übung 3.3.3** Am Ende von Abschnitt 3.3.3 haben wir einige mögliche Verbesserungen für Ausdrücke angeführt, die Straßenangaben beschreiben. Modifizieren Sie den dort entwickelten Ausdruck, sodass er alle aufgeführten Möglichkeiten abdeckt.

3.4 Algebraische Gesetze für reguläre Ausdrücke

In Beispiel 3.5 wurde verdeutlicht, dass es möglich sein muss, reguläre Ausdrücke zu vereinfachen, damit die Ausdrücke handhabbar bleiben. Wir haben dort einige Ad-hoc-Argumente dafür angeführt, warum ein Ausdruck durch einen anderen ersetzt werden kann. In allen Fällen ging es im Grunde genommen immer darum, dass die Ausdrücke in dem Sinn *äquivalent* waren, als sie dieselbe Sprache definierten. In die-

sem Abschnitt stellen wir eine Gruppe von algebraischen Gesetzen vor, nach denen sich die Frage der Äquivalenz von Ausdrücken auf einer etwas abstrakteren Ebene behandeln lässt. Statt bestimmte reguläre Ausdrücke zu untersuchen, betrachten wir Paare von regulären Ausdrücken mit Variablen als Argumente. Zwei Ausdrücke mit Variablen sind *äquivalent*, wenn wir die Variablen durch beliebige Sprachen ersetzen können und die beiden Ausdrücke auch dann stets dieselbe Sprache beschreiben.

Es folgt ein Beispiel für dieses Verfahren aus der Algebra der Arithmetik. Die Aussage 1 + 2 = 2 + 1 ist eine Sache. Dies ist ein Beispiel für das Kommutativgesetz der Addition, das sich leicht überprüfen lässt, indem der Additionsoperator auf beide Seiten angewandt und das Ergebnis 3 = 3 berechnet wird. Das *Kommutativgesetz der Addition* sagt allerdings mehr aus: Es besagt, $x + y = y + x$, wobei x und y Variablen sind, die durch zwei beliebige Zahlen ersetzt werden können. Das heißt, wir können zwei Zahlen in beliebiger Reihenfolge addieren und erhalten stets das gleiche Ergebnis, gleichgültig welche Zahlen wir verwenden.

Wie für arithmetische Ausdrücke gilt auch für reguläre Ausdrücke eine Reihe von Gesetzen. Viele dieser Gesetze ähneln den Gesetzen der Arithmetik, wenn man die Vereinigung von Mengen der Addition und die Verkettung der Multiplikation gleichsetzt. Es gibt jedoch einige Bereiche, in denen diese Analogie nicht stimmt, und es gibt auch einige Gesetze, die für reguläre Ausdrücke gelten und die keine Entsprechung in der Arithmetik haben; dies gilt insbesondere für den Sternoperator. Die nächsten Unterabschnitte bilden einen Katalog der wichtigsten Gesetze. Wir schließen diesen Abschnitt mit einer Erörterung der Verfahren, mit denen man prüfen kann, ob ein angenommenes Gesetz für reguläre Ausdrücke tatsächlich ein Gesetz ist, d. h. ob es für alle Sprachen gilt, die für die Variablen eingesetzt werden können.

3.4.1 Assoziativität und Kommutativität

Kommutativität ist die Eigenschaft eines Operators, die besagt, dass wir die Reihenfolge der Operanden vertauschen können, ohne dass sich das Ergebnis dadurch ändert. Oben wurde ein Beispiel aus der Arithmetik angeführt: $x + y = y + x$. *Assoziativität* ist die Eigenschaft eines Operators, die es zulässt, dass Operanden unterschiedlich gruppiert werden, wenn der Operator zweimal angewandt wird. Beispielsweise lautet das Assoziativgesetz der Multiplikation $(x \times y) \times z = x \times (y \times z)$. Die folgenden drei Gesetze ähnlicher Art gelten für reguläre Ausdrücke:

- $L + M = M + L$. Dieses Gesetz, das *Kommutativgesetz der Mengenvereinigung*, besagt, dass zwei Sprachen in beliebiger Reihenfolge vereinigt werden können.
- $(L + M) + N = L + (M + N)$. Dieses Gesetz, das *Assoziativgesetz der Mengenvereinigung*, besagt, dass drei Sprachen vereinigt werden können, indem man entweder zuerst die Vereinigung der ersten beiden oder zuerst die Vereinigung der letzten beiden bildet. Beachten Sie, dass wir aus der Kombination dieses Gesetzes mit dem Kommutativgesetz der Mengenvereinigung schließen können, dass jede beliebige Sammlung von Sprachen in beliebiger Reihenfolge und Gruppierung vereinigt werden kann, ohne dass sich das Ergebnis der Vereinigung ändert. Eine Zeichenreihe ist also genau dann in $L_1 \cup L_2 \cup ... \cup L_k$ enthalten, wenn sie in einer oder mehreren der Sprachen L_i enthalten ist.
- $(LM)N = L(MN)$. Dieses Gesetz, das *Assoziativgesetz der Verkettung*, besagt, dass drei Sprachen verkettet werden können, indem entweder die ersten beiden zuerst oder die letzten beiden zuerst verkettet werden.

In dieser Liste fehlt das »Gesetz« $LM = ML$, das besagen würde, dass die Verkettung kommutativ ist. Dieses Gesetz ist allerdings falsch.

Beispiel 3.10 Betrachten Sie die regulären Ausdrücke **01** und **10**. Diese Ausdrücke beschreiben die Sprachen {01} bzw. {10}. Da sich diese Sprachen unterscheiden, kann kein allgemeines Gesetz $LM = LM$ gelten. Wäre es gültig, könnten wir den regulären Ausdruck **0** für L und den Ausdruck **1** für M einsetzen und fälschlicherweise folgern, dass **01** = **10**. ■

3.4.2 Einheiten und Annihilatoren

Die *Einheit* bezüglich eines Operators ist ein Wert, für den gilt, dass bei Anwendung des Operators auf die Einheit und einen anderen Wert der andere Wert das Ergebnis bildet. Beispielsweise ist 0 die Einheit bezüglich der Addition, da $0 + x = x + 0 = x$, und 1 ist die Einheit bezüglich der Multiplikation, da $1 \times x = x \times 1 = x$. Der *Annihilator* bezüglich eines Operators ist ein Wert, für den gilt, dass bei Anwendung des Operators auf den Annihilator und einen anderen Wert der Annihilator das Ergebnis bildet. Beispielsweise ist 0 der Annihilator bezüglich der Multiplikation, da $0 \times x = x \times 0 = 0$. Es gibt keinen Annihilator für die Addition.

Es gibt drei Gesetze für reguläre Ausdrücke, die diese Konzepte betreffen und nachfolgend aufgeführt sind.

- $\emptyset + L = L + \emptyset = L$. Dieses Gesetz stellt fest, dass $\emptyset$ die Einheit bezüglich der Mengenvereinigung ist. Das heißt, $\emptyset$ vermittelt in Verbindung mit der Vereinigung die Identitätsfunktion, weil L vereinigt mit $\emptyset$ identisch L ist. Dieses Gesetz wird deshalb das Identitätsgesetz der Vereinigung genannt.
- $\varepsilon L = L\varepsilon = L$. Dieses Gesetz stellt fest, dass ε die Einheit bezüglich der Verkettung ist. Das heißt, ε vermittelt in Verbindung mit der Verkettung die Identitätsfunktion, weil L verkettet mit ε identisch L ist. Dieses Gesetz wird deshalb das Identitätsgesetz der Verkettung genannt.
- $\emptyset L = L\emptyset = \emptyset$. Dieses Gesetz stellt fest, dass $\emptyset$ der Annihilator der Verkettung ist.

Diese Gesetze sind mächtige Werkzeuge zur Vereinfachung von Ausdrücken. Wenn beispielsweise die Vereinigung verschiedener Ausdrücke vorliegt, von denen einige gleich $\emptyset$ sind oder zu $\emptyset$ vereinfacht wurden, dann können die betreffenden Ausdrücke aus der Vereinigung eliminiert werden. Analog gilt, wenn eine Verkettung verschiedener Ausdrücke vorliegt, von denen einige gleich ε sind oder zu ε vereinfacht wurden, dann können die betreffenden Ausdrücke aus der Verkettung eliminiert werden. Schließlich gilt, wenn eine beliebige Anzahl von Ausdrücken verkettet wird und nur einer dieser Ausdrücke gleich $\emptyset$ ist, dann kann die gesamte Verkettung durch $\emptyset$ ersetzt werden.

3.4.3 Distributivgesetze

Ein Distributivgesetz betrifft zwei Operatoren und behauptet, dass ein Operator auf jedes Argument des anderen Operators einzeln angewandt werden kann. Das gebräuchlichste Beispiel aus der Arithmetik ist das Distributivgesetz der Multiplikation bezüglich der Addition, das heißt: $x \times (y + z) = x \times y + x \times z$. Da die Multiplikation kommutativ ist, ist es gleichgültig, ob die Multiplikation links oder rechts von der

Summe steht. Es gibt jedoch ein analoges Gesetz für reguläre Ausdrücke, das wir in zwei Formen angeben müssen, da die Verkettung nicht kommutativ ist. Diese Gesetze lauten:

- $L(M + N) = LM + LN$. Dieses Gesetz wird als linkes Distributivgesetz der Verkettung bezüglich der Vereinigung bezeichnet.
- $(M + N)L = ML + NL$. Dieses Gesetz wird als rechtes Distributivgesetz der Verkettung bezüglich der Vereinigung bezeichnet.

Wir wollen nun das linke Distributivgesetz beweisen. Das andere Gesetz wird auf ähnliche Weise bewiesen. Der Beweis bezieht sich ganz allgemein auf Sprachen. Er hängt nicht davon ab, ob die Sprachen durch reguläre Ausdrücke beschrieben werden.

Satz 3.11 Wenn L, M und N Sprachen sind, dann gilt:

$$L(M \cup N) = LM \cup LN$$

Beweis: Der Beweis ähnelt einem anderen Beweis zu einem Distributivgesetz, der in Satz 1.10 vorgestellt wurde. Wir müssen zeigen, dass eine Zeichenreihe w genau dann in $L(M \cup N)$ enthalten ist, wenn sie in $LM \cup LN$ enthalten ist.

(Nur-wenn-Teil) Wenn w in $L(M \cup N)$ enthalten ist, dann gilt $w = xy$, wobei x Element von L und y Element von M oder von N ist. Wenn y in M enthalten ist, dann ist xy in LM enthalten und daher auch in $LM \cup LN$.

Ähnlich gilt: Wenn y in N enthalten ist, dann ist xy in LN enthalten und daher auch in $LM \cup LN$.

(Wenn-Teil) Angenommen, w sei in $LM \cup LN$ enthalten. Dann ist w Element von LM oder von LN. Sei zunächst w in LM. Dann ist $w = xy$, wobei x Element von L und y Element von M ist. Da y Element von M ist, ist y auch in $M \cup N$ enthalten. Folglich ist xy in $L(M \cup N)$ enthalten. Wenn w nicht in LM enthalten ist, dann ist w mit Sicherheit in LN enthalten. Ein ähnliches Argument wie zuvor zeigt, dass w in $L(M \cup N)$ enthalten ist. ■

Beispiel 3.12 Betrachten Sie den regulären Ausdruck $\mathbf{0}+\mathbf{01}^*$. Wir können den Ausdruck $\mathbf{0}$ aus der Vereinigung »herauslösen«, aber zuerst müssen wir erkennen, dass der Ausdruck $\mathbf{0}$ als Verkettung von $\mathbf{0}$ mit ε dargestellt werden kann. Das heißt, wir wenden das Identitätsgesetz der Verkettung an, um $\mathbf{0}$ durch $\mathbf{0}\varepsilon$ zu ersetzen, wodurch wir den Ausdruck $\mathbf{0}\varepsilon + \mathbf{01}^*$ erhalten. Nun können wir das linke Distributivgesetz anwenden, um diesen Ausdruck durch $\mathbf{0}(\varepsilon + \mathbf{1}^*)$ zu ersetzen. Wenn wir weiter erkennen, dass ε in $L(\mathbf{1}^*)$ enthalten ist, dann wissen wir, dass $\varepsilon + \mathbf{1}^* = \mathbf{1}^*$, und können den ursprünglichen Ausdruck vereinfachen zu $\mathbf{01}^*$. ■

3.4.4 Das Idempotenzgesetz

Ein Operator wird als *idempotent* bezeichnet, wenn seine Anwendung auf zwei Operanden mit dem gleichen Wert eben diesen Wert als Ergebnis hat. Die üblichen arithmetischen Operatoren sind nicht idempotent. Im Allgemeinen gilt $x + x \neq x$ und $x \times x \neq x$, obwohl es einige Werte für x gibt, bei denen die Gleichheit gilt (z. B. $0 + 0 = 0$). Vereinigung und Durchschnitt sind jedoch gebräuchliche Beispiele für idempotente Operatoren. Für reguläre Ausdrücke können wir daher folgendes Gesetz formulieren:

- $L + L = L$. Dieses Gesetz, das Idempotenzgesetz für die Vereinigung, besagt, dass die Vereinigung von zwei identischen Ausdrücken eben diesen Ausdruck ergibt, d. h. wir können die Vereinigung der beiden Ausdrücke durch ein Exemplar dieses Ausdrucks ersetzen.

3.4.5 Gesetze bezüglich der Hüllenbildung

Es gibt eine Reihe von Gesetzen, die den Sternoperator und seine Unix-Varianten + und ? betreffen. Wir werden diese Gesetze im Folgenden aufführen und erklären, warum sie gültig sind.

- $(L^*)^* = L^*$. Dieses Gesetz besagt, dass durch die Anwendung des Sternoperators auf einen Ausdruck, der bereits abgeschlossen ist, die Sprache nicht verändert wird. Die Sprache von $(L^*)^*$ besteht aus allen Zeichenreihen, die durch Verkettung der in der Sprache von L^* enthalten Zeichenreihen gebildet werden. Diese Zeichenreihen setzen sich aber wiederum aus Zeichenreihen der Sprache L zusammen. Folglich ist jede in $(L^*)^*$ enthaltene Zeichenreihe auch eine Verkettung von Zeichenreihen aus L und somit in der Sprache von L^* enthalten.
- $\emptyset^* = \varepsilon$. Die Hülle von $\emptyset$ enthält nur die Zeichenreihe ε, wie wir in Beispiel 3.6 erörtert haben.
- $\varepsilon^* = \varepsilon$. Es lässt sich leicht überprüfen, dass die einzige Zeichenreihe, die durch die Verkettung einer beliebigen Anzahl von leeren Zeichenreihen gebildet werden kann, die leere Zeichenreihe selbst ist.
- $L^+ = LL^* = L^*L$. Sie wissen, dass L^+ definiert ist als $L + LL + LLL + \ldots$. Zudem gilt $L^* = \varepsilon + L + LL + LLL + \ldots$. Daraus folgt:

  $$LL^* = L\varepsilon + LL + LLL + LLLL \ldots$$

  Wenn wir uns in Erinnerung rufen, dass $L\varepsilon = L$, dann erkennen wir, dass die unendlichen Ausdrücke für LL^* und für L^+ identisch sind. Der Beweis, dass $L^+ = L^*L$ gilt, ist ähnlich zu führen.[4]
- $L^* = L^+ + \varepsilon$. Dies ist leicht zu beweisen, da L^+ jeden Term aus L^* enthält, mit Ausnahme von ε. Beachten Sie, dass der Zusatz + ε nicht erforderlich ist, wenn die Sprache L die Zeichenreihe ε bereits enthält; für diesen Sonderfall gilt: $L^+ = L^*$.
- $L? = \varepsilon + L$. Diese Regel ist nicht anders als die Definition des ?-Operators.

3.4.6 Gesetze für reguläre Ausdrücke entdecken

Jedes der obigen Gesetze wurde formal oder informell bewiesen. Man könnte nun eine unendliche Vielzahl für reguläre Ausdrücke geltender Gesetze vermuten. Gibt es eine allgemeine Methodologie, die Beweise gültiger Gesetze erleichtert? Es zeigt sich, dass die Gültigkeit eines Gesetzes sich auf die Frage der Gleichheit zweier Sprachen reduzieren lässt. Interessanterweise ist diese Technik eng mit den Operatoren für

4. Beachten Sie, dass also jede Sprache L und ihre Hülle L^* kommutativ sind (bezüglich der Verkettung): $LL^* = L^*L$. Diese Regel widerspricht der Tatsache nicht, dass die Verkettung im Allgemeinen nicht kommutativ ist.

reguläre Ausdrücke verknüpft und kann nicht auf Ausdrücke, die andere Operatoren (wie z. B. den Durchschnitt) enthalten, erweitert werden.

Um dieses Verfahren zu demonstrieren, wollen wir ein vermutetes Gesetz betrachten:

$$(L + M)^* = (L^*M^*)^*$$

Dieses Gesetz besagt Folgendes: Wenn L und M Sprachen sind und der Sternoperator auf deren Vereinigung angewandt wird, dann erhalten wir dieselbe Sprache wie durch die Anwendung des Sternoperators auf die Sprache L^*M^*, das heißt der Anwendung des Sternoperators auf alle Zeichenreihen, die aus null oder mehr Elementen von L gefolgt von null oder mehr Elementen von M bestehen.

Zum Beweis dieses Gesetzes nehmen wir zuerst an, dass die Zeichenreihe w in der Sprache von $(L + M)^*$ enthalten sei.[5] Wir können dann sagen, w = $w_1w_2 \ldots w_k$ für ein k, wobei jede Zeichenreihe w_i in L oder M enthalten ist. Daraus folgt, dass jede Zeichenreihe w_i in der Sprache von L^*M^* enthalten ist. Dies lässt sich wie folgt begründen: Wenn w_i in L enthalten ist, dann wähle die Zeichenreihe w_i aus L; diese Zeichenreihe ist auch in L^* enthalten. Wähle keine Zeichenreihen aus M; das heißt, wähle ε aus M^*. Wenn w_i in M enthalten ist, dann ist die Argumentation ähnlich. Nachdem bewiesen wurde, dass jedes w_i in L^*M^* enthalten ist, folgt daraus, dass w in der Hülle dieser Sprache enthalten ist.

Um den Beweis zu vervollständigen, müssen wir zudem die Umkehrung beweisen, dass Zeichenreihen, die in $(L^*M^*)^*$ enthalten sind, auch in $(L + M)^*$ enthalten sind. Wir übergehen diesen Teil des Beweises, da unser Ziel nicht im Beweis dieses Gesetzes besteht, sondern in der Hervorhebung der folgenden wichtigen Eigenschaft regulärer Ausdrücke:

Jeder reguläre Ausdruck mit Variablen kann als *konkreter* regulärer Ausdruck (ein Ausdruck ohne Variablen) betrachtet werden, wenn man sich jede Variable als ein bestimmtes Symbol vorstellt. Beispielsweise können im Ausdruck $(L + M)^*$ die Variablen L und M durch die Symbole a bzw. b ersetzt werden, womit wir den regulären Ausdruck $(\mathbf{a} + \mathbf{b})^*$ erhalten.

Die Sprache des konkreten Ausdrucks gibt uns Aufschluss über die Form der Zeichenreihen der Sprache, die aus dem ursprünglichen Ausdruck gebildet wird, wenn wir die Variablen durch Sprachen ersetzen. In unserer Analyse von $(L + M)^*$ haben wir so beobachtet, dass jede Zeichenreihe w, die aus einer Folge von Elementen aus L oder M besteht, in der Sprache $(L + M)^*$ enthalten ist. Wir kommen zu diesem Schluss, wenn wir die Sprache des konkreten Ausdrucks $(\mathbf{a} + \mathbf{b})^*$ betrachten, bei der es sich offensichtlich um die Menge aller aus den Zeichen a und b bestehenden Zeichenreihen handelt. Wir könnten jede in L enthaltene Zeichenreihe für ein Vorkommen von a einsetzen, und wir könnten jede in M enthaltene Zeichenreihe für ein Vorkommen von b einsetzen, wobei durchaus verschiedene Zeichenreihen für die verschiedenen Vorkommen von a und b verwendet werden können. Wenn wir diese Substitutionen an allen in $(\mathbf{a} + \mathbf{b})^*$ enthaltenen Zeichenreihen vornehmen, dann erhalten wir alle Zeichenreihen, die gebildet werden, indem Zeichenreihen aus L und/oder M in beliebiger Reihenfolge miteinander verkettet werden.

5. Der Einfachheit halber werden wir die regulären Ausdrücke und ihre Sprachen als identisch behandeln und im Text nicht ausdrücklich durch den Vorsatz »die Sprache von« vor jedem regulären Ausdruck unterscheiden.

Die obige Aussage mag offensichtlich erscheinen, aber wie im Exkurs »Über reguläre Ausdrücke hinausgehende Erweiterungen des Tests können fehlschlagen« hervorgehoben wird, ist sie nicht einmal wahr, wenn andere Operatoren zu den drei Operatoren regulärer Ausdrücke hinzugefügt werden. Wir beweisen im nächsten Satz das allgemeine Prinzip für reguläre Ausdrücke.

Satz 3.13 Sei E ein regulärer Ausdruck mit den Variablen L_1, L_2, ..., L_m. Bilde den konkreten regulären Ausdruck C, indem jedes Vorkommen von L_j durch das Symbol a_j mit $j = 1, 2, \ldots, m$ ersetzt wird. Dann gilt für beliebige Sprachen L_1, L_2, ..., L_m, dass jede Zeichenreihe w aus $L(E)$ dargestellt werden kann als $w = w_1w_2 \ldots w_k$, wobei jede Zeichenreihe w_i in einer der Sprachen L_{j_i} und die Zeichenreihe $a_{j_1}a_{j_2}\ldots a_{j_k}$ in der Sprache $L(C)$ enthalten ist. Weniger formal ausgedrückt: Wir können L(E) konstruieren, indem wir jede in $L(C)$ enthaltene Zeichenreihe, die wir hier $a_{j_1}a_{j_2}\ldots a_{j_k}$ nennen, nehmen und für jede der Symbole a_{j_i} eine Zeichenreihe aus der zugehörigen Sprache L_{j_i} einsetzen.

Beweis: Der Beweis ist eine strukturelle Induktion über den Ausdruck E.

Induktionsbeginn: Die Basisfälle sind gegeben, wenn E *aus* ε, $\emptyset$ oder einer Variablen L besteht. In den ersten beiden Fällen gibt es nichts zu beweisen, da der konkrete Ausdruck C mit E identisch ist. Wenn E eine Variable L ist, dann gilt $L(E) = L$. Der konkrete Ausdruck C ist einfach **a**, wobei a das L entsprechende Symbol ist. Folglich ist $L(C) = \{a\}$. Wenn wir das Symbol a in dieser einen Zeichenreihe durch eine beliebige Zeichenreihe aus L ersetzen, dann erhalten wir die Sprache L, die gleich $L(E)$ ist.

Induktionsschritt: Es sind drei Fälle zu unterscheiden, die vom letzten Operator von E abhängen. Nehmen wir zuerst an, $E = F + G$, d. h. die Vereinigung ist der letzte Operator. Seien C und D die konkreten Ausdrücke, die aus F bzw. G gebildet werden, indem konkrete Symbole für die Sprachvariablen in diese Ausdrücke eingesetzt werden. Beachten Sie, dass sowohl in F als auch in G alle Vorkommen einer bestimmten Variablen durch dasselbe Symbol ersetzt werden müssen. Wir erhalten dann für E den konkreten Ausdruck $C + D$, und $L(C + D) = L(C) + L(D)$.

Angenommen, w ist eine in $L(E)$ enthaltene Zeichenreihe, wenn die Sprachvariablen von E durch bestimmte Sprachen ersetzt werden. Dann ist w Element von $L(F)$ oder von $L(G)$. Nach der Induktionsannahme erhalten wir w, indem wir mit einer konkreten Zeichenreihe aus $L(C)$ oder $L(D)$ beginnen und für die Symbole Zeichenreihen der entsprechenden Sprachen substituieren. Folglich kann die Zeichenreihe w in jedem Fall konstruiert werden, indem wir mit einer konkreten Zeichenreihe aus $L(C + D)$ beginnen und dieselben Ersetzungen von Symbolen durch Zeichenreihen durchführen.

Wir müssen zudem die Fälle betrachten, in denen E gleich FG oder F^* ist. Die Argumentation ähnelt allerdings der oben für die Vereinigung beschriebenen, und wir überlassen die Vervollständigung dieses Beweises dem Leser. ■

3.4.7 Test eines für reguläre Ausdrücke geltenden Gesetzes der Algebra

Wir können nun den Test zur Beantwortung der Frage, ob ein Gesetz für reguläre Ausdrücke gültig ist, formulieren und beweisen. Der Test zur Beantwortung der Frage, ob

$E = F$ wahr ist, wobei E und F zwei reguläre Ausdrücke mit der gleichen Menge von Variablen sind, lautet:

1. Wandle E und F in konkrete reguläre Ausdrücke C und D um, indem jede Variable durch ein konkretes Symbol ersetzt wird.
2. Prüfe, ob $L(C) = L(D)$. Falls dies zutrifft, dann ist $E = F$ wahr und somit ein gültiges Gesetz, andernfalls ist das »Gesetz« falsch. Beachten Sie, dass wir den Test zur Beantwortung der Frage, ob zwei reguläre Ausdrücke dieselbe Sprache definieren, erst in Abschnitt 4.4 behandeln werden. Wir können jedoch Ad-hoc-Methoden einsetzen, um die Gleichheit von Sprachpaaren zu entscheiden, für die wir uns interessieren. Rufen Sie sich dazu auch in Erinnerung: Wenn zwei Sprachen nicht gleich sind, dann genügt zum Beweis die Angabe eines Gegenbeispiels, nämlich einer einzigen Zeichenreihe, die zu einer der beiden Sprachen gehört, aber nicht zur anderen.

Satz 3.14 Der obige Test identifiziert für reguläre Ausdrücke geltende Gesetze korrekt.

Beweis: Wir werden zeigen, dass $L(E) = L(F)$ genau dann für alle Sprachen gilt, die für die Variablen von E und F eingesetzt werden, wenn $L(C) = L(D)$.

(Nur-wenn-Teil) Angenommen, $L(E) = L(F)$ gilt für alle möglichen Sprachen, die für die Variablen eingesetzt werden können. Wähle für jede Variable L das konkrete Symbol a, das L in den Ausdrücken C und D ersetzt. Dann gilt für diesen Fall $L(C) = L(E)$ und $L(D) = L(F)$. Da $L(E) = L(F)$ gegeben ist, folgt $L(C) = L(D)$.

(Wenn-Teil) Angenommen, $L(C) = L(D)$. Nach Satz 3.13 werden $L(E)$ und $L(F)$ jeweils konstruiert, indem die konkreten Symbole der Zeichenreihen in $L(C)$ und $L(D)$ durch Zeichenreihen der Sprache, der diese Symbole zugeordnet sind, ersetzt werden. Wenn die Zeichenreihen von $L(C)$ und $L(D)$ gleich sind, dann sind auch die beiden Sprachen, die auf diese Weise konstruiert werden, gleich, d. h. $L(E) = L(F)$. ■

Beispiel 3.15 Betrachten Sie den Gesetzesvorschlag $(L + M)^* = (L^*M^*)^*$. Wenn wir die Variablen L und M durch die konkreten Symbole a und b ersetzen, dann erhalten wir die regulären Ausdrücke $(\mathbf{a} + \mathbf{b})^* = (\mathbf{a}^*\mathbf{b}^*)^*$. Es lässt sich mühelos nachweisen, dass beide Ausdrücke die Sprache beschreiben, die alle aus den Symbolen a und b bestehenden Zeichenreihen umfasst. Folglich stehen die beiden konkreten Ausdrücke für dieselbe Sprache, und das Gesetz gilt.

Betrachten Sie $L^* = L^*L^*$ nun als weiteres Beispiel für ein Gesetz. Die konkreten Sprachen lauten $\mathbf{a}^*$ bzw. $\mathbf{a}^*\mathbf{a}^*$, und jede dieser Sprachen umfasst die Menge aller aus dem Symbol a bestehenden Zeichenreihen. Auch hier stellen wir fest, dass das gefundene Gesetz gültig ist, d. h. die Verkettung einer abgeschlossenen Sprache mit sich selbst ergibt eben diese Sprache.

Als letztes Beispiel betrachten Sie das angebliche Gesetz $L + ML = (L + M)L$. Wenn wir die Symbole a und b für die Variablen L und M wählen, dann erhalten wir die konkreten Ausdrücke $\mathbf{a} + \mathbf{ba} = (\mathbf{a} + \mathbf{b})\mathbf{a}$. Die Sprachen dieser Ausdrücke sind allerdings nicht gleich. Beispielsweise ist die Zeichenreihe aa in der Sprache des zweiten, nicht aber des ersten enthalten. Folglich ist dieses angebliche Gesetz falsch. ■

Über reguläre Ausdrücke hinausgehende Erweiterungen des Tests können fehlschlagen

Lassen Sie uns eine erweiterte Algebra für reguläre Ausdrücke betrachten, die den Schnittmengenoperator enthält. Wie wir in Satz 4.8 sehen werden, wird interessanterweise die Menge der beschreibbaren Sprachen durch das Hinzufügen des Operators $\cap$ zu den drei Operatoren für reguläre Ausdrücke nicht erweitert. Allerdings wird dadurch der Test für die Gültigkeit algebraischer Gesetze ungültig.

Betrachten Sie das »Gesetz« $L \cap M \cap N = L \cap M$; das heißt, die Schnittmenge von beliebigen drei Sprachen ist identisch mit der Schnittmenge der ersten beiden Sprachen. Dieses Gesetz ist offenkundig falsch. Sei beispielsweise $L = M = \{a\}$ und $N = \emptyset$. Der auf der Konkretisierung von Variablen basierende Test würde den Unterschied hier nicht aufdecken. Das heißt, wenn wir L, M und N durch die Symbole a, b und c ersetzen würden, dann würden wir testen, ob $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. Da beide Seiten gleich der leeren Menge sind, wären die Sprachen gleich, und der Test würde implizieren, dass das »Gesetz« gilt.

3.4.8 Übungen zum Abschnitt 3.4

Übung 3.4.1 Prüfen Sie die Gültigkeit folgender Identitäten regulärer Ausdrücke:

* a) $R + S = S + R$

b) $(R + S) + T = R + (S + T)$

c) $(RS)T = R(ST)$

d) $R\,(S + T) = RS + RT$

e) $(R + S)\,T = RT + ST$

* f) $(R^*)^* = R^*$

g) $(\varepsilon + R)^* = R^*$

h) $(R^*S^*)^* = (R + S)^*$

! **Übung 3.4.2** Beweisen Sie die Wahrheit oder Falschheit der folgenden Aussagen über reguläre Ausdrücke:

* a) $(R + S)^* = R^* + S^*$

b) $(RS + R)^*R = R(SR + R)^*$

* c) $(RS + R)^*RS = (RR^*S)^*$

d) $(R + S)^*S = (R^*S)^*$

e) $S(RS + S)^*R = RR^*S(RR^*S)^*$

Übung 3.4.3 In Beispiel 3.6 haben wir den folgenden regulären Ausdruck entwickelt:

$$(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1}) + (\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$$

Entwickeln Sie unter Verwendung der Distributivgesetze zwei verschiedene, einfachere, äquivalente Ausdrücke.

Übung 3.4.4 Am Anfang von Abschnitt 3.4.6 haben wir einen Teil des Beweises, dass $(L^*M^*)^* = (L + M)^*$, dargestellt. Vervollständigen Sie den Beweis, indem Sie zeigen, dass in $(L^*M^*)^*$ enthaltene Zeichenreihen auch in $(L + M)^*$ enthalten sind.

! **Übung 3.4.5** Vervollständigen Sie den Beweis von Satz 3.13, indem Sie die Fälle behandeln, in denen der reguläre Ausdruck E die Form FG oder die Form F^* hat.

3.5 Zusammenfassung von Kapitel 3

- *Reguläre Ausdrücke* : Diese algebraische Notation beschreibt genau dieselben Sprachen wie endliche Automaten: die regulären Sprachen. Als Operatoren für reguläre Ausdrücke sind die Vereinigung, die Verkettung (auch Konkatenation oder »Punktoperator« genannt) und der Sternoperator (oder Kleenesche Hülle) definiert.
- *Reguläre Ausdrücke in der Praxis* : Systeme wie Unix und verschiedene Unix-Befehle verwenden eine erweiterte Notation für reguläre Ausdrücke, die viele Kürzel für gebräuchliche Ausdrücke bietet. Zeichenklassen erlauben es, Symbolmengen einfach darzustellen, während Operatoren, wie »ein oder mehr Vorkommen von« und »höchstens ein Vorkommen von« die regulären Operatoren für reguläre Ausdrücke ergänzen.
- *Äquivalenz von regulären Ausdrücken und endlichen Automaten*: Wir können einen DEA durch eine induktive Konstruktion in einen regulären Ausdruck umwandeln. In einer solchen Konstruktion werden Ausdrücke für Beschriftungen von Pfaden konstruiert, die zunehmend umfangreichere Mengen von Zuständen durchlaufen. Alternativ können wir ein Verfahren zur Eliminierung von Zuständen einsetzen, um den regulären Ausdruck für einen DEA zu bilden. In der umgekehrten Richtung können wir aus einem regulären Ausdruck rekursiv einen ε-NEA konstruieren und diesen ε-NEA dann bei Bedarf in einen DEA umwandeln.
- *Die Algebra regulärer Ausdrücke* : Viele der algebraischen Gesetze der Arithmetik gelten für reguläre Ausdrücke, obwohl es sehr wohl Unterschiede gibt. Vereinigung und Verkettung sind assoziativ, aber nur die Vereinigung ist kommutativ. Die Verkettung ist bezüglich der Vereinigung distributiv. Die Vereinigung ist idempotent.
- *Algebraische Identitäten testen* : Wir können feststellen, ob eine Äquivalenz regulärer Ausdrücke, die Variablen als Argumente enthalten, wahr ist, indem wir die Variablen durch verschiedene Konstanten ersetzen und testen, ob die resultierenden Sprachen gleich sind.

LITERATURANGABEN ZU KAPITEL 3

Das Konzept der regulären Ausdrücke und der Beweis ihrer Äquivalenz mit endlichen Automaten ist das Werk von S. C. Kleene [3]. Die Konstruktion eines ε-NEA aus einem regulären Ausdruck, die hier vorgestellt wurde, wird als »McNaughton-Yamada-Konstruktion« bezeichnet und stammt aus [4]. Der Test zur Überprüfung der Äquivalenz von regulären Ausdrücken, in dem Variablen als Konstanten behandelt werden, wurde von J. Gischer [2] beschrieben. Dieser Bericht zeigte, dass das Hinzufügen verschiedener anderer Operatoren, wie Durchschnitt oder ordnungserhaltende Durchmischung (siehe Übung 7.3.4) den Test fehlschlagen lässt, obwohl diese Operatoren die Klasse der darstellbaren Sprachen nicht erweitern.

Schon vor der Entwicklung von Unix untersuchte K. Thompson die Verwendung regulärer Ausdrücke in Kommandos wie `grep`, und sein Algorithmus zur Verarbeitung solcher Kommandos findet sich in [5]. Die frühe Unix-Entwicklung führte zur Entwicklung verschiedener anderer Kommandos, die von der Notation erweiterter regulärer Ausdrücke stark Gebrauch machen, z. B. das Kommando `lex` von M. Lesk. Eine Beschreibung dieses Kommandos und anderer auf regulären Ausdrücken basierenden Techniken sind in [1] enthalten.

1. A. V. Aho, R. Sethi und J. D. Ullman [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA.

2. J. L. Gischer [1984]. STAN-CS-TR-84-1033.

3. S. C. Kleene [1956]. »Representation of events in nerve nets and finite automata«, in C. E. Shannon und J. McCarthy, *Automata Studies*, Princeton Univ. Press, S. 3–42.

4. R. McNaughton und H. Yamada [Jan. 1960]. »Regular expressions and state graphs for automata«, *IEEE Trans. Electronic Computers* **9**:1, S. 39–47.

5. K. Thompson [Juni 1968]. »Regular expression search algorithm«, *Comm. ACM* **11**:6, S. 419–422.

KAPITEL 4

Eigenschaften regulärer Sprachen

Dieses Kapitel untersucht die Eigenschaften regulärer Sprachen. Unser erstes Untersuchungsinstrument ist ein Beweisverfahren dafür, dass bestimmte Sprachen nicht regulär sind. Dieser Satz, der als »Pumping-Lemma« bezeichnet wird, wird in Abschnitt 4.1 vorgestellt.

Ein wichtiger Merkmalstyp regulärer Sprachen wird als »Eigenschaft der Abgeschlossenheit« bezeichnet. Diese Eigenschaften ermöglichen die Entwicklung von Modulen zur Erkennung von Sprachen, die durch bestimmte Operationen aus anderen Sprachen gebildet werden. Beispielsweise ist der Durchschnitt zweier regulärer Sprachen auch regulär. Wenn wir Automaten zur Erkennung zweier verschiedener regulärer Sprachen besitzen, können wir einen Automaten mechanisch konstruieren, der genau den Durchschnitt dieser beiden Sprachen erkennt. Da der Automat für den Durchschnitt über viel mehr Zustände verfügen kann als jeder der beiden gegebenen Automaten, kann diese »Eigenschaft der Abgeschlossenheit« ein nützliches Hilfsmittel zur Konstruktion komplexer Automaten darstellen. In Abschnitt 2.1 wurde diese Konstruktion in einer grundlegenden Weise verwendet.

Einige andere wichtige Merkmale regulärer Sprachen werden als »Entscheidbarkeitseigenschaften« bezeichnet. Das Studium dieser Eigenschaften liefert uns Algorithmen zur Beantwortung wichtiger Fragen zu Automaten. Ein zentrales Beispiel ist ein Algorithmus zur Beantwortung der Frage, ob zwei Automaten dieselbe Sprache definieren. Eine Konsequenz unserer Fähigkeit, diese Frage zu entscheiden, ist, dass wir Automaten »minimieren«, d. h. ein Äquivalent für einen gegebenen Automaten finden können, das über die minimale Anzahl von Zuständen verfügt. Dieses Problem war Jahrzehnte lang im Design von Schaltkreisen wichtig, da die Kosten eines Schaltkreises (die Fläche, die der Schaltkreis auf dem Chip einnimmt) in der Regel sinken, wenn die Anzahl von Zuständen des Automaten, der von einem Schaltkreis implementiert wird, verringert wird.

4.1 Beweis der Nichtregularität von Sprachen

Wir haben festgestellt, dass die Klasse der Sprachen, die als reguläre Sprachen bezeichnet werden, mindestens vier Beschreibungen besitzen. Es handelt sich um die Sprachen, die von DEAs, NEAs und ε-NEAs akzeptiert werden. Es sind zudem die Sprachen, die durch reguläre Ausdrücke definiert werden.

Nicht jede Sprache ist eine reguläre Sprache. In diesem Abschnitt stellen wir eine mächtige Technik namens »Pumping-Lemma« vor, mit der gezeigt werden kann, dass bestimmte Sprachen nicht regulär sind. Wir führen dann verschiedene Beispiele nichtregulärer Sprachen an. In Abschnitt 4.2 werden wir zeigen, wie das Pumping-Lemma in Verbindung mit den Eigenschaften der Abgeschlossenheit von regulären Sprachen zum Beweis der Nichtregularität anderer Sprachen eingesetzt werden kann.

4.1.1 Das Pumping-Lemma für reguläre Sprachen

Lassen Sie uns die Sprache $L_{01} = \{0^n1^n \mid n \geq 1\}$ betrachten. Diese Sprache enthält alle Zeichenreihen 01, 0011, 000111 etc, die aus einer oder mehr Nullen gefolgt von der gleichen Anzahl von Einsen bestehen. Wir behaupten, dass L_{01} keine reguläre Sprache ist. Die intuitive Argumentation lautet, dass L_{01} die Sprache eines DEA A sein müsste, wenn L_{01} eine reguläre Sprache wäre. Dieser Automat hat eine bestimmte Anzahl von Zuständen, sagen wir k Zustände. Stellen Sie sich vor, der Automat erhält k Nullen als Eingabe. Er befindet sich in irgendeinem Zustand, nachdem er die $k + 1$ Präfixe der Eingabe verarbeitet hat: ε, 0, 00, ... , 0^k. Da es nur k Zustände gibt, besagt das Schubfachprinzip, dass A nach dem Lesen zweier verschiedener Präfixe, die wir 0^i und 0^j nennen, sich im gleichen Zustand befinden muss, sagen wir Zustand q.

Nehmen wir nun stattdessen an, dass der Automat A nach dem Lesen von i oder j Nullen Einsen als Eingabe erhält. Nachdem er i Einsen gelesen hat, muss er akzeptieren, wenn er zuvor i Nullen, nicht aber, wenn er zuvor j Nullen gelesen hat. Da er sich im Zustand q befand, als die Eingabe von Einsen begann, kann er sich nicht daran »erinnern«, ob er i oder j Nullen gelesen hat. Daher können wir den Automaten A »täuschen« und dazu bewegen, das Falsche zu tun – zu akzeptieren, wenn er nicht akzeptieren sollte, oder nicht zu akzeptieren, wenn er es sollte.

Die obige Argumentation ist informell, lässt sich aber präzisieren. Allerdings können wir auch, wie nachfolgend beschrieben, unter Verwendung eines allgemeinen Ergebnisses zu dem Schluss kommen, dass die Sprache L_{01} keine reguläre Sprache ist.

Satz 4.1 (*Das Pumping-Lemma für reguläre Sprachen*) Sei L eine reguläre Sprache. Dann gibt es eine Konstante n (die von L abhängt), derart dass für jede Zeichenreihe w in L mit $|w| \geq n$ gilt, dass wir w in drei Zeichenreihen $w = xyz$ zerlegen können, für die gilt:

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. Für alle $k \geq 0$ gilt, dass die Zeichenreihe xy^kz auch in L enthalten ist.

Das heißt, wir können stets eine nichtleere Zeichenreihe y nicht allzu weit vom Beginn von w finden, die beliebig oft wiederholt oder gelöscht (wenn $k = 0$) werden kann, wobei die resultierende Zeichenreihe nach wie vor in der Sprache L enthalten ist.

Beweis: Angenommen, L sei regulär. Dann gilt $L = L(A)$ für einen DEA A. Angenommen, A besitzt n Zustände. Nun betrachten wir eine beliebige Zeichenreihe w aus L mit einer Länge, die größer oder gleich n ist, sagen wir $w = a_1a_2 \ldots a_m$, wobei $m \geq n$ und jedes a_i ein Eingabesymbol ist. Wir definieren für $i = 0, 1, \ldots, n$ den Zustand p_i als $\hat{\delta}(q_0, a_1a_2 \ldots a_i)$, wobei δ die Übergangsfunktion von A und q_0 der Startzustand von A ist. Das

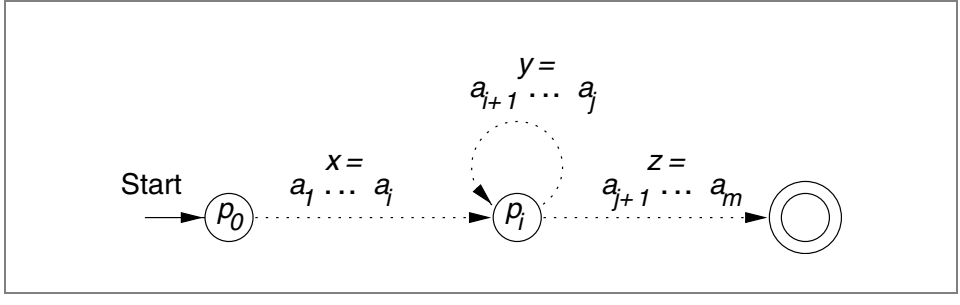
heißt, p_i ist der Zustand, in dem sich A nach dem Lesen der ersten i Symbole von w befindet. Beachten Sie, dass $p_0 = q_0$.

Nach dem Schubfachprinzip ist es nicht möglich, dass die $n + 1$ Zustände p_i für $i = 0, 1, \ldots n$ paarweise verschieden sind, da es nur n verschiedene Zustände gibt. Folglich können wir zwei verschiedene ganze Zahlen i und j finden mit $0 \leq i < j \leq n$, derart dass $p_i = p_j$. Wir können $w = xyz$ nun wie folgt zerlegen:

1. $x = a_1 a_2 \ldots a_i$
2. $y = a_{i+1} a_{i+2} \ldots a_j$
3. $z = a_{j+1} a_{j+2} \ldots a_m$

Das heißt, x führt einmal zum Zustand p_i, y führt von p_i zurück nach p_i (da p_i gleich p_j ist), und z ist der Rest von w. Die Beziehungen zwischen den Zeichenreihen und Zuständen sind in Abbildung 4.1 dargestellt. Beachten Sie, dass x leer sein kann, falls $i = 0$. Zudem kann z leer sein, nämlich wenn $j = n = m$. Allerdings kann y nicht leer sein, da i kleiner als j sein muss.

Abbildung 4.1: Jede Zeichenreihe, deren Länge nicht kleiner als die Anzahl der Zustände ist, muss bewirken, dass ein Zustand zweimal durchlaufen wird



Betrachten wir nun, was passiert, wenn der Automat A die Eingabe xy^kz mit einem beliebigen Wert für $k \geq 0$ erhält. Wenn $k = 0$, dann wechselt der Automat nach der Eingabe x vom Startzustand q_0 (der gleich p_0 ist) in den Zustand p_i. Da p_i gleich p_j ist, muss es so sein, dass A von p_i aus nach der Eingabe z in den in **Abbildung 4.1** dargestellten akzeptierenden Zustand überführt wird. Folglich akzeptiert A die Zeichenreihe xz.

Wenn $k > 0$, dann wechselt A nach der Eingabe x in den Zustand p_i, kehrt auf die Eingabe y^k hin k-mal von p_i zu p_i zurück und geht dann nach der Eingabe z in den akzeptierenden Zustand über. Folglich wird xy^kz für jeden Wert $k \geq 0$ von A akzeptiert, d. h. xy^kz ist in der Sprache L enthalten. ■

4.1.2 Anwendungen des Pumping-Lemmas

Wir wollen nun einige Beispiele betrachten, die zeigen, wie das Pumping-Lemma eingesetzt wird. In jedem der beschriebenen Fälle geben wir eine Sprache an und beweisen mithilfe des Pumping-Lemmas, dass die Sprache nicht regulär ist.

Das Pumping-Lemma als Zwei-Personen-Spiel

Rufen Sie sich unsere Diskussion aus Abschnitt 1.2.3 in Erinnerung, in der wir aufzeigten, dass man sich einen Satz, dessen Aussage mehrere sich abwechselnde All- und Existenzquantoren enthält, als Spiel für zwei Spieler vorstellen kann. Das Pumping-Lemma ist ein wichtiges Beispiel für diese Art von Sätzen, da es abwechselnd zwei All- und zwei Existenzquantoren enthält: »**Für alle** regulären Sprachen L **gibt es** ein n, derart dass **für alle** w in L mit $|w| \geq n$ gilt, dass **es** eine Zerlegung $w = xyz$ **gibt**, derart dass ...« Wir können die Anwendung des Pumping-Lemmas als Spiel betrachten, in dem

1. Spieler 1 die Sprache L auswählt, deren Nichtregularität es zu beweisen gilt.
2. Spieler 2 die Zahl n auswählt, Spieler 1 aber nicht offenbart, d. h. Spieler 1 muss ein Spiel für alle möglichen Werte n entwerfen.
3. Spieler 1 wählt die Zeichenreihe w aus, die von n abhängen kann und die mindestens die Länge n haben muss.
4. Spieler 2 zerlegt w in x, y und z, wobei er die Beschränkungen beachtet, die durch das Pumping-Lemma auferlegt werden: $y \neq \varepsilon$ und $|xy| \leq n$. Wieder braucht Spieler 2 die Werte von x, y und z dem Spieler 1 nicht mitzuteilen, die allerdings die Bedingungen erfüllen müssen.
5. Spieler 1 »gewinnt« durch die Nennung eines k, das eine Funktion von n, x, y und z sein kann, sodass xy^kz nicht in L enthalten ist.

Beispiel 4.2 Wir wollen nun zeigen, dass die Sprache L_{eq}, die aus allen Zeichenreihen mit der gleichen Anzahl von Nullen und Einsen (die sich in keiner bestimmten Reihenfolge befinden müssen) besteht, keine reguläre Sprache ist. Im Sinn des im Exkurs »Das Pumping-Lemma als Zwei-Personen-Spiel« beschriebenen Spiels, werden wir Spieler 1 sein und müssen uns mit den von Spieler 2 getroffenen Spielzügen auseinander setzen. Angenommen, n ist die Konstante, die es gemäß dem Pumping-Lemma geben muss, wenn L_{eq} regulär ist; d. h. »Spieler 2« wählt n. Wir wählen $w = 0^n1^n$, d. h. n Nullen gefolgt von n Einsen, eine Zeichenreihe, die mit Sicherheit in L_{eq} enthalten ist.

Nun zerlegt »Spieler 2« unsere Zeichenreihe w in xyz. Wir wissen nur, dass $y \neq \varepsilon$ und dass $|xy| \leq n$. Diese Information ist jedoch sehr hilfreich, und wir »gewinnen« folgendermaßen: Da $|xy| \leq n$ und xy am Anfang von w steht, wissen wir, dass x und y nur aus Nullen bestehen können. Das Pumping-Lemma besagt, dass xz in L_{eq} enthalten ist, wenn L_{eq} regulär ist. Dieser Schluss entspricht dem Fall $k = 0$ im Pumping-Lemma.[1] xz enthält also n Einsen, da alle Einsen von w in z enthalten sind. Aber xz enthält weniger als n Nullen, da hier die Nullen der Zeichenreihe y fehlen. Da $y \neq \varepsilon$, wissen wir, dass x und z zusammengenommen über nicht mehr als $n - 1$ Nullen verfügen können. Nachdem wir von der Annahme ausgingen, L_{eq} sei eine reguläre Sprache, haben wir nun eine Tatsache bewiesen, deren Falschheit bekannt ist, nämlich dass xz in L_{eq} enthalten ist. Wir haben damit einen Widerspruchsbeweis für die Tatsache, dass L_{eq} keine reguläre Sprache ist. ■

1. Beachten Sie im Folgenden, dass uns auch die Wahl von $k = 2$ und sogar jeder Wert von k außer 1 zum Erfolg geführt hätte.

Beispiel 4.3 Wir wollen zeigen, dass die Sprache L_{pr}, die alle aus Einsen bestehenden Zeichenreihen umfasst, deren Länge eine Primzahl ist, keine reguläre Sprache ist. Angenommen, sie wäre eine reguläre Sprache. Dann würde es eine Konstante n geben, die die Bedingungen des Pumping-Lemmas erfüllt. Betrachten wir eine Primzahl $p \geq n + 2$. Es muss eine solche Primzahl p geben, da die Anzahl der Primzahlen unendlich ist. Sei $w = 1^p$.

Nach dem Pumping-Lemma können wir $w = xyz$ so zerlegen, dass $y \neq \varepsilon$ und $|xy| \leq n$. Sei $|y| = m$. Dann gilt $|xz| = p - m$. Betrachten wir nun die Zeichenreihe $xy^{p-m}z$, die gemäß dem Pumping-Lemma in L_{pr} enthalten sein muss, wenn L_{pr} tatsächlich regulär ist. Allerdings ist

$$|xy^{p-m}z| = |xz| + (p - m)|y| = p - m + (p - m)m = (m + 1)(p - m)$$

Es sieht so aus, als sei $|xy^{p-m}z|$ keine Primzahl, da sie über die beiden Faktoren $m + 1$ und $p - m$ verfügt. Wir müssen jedoch überprüfen, dass keiner dieser beiden Faktoren 1 ist, denn in diesem Fall könnte $(m + 1)(p - m)$ doch eine Primzahl sein; aber $m + 1 > 1$, da aus $y \neq \varepsilon$ hervorgeht, dass $m \geq 1$. Zudem ist $p - m > 1$, da $p \geq n + 2$ als gegeben vorausgesetzt wurde, und $m \leq n$, weil

$$m = |y| \leq |xy| \leq n.$$

Folglich ist $p - m \geq 2$.

Wir begannen mit der Annahme, dass die fragliche Sprache regulär sei, und haben wieder einen Widerspruch hergeleitet, indem wir gezeigt haben, dass eine Zeichenreihe nicht in der Sprache enthalten ist, die gemäß dem Pumping-Lemma in der Sprache enthalten sein müsste. Infolgedessen können wir schlussfolgern, dass L_{pr} keine reguläre Sprache ist. ■

4.1.3 Übungen zum Abschnitt 4.1

Übung 4.1.1 Beweisen Sie, dass die folgenden Sprachen keine regulären Sprachen sind:

a) $\{0^n1^n \mid n \geq 1\}$. Bei dieser Sprache, die aus Zeichenreihen von Nullen gefolgt von jeweils gleich langen Zeichenreihen von Einsen besteht, handelt es sich um die Sprache L_{01}, die wir am Beginn dieses Abschnitts informell betrachtet haben. Hier sollten Sie das Pumping-Lemma im Beweis verwenden.

b) Die Menge aller Zeichenreihen mit korrekt vernesteten linken und rechten Klammern. Es handelt sich um die Zeichenreihen von runden Klammern »(« und »)«, die in einem wohlgeformten arithmetischen Ausdruck vorkommen können.

* c) $\{0^n10^n \mid n \geq 1\}$

d) $\{0^n1^m2^n \mid n$ und m sind zufällig gewählte ganze Zahlen$\}$

e) $\{0^n1^m \mid n \leq m\}$

f) $\{0^n1^{2n} \mid n \geq 1\}$

! **Übung 4.1.2** Beweisen Sie, dass die folgenden Sprachen keine regulären Sprachen sind:

* a) $\{0^n \mid n$ ist eine Quadratzahl$\}$

b) $\{0^n \mid n$ ist eine Kubikzahl$\}$

c) $\{0^n \mid n$ ist eine Zweierpotenz$\}$

d) Die Menge der Zeichenreihen aus Nullen und Einsen, deren Länge eine Quadratzahl ist

e) Die Menge der Zeichenreihen aus Nullen und Einsen, die die Form ww haben

f) Die Menge der Zeichenreihen aus Nullen und Einsen, die die Form ww^R haben, d. h. aus einer Zeichenreihe gefolgt von deren Spiegelung bestehen (in Abschnitt 4.2.2 finden Sie eine formale Definition der Spiegelung einer Zeichenreihe)

g) Die Menge der Zeichenreihen aus Nullen und Einsen, die die Form $w\overline{w}$ haben, wobei $\overline{w}$ aus w gebildet wird, indem alle Nullen durch Einsen und alle Einsen durch Nullen ersetzt werden; so ist etwa $\overline{011} = 100$; 011100 ein Beispiel für eine Zeichenreihe dieser Sprache

h) Die Menge der Zeichenreihen der Form $w1^n$, wobei w eine Zeichenreihe aus Nullen und Einsen der Länge n ist

!! **Übung 4.1.3** Beweisen Sie, dass die folgenden Sprachen keine regulären Sprachen sind:

a) Die Menge der Zeichenreihen aus Nullen und Einsen, die mit einer Eins beginnen und in der Interpretation als ganze Zahl eine Primzahl darstellen

b) Die Menge der Zeichenreihen der Form 0^i1^j, derart dass 1 der größte gemeinsame Teiler von i und j ist

! **Übung 4.1.4** Wenn wir versuchen, das Pumping-Lemma auf eine reguläre Sprache anzuwenden, dann »gewinnt der Gegner« und wir können den Beweis nicht vollständig führen. Zeigen Sie, worin die Schwierigkeit besteht, wenn wir für L eine der folgenden Sprachen einsetzen:

* a) Die leere Menge

* b) $\{00, 11\}$

* c) $(\mathbf{00} + \mathbf{11})^*$

d) $\mathbf{01}^*\mathbf{0}^*\mathbf{1}$

4.2 Abgeschlossenheitseigenschaften regulärer Sprachen

In diesem Abschnitt beweisen wir verschiedene Sätze der Form »Wenn bestimmte Sprachen regulär sind und eine Sprache L durch bestimmte Operationen (z. B. L ist die Vereinigung der beiden regulären Sprachen) aus ihnen gebildet wird, dann ist L auch regulär.« Diese Sätze werden häufig als *Abgeschlossenheitseigenschaften* regulärer Sprachen bezeichnet, da sie zeigen, dass die Klasse der regulären Sprachen bezüglich der genannten Operation abgeschlossen ist. *Abgeschlossenheitseigenschaften* geben der Vorstellung Ausdruck, dass gewisse, mit einer (oder mehreren) regulären Sprache(n) verwandte Sprachen auch regulär sind. Sie stellen zudem ein interessantes Beispiel dafür dar, wie die äquivalenten Repräsentationen regulärer Sprachen (Automaten und reguläre Ausdrücke) sich in unserem Verständnis dieser Sprachklasse gegenseitig verstärken, da sich häufig eine Repräsentation viel besser als die anderen zum Beweis einer Abgeschlossenheitseigenschaft eignet. Nachfolgend sind die wichtigsten Abgeschlossenheitseigenschaften regulärer Sprachen zusammengefasst:

1. Die Vereinigung zweier regulärer Sprachen ist regulär.
2. Der Durchschnitt zweier regulärer Sprachen ist regulär.
3. Das Komplement einer regulären Sprachen ist regulär.
4. Die Differenz zweier regulärer Sprachen ist regulär.
5. Die Spiegelung einer regulären Sprache ist regulär.
6. Die Hülle (Sternoperator) einer regulären Sprache ist regulär.
7. Die Verkettung von regulären Sprachen ist regulär.
8. Ein Homomorphismus (Ersetzung von Symbolen durch Zeichenreihen) einer regulären Sprache ist regulär.
9. Der inverse Homomorphismus einer regulären Sprache ist regulär.

4.2.1 Abgeschlossenheit regulärer Sprachen bezüglich Boolescher Operationen

Als erste Abgeschlossenheitseigenschaften stellen wir die Booleschen Operationen vor: Vereinigung, Durchschnitt und Komplement.

1. Seien L und M Sprachen über dem Alphabet Σ; dann ist $L \cup M$ die Sprache, die alle Zeichenreihen enthält, die in L oder in M oder in beiden Sprachen enthalten sind.
2. Seien L und M Sprachen über dem Alphabet Σ; dann ist $L \cap M$ die Sprache, die alle Zeichenreihen enthält, die sowohl in L als auch in M enthalten sind.
3. Sei L eine Sprache über dem Alphabet Σ; dann ist $\overline{L}$, das Komplement von L, die Menge der in Σ^* enthaltenen Zeichenreihen, die nicht in L enthalten sind.

Es stellt sich heraus, dass die regulären Sprachen bezüglich aller drei Booleschen Operationen abgeschlossen sind. Wie wir sehen werden, verfolgen die Beweise jedoch unterschiedliche Ansätze.

Was geschieht, wenn die Sprachen unterschiedliche Alphabete haben?

Wenn wir die Vereinigung oder den Durchschnitt von zwei Sprachen L und M bilden, dann können diese Sprachen unterschiedliche Alphabete besitzen. Es ist beispielweise möglich, dass $L \subseteq \{a, b\}^*$, während $M \subseteq \{b, c, d\}^*$. Wenn eine Sprache L aus Zeichenreihen besteht, die sich aus Symbolen des Alphabets Σ zusammensetzen, dann kann man L auch als Sprache über jedem endlichen Alphabet betrachten, das eine Obermenge von Σ ist. Folglich können wir die beiden Sprachen L und M aus dem obigen Beispiel auch als Sprachen über dem Alphabet $\{a, b, c, d\}$ betrachten. Die Tatsache, dass keine der in L enthaltenen Zeichenreihen die Symbole c und d enthält, ist ebenso irrelevant wie die Tatsache, dass keine der in M enthaltenen Zeichenreihen das Symbol a enthält.

Analog hierzu können wir zur Bildung des Komplements einer Sprache L, die eine Teilmenge von Σ_1^* ist, das Komplement *in Bezug auf ein* Alphabet Σ_2 bilden, das eine Obermenge des Alphabets Σ_1 darstellt. Wenn wir so vorgehen, dann ist das Komplement von L gleich $\Sigma_2^* - L$; d. h. das Komplement von L in Bezug auf Σ_2 umfasst (neben anderen Zeichenreihen) alle Zeichenreihen aus Σ_2^*, die mindestens ein Symbol aufweisen, das in Σ_2, aber nicht in Σ_1 enthalten ist. Hätten wir dagegen das Komplement von L in Bezug auf Σ_1 gebildet, dann wäre keine Zeichenreihe, die Symbole aus $\Sigma_2 - \Sigma_1$ enthält, in $\overline{L}$ enthalten. Daher sollten wir streng genommen stets das Alphabet angeben, das der Komplementbildung zu Grunde liegt. Oft ist es jedoch offensichtlich, welches Alphabet gemeint ist. Wenn L z. B. durch einen Automaten definiert wird, dann beinhaltet die Spezifikation des Automaten ein Alphabet. Folglich werden wir häufig von dem »Komplement« reden, ohne das Alphabet anzugeben.

Abgeschlossenheit bezüglich der Vereinigung

Satz 4.4 Wenn L und M reguläre Sprachen sind, dann ist auch $L \cup M$ eine reguläre Sprache.

Beweis: Dieser Beweis ist einfach. Da L und M regulär sind, sind sie darstellbar durch reguläre Ausdrücke; sei $L = L(R)$ und $M = L(S)$. Dann folgt aus der Definition des Operators + für reguläre Ausdrücke unmittelbar, dass $L \cup M = L(R + S)$. ■

Abgeschlossenheit bezüglich der Komplementbildung

Der Satz für die Vereinigung wurde dadurch sehr vereinfacht, dass die Sprachen als reguläre Ausdrücke beschrieben wurden. Als Nächstes wollen wir die Komplementbildung betrachten. Wissen Sie, wie man einen regulären Ausdruck so abändert, dass er das Komplement einer Sprache definiert? Nun, wir wissen es auch nicht. Der Beweis ist jedoch möglich, da man, wie wir in Satz 4.5 sehen werden, von einem DEA ausgehend einen DEA konstruieren kann, der das Komplement akzeptiert. Folglich können wir, ausgehend von einem regulären Ausdruck, auf folgende Weise einen regulären Ausdruck für dessen Komplement finden:

1. Wandle den regulären Ausdruck in einen ε-NEA um.
2. Wandle diesen ε-NEA mithilfe der Teilmengenkonstruktion in einen DEA um.
3. Bilde das Komplement für die akzeptierenden Zustände dieses DEA.

4. Wandle den Komplement-DEA unter Verwendung der in den Abschnitten 3.2.1 oder 3.2.2 beschriebenen Konstruktionen wieder in einen regulären Ausdruck um.

Abgeschlossenheit bezüglich regulärer Operationen

Der Beweis, dass reguläre Sprachen bezüglich der Vereinigung abgeschlossen sind, war äußerst einfach, da die Vereinigung eine der drei Operationen ist, die reguläre Ausdrücke definieren. Derselbe Gedanke wie in Satz 4.4 gilt genauso für die Verkettung und die Hüllenbildung. Das heißt:

- Wenn L und M reguläre Sprachen sind, dann ist auch LM eine reguläre Sprache.
- Wenn L eine reguläre Sprache ist, dann ist auch L^* eine reguläre Sprache.

Satz 4.5 Wenn L eine reguläre Sprache über dem Alphabet Σ ist, dann ist $\overline{L} = \Sigma^* - L$ auch eine reguläre Sprache.

Beweis: Sei $L = L(A)$ für einen DEA $A = (Q, \Sigma, \delta, q_0, F)$. Dann gilt $\overline{L} = L(B)$, wobei B für den DEA $(Q, \Sigma, \delta, q_0, Q - F)$ steht. Das heißt, B unterscheidet sich von A nur dadurch, dass die akzeptierenden Zustände von A den nicht akzeptierenden Zuständen von B entsprechen und umgekehrt. Dann ist w genau dann in $L(B)$ enthalten, wenn $\hat{\delta}(q_0, w)$ in $Q - F$ enthalten ist, was genau dann der Fall ist, wenn w nicht in $L(A)$ enthalten ist. ■

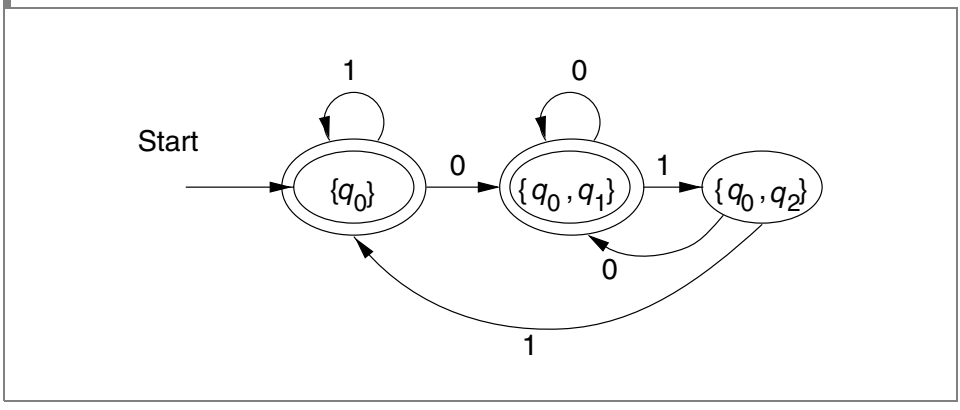
Beachten Sie, dass es im obigen Beweis wichtig ist, dass $\hat{\delta}(q_0, w)$ stets einen Zustand beschreibt, d. h. dass in A keine Übergänge fehlen. Wäre dies der Fall, dann könnten bestimmte Zeichenreihen weder zu einem akzeptierenden noch zu einem nicht akzeptierenden Zustand von A führen, und diese Zeichenreihen würden sowohl in $L(A)$ als auch in $L(B)$ fehlen. Glücklicherweise haben wir den DEA so definiert, dass von jedem Zustand aus für jedes in Σ enthaltene Symbol ein Zustandsübergang erfolgt, sodass jede Zeichenreihe entweder zu einem in F oder in Q - F enthaltenen Zustand führt.

Beispiel 4.6 Sei A der Automat aus Abbildung 2.9. Dieser DEA A akzeptiert alle Zeichenreihen aus Nullen und Einsen, die auf 01 enden, und keine anderen. Die Sprache dieses Automaten wird durch folgenden regulären Ausdruck beschrieben: $L(A) = (\mathbf{0} + \mathbf{1})^*\mathbf{01}$. Das Komplement von $L(A)$ besteht daher aus allen Zeichenreihen aus Nullen und Einsen, die *nicht* mit 01 enden.

Abbildung 4.2 zeigt den Automaten für $\{0, 1\}^* - L(A)$. Dieser Automat unterscheidet sich von dem in Abbildung 2.9. nur dadurch, dass der akzeptierende Zustand zu einem nicht akzeptierenden und die beiden nicht akzeptierenden Zustände zu akzeptierenden wurden. ■

Beispiel 4.7 In diesem Beispiel wenden wir Satz 4.5 an, um zu zeigen, dass eine bestimmte Sprache nicht regulär ist. In Beispiel 4.2 haben wir gezeigt, dass die Sprache L_{eq}, die aus Zeichenreihen mit der gleichen Anzahl von Nullen und Einsen besteht, nicht regulär ist. Dieser Beweis bestand in der unmittelbaren Anwendung des Pumping-Lemmas. Jetzt betrachten wir die Sprache M, die aus jenen Zeichenreihen besteht, die eine ungleiche Anzahl von Nullen und Einsen enthalten.

Abbildung 4.2: Der DEA, der das Komplement der Sprache (**0** + **1**)*__01__ akzeptiert



Es lässt sich nicht so leicht mithilfe des Pumping-Lemmas zeigen, dass die Sprache M nicht regulär ist. Wenn wir mit einer in M enthaltenen Zeichenreihe w beginnen, sie in $w = xyz$ aufteilen und y wiederholt einfügen oder löschen, dann stellen wir möglicherweise fest, dass y eine Zeichenreihe wie 01 ist, die über die gleiche Anzahl von Nullen und Einsen verfügt. Falls dem so wäre, dann hätte xy^kz für kein k die gleiche Anzahl von Nullen und Einsen, da xyz nicht die gleiche Anzahl von Nullen und Einsen enthält und diese Anzahl beim Hinzufügen bzw. Löschen von y jeweils um den gleichen Betrag erhöht bzw. vermindert wird. Folglich könnten wir nie das Pumping-Lemma einsetzen, um einen Widerspruch zur Behauptung, M sei regulär, zu konstruieren.[2]

M ist aber tatsächlich nicht regulär. Das liegt i.W. daran, dass $M = \overline{L}_{eq}$. Weil nämlich das Komplement des Komplements die Menge ist, mit der wir begonnen haben, folgt, dass $L_{eq} = \overline{M}$. Wenn M regulär ist, dann ist nach Satz 4.5 auch L_{eq} regulär. Wir wissen jedoch, dass L_{eq} nicht regulär ist. Damit haben wir einen Widerspruchsbeweis dafür, dass M nicht regulär ist. ■

Abgeschlossenheit bezüglich des Durchschnitts

Lassen Sie uns nun den Durchschnitt zweier Sprachen betrachten. Wir müssen hier wenig tun, da die drei Booleschen Operationen nicht unabhängig sind. Wenn wir über Methoden zur Komplementbildung und Vereinigung verfügen, dann können wir den Durchschnitt der Sprachen L und M durch die Identität

$$L \cap M = \overline{\overline{L} \cup \overline{M}} \tag{4.1}$$

erhalten.

Im Allgemeinen ist der Durchschnitt zweier Mengen die Menge der Elemente, die nicht in den Komplementen beider Mengen enthalten sind. Diese Beobachtung, die in Gleichung (4.1) ausgedrückt ist, ist eines der *DeMorganschen Gesetze*. Das andere Gesetz entspricht diesem, wobei Durchschnitt und Vereinigung vertauscht sind; d. h. $L \cup M = \overline{\overline{L} \cap \overline{M}}$.

2. Anmerkung des Übersetzers: Diese Feststellung ist so nicht richtig. Tatsächlich kann man mit Hilfe des Pumping-Lemmas beweisen, dass M nicht regulär ist. Allerdings ist der Beweis nicht so einfach wie der hier angegebene.

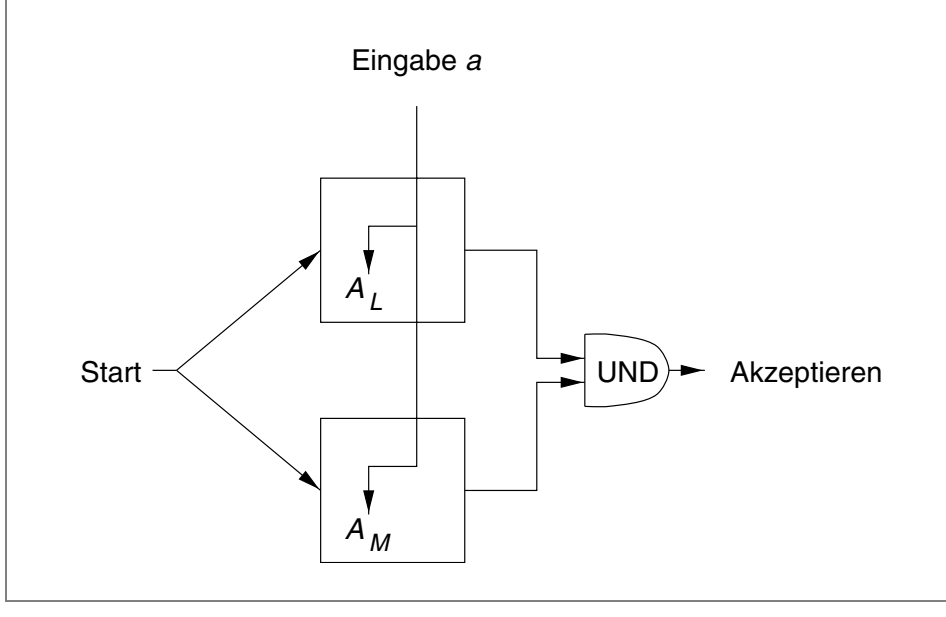
Wir können allerdings auch direkt einen DEA für den Durchschnitt zweier Sprachen konstruieren. Diese Konstruktion, die im Grunde genommen zwei DEAs parallel ausführt, ist für sich genommen nützlich. Wir haben sie beispielsweise verwendet, um den in Abbildung 2.3 dargestellten Automaten zu konstruieren, der das »Produkt« dessen repräsentiert, was die anderen beiden Beteiligten – Bank und Geschäft – taten. Wir werden die *Produktkonstruktion* im nächsten Satz formalisieren.

Satz 4.8 Wenn L und M reguläre Sprachen sind, dann ist auch $L \cap M$ eine reguläre Sprache.

Beweis: Seien L und M die Sprachen der Automaten $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ und $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$. Beachten Sie, dass wir annehmen, dass die Alphabete der beiden Automaten identisch sind; d. h. Σ ist die Vereinigung der Alphabete von L und M, falls diese Alphabete verschieden sind. Die Produktkonstruktion ist sowohl für NEAs als auch für DEAs einsetzbar; um die Argumentation jedoch so einfach wie möglich zu halten, nehmen wir an, dass A_L und A_M DEAs sind.

Für $L \cap M$ konstruieren wir einen Automaten A, der sowohl A_L als auch A_M simuliert. Bei den Zuständen von A handelt es sich um Paare von Zuständen, wobei der erste von A_L und der zweite von A_M stammt. Zum Entwurf der Zustandsübergänge von A nehmen wir an, A befände sich im Zustand (p, q), wobei p der Zustand von A_L und q der Zustand von A_M ist. Wenn a das Eingabesymbol ist, können wir beobachten, wie A_L auf diese Eingabe reagiert. Angenommen, A_L geht in den Zustand s über. Wir können auch ablesen, wie A_M auf die Eingabe a reagiert. Nehmen wir an, A_M wechselt in den Zustand t. Der nächste Zustand von A ist demnach (s, t). Auf diese Weise hat A die Arbeitsweise von A_L und A_M simuliert. Dieser Gedankengang ist in Abbildung 4.3 skizziert.

Abbildung 4.3: Ein Automat, der zwei andere Automaten simuliert und ausschließlich dann akzeptiert, wenn die beiden anderen Automaten akzeptieren



Die verbleibenden Details sind einfach. Der Startzustand von A entspricht dem Paar (q_L, q_M) der Startzustände von A_L und A_M. Da der Automat genau dann akzeptieren soll, wenn beide Automaten akzeptieren, wählen wir als die akzeptierenden Zustände von A all jene Paare (p, q), derart dass p ein akzeptierender Zustand von A_L und q ein akzeptierender Zustand von A_M ist. Formal definieren wir:

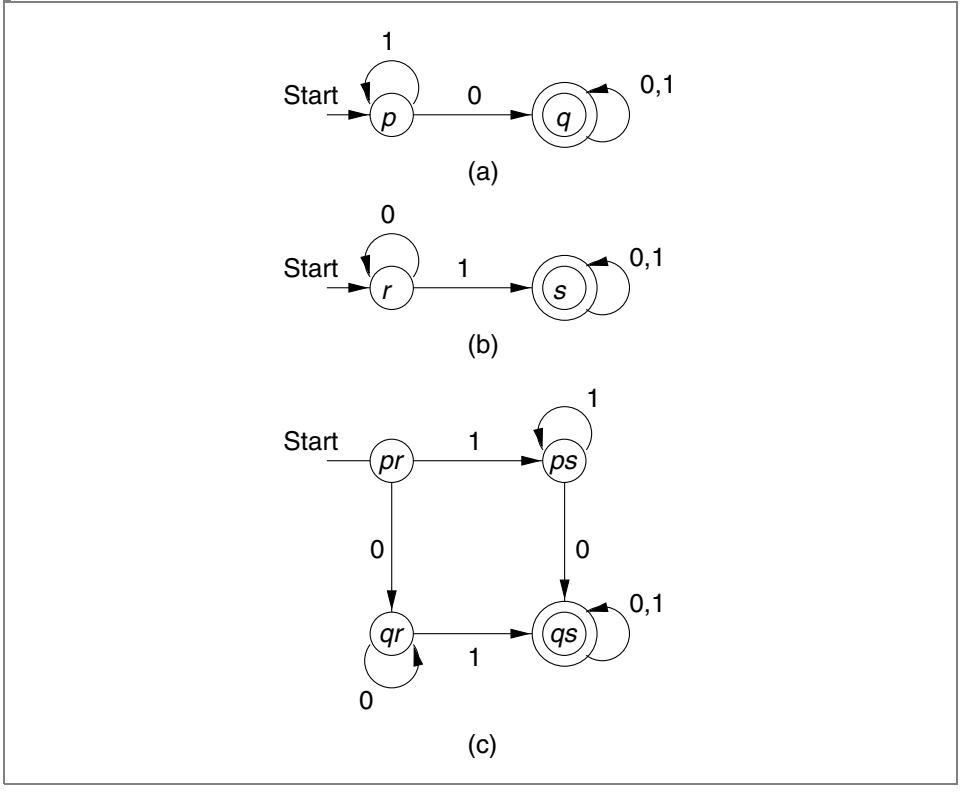
$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F_L \times F_M)$$

wobei $\delta((p, q),a) = (\delta_L(p, a), \delta_M(q, a))$.

Warum $L(A) = L(A_L) \cap L(A_M)$, wird deutlich, wenn man erstens sieht, dass eine einfache Induktion über $|w|$ zeigt, dass $\hat{\delta}((q_L,q_M), w) = (\hat{\delta}_L(q_L,w), \hat{\delta}_M(q_M,w))$. A akzeptiert w zweitens genau dann, wenn $\hat{\delta}(q_L,q_M), w)$ ein Paar akzeptierender Zustände repräsentiert. Das heißt, $\hat{\delta}_L(q_L,w)$ muss in F_L und $\hat{\delta}_M(q_M,w)$ muss in F_M enthalten sein. Anders ausgedrückt, w wird von A genau dann akzeptiert, wenn sowohl A_L als auch A_M w akzeptieren. Folglich akzeptiert A den Durchschnitt von L und M. ■

Beispiel 4.9 Abbildung 4.4 zeigt zwei DEAs. Der Automat in Abbildung 4.4 (a) akzeptiert alle Zeichenreihen, die eine Null enthalten, während der Automat in Abbildung 4.4 (b) alle Zeichenreihen akzeptiert, die eine Eins enthalten. Wir zeigen in Abbildung 4.4 (c) das Produkt dieser beiden Automaten. Die Zustände dieses Automaten sind mit den Zustandspaaren beschriftet, die aus den Zuständen der in (a) und (b) dargestellten Automaten bestehen.

Abbildung 4.4: Die Produktkonstruktion



Man kann mühelos argumentieren, dass dieser Automat den Durchschnitt der beiden Sprachen akzeptiert: jene Zeichenreihen, die sowohl eine Null als auch eine Eins enthalten. Der Zustand *pr* repräsentiert nur den Anfangszustand, in dem weder eine Null noch eine Eins eingegeben wurde. Der Zustand *qr* bedeutet, dass bislang nur Nullen gelesen wurden, während der Zustand *ps* die Situation wiedergibt, in der bislang nur Einsen gelesen wurden. Der akzeptierende Zustand *qs* repräsentiert die Situation, in der sowohl eine Eins als auch eine Null eingegeben wurde. ■

Abgeschlossenheit bezüglich der Differenzmenge

Es gibt eine vierte Operation, die häufig auf Mengen angewandt wird und mit den Booleschen Operationen verwandt ist: die Differenz. Die Differenz der Sprachen L und M wird durch den Ausdruck $L - M$ beschrieben und enthält die Menge der Zeichenreihen, die in L, nicht aber in M enthalten sind. Die regulären Sprachen sind auch bezüglich dieser Operation abgeschlossen. Der Beweis folgt unmittelbar aus den oben bewiesenen Sätzen.

Satz 4.10 Wenn L und M reguläre Sprachen sind, dann ist auch $L - M$ eine reguläre Sprache.

Beweis: Wir stellen fest, dass $L - M = L \cap \overline{M}$ ist. Nach Satz 4.5 ist $\overline{M}$ regulär, und nach Satz 4.8 ist L $\cap \overline{M}$ regulär. Infolgedessen ist $L - M$ regulär. ■

4.2.2 Spiegelung

Die Spiegelung einer Zeichenreihe $a_1a_2 \ldots a_n$ besteht aus den in der umgekehrten Reihenfolge angegebenen Symbolen der Zeichenreihe, d. h. $a_n \ldots a_2a_1$. Wir verwenden die Notation w^R für die Spiegelung der Zeichenreihe w. Demnach ist 0010^R gleich 0100 und $\varepsilon^R = \varepsilon$.

Bei der Spiegelung einer Sprache L, die durch L^R angegeben wird, handelt es sich um die Sprache, die aus allen gespiegelten Zeichenreihen der Sprache L besteht. Wenn beispielsweise $L = \{001, 10, 111\}$, dann ist $L^R = \{100, 01, 111\}$.

Die Spiegelung ist eine weitere Operation, die die Regularität von Sprachen nicht beeinflusst. Das heißt, wenn L eine reguläre Sprache ist, dann ist auch L^R eine reguläre Sprache. Dies lässt sich durch zwei einfache Beweise zeigen, von denen der eine auf Automaten und der andere auf regulären Ausdrücken basiert. Wir werden den auf Automaten basierenden Beweis informell beschreiben und es Ihnen überlassen, die Details zu ergänzen. Anschließend beweisen wir den Satz formal unter Verwendung regulärer Ausdrücke.

Aus einer gegebenen Sprache L, die die Sprache $L(A)$ eines endlichen Automaten ist, der möglicherweise nichtdeterministisch ist und über ε-Übergänge verfügt, können wir wie folgt einen Automaten für L^R konstruieren:

1. Kehre alle Pfeile im Übergangsdiagramm von A um.
2. Setze den Startzustand von A als einzigen akzeptierenden Zustand des neuen Automaten ein.
3. Erstelle einen neuen Startzustand p_0 mit Übergängen für ε zu allen akzeptierenden Zuständen von A.

Das Ergebnis ist ein Automat, der A »in umgekehrter Richtung« simuliert und daher eine Zeichenreihe w genau dann akzeptiert, wenn A w^R akzeptiert. Nun beweisen wir den Satz zur Spiegelung formal.

Satz 4.11 Wenn L eine reguläre Sprache ist, dann ist auch L^R eine reguläre Sprache.

Beweis: Angenommen, L werde durch den regulären Ausdruck E definiert. Der Beweis besteht aus einer strukturellen Induktion über den regulären Aufbau von E. Wir zeigen, dass es einen anderen regulären Ausdruck E^R gibt, derart dass $L(E^R) = (L(E))^R$, d. h. die Sprache von E^R ist die Spiegelung von E.

Induktionsbeginn: Wenn E für ein Symbol a gleich ε, $\varnothing$ oder **a** ist, dann ist E^R mit E identisch. Das heißt, wir wissen, dass $\{\varepsilon\}^R = \{\varepsilon\}$, $\{\varnothing\}^R = \{\varnothing\}$ und $\{a\}^R = \{a\}$.

Induktionsschritt: Abhängig von der Form von E sind drei Fälle zu unterscheiden:

1. $E = E_1 + E_2$. Dann gilt $E^R = E_1^R + E_2^R$. Begründet wird dies dadurch, dass man die Spiegelung der Vereinigung zweier Sprachen erhält, indem man die Spiegelungen der beiden Sprachen berechnet und diese vereinigt.

2. $E = E_1E_2$. Dann gilt $E^R = E_2^RE_1^R$. Beachten Sie, dass wir in der Spiegelung sowohl die beiden Sprachen in der umgekehrten Reihenfolge angeben als auch die Sprachen selbst. Wenn z. B. $L(E_1) = \{01, 111\}$ und $L(E_2) = \{00, 10\}$, dann ist $L(E_1E_2) = \{0100, 0110, 11100, 11110\}$. Die Spiegelung der vereinigten Sprache lautet:

   $$\{0010, 0110, 00111, 01111\}$$

   Wenn wir die Spiegelungen von $L(E_2)$ und $L(E_1)$ in dieser Reihenfolge verketten, erhalten wir

   $$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$

   welches die gleiche Sprache wie $(L(E_1\ E_2))^R$ ist. Im Allgemeinen gilt, wenn ein in $L(E)$ enthaltenes Wort w die Verkettung von w_1 aus $L(E_1)$ und w_2 aus $L(E_2)$ darstellt, dann gilt $w^R = w_2^Rw_1^R$.

3. $E = E_1^*$; dann gilt $E^R = (E_1^R)^*$. Die Begründung lautet, dass jede Zeichenreihe w aus $L(E)$ in der Form $w_1w_2 \ldots w_n$ dargestellt werden kann, wobei jedes w_i in $L(E)$ enthalten ist. Nun gilt:

   $$w^R = w_n^Rw_{n-1}^R \ldots w_1^R$$

   Jede Zeichenreihe w_i^R ist in $L(E^R)$ enthalten, daher ist w^R in $L(E_1^R)^*$ enthalten. Umgekehrt gilt, jede in $L((E_1^R)^*)$ enthaltene Zeichenreihe hat die Form $w_1w_2 \ldots w_n$, wobei jede Zeichenreihe w_i die Spiegelung einer in $L(E_1)$ enthaltenen Zeichenreihe ist. Die Spiegelung der Zeichenreihe $w_1w_2 \ldots w_n$, also $w_n^Rw_{n-1}^R \ldots w_1^R$, ist daher eine Zeichenreihe aus $L(E_1^*)$, also aus $L(E)$. Wir haben damit gezeigt, dass eine Zeichenreihe genau dann in $L(E)$ enthalten ist, wenn ihre Spiegelung in $L((E_1^R)^*)$ enthalten ist. ■

Beispiel 4.12 Sei L definiert durch den regulären Ausdruck $(\mathbf{0} + \mathbf{1})\mathbf{0}^*$. Dann ist L^R nach der Regel für die Verkettung die Sprache von $(\mathbf{0}^*)^R(\mathbf{0} + \mathbf{1})^R$. Wenn wir die Regeln für die Hüllenbildung und die Vereinigung auf die beiden Teile anwenden und dann

die Basisregel anwenden, die besagt, dass die Spiegelungen von **0** und **1** unverändert bleiben, stellen wir fest, dass L^R durch den regulären Ausdruck $\mathbf{0}^*(\mathbf{0} + \mathbf{1})$ beschrieben wird. ■

4.2.3 Homomorphismus

Ein Zeichenreihen-Homomorphismus ist eine Funktion von Zeichenreihen, die bewirkt, dass jedes Symbol durch eine bestimmte Zeichenreihe ersetzt wird.

Beispiel 4.13 Die Funktion h, die durch $h(0) = ab$ und $h(1) = \varepsilon$ definiert ist, ist ein Homomorphismus. Wenn ihr eine Zeichenreihe aus Nullen und Einsen übergeben wird, ersetzt sie alle Nullen durch die Zeichenreihe ab und alle Einsen durch die leere Zeichenreihe. Wenn h beispielsweise die Zeichenreihe 0011 übergeben wird, dann ist das Ergebnis $abab$. ■

Formal ausgedrückt, wenn h ein Homomorphismus über dem Alphabet Σ und $w = a_1a_2 \ldots a_n$ eine Zeichenreihe aus Symbolen des Alphabets Σ ist, dann ist $h(w) = h(a_1)h(a_2) \ldots h(a_n)$. Das heißt, wir wenden h auf jedes Symbol von w an und verketten die Ergebnisse der Reihenfolge nach. Wenn h z. B. der Homomorphismus aus dem Beispiel 4.13 und $w = 0011$ ist, dann ist wie in diesem Beispiel gefordert $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\varepsilon)(\varepsilon)$.

Wir können einen Homomorphismus zudem auf eine Sprache anwenden, indem wir ihn auf jede in der Sprache enthaltene Zeichenreihe anwenden. Das heißt, wenn L eine Sprache über dem Alphabet Σ und h ein Homomorphismus über Σ ist, dann ist $h(L) = \{h(w) \mid w \text{ ist in } L \text{ enthalten}\}$. Wenn beispielsweise L die Sprache des regulären Ausdrucks $\mathbf{10}^*\mathbf{1}$ ist, d. h. eine beliebige Anzahl von Nullen, die links und rechts von einer einzelnen Eins umgeben sind, dann ist für das o.a. h $h(L)$ die Sprache $(\mathbf{ab})^*$. Erklären lässt sich dies dadurch, dass die Funktion h aus Beispiel 4.13 die Einsen im Endeffekt löscht, da sie durch die leere Zeichenreihe ε ersetzt werden, und statt der Nullen ab einsetzt. Mithilfe dieses Verfahrens, in dem der Homomorphismus direkt auf einen regulären Ausdruck angewandt wird, kann bewiesen werden, dass reguläre Sprachen bezüglich Homomorphismen abgeschlossen sind.

Satz 4.14 Wenn L eine reguläre Sprache über dem Alphabet Σ und h ein Homomorphismus über Σ ist, dann ist auch $h(L)$ eine reguläre Sprache.

Beweis: Sei $L = L(R)$ für einen regulären Ausdruck R. Allgemein gilt, wenn E ein regulärer Ausdruck mit Symbolen aus Σ ist, dann ist $h(E)$ der Ausdruck, den wir erhalten, indem jedes Symbol a aus Σ in E durch $h(a)$ ersetzt wird. Wir behaupten, dass $h(R)$ die Sprache $h(L)$ definiert.

Der Beweis ist eine einfache strukturelle Induktion, die besagt, wann immer wir h auf einen Teilausdruck E von R anwenden, um $h(E)$ zu erhalten, ist die Sprache von $h(E)$ dieselbe Sprache, die wir durch die Anwendung von h auf $L(E)$ erhalten. Formal ausgedrückt, $L(h(E)) = h(L(E))$.

Induktionsbeginn: Wenn E gleich ε oder $\varnothing$ ist, dann ist $h(E)$ mit E identisch, da h weder die leere Zeichenreihe ε noch die leere Sprache $\varnothing$ verändert. Somit ist $L(h(E)) = L(E)$. Ist E allerdings $\varnothing$ oder ε, dann enthält $L(E)$ entweder keine Zeichenreihen oder eine Zeichenreihe ohne Symbole. Folglich ist in beiden Fällen $h(L(E)) = L(E)$. Wir schließen daraus, dass $L(h(E)) = L(E) = h(L(E))$.

Ansonsten gibt es nur noch den Basisfall, dass $E = \mathbf{a}$ für ein Symbol a aus Σ ist. In diesem Fall ist $L(E) = \{a\}$, daher ist $h(L(E)) = \{h(a)\}$. $h(E)$ ist der reguläre Ausdruck, der die Zeichenreihe $h(a)$ repräsentiert. Folglich ist $L(h(E))$ auch $\{h(a)\}$, und wir schließen daraus, dass $L(h(E)) = h(L(E))$.

Induktionsschritt: Es sind drei einfache Fälle zu unterscheiden. Wir werden nur den Fall der Vereinigung beweisen, in dem gilt $E = F + G$. Die Art und Weise, in der Homomorphismen auf reguläre Ausdrücke angewandt werden, stellt sicher, dass $h(E) = h(F + G) = h(F) + h(G)$. Wir wissen zudem, dass $L(E) = L(F) \cup L(G)$; gemäß der Definition des Operators »+« für reguläre Ausdrücke gilt also:

$$L(h(E)) = L(h(F) + h(G)) = L(h(F)) \cup L(h(G)) \tag{4.2}$$

Da h auf die Sprache angewandt wird, indem es auf jede in der Sprache enthaltene Zeichenreihe angewandt wird, folgt schließlich:

$$h(L(E)) = h(L(F) \cup L(G)) = h(L(F)) \cup h(L(G)) \tag{4.3}$$

Nun können wir nach der Induktionshypothese behaupten, dass $L(h(F)) = h(L(F))$ und $L(h(G)) = h(L(G))$. Folglich sind die letzten Ausdrücke in (4.2) und (4.3) äquivalent, und daher sind auch die jeweiligen ersten Ausdrücke äquivalent, d. h. $L(h(E)) = h(L(E))$.

Wir werden die Fälle, in denen der Ausdruck E eine Verkettung oder Hülle ist, nicht beweisen; das Verfahren ähnelt in beiden Fällen jedoch dem oben beschriebenen. Daraus folgt, dass $L(h(R))$ identisch ist mit $h(L(R))$; das heißt, wird der Homomorphismus h auf den regulären Ausdruck für die Sprache L angewandt, dann ergibt sich ein regulärer Ausdruck, der die Sprache $h(L)$ definiert. ■

4.2.4 Inverser Homomorphismus

Homomorphismen können ebenso »rückwärts« angewandt werden, und auch in diesem Fall bleibt die Regularität von Sprachen erhalten. Angenommen, h ist ein Homomorphismus zur Ersetzung von Zeichenreihen aus einem Alphabet Σ durch Zeichenreihen eines anderen Alphabets T.[3] (das mit dem ersten identisch sein kann) Sei L eine Sprache über dem Alphabet T; dann ist $h^{-1}(L)$ (gelesen: »h invers von L«) die Menge aller Zeichenreihen w aus Σ^*, derart dass $h(w)$ in L enthalten ist. In Abbildung 4.5 ist in Teil (a) dargestellt, wie sich ein Homomorphismus auf die Sprache L auswirkt, und in Teil (b), wie sich ein inverser Homomorphismus auswirkt.

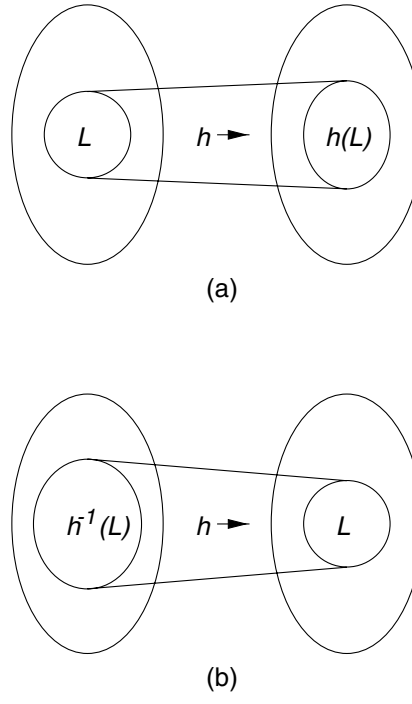
Beispiel 4.15 Sei L die Sprache des regulären Ausdrucks $(\mathbf{00} + \mathbf{1})^*$. Das heißt, L besteht aus allen Zeichenreihen aus Nullen und Einsen, in denen die Nullen paarweise auftreten. Folglich sind die Zeichenreihen 0010011 und 10000111 in L enthalten, 000 und 10100 dagegen nicht.

Sei h der Homomorphismus, der durch $h(a) = 01$ und $h(b) = 10$ definiert ist. Wir behaupten, dass $h^{-1}(L)$ die Sprache des regulären Ausdrucks $(\mathbf{ba})^*$ ist, d. h. alle Zeichenreihen sich wiederholender Paare ba umfasst. Wir werden beweisen, dass $h(w)$ genau dann in L enthalten ist, wenn w die Form $baba$ ... ba hat.

(Wenn-Teil) Angenommen, w besteht aus n Wiederholungen von ba, wobei $n \geq 0$. Beachten Sie, dass $h(ba) = 1001$; daher besteht $h(w)$ aus n Wiederholungen von 1001. Da

3. Dieses »T« ist als griechischer Großbuchstabe *Tau* zu betrachten, also der Buchstabe nach *Sigma*.

Abbildung 4.5: Wirkung eines Homomorphismus und eines inversen Homomorphismus



1001 zwei Einsen und ein Paar Nullen enthält, wissen wir, dass 1001 in L enthalten ist. Daher besteht auch jede Wiederholung von 1001 aus Einsen und 00-Paaren und ist somit in L enthalten. Folglich ist $h(w)$ in L enthalten.

(Nur-wenn-Teil) Wir müssen nun annehmen, dass $h(w)$ in L enthalten ist, und zeigen, dass w die Form *baba ... ba* hat. Es gibt vier Bedingungen, unter denen eine Zeichenreihe *nicht* diese Form hat, und wir werden zeigen, dass $h(w)$ nicht in L enthalten ist, wenn eine dieser Bedingungen erfüllt ist. Das heißt, wir beweisen die Umkehrung der Aussage, die es zu beweisen gilt.

1. Wenn die Zeichenreihe w mit a beginnt, dann beginnt $h(w)$ mit 01. Daher enthält die Zeichenreihe eine einzelne Null und ist nicht in L enthalten.
2. Wenn die Zeichenreihe w auf b endet, dann endet $h(w)$ auf 10 und enthält wieder eine einzelne Null.
3. Wenn die Zeichenreihe w zwei aufeinander folgende Symbole a enthält, dann enthält $h(w)$ die Teilzeichenreihe 0101. Auch hier enthält die Zeichenreihe w eine einzelne Null.
4. Ähnlich liegt der Fall, wenn die Zeichenreihe w zwei aufeinander folgende Symbole b enthält. $h(w)$ enthält dann die Teilzeichenreihe 1010 und damit eine einzelne Null.

Daraus folgt: Wenn einer dieser Fälle zutrifft, dann ist $h(w)$ nicht in L enthalten. Zudem gilt, dass w die Form *baba ... ba* hat, sofern keiner der Fälle (1) bis (4) zutrifft.

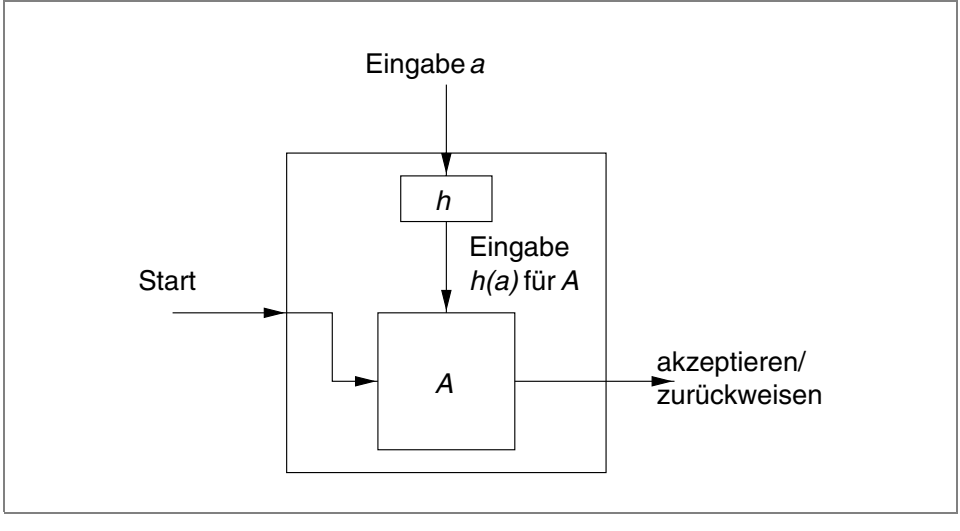
Um das zu sehen, wollen wir annehmen, dass keine der Bedingungen (1) bis (4) erfüllt ist. Dann wissen wir auf Grund von (1), dass w mit b beginnen muss, und aus (2) geht hervor, dass w auf a enden muss. Aus (3) und (4) können wir schließen, dass sich die Symbole a und b in w abwechseln müssen. Folglich ist die logische ODER-Verknüpfung von (1) bis (4) äquivalent mit der Aussage »w hat nicht die Form *baba ... ba*«. Wir haben bewiesen, dass die ODER-Verknüpfung von (1) bis (4) impliziert, dass $h(w)$ nicht in L enthalten ist. Diese Aussage stellt die Umkehrung der gewünschten Aussage dar: »Wenn $h(w)$ in L enthalten ist, dann hat w die Form *baba ... ba*.« ■

Wer werden als Nächstes beweisen, dass der inverse Homomorphismus einer regulären Sprache auch regulär ist, und dann zeigen, wie dieser Satz eingesetzt werden kann.

Satz 4.16 Wenn h ein Homomorphismus von Alphabet Σ nach Alphabet T und L eine reguläre Sprache über T ist, dann ist auch $h^{-1}(L)$ eine reguläre Sprache.

Beweis: Der Beweis beginnt mit einem DEA A für L. Wir konstruieren aus A und h unter Verwendung des in Abbildung 4.6 dargestellten Plans einen DEA für $h^{-1}(L)$. Dieser DEA verwendet die Zustände von A, übersetzt die Eingabesymbole jedoch gemäß h, bevor er den Übergang in den nächsten Zustand vollzieht.

Abbildung 4.6: Der DEA für $h^{-1}(L)$ wendet h auf seine Eingabe an und simuliert dann den DEA für L



Formal ausgedrückt, L sei $L(A)$, wobei DEA $A = (Q, \mathrm{T}, \delta, q_0, F)$. Definiere einen DEA

$$\mathrm{B} = (Q, \Sigma, \gamma, q_0, F)$$

dessen Übergangsfunktion γ nach der Regel $\gamma(q, a) = \hat{\delta}(q, h(a))$ gebildet wird. Das heißt, die Übergänge, die B auf die Eingabe a hin durchführt, sind ein Ergebnis der Folge von Übergängen, die A auf die Zeichenreihe $h(a)$ hin vollzieht. Bedenken Sie, dass es sich

bei $h(a)$ um ε, um ein Symbol oder um eine Zeichenreihe handeln kann. Die Übergangsfunktion $\hat{\delta}$ ist allerdings so definiert, dass sie alle diese Fälle behandeln kann.

Mit einer einfachen Induktion über $|w|$ lässt sich zeigen, dass $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$. Da A und B über dieselben akzeptierenden Zustände verfügen, akzeptiert B die Zeichenreihe w genau dann, wenn A $h(w)$ akzeptiert. Anders ausgedrückt, B akzeptiert genau die Zeichenreihen, die in $h^{-1}(L)$ enthalten sind. ■

Beispiel 4.17 In diesem Beispiel werden wir inverse Homomorphismen und einige andere Abgeschlossenheitseigenschaften regulärer Mengen verwenden, um eine Merkwürdigkeit in Bezug auf endliche Automaten zu beweisen. Angenommen, wir forderten, dass ein DEA jeden Zustand mindestens einmal besucht, wenn er seine Eingabe akzeptiert. Genauer gesagt, angenommen, $A = (Q, \Sigma, \delta, q_0, F)$ sei ein DEA, und uns interessiert die Sprache L aller Zeichenreihen w aus Σ^*, derart dass $\hat{\delta}(q_0, w)$ in F enthalten ist und es zudem für jeden Zustand q aus Q ein Präfix x_q von w gibt, derart dass $\hat{\delta}(q_0, x_q) = q$. Ist L regulär? Wir können dies beweisen, allerdings ist, die Konstruktion kompliziert.

Zuerst beginnen wir mit der Sprache M, die $L(A)$ ist, d. h. die Menge der Zeichenreihen, die der DEA A in der üblichen Weise akzeptiert, also ungeachtet dessen, welche Zustände er während der Verarbeitung der Eingabe besucht. Beachten Sie, dass $L \subseteq M$, da die Definition von L den in $L(A)$ enthaltenen Zeichenreihen eine zusätzliche Bedingung auferlegt. Unser Beweis, dass L regulär ist, beginnt mit einem inversen Homomorphismus, bei dem die Zustände von A in die Eingabesymbole eingesetzt werden. Genauer gesagt, wir wollen ein neues Alphabet T definieren, das aus Symbolen besteht, die man sich als Tripel $[paq]$ vorstellen kann, wobei gilt:

1. p und q sind in Q enthaltene Zustände.
2. a ist ein in Σ enthaltenes Symbol.
3. $\delta(p, a) = q$.

Wir können die in T enthaltenen Symbole als Repräsentationen der Zustandsübergänge von A betrachten. Es ist unbedingt zu beachten, dass die Notation $[paq]$ ein einziges Symbol darstellt und es sich nicht um die Verkettung von drei Symbolen handelt. Wir könnten auch einzelne Buchstaben zur Benennung verwenden, doch dann wäre ihre Beziehung zu p, q und a schwer zu beschreiben.

Nun definieren wir den Homomorphismus $h([paq]) = a$ für alle p, a und q. Das heißt, h entfernt die Zustandskomponenten aus jedem der in T enthaltenen Symbole, sodass nur das Symbol aus Σ übrig bleibt. Unser erster Schritt zum Beweis der Regularität von L besteht in der Konstruktion der Sprache $L_1 = h^{-1}(M)$. Da M regulär ist, ist nach Satz 4.16 auch L_1 regulär. Bei den in L_1 enthaltenen Zeichenreihen handelt es sich um die Zeichenreihen aus M, deren einzelnen Symbolen ein Zustandspaar angehängt wurde und die so einen Zustandsübergang repräsentieren.

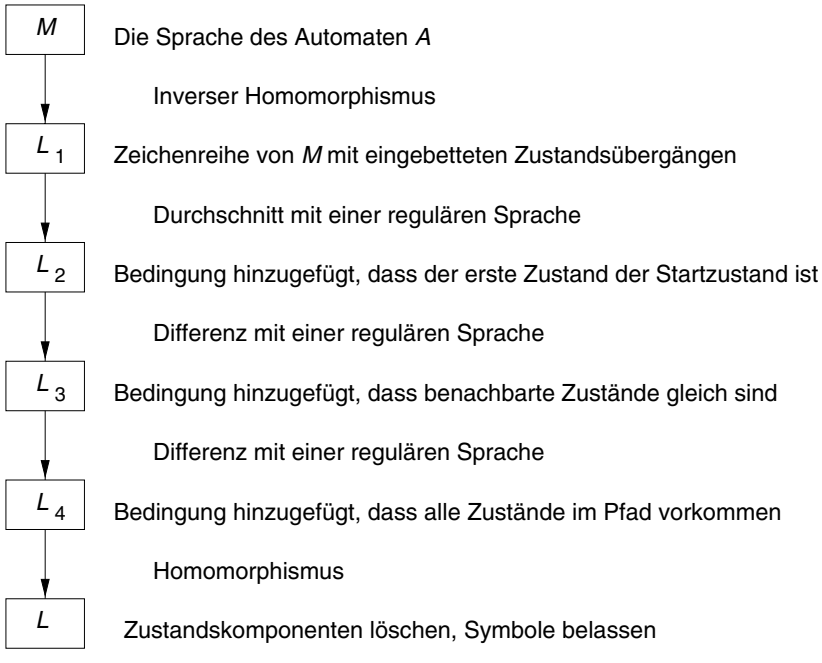
Betrachten Sie zur Veranschaulichung den Automaten mit zwei Zuständen in Abbildung 4.4 (a). Das Alphabet Σ ist $\{0, 1\}$, und das Alphabet T besteht aus den vier Symbolen $[p0q]$, $[q0q]$, $[p1p]$ und $[q1q]$. Es gibt beispielsweise für die Eingabe 0 einen Übergang von Zustand p nach q, und daher ist $[p0q]$ eines der in T enthaltenen Symbole. Die Zeichenreihe 101 wird vom Automaten akzeptiert. Die Anwendung von h^{-1} auf diese Zeichenreihe ergibt $2^3 = 8$ Zeichenreihen, wobei $[p1p][p0q][q1q]$ und $[q1q][q0q][p1p]$ zwei Beispiele für diese Zeichenreihen sind.

Wir werden nun L aus L_1 konstruieren, indem wir eine Reihe weiterer Operationen ausführen, die die Regularität erhalten. Unser erstes Ziel besteht darin, all jene Zeichenreihen aus L_1 zu eliminieren, die Zustandsfolgen nicht korrekt beschreiben. Das heißt, wir können uns vorstellen, dass ein Symbol wie [*paq*] Folgendes aussagt: Der Automat war im Zustand *p*, las die Eingabe *a* und ging daraufhin in den Zustand *q* über. Die Symbolfolge muss drei Bedingungen erfüllen, wenn sie eine akzeptierende Berechnung von *A* repräsentieren soll:

1. Das erste Symbol muss als ersten Zustand q_0 enthalten, den Startzustand von *A*.
2. Jeder Übergang muss da beginnen, wo der letzte aufgehört hat. Das heißt, in einem Symbol muss der erste Zustand gleich dem zweiten Zustand des vorherigen Symbols sein.
3. Der zweite Zustand des letzten Symbols muss in *F* enthalten sein. Diese Bedingung wird tatsächlich schon garantiert, wenn wir (1) und (2) erzwingen, da wir wissen, dass jede in L_1 enthaltene Zeichenreihe aus einer von *A* akzeptierten Zeichenreihe resultiert.

Abbildung 4.7 zeigt den Plan zur Konstruktion von *L*.

Abbildung 4.7: Wie die Sprache *L* mithilfe von Operationen, die die Regularität von Sprachen erhalten, aus der Sprache *M* konstruiert wird



Wir erzwingen (1), indem wir den Durchschnitt von L_1 und der Menge von Zeichenreihen bilden, die für ein Symbol a und einen Zustand q mit einem Symbol der Form $[q_0aq]$ beginnen. Das heißt, sei E_1 der Ausdruck $[q_0a_1q_1] + [q_0a_2q_2] + ...$, wobei a_iq_i alle Paare in Σ x Q umfasst, derart dass $\delta(q_0, a_i) = q_i$, dann sei $L_2 = L_1 \cap L(E_1\,T^*)$. Da $E_1\,T^*$ ein regulärer Ausdruck ist, der alle Zeichenreihen aus T^* beschreibt, die mit dem Startzustand beginnen, enthält L_2 alle Zeichenreihen, die durch die Anwendung von h^{-1} auf die Sprache M gebildet werden und den Startzustand als erste Komponente des ersten Symbols enthalten; d. h. L_2 erfüllt (1).

Um Bedingung (2) zu erzwingen, ist es am einfachsten, alle Zeichenreihen, die diese Bedingung nicht erfüllen, von L_2 (mit dem Operator für die Mengendifferenz) abzuziehen. Sei E_2 der reguläre Ausdruck, der aus der Summe (Vereinigung) der Verkettungen aller Symbolpaare besteht, die nicht zusammenpassen, d. h. Paare der Form $[paq][rbs]$, wobei $q \neq r$; dann ist $T^*E_2\,T^*$ ein regulärer Ausdruck, der alle Zeichenreihen beschreibt, die die Bedingung (2) nicht erfüllen.

Wir können nun definieren $L_3 = L_2 - L(T^*E_2\,T^*)$. Die Zeichenreihen in L_3 erfüllen die Bedingung (1), weil die Zeichenreihen in L_2 mit dem Startsymbol beginnen müssen. Sie erfüllen Bedingung (2), weil die Subtraktion von $L(T^*E_2\,T^*)$ jede Zeichenreihe entfernt, die diese Bedingung verletzt. Schließlich erfüllen sie Bedingung (3), dass der letzte Zustand ein akzeptierender sein muss, weil wir lediglich mit Zeichenreihen aus M begonnen haben, die alle zu einem akzeptierenden Zustand von A führen. Infolgedessen besteht L_3 aus den Zeichenreihen aus M, in denen die Zustände der akzeptierenden Berechnung als Bestandteile in die einzelnen Symbole eingebettet sind. Beachten Sie, dass die Sprache L_3 regulär ist, weil auf die reguläre Menge M nur eine Folge von Operationen (inverser Homomorphismus, Durchschnitt und Mengendifferenz) angewandt wird, die die Regularität erhalten.

Unser Ziel bestand ursprünglich darin, nur jene Zeichenreihen aus M zu akzeptieren, während deren akzeptierender Berechnung jeder Zustand besucht wurde. Wir können diese Bedingung durch weitere Anwendungen des Mengendifferenzoperators erzwingen. Das heißt, für jeden Zustand q sei E_q der reguläre Ausdruck, der die Summe aller in T enthaltenen Symbole beschreibt, derart dass q weder an der ersten noch an der zweiten Position vorkommt. Wenn wir $L(E_q^*)$ von L_3 subtrahieren, dann erhalten wir die Zeichenreihen, die eine akzeptierende Berechnung von A repräsentieren und die mindestens einmal zum Zustand q führen. Wenn wir von L_3 alle Sprachen $L(E_q^*)$ für q aus Q subtrahieren, dann erhalten wir die akzeptierenden Berechnungen von A, die alle Zustände berühren. Wir nennen diese Sprache L_4. Nach Satz 4.10 wissen wir, dass auch L_4 eine reguläre Sprache ist.

Der letzte Schritt besteht darin, durch Beseitigung der Zustandskomponenten L aus L_4 zu konstruieren. Das heißt, $L = h(L_4)$. L ist die Menge der in Σ^* enthaltenen Zeichenreihen, die von A akzeptiert werden und während ihrer Berechnung mindestens einmal zu jedem Zustand von A führen. Da reguläre Sprachen bezüglich Homomorphismen abgeschlossen sind, wissen wir, dass L regulär ist. ■

4.2.5 Übungen zum Abschnitt 4.2

Übung 4.2.1 Angenommen, h ist ein Homomorphismus vom Alphabet {0, 1, 2} zum Alphabet {a, b} und wie folgt definiert: $h(0) = a$; $h(1) = ab$ und $h(2) = ba$.

* a) Was ergibt $h(0120)$?

b) Was ergibt $h(21120)$?

* c) Wenn L die Sprache $L(\mathbf{01}^*\mathbf{2})$ ist, was ist dann $h(L)$?

d) Wenn L die Sprache $L(\mathbf{0} + \mathbf{12})$ ist, was ist dann $h(L)$?

* e) Angenommen, L ist die Sprache $\{ababa\}$, d. h. die Sprache, die lediglich aus der einen Zeichenreihe *ababa* besteht. Was ist dann $h^{-1}(L)$?

! f) Wenn L die Sprache $L(\mathbf{a}(\mathbf{ba})^*)$ ist, was ist dann $h^{-1}(L)$?

*! **Übung 4.2.2** Wenn L eine Sprache und a ein Symbol ist, dann ist L/a, der Quotient von L und a, die Menge der Zeichenreihen w, derart dass wa in L enthalten ist. Wenn beispielsweise $L = \{a, aab, baa\}$, dann ist $L/a = \{\varepsilon, ba\}$. Beweisen Sie, dass L/a regulär ist, wenn L regulär ist. *Hinweis*: Beginnen Sie mit einem DEA für L und betrachten Sie die Menge der akzeptierenden Zustände.

! **Übung 4.2.3** Wenn L eine Sprache und a ein Symbol ist, dann ist $a\backslash L$ die Menge der Zeichenreihen w, derart dass aw in L enthalten ist. Wenn beispielsweise $L = \{a, aab, baa\}$, dann ist $a\backslash L = \{\varepsilon, ab\}$. Beweisen Sie, dass $a\backslash L$ regulär ist, wenn L regulär ist. *Hinweis*: Denken Sie daran, dass reguläre Sprachen bezüglich der Spiegelung und der Quotientenoperation aus 4.2.2 abgeschlossen sind.

! **Übung 4.2.4** Welche der folgenden Identitäten sind wahr?

a) $(L/a)a = L$ (die linke Seite repräsentiert die Verkettung der Sprachen L/a und $\{a\}$)

b) $a(a\backslash L) = L$ (hier ist wieder eine Verkettung mit {a} gemeint)

c) $(La)/a = L$

d) $a\backslash(aL) = L$

Übung 4.2.5 Die in Übung 4.2.3 beschriebene Operation wird gelegentlich auch als »Ableitung« betrachtet und $a\backslash L$ wird wie folgt dargestellt: $\frac{dL}{da}$. Diese Ableitungen werden auf reguläre Ausdrücke in ähnlicher Weise angewandt, wie normale Ableitungen auf arithmetische Ausdrücke. Wenn R ein regulärer Ausdruck ist, dann soll $\frac{dR}{da}$ dasselbe bedeuten wie $\frac{dL}{da}$, wenn $L = L(R)$.

a) Zeigen Sie, dass $\frac{d(R+S)}{da} = \frac{dR}{da} + \frac{dS}{da}$.

*! b) Formulieren Sie die Regel für die Ableitung von RS. *Hinweis*: Sie müssen zwei Fälle unterscheiden: Wenn $L(R)$ die leere Zeichenfolge ε enthält und wenn nicht. Diese Regel entspricht nicht ganz der »Produktregel« für gewöhnliche Ableitungen, ähnelt ihr jedoch.

! c) Formulieren Sie die Regel für die Ableitung einer Hülle, d. h. für $\frac{d(R^*)}{da}$.

d) Verwenden Sie die Regeln aus (a) bis (c), um die Ableitung für den regulären Ausdruck $(\mathbf{0} + \mathbf{1})^*\mathbf{011}$ in Bezug auf 0 und 1 zu finden.

* e) Charakterisieren Sie diejenigen Sprachen L, für die gilt $\frac{dL}{d0} = \varnothing$.

*! f) Charakterisieren Sie diejenigen Sprachen L, für die gilt $\frac{dL}{d0} = L$.

! **Übung 4.2.6** Zeigen Sie, dass die regulären Sprachen bezüglich der folgenden Operationen abgeschlossen sind:

a) $min(L) = \{w \mid w$ ist in L enthalten, aber kein eigentliches Präfix von w ist in L enthalten$\}$.

b) $max(L) = \{w \mid w$ ist in L enthalten, und für kein anderes x als ε ist wx in L enthalten$\}$.

c) $init(L) = \{w \mid$ für ein x ist wx in L enthalten$\}$.

Hinweis: Wie bei Übung 4.2.2 ist es am einfachsten mit einem DEA für L zu beginnen und eine Konstruktion anzugeben, um die gewünschte Sprache zu erhalten.

! **Übung 4.2.7** Wenn $w = a_1a_2 \ldots a_n$ und $x = b_1b_2 \ldots b_n$ Zeichenreihen derselben Länge sind, definieren Sie $alt(w, x)$ als die Zeichenreihe, in der die Symbole von w und x abwechseln, wobei mit w begonnen werden soll, d. h. $a_1b_1a_2b_2 \ldots a_nb_n$. Wenn L und M Sprachen sind, definieren Sie $alt(L, M)$ als die Menge der Zeichenreihen der Form $alt(w, x)$, wobei w eine beliebige Zeichenreihe aus L und x eine beliebige Zeichenreihe aus M derselben Länge ist. Beweisen Sie, dass $alt(L, M)$ regulär ist, wenn L und M regulär sind.

*!! **Übung 4.2.8** Sei L eine Sprache. Definieren Sie $half(L)$ als die Menge der ersten Hälften der in L enthaltenen Zeichenreihen, d. h. $\{w \mid$ für ein x mit $|x| = |w|$ ist wx in $L\}$. Wenn z. B. $L = \{\varepsilon, 0010, 011, 010110\}$, dann ist $half(L) = \{\varepsilon, 00, 010\}$. Beachten Sie, dass Zeichenreihen mit einer ungeraden Anzahl von Zeichen nicht zu $half(L)$ beitragen. Beweisen Sie, dass $half(L)$ eine reguläre Sprache ist, wenn L regulär ist.

!! **Übung 4.2.9** Wir können Übung 4.2.8 für eine Reihe von Funktionen verallgemeinern, die die Größe des Anteils an den Zeichenreihen festlegen. Wenn f eine Funktion ganzer Zahlen ist, definieren Sie $f(L)$ wie folgt: $\{w \mid$ für ein x mit $|x| = f(|w|)$ ist wx in $L\}$. Die Operation *half* entspricht z. B. der Identitätsfunktion $f(n) = n$, da die Definition von $half(L)$ die Bedingung $|x| = |w|$ enthält. Zeigen Sie, dass $f(L)$ eine reguläre Sprache ist, wenn L eine reguläre Sprache ist und wenn f für eine der folgenden Funktionen steht:

a) $f(n) = 2n$ (d. h. nimm das erste Drittel der Zeichenreihen)

b) $f(n) = n^2$ (d. h. die Anzahl der zu übernehmenden Zeichen ist gleich der Quadratwurzel der Anzahl der Zeichen, die nicht übernommen werden)

c) $f(n) = 2^n$ (d. h. die Anzahl der zu übernehmenden Zeichen ist gleich dem dualen Logarithmus der Anzahl der Zeichen, die nicht übernommen werden)

!! **Übung 4.2.10** Angenommen, L ist irgendeine Sprache, die nicht unbedingt regulär zu sein braucht und deren Alphabet gleich $\{0\}$ ist; d. h. die Zeichenreihen von L bestehen ausschließlich aus Nullen. Beweisen Sie, dass L^* regulär ist. *Hinweis*: Dieser Satz mag auf den ersten Blick absurd scheinen. Ein Beispiel kann Ihnen vielleicht verdeutlichen, warum er wahr ist. Betrachten Sie die Sprache $L = \{0^i \mid i$ ist eine Primzahl$\}$, von der wir aus Beispiel 4.3 wissen, dass sie nicht regulär ist. Die Zeichenreihen 00 und 000 sind in L enthalten, da 2 und 3 Primzahlen sind. Folglich können wir für $j \geq 2$ zeigen, dass 0^j in L^* enthalten ist. Wenn j gerade ist, dann verwenden wir $j/2$-mal 00, und wenn j ungerade ist, dann verwenden wir einmal 000 und $(j - 3)/2$-mal 00. Daraus ergibt sich $L^* = \mathbf{000}^* \cup \{\varepsilon\}$.

!! **Übung 4.2.11** Zeigen Sie, dass die regulären Sprachen bezüglich der folgenden Operation abgeschlossen sind: $cycle(L) = \{w \mid$ wir können w in der Form $w = xy$ schreiben, sodass yx in L enthalten ist$\}$. Wenn z. B. $L = \{01, 011\}$, dann ist $cycle(L) = \{01, 10, 011, 110, 101\}$. *Hinweis*: Beginnen Sie mit einem DEA für L und konstruieren Sie einen ε-NEA für $cycle(L)$.

!! **Übung 4.2.12** Sei $w_1 = a_0a_0a_1$ und $w_i = w_{i-1}w_{i-1}a_i$ für alle $i > 1$. Beispielsweise ist $w_3 = a_0a_0a_1a_0a_0a_1a_2a_0a_0a_1a_0a_0a_1a_2a_3$. Der kürzeste reguläre Ausdruck für die Sprache $L_n = \{w_n\}$, d. h. die Sprache, die aus der Zeichenreihe w_n besteht, ist die Zeichenreihe w_n, und die Länge dieses Ausdrucks ist $2^{n+1} - 1$. Wenn wir jedoch den Durchschnittsoperator zulassen, dann können wir einen Ausdruck für L_n formulieren, dessen Länge $O(n^2)$ ist. Finden Sie einen solchen Ausdruck. *Hinweis*: Finden Sie n Sprachen, die jeweils durch einen regulären Ausdruck der Länge $O(n)$ beschrieben werden und deren Schnittmenge L_n ist.

! **Übung 4.2.13** Die Abgeschlossenheitseigenschaften erleichtern Beweise, dass bestimmte Sprachen nicht regulär sind. Beginnen Sie mit der Tatsache, dass die Sprache

$$L_{0n1n} = \{0^n1^n \mid n \geq 0\}$$

keine reguläre Menge ist. Beweisen Sie, dass die folgenden Sprachen nicht regulär sind, indem Sie sie unter Verwendung von Operationen, die bekanntermaßen die Regularität erhalten, in L_{0n1n} transformieren:

* a) $\{0^i1^j \mid i \neq j\}$

b) $\{0^n1^m2^{n-m} \mid n \geq m \geq 0\}$

Übung 4.2.14 In Satz 4.8 beschrieben wir die »Produktkonstruktion«, mit der aus zwei DEAs ein DEA konstruiert wurde, dessen Sprache in der Schnittmenge der Sprachen der ersten beiden DEAs enthalten ist.

a) Zeigen Sie, wie die Produktkonstruktion für NEAs (ohne ε-Übergänge) durchgeführt wird.

! b) Zeigen Sie, wie die Produktkonstruktion für ε-NEAs durchgeführt wird.

* c) Zeigen Sie, in welcher Weise die Produktkonstruktion modifiziert werden muss, damit der resultierende DEA die Differenz der Sprachen der beiden gegebenen DEAs akzeptiert.

d) Zeigen Sie, in welcher Weise die Produktkonstruktion modifiziert werden muss, damit der resultierende DEA die Vereinigung der Sprachen der beiden gegebenen DEAs akzeptiert.

Übung 4.2.15 Im Beweis von Satz 4.8 behaupteten wir, dass durch Induktion über die Länge von w bewiesen werden könnte, dass

$$\hat{\delta}((q_L,q_M), w) = (\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w))$$

Führen Sie diesen induktiven Beweis.

Übung 4.2.16 Vervollständigen Sie den Beweis von Satz 4.14, indem Sie die Fälle behandeln, in denen E eine Verkettung von zwei Teilausdrücken bzw. die Hülle eines Ausdrucks darstellt.

Übung 4.2.17 In Satz 4.16 haben wir einen Induktionsbeweis über die Länge von w ausgelassen, nämlich dass $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$. Beweisen Sie diese Aussage.

4.3 Entscheidbarkeit regulärer Sprachen

In diesem Abschnitt betrachten wir, wie man wichtige Fragen zu regulären Sprachen beantworten kann. Zuerst müssen wir darauf eingehen, was es bedeutet, eine Frage zu einer Sprache zu stellen. Sprachen sind normalerweise unendlich, und man kann also im Allgemeinen die Zeichenreihen einer Sprache nicht jemandem zeigen und dann eine Frage stellen, zu deren Beantwortung die unendliche Menge von Zeichenreihen untersucht werden müsste. Stattdessen präsentieren wir eine Sprache, indem wir eine der endlichen Repräsentationen angeben: einen DEA, NEA, ε-NEA oder regulären Ausdruck.

Natürlich wird die auf diese Weise beschriebene Sprache regulär sein, und es gibt tatsächlich gar keine Möglichkeit, eine völlig beliebige Sprache endlich zu repräsentieren. In späteren Kapiteln werden wir endliche Verfahren zur Darstellung von weiteren Sprachen neben den regulären kennen lernen, sodass wir Fragen zu Sprachen im Hinblick auf diese allgemeineren Klassen betrachten können. Für viele Fragen gibt es allerdings nur für die Klasse der regulären Sprachen Algorithmen. Diese Fragen werden »unentscheidbar« (es gibt keinen Algorithmus zu deren Beantwortung), wenn sie zu »ausdrucksstärkeren« Notationen (d. h. Notationen, mit denen umfangreichere Mengen von Sprachen beschrieben werden können) gestellt werden als zu den Repräsentationen, die wir für reguläre Sprachen entwickelt haben.

Wir beginnen unser Studium der Algorithmen für Fragen zu regulären Sprachen, indem wir Möglichkeiten betrachten, wie die Repräsentation einer Sprache in eine andere Repräsentation derselben Sprache umgewandelt werden kann. Insbesondere werden wir die Zeitkomplexität der Algorithmen betrachten, die diese Umwandlungen durchführen. Anschließend erörtern wir einige grundlegende Fragen zu Sprachen:

1. Ist die beschriebene Sprache leer?
2. Ist eine bestimmte Zeichenreihe w in der beschriebenen Sprache?
3. Beschreiben zwei Repräsentationen einer Sprache tatsächlich dieselbe Sprache? Diese Frage wird häufig als »Äquivalenz« von Sprachen bezeichnet.

4.3.1 Wechsel zwischen Repräsentationen

Wir wissen, dass wir jede der vier Repräsentationen von regulären Sprachen in jede der drei anderen Repräsentationen umwandeln können. Abbildung 3.1 hat die Pfade gezeigt, die von einer Repräsentation zu den anderen führen. Obwohl es Algorithmen für die verschiedenen Umwandlungen gibt, sind wir nicht nur an der Möglichkeit einer Umwandlung, sondern auch an dem damit verbundenen Zeitaufwand interessiert. Insbesondere ist es wichtig, zwischen Algorithmen, die einen exponentiellen Zeitaufwand (als Funktion der Größe ihrer Eingabe) erfordern und daher nur für relativ kleine Beispiele verwendbar sind, und solchen, die einen bezüglich der Größe ihrer

Eingabe linearen, quadratischen oder in geringem Umfang polynomiellen Zeitaufwand erfordern. Letztere Algorithmen sind in dem Sinn »realistisch«, als wir erwarten, dass sie auch für umfangreiche Beispiele ausführbar sind. Wir werden die Zeitkomplexität aller hier erörterten Umwandlungen betrachten.

NEAs in DEAs umwandeln

Wenn wir mit einem NEA oder einem ε-NEA beginnen und ihn in einen DEA umwandeln, dann kann der Zeitaufwand in der Anzahl der Zustände des NEA exponentiell sein. Die Berechnung der ε-Hülle von n Zuständen nimmt $O(n^3)$ Zeit in Anspruch. Wir müssen von jedem der n Zustände allen mit ε beschrifteten Pfeilen nachgehen. Wenn es n Zustände gibt, dann kann es nicht mehr als n^2 Pfeile geben. Überlegte Buchführung und gut entworfene Datenstrukturen können sicherstellen, dass die Erforschung von jedem Zustand aus in $O(n^2)$ Zeit erfolgt. Mit einem transitiven Hüllenbildungsalgorithmus wie dem Warshall-Algorithmus kann die gesamte ε-Hülle sogar in einem Durchgang berechnet werden.[4]

Nachdem die ε-Hülle berechnet wurde, können wir mithilfe der Teilmengenkonstruktion den äquivalenten DEA berechnen. Hierbei ergeben die Zustände des DEA, deren Anzahl bis zu 2^n betragen kann, die dominanten Kosten. Wir können unter Zuhilfenahme der ε-Hüllendaten und der Übergangstabelle des NEA in $O(n3)$ Zeit für jeden Zustand die Übergänge für alle Eingabesymbole berechnen. Angenommen, wir möchten für den DEA $\delta(\{q_1, q_2, ..., q_k\}, a)$ berechnen. Von jedem Zustand q_1 aus können auf mit ε beschrifteten Pfaden n Zustände erreichbar sein, und jeder dieser Zustände kann bis zu n Pfeile mit der Beschriftung a besitzen. Indem wir ein mit Zuständen indiziertes Array anlegen, können wir die Vereinigung von bis zu n Mengen von bis zu n Zuständen in einem zu n^2 proportionalen Zeitraum berechnen.

Auf diese Weise können wir für jeden Zustand q_i die Menge der Zustände berechnen, die auf einem Pfad mit der Beschriftung a (möglicherweise einschließlich Pfeilen mit der Beschriftung ε) erreicht werden können. Da $k \leq n$, gibt es höchstens n Zustände zu behandeln. Wir berechnen die erreichbaren Zustände für jeden Zustand in $O(n^2)$ Zeit. Folglich beträgt der gesamte Zeitraum, der zur Berechnung von erreichbaren Zuständen aufgewendet werden muss, $O(n^3)$. Die Vereinigung der Mengen von erreichbaren Zuständen erfordert nur $O(n^2)$ zusätzliche Zeit, und wir schließen daraus, dass die Berechnung eines DEA-Übergangs $O(n^3)$ Zeit erfordert.

Beachten Sie, dass die Anzahl der Eingabesymbole konstant ist und nicht von n abhängt. Folglich berücksichtigen wir in dieser und anderen Schätzungen der Ausführungszeit die Anzahl der Eingabesymbole nicht als Faktor. Die Größe des Eingabealphabets beeinflusst den konstanten Faktor, der in der »O«-Notation verborgen ist, aber sonst nichts.

Unsere Schlussfolgerung ist, dass die Ausführungszeit einer NEA-in-DEA-Umwandlung (einschließlich NEAs mit ε-Übergängen) $O(n^3 2^n)$ beträgt. Nun ist es in der Praxis häufig so, dass eine viel geringere Anzahl von Zuständen als 2^n erzeugt wird, oft sogar nur n Zustände. Wir können also den oberen Grenzwert für die Ausführungszeit mit $O(n^3 s)$ angeben, wobei s die Anzahl der Zustände angibt, die der DEA tatsächlich besitzt.

4. Transitive Hüllenbildungsalgorithmen werden in A. V. Aho, J. E. Hopcroft und J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1984, näher behandelt.

DEAs in NEAs umwandeln

Diese Umwandlung ist einfach und erfordert $O(n)$ Zeit für einen DEA mit n Zuständen. Wir müssen lediglich die Übergangstabelle des DEA abändern, indem wir die Zustände in geschweifte Klammern setzen, um sie als Mengen zu kennzeichnen, und eine Spalte für ε hinzufügen, wenn es sich um einen ε-NEA handelt. Da wir die Anzahl der Eingabesymbole (d. h. die Breite der Übergangstabelle) als eine Konstante behandeln, beansprucht das Kopieren und Bearbeiten der Tabelle $O(n)$ Zeit.

Automaten in reguläre Ausdrücke umwandeln

Wenn wir die Konstruktion aus Abschnitt 3.2.1 betrachten, stellen wir fest, dass wir in jeder von n Runden (wobei n die Anzahl der Zustände des DEA ist) die Größe der konstruierten regulären Ausdrücke vervierfachen können, da jeder aus vier Ausdrücken der vorherigen Runde erstellt wird. Folglich kann das einfache Niederschreiben von n^3 Ausdrücken $O(n^3 4^n)$ Zeit beanspruchen. Die verbesserte Konstruktion aus Abschnitt 3.2.2 reduziert den konstanten Faktor, aber wirkt sich nicht auf den im schlechtesten Fall exponentiellen Zeitaufwand aus.

Die gleiche Konstruktion wird in der gleichen Ausführungszeit berechnet wenn es sich bei der Eingabe um einen NEA oder gar einen ε-NEA handelt, obwohl wir diese Aussagen nicht bewiesen haben. Es ist jedoch wichtig, diese Konstruktion direkt auf den NEA anzuwenden. Wenn wir nämlich einen NEA zuerst in einen DEA und den DEA dann in einen regulären Ausdruck umwandeln, könnte der Zeitaufwand $O(n^3 4^{n^3 2^n})$ betragen und damit doppelt exponentiell sein.

Reguläre Ausdrücke in Automaten umwandeln

Die Umwandlung eines regulären Ausdrucks in einen ε-NEA erfordert einen linearen Zeitaufwand. Wir müssen dazu die Ausdrücke mithilfe einer Technik, die zur Verarbeitung eines regulären Ausdrucks der Länge n nur $O(n)$ Zeit erfordert, effizient strukturell gliedern[5]. Das Ergebnis ist ein Baum mit einem Knoten für jedes einzelne Zeichen des regulären Ausdrucks (wobei Klammern nicht in dem Baum aufscheinen müssen; sie dienen lediglich zur Gliederung des Ausdrucks).

Sobald wir einen Baum für den regulären Ausdruck besitzen, können wir den Baum von unten nach oben verarbeiten und den ε-NEA für jeden Knoten erstellen. Die Konstruktionsregeln für die Umwandlung eines regulären Ausdrucks, die in Abschnitt 3.2.3 dargestellt wurden, fügen für jeden Knoten des Baums nie mehr als zwei Zustände und vier Pfeile hinzu. Folglich hat der resultierende ε-NEA sowohl $O(n)$ Zustände als auch $O(n)$ Pfeile. Zudem ist der zur Erstellung dieser Elemente erforderliche Arbeitsaufwand an jedem Knoten des Baums konstant, sofern die Funktion, die jeden Teilbaum bearbeitet, Zeiger auf den Startzustand und die akzeptierenden Zustände des Automaten zurückgibt.

Wir schließen daraus, dass die Konstruktion eines ε-NEA aus einem regulären Ausdruck einen Zeitaufwand erfordert, der linear in der Größe des Ausdrucks ist. Wir können die ε-Übergänge aus einem ε-NEA mit n Zuständen eliminieren, um in $O(n^2)$ Zeit einen gewöhnlichen NEA zu erstellen, ohne die Anzahl der Zustände zu erhöhen. Es kann allerdings einen exponentiellen Zeitaufwand erfordern, um daraus einen DEA zu konstruieren.

---

5. Parsing-Methoden, die diese Aufgabe in $O(n)$ Zeit ausführen können, werden in Y. V. Aho, R. Sethi und J.D. Ullman, *Compiler Design: Principles, Tools, and Techniques*, Addison-Wesley, 1986, behandelt.

4.3.2 Prüfen, ob eine reguläre Sprache leer ist

Auf den ersten Blick scheint die Beantwortung der Frage, ob eine reguläre Sprache leer ist, offensichtlich: ∅ ist leer, und alle anderen Sprachen sind es nicht. Wie wir am Beginn von Abschnitt 4.3 erörtert haben, wird das Problem allerdings nicht durch eine explizite Auflistung der in L enthaltenen Zeichenreihen gelöst. Stattdessen haben wir eine Repräsentation von L und müssen entscheiden, ob diese Repräsentation die Sprache ∅ beschreibt.

Handelt es sich bei der Repräsentation um einen endlichen Automaten, dann stellt sich die Frage, ob die Sprache leer ist, in der Form, ob es irgendeinen Pfad vom Startzustand zu einem akzeptierenden Zustand gibt. Die Entscheidung, ob wir vom Startzustand aus einen akzeptierenden Zustand erreichen können, ist ein einfaches Beispiel für die grafische Darstellung der Erreichbarkeit, im Prinzip ähnlich der Berechnung der ε-Hülle, die wir in Abschnitt 2.5.3 erörtert haben. Der Algorithmus lässt sich durch dieses rekursive Verfahren zusammenfassen.

INDUKTIONSBEGINN: Der Startzustand ist vom Startzustand aus mit Sicherheit erreichbar.

INDUKTIONSSCHRITT: Wenn Zustand q vom Startzustand aus erreichbar ist und es einen Pfeil von q nach p gibt, der mit einem Eingabesymbol oder ε (wenn es sich um einen ε-NEA handelt) beschriftet ist, dann ist p erreichbar.

Auf diese Weise können wir die Menge der erreichbaren Zustände berechnen. Wenn darunter ein akzeptierender Zustand ist, beantworten wir die Frage mit »Nein« (die Sprache des Automaten ist *nicht* leer), andernfalls beantworten wir die Frage mit »Ja«. Beachten Sie, dass die Berechnung der Erreichbarkeit höchstens einen Zeitaufwand von $O(n^2)$ erfordert, wenn der Automat über n Zustände verfügt, und im schlimmsten Fall proportional zur Anzahl der Pfeile im Übergangsdiagramm des Automaten ist, die kleiner als n^2 und nicht größer als $O(n^2)$ sein kann.

Wenn ein regulärer Ausdruck, der die Sprache L repräsentiert, statt eines Automaten vorliegt, können wir den Ausdruck in einen ε-NEA umwandeln und in der oben beschriebenen Weise verfahren. Da der Automat, der aus einem regulären Ausdruck der Länge n resultiert, höchstens $O(n)$ Zustände und Übergänge besitzt, erfordert der Algorithmus einen Zeitaufwand von $O(n)$.

Wir können den regulären Ausdruck aber auch direkt daraufhin untersuchen, ob er eine leere Sprache beschreibt. Klar ist, dass die Sprache mit Sicherheit nicht leer ist, wenn der reguläre Ausdruck kein Vorkommen von ∅ enthält. Ist das Symbol ∅ vorhanden, dann kann die Sprache unter Umständen leer sein. Die folgenden rekursiven Regeln geben darüber Aufschluss, ob ein regulärer Ausdruck eine leere Sprache beschreibt.

INDUKTIONSBEGINN: ∅ repräsentiert die leere Sprache; ε und **a** für ein Symbol a repräsentieren eine nichtleere Sprache.

INDUKTIONSSCHRITT: Angenommen R ist ein regulärer Ausdruck, dann sind vier Fälle zu unterscheiden, abhängig von der Art und Weise, wie R konstruiert ist:

1. $R = R_1 + R_2$. Dann ist die Sprache $L(R)$ genau dann leer, wenn sowohl $L(R_1)$ als auch $L(R_2)$ leer ist.
2. $R = R_1R_2$. Dann ist die Sprache $L(R)$ genau dann leer, wenn $L(R_1)$ oder $L(R_2)$ leer ist.
3. $R = R_1^*$. Dann ist $L(R)$ nicht leer. Sie enthält in jedem Fall mindestens ε.

4. $R = (R_1)$. Dann ist die Sprache $L(R)$ genau dann leer, wenn $L(R_1)$ leer ist, da diese Sprachen identisch sind.

4.3.3 Zugehörigkeit zu einer regulären Sprache prüfen

Wenn eine Zeichenreihe w und die Sprache L gegeben sind, besteht die nächste wichtige Frage darin, ob w in L enthalten ist. Während w explizit angegeben wird, wird L durch einen Automaten oder einen regulären Ausdruck repräsentiert.

Wird L durch einen DEA beschrieben, dann ist der Algorithmus einfach. Simuliere den DEA, der die Zeichenreihe der Eingabesymbole w verarbeitet, ab dem Startzustand. Wenn der DEA einen akzeptierenden Zustand erreicht, dann ist die Frage zu bejahen; andernfalls ist sie zu verneinen. Dieser Algorithmus ist äußerst schnell. Wenn $|w| = n$ ist und der Automat durch eine passende Datenstruktur repräsentiert wird, wie z. B. ein zweidimensionales Array als Übergangstabelle, dann erfordert jeder Übergang einen konstanten Zeitaufwand, und der gesamte Test erfordert $O(n)$ Zeit.

Wird L durch eine andere Repräsentation als einen DEA beschrieben, könnten wir diese in einen DEA umwandeln und den oben beschriebenen Test durchführen. Dieser Ansatz könnte einen Zeitaufwand erfordern, der exponentiell zur Größe der Repräsentation ist, obwohl er in Bezug auf $|w|$ linear ist. Handelt es sich bei der Repräsentation allerdings um einen NEA oder einen ε-NEA, dann ist es einfacher und effizienter, den NEA direkt zu simulieren. Das heißt, wir verarbeiten der Reihe nach die einzelnen Zeichen von w und führen Buch über die Menge der Zustände, die der NEA annehmen kann, und können so jeden mit dem entsprechenden Präfix beschrifteten Pfad verfolgen. Dieser Ansatz wurde in Abbildung 2.8 vorgestellt.

Besitzt w die Länge n und der NEA s Zustände, dann beträgt die Ausführungszeit dieses Algorithmus $O(ns^2)$. Die einzelnen Eingabesymbole können verarbeitet werden, indem wir die vorherige Zustandsmenge nehmen, die höchstens s Zustände umfasst, und die Nachfolger der einzelnen Zustände dieser Menge betrachten. Wir bilden die Vereinigung von höchstens s Mengen von höchstens s Zuständen, wofür ein Zeitaufwand von $O(s^2)$ erforderlich ist.

Falls der NEA über ε-Übergänge verfügt, dann müssen wir die ε-Hülle berechnen, bevor wir mit der Simulation beginnen. Dann umfasst die Berechnung jedes Eingabesymbols a zwei Phasen, die jeweils $O(s^2)$ Zeit erfordern. Zuerst nehmen wir die vorherige Zustandsmenge und suchen nach den Nachfolgern für das Eingabesymbol a. Als Nächstes berechnen wir die ε-Hülle für diese Menge von Zuständen. Die anfängliche Zustandsmenge für die Simulation besteht in der ε-Hülle des Anfangszustands des NEA.

Handelt es sich bei der Repräsentation von L um einen regulären Ausdruck der Länge s, dann können wir diesen in $O(s)$ Zeit in einen ε-NEA mit höchsten $2s$ Zuständen umwandeln. Anschließend führen wir die oben beschriebene Simulation aus, welche bei einer Eingabe w der Länge n einen Zeitaufwand von $O(ns^2)$ erfordert.

4.3.4 Übungen zum Abschnitt 4.3

* **Übung 4.3.1** Formulieren Sie einen Algorithmus, mit dem sich ermitteln lässt, ob eine reguläre Sprache L unendlich ist. *Hinweis*: Verwenden Sie das Pumping-Lemma, um zu zeigen, dass die Sprache unendlich sein muss, wenn sie eine Zeichenreihe enthält, deren Länge eine bestimmte Untergrenze überschreitet.

Übung 4.3.2 Geben Sie einen Algorithmus an, mit dem sich ermitteln lässt, ob eine reguläre Sprache L mindestens 100 Zeichenreihen enthält.

Übung 4.3.3 Angenommen L ist eine reguläre Sprache über dem Alphabet Σ. Formulieren Sie einen Algorithmus, mit dem sich ermitteln lässt, ob $L = \Sigma^*$, d. h. ob L alle Zeichenreihen über ihrem Alphabet enthält.

Übung 4.3.4 Formulieren Sie einen Algorithmus, mit dem sich ermitteln lässt, ob zwei reguläre Sprachen L_1 und L_2 mindestens eine gemeinsame Zeichenreihe enthalten.

Übung 4.3.5 Formulieren Sie einen Algorithmus, mit dem sich für zwei reguläre Sprachen L_1 und L_2 über dem gleichen Alphabet Σ ermitteln lässt, ob eine Zeichenreihe in Σ^* existiert, die weder in L_1 noch in L_2 enthalten ist.

4.4 Äquivalenz und Minimierung von Automaten

Im Gegensatz zu den vorigen Fragen (Leere und Zugehörigkeit), deren Algorithmen relativ einfach waren, erfordert die Beantwortung der Frage, ob zwei Beschreibungen regulärer Sprachen dieselbe Sprache repräsentieren, erheblich größere Anstrengung. In diesem Abschnitt erörtern wir, wie sich testen lässt, ob zwei Beschreibungen regulärer Sprachen in dem Sinn *äquivalent* sind, als sie dieselbe Sprache definieren. Eine wichtige Folge dieses Tests besteht in der Erkenntnis, dass es eine Möglichkeit gibt, einen DEA zu minimieren. Das heißt, wir können zu jedem DEA A einen äquivalenten DEA finden, der unter allen mit A äquivalenten DEAs eine minimale Anzahl von Zuständen besitzt. Tatsächlich ist dieser DEA im Wesentlichen sogar eindeutig: Wenn zwei äquivalente DEAs mit minimaler Anzahl von Zuständen gegeben sind, dann gibt es stets eine Möglichkeit, die Zustände so umzubenennen, dass die DEAs identisch werden.

4.4.1 Prüfen, ob Zustände äquivalent sind

Wir beginnen damit, Fragen zu den Zuständen eines einzigen DEA zu stellen. Unser Ziel ist es, herauszufinden, wann zwei verschiedene Zustände p und q durch einen einzigen Zustand ersetzt werden können, der sich sowohl wie p als auch wie q verhält. Wir sagen, die Zustände p und q sind *äquivalent*, wenn Folgendes gilt:

- Für alle Eingabezeichenreihen w ist $\hat{\delta}(p, w)$ genau dann ein akzeptierender Zustand, wenn $\hat{\delta}(q, w)$ ein akzeptierender Zustand ist.

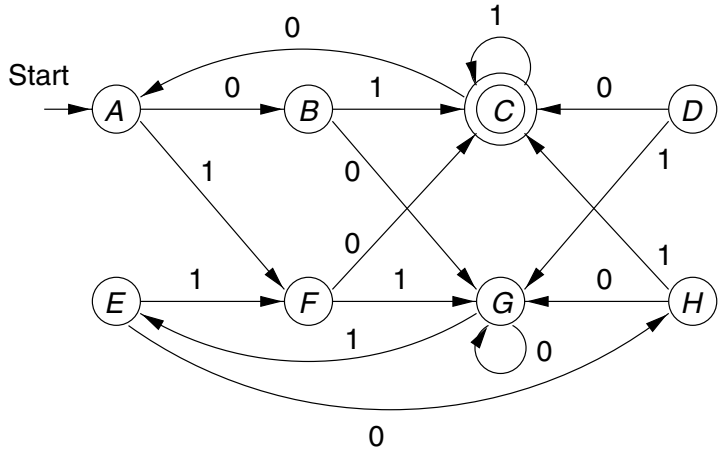
Weniger formal ausgedrückt: Es ist unmöglich, äquivalente Zustände p und q zu unterscheiden, indem wir bei einem der Zustände beginnen und fragen, ob eine gegebene Eingabezeichenreihe zur Akzeptanz führt, wenn der Automat in diesem (unbekannten) Zustand gestartet wird. Beachten Sie, wir fordern *nicht*, dass es sich bei $\hat{\delta}(p, w)$ und $\hat{\delta}(q, w)$ um denselben Zustand handelt, nur dass entweder beide oder keiner von beiden akzeptierende Zustände sind.

Wenn zwei Zustände nicht äquivalent sind, dann bezeichnen wir sie als *unterscheidbar*. Das heißt, Zustand p ist vom Zustand q unterscheidbar, wenn es mindestens eine Zeichenreihe w gibt, derart dass einer der Zustände $\hat{\delta}(p, w)$ und $\hat{\delta}(q, w)$ akzeptiert und der andere nicht.

Beispiel 4.18 Betrachten Sie den DEA aus Abbildung 4.8, dessen Übergangsfunktion wir in diesem Beispiel δ nennen wollen. Bestimmte Zustandspaare sind offensichtlich nicht äquivalent. Beispielsweise sind C und G nicht äquivalent, weil ein Zustand akzeptierend ist und der andere nicht. Das heißt, die leere Zeichenreihe unterscheidet diese beiden Zustände, da $\hat{\delta}(C, \varepsilon)$ akzeptierend ist und $\hat{\delta}(G, \varepsilon)$ ist es nicht.

Betrachten Sie die Zustände A und G. Die Zeichenreihe ε unterscheidet sie nicht, weil sie beide nicht akzeptierend sind. Die Zeichenreihe *0* unterscheidet sie nicht, weil die Eingabe 0 einen Übergang in B bzw. G bewirkt und diese beiden Zustände nicht akzeptierend sind. Allerdings unterscheidet die Eingabe 01 A von G, weil $\hat{\delta}(A, 01) = C$ und $\hat{\delta}(G, 01) = E$; C ist ein akzeptierender Zustand und E ist es nicht. Jede Eingabezeichenreihe, die in A und G Übergänge in Zustände bewirkt, von denen nur einer akzeptierend ist, reicht für den Beweis aus, dass A und G nicht äquivalent sind.

Abbildung 4.8: Ein Automat mit äquivalenten Zuständen



Betrachten Sie dagegen die Zustände A und E. Keiner von beiden ist akzeptierend und daher unterscheiden sie sich nicht hinsichtlich ε. Auf die Eingabe 1 hin wechseln beide in den Zustand F. Folglich kann keine Eingabezeichenreihe, die mit 1 beginnt, zu einer Unterscheidung zwischen A und E führen, denn für jede Zeichenreihe x gilt $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x)$.

Wir wollen nun betrachten, wie sich die Zustände A und E im Fall von Eingaben verhalten, die mit 0 beginnen. Dann erfolgt ein Übergang zum Zustand B bzw. H. Da diese beiden Zustände nicht akzeptierend sind, führt auch die Zeichenreihe 0 zu keiner Unterscheidung zwischen A und E. Die Zustände B bzw. H ändern in dieser Beziehung nichts, da sie beide auf die Eingabe 0 hin zum Zustand G und auf die Eingabe 1 hin zum Zustand C übergehen. Daher führen sämtliche Eingaben, die mit 0 beginnen, zu keiner Unterscheidung zwischen A und E. Wir schließen daraus, dass keine Eingabezeichenreihe zu einer Unterscheidung zwischen A und E führen wird; d. h. diese beiden Zustände sind äquivalent. ■

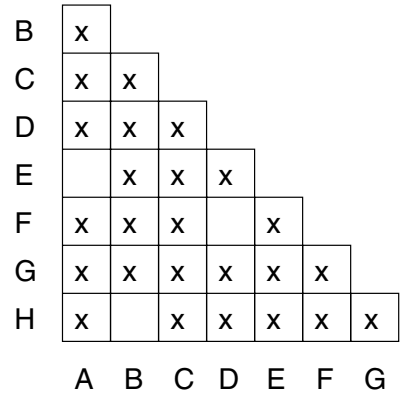
Um äquivalente Zustände zu finden, versuchen wir unser Möglichstes, unterscheidbare Zustandspaare aufzuspüren. Es mag vielleicht überraschen, ist aber wahr, dass jedes Zustandspaar, das wir als nicht unterscheidbar einstufen können, äquivalent ist, wenn wir den unten beschriebenen Algorithmus anwenden und unser Möglichstes tun. Der Algorithmus, den wir als »*Table-filling-Algorithmus*« bezeichnen, entdeckt rekursiv unterscheidbare Zustandspaare in einem DEA $A = (Q, \Sigma, \delta, q_0, F)$.

INDUKTIONSBEGINN: Wenn p ein akzeptierender Zustand und q ein nicht akzeptierender Zustand ist, dann ist das Paar $\{p, q\}$ unterscheidbar.

INDUKTIONSSCHRITT: Seien p und q Zustände, derart dass für ein Eingabesymbol a die Zustände $r = \delta(p, a)$ und $s = \delta(q, a)$ bekanntermaßen unterscheidbar sind. Dann ist $\{p, q\}$ ein Paar unterscheidbarer Zustände. Diese Regel ist aus dem Grund vernünftig, weil es eine Zeichenreihe w geben muss, die r von s unterscheidet; d. h. genau einer der Zustände $\hat{\delta}(r, w)$ und $\hat{\delta}(s, w)$ ist ein akzeptierender Zustand. Demnach muss die Zeichenreihe aw p von q unterscheiden, da $\hat{\delta}(p, aw)$ und $\hat{\delta}(q, aw)$ den Zuständen $\hat{\delta}(r, w)$ und $\hat{\delta}(s, w)$ entsprechen.

Beispiel 4.19 Wir wollen den Table-filling-Algorithmus auf den DEA aus Abbildung 4.8 anwenden. Die resultierende Tabelle ist in Abbildung 4.9 dargestellt, wobei x für ein Paar von unterscheidbaren Zuständen steht und die leeren Quadrate die Paare bezeichnen, die sich als äquivalent erwiesen haben. Anfangs enthält die Tabelle keine Einträge mit x.

Abbildung 4.9: Tabelle nicht äquivalenter Zustände



Auf Grund des Induktionsbeginns und da C der einzige akzeptierende Zustand ist, tragen wir in die Zeile und Spalte für C ein x in die Tabelle ein. Nachdem wir nun einige unterscheidbare Paare kennen, können wir andere entdecken. Da das Paar $\{C, H\}$ unterscheidbar ist und von den Zuständen E und F auf die Eingabe 0 hin ein Übergang nach H bzw. C erfolgt, wissen wir, dass auch $\{E, F\}$ ein unterscheidbares Paar ist. In der Tat lassen sich alle x-Einträge in Abbildung 4.9, mit Ausnahme des Paars $\{A, G\}$, ermitteln, indem man die Übergänge von Zustandspaaren bei der Eingabe 0 oder 1

betrachtet und beobachtet, dass (bei einer dieser Eingaben) ein Zustand zu C führt und der andere nicht. Wir können in der nächsten Runde zeigen, dass das Paar $\{A, G\}$ unterscheidbar ist, da bei der Eingabe 1 ein Übergang zu F bzw. E erfolgt und wir bereits wissen, dass das Paar $\{E, F\}$ unterscheidbar ist.

Wir können ansonsten jedoch keine unterscheidbaren Paare finden. Die drei verbleibenden Paare $\{A, E\}$, $\{B, H\}$ und $\{D, F\}$ sind daher äquivalente Paare. Überlegen Sie beispielsweise, warum wir nicht folgern können, dass $\{A, E\}$ ein unterscheidbares Paar ist. Bei der Eingabe 0 geht A in B und E in H über und $\{B, H\}$ hat sich bislang noch nicht als unterscheidbar erwiesen. Auf die Eingabe 1 hin gehen sowohl A als auch E zu F über, sodass sich die Zustände auf diese Weise auch nicht unterscheiden lassen. Die übrigen beiden Paare $\{B, H\}$ und $\{D, F\}$ werden sich nie unterscheiden lassen, da sie über identische Übergänge für 0 und 1 verfügen. Folglich endet der Table-filling-Algorithmus mit der in Abbildung 4.9 gezeigten Tabelle, die die korrekte Bestimmung äquivalenter und nicht äquivalenter Zustände liefert. ■

Satz 4.20 Wenn zwei Zustände durch den Table-filling-Algorithmus nicht unterschieden werden, dann sind diese Zustände äquivalent.

Beweis: Wir wollen wieder annehmen, dass von dem DEA $A = (Q, \Sigma, \delta, q_0, F)$ die Rede ist. Angenommen, der Satz ist falsch; d. h. es gibt mindestens ein Zustandspaar $\{p, q\}$, derart dass

1. die Zustände p und q unterscheidbar sind in dem Sinn, dass es eine Zeichenreihe w gibt, für die gilt, dass genau einer der Zustände $\hat{\delta}(p, w)$ und $\hat{\delta}(q, w)$ akzeptierend ist, und
2. der Table-filling-Algorithmus nicht findet, dass die Zustände p und q unterscheidbar sind.

Wir wollen ein solches Zustandspaar ein *schlechtes Paar* nennen.

Wenn es schlechte Paare gibt, dann muss es unter ihnen Paare geben, die sich durch eine kürzeste Zeichenreihe unter allen, die schlechte Paare unterscheiden, unterscheiden lassen. Sei $\{p, q\}$ solch ein schlechtes Paar und sei $w = a_1a_2 \ldots a_n$ eine Zeichenreihe mit minimaler Länge unter allen, die p von q unterscheiden. Dann ist genau einer der Zustände $\hat{\delta}(p, w)$ und $\hat{\delta}(q, w)$ ein akzeptierender Zustand.

Wir stellen zuerst fest, dass w nicht gleich ε sein kann, da Zustandspaare, die durch ε unterschieden werden, beim Induktionsbeginn des Table-filling-Algorithmus markiert werden. Folglich ist $n \geq 1$.

Betrachten Sie die Zustände $r = \delta(p, a_1)$ und $s = \delta(q, a_1)$. Die Zustände r und s sind durch die Zeichenreihe $a_2a_3 \ldots a_n$ unterscheidbar, da diese Zeichenreihe zu den Zuständen $\hat{\delta}(p, w)$ und $\hat{\delta}(q, w)$ führt. Die Zeichenreihe, die r und s unterscheidet, ist allerdings kürzer als irgendeine Zeichenreihe, die ein schlechtes Paar unterscheidet. Folglich kann $\{r, s\}$ kein schlechtes Paar sein. Also muss der Table-filling-Algorithmus erkannt haben, dass diese Zustände unterscheidbar sind.

Der induktive Teil des Table-filling-Algorithmus endet aber erst dann, wenn die Schlussfolgerung gezogen wurde, dass auch p und q unterscheidbar sind, da der Algorithmus erkannt hat, dass $r = \delta(p, a_1)$ von $s = \delta(q, a_1)$ unterscheidbar ist. Wir haben damit einen Widerspruch zu unserer Annahme hergeleitet, dass es schlechte Paare gibt. Wenn es keine schlechten Paare gibt, dann wird jedes Paar unterscheidbarer Zustände durch den Table-filling-Algorithmus unterschieden, und somit ist der Satz wahr. ■

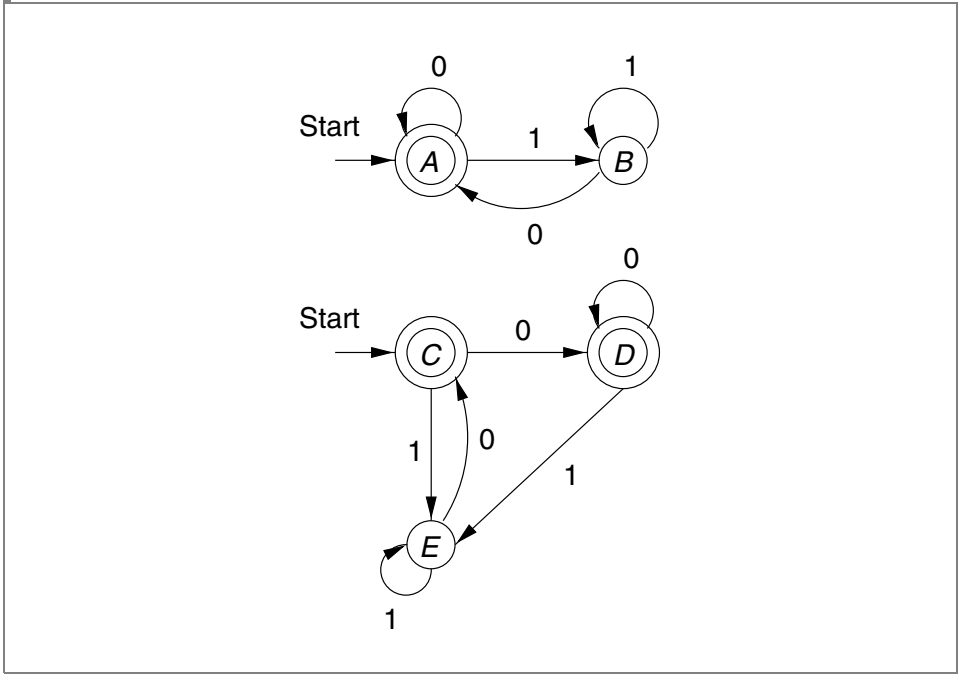
4.4.2 Prüfen, ob reguläre Sprachen äquivalent sind

Der Table-filling-Algorithmus bietet uns eine Möglichkeit, auf einfache Weise festzustellen, ob reguläre Sprachen gleich sind. Angenommen, die Sprachen L und M werden auf irgendeine Weise beschrieben, z. B. eine durch einen regulären Ausdruck und die andere durch einen NEA. Transformieren Sie beide Darstellungen jeweils in einen DEA, und sorgen Sie (ggf. durch Umbenennung der Zustände) dafür, dass deren Zustandsmengen disjunkt sind. Fassen Sie dann die Zustandsübergangstafeln dieser beiden DEAs in einer einzigen Zustandsübergangstafel zusammen.

Stellen Sie sich nun einen DEA vor, dessen Zustände die Vereinigung der Zustände der DEAs für L und M darstellen. Technisch gesehen verfügt dieser DEA über zwei Startzustände. Der Startzustand ist jedoch, was die Bestimmung der Äquivalenz von Zuständen betrifft, eigentlich irrelevant, und daher können wir einen beliebigen Zustand als alleinigen Startzustand einsetzen.

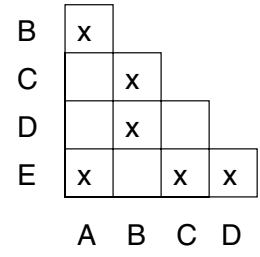
Nun prüfen wir mithilfe des Table-filling-Algorithmus, ob die Startzustände der ursprünglichen DEAs äquivalent sind. Falls Sie äquivalent sind, dann ist $L = M$, falls nicht, dann gilt $L \neq M$.

Abbildung 4.10: Zwei äquivalente DEAs



Beispiel 4.21 Betrachten Sie die beiden DEAs in Abbildung 4.10. Jeder dieser beiden DEAs akzeptiert die leere Zeichenreihe und alle Zeichenreihen, die mit 0 enden, d. h. die Sprache des regulären Ausdrucks $\varepsilon + (\mathbf{0} + \mathbf{1})^*\mathbf{0}$. Wir können uns vorstellen, dass Abbildung 4.10 einen einzigen DEA mit den fünf Zuständen A bis E repräsentiert. Wenn wir den Table-filling-Algorithmus auf diesen Automaten anwenden, erhalten wir das in Abbildung 4.11 dargestellte Ergebnis.

Abbildung 4.11: Tabelle der Unterscheidungsmerkmale für die DEAs aus Abbildung 4.10



Diese Tabelle ergab sich daraus, dass wir für alle Zustandspaare, bei denen genau ein Zustand akzeptierend ist, den Eintrag x eingefügt haben. Damit ist bereits alles erledigt. Bei den vier verbleibenden Paaren $\{A, C\}$, $\{A, D\}$, $\{C, D\}$ und $\{B, E\}$ handelt es sich um äquivalente Paare. Sie sollten sich vergewissern, dass im induktiven Teil des Table-filling-Algorithmus keine weiteren unterscheidbaren Paare entdeckt werden. Anhand der in Abbildung 4.11 gezeigten Tabelle können wir beispielsweise das Paar $\{A, D\}$ nicht unterscheiden, weil die Zustände bei der Eingabe 0 zu sich selbst und bei der Eingabe 1 zum Paar $\{B, E\}$ übergehen, das sich noch nicht als unterscheidbar erwiesen hat. Da A und C nach dieser Tabelle äquivalent sind und die Startzustände der ursprünglichen Automaten darstellen, schließen wir daraus, dass diese DEAs dieselbe Sprache akzeptieren. ■

Der zum Ausfüllen der Tabelle und folglich zur Entscheidung der Frage, ob zwei Zustände äquivalent sind, erforderliche Zeitaufwand ist polynomiell in der Anzahl der Zustände. Wenn es n Zustände gibt, dann gibt es $\binom{n}{2}$ oder $n(n-1)/2$ Zustandspaare. In einer Runde überprüfen wir alle Zustandspaare daraufhin, ob einer ihrer Nachfolger sich als unterscheidbar erwiesen hat. Daher nimmt diese Runde mit Sicherheit nicht mehr als $O(n^2)$ Zeit in Anspruch. Der Algorithmus wird zudem beendet, wenn eine Runde keine weiteren Tabelleneinträge ergibt. Folglich kann es nicht mehr als $O(n^2)$ Runden geben, und $O(n^4)$ ist mit Sicherheit Obergrenze für die Ausführungszeit des Table-filling-Algorithmus.

Eine geschickte Ausführung des Algorithmus kann die Tabelle allerdings in $O(n^2)$ Zeit ausfüllen. Dazu wird für jedes Zustandspaar $\{r, s\}$ eine Liste mit den Paaren $\{p, q\}$ angelegt, die von $\{r, s\}$ abhängen, d. h. unterscheidbar sind, wenn $\{r, s\}$ unterscheidbar ist. Wir erstellen diese Listen $l(\{r, s\})$, indem wir jedes Zustandspaar $\{p, q\}$ untersuchen und für jedes Eingabesymbol a in die Liste $l(\{\delta(p, a), \delta(q, a)\})$ eintragen, weil $\delta(p, a)$ und $\delta(q, a)$ Nachfolgezustände von p bzw. q bezüglich a sind, und $\{p, q\}$ also von $\{\delta(p, a), \delta(q, a)\}$ abhängt.

Wenn wir ein Eingabesymbol finden, durch das das Zustandspaar $\{r, s\}$ unterscheidbar wird, gehen wir die Liste $l(\{r, s\})$ durch. Jedes Paar auf dieser Liste, das noch nicht als unterscheidbar gekennzeichnet ist, wird dann als solches gekennzeichnet.

Der Gesamtarbeitsaufwand dieses Algorithmus ist zur Gesamtsumme der Listenlängen proportional, da wir entweder etwas zu den Listen hinzufügen (Initialisierung) oder ein Element der Liste zum ersten und letzten Mal überprüfen (wenn wir die Liste $l(\{r, s\})$ für ein Paar $\{r, s\}$ durchgehen, das sich als unterscheidbar erwiesen

hat). Da die Größe des Eingabealphabets als konstant betrachtet wird, wird jedes Zustandspaar in $O(1)$ Listen eingefügt. Weil es $O(n^2)$ Paare gibt, beträgt der Gesamtarbeitsaufwand $O(n^2)$.

4.4.3 Minimierung von DEAs

Eine andere wichtige Konsequenz der Prüfung der Äquivalenz von Zuständen besteht darin, dass wir DEAs »minimieren« können. Das heißt, für jeden DEA können wir einen äquivalenten DEA finden, der so wenige Zustände besitzt wie kein anderer DEA, der dieselbe Sprache akzeptiert. Abgesehen von der Tatsache, dass Zustände beliebig benannt werden können, ist dieser Minimal-DEA, der über die minimale Anzahl von Zuständen verfügt, zudem für eine Sprache eindeutig.

Der Minimierung von DEAs liegt der Gedanke zu Grunde, dass es das Konzept der Zustandsäquivalenz ermöglicht, die Zustände in Blöcke aufzuteilen, derart dass

1. alle Zustände eines Blocks äquivalent sind.
2. keine zwei Zustände, die aus verschiedenen Blöcken gewählt werden, äquivalent sind.

Vor dieser Aufteilung müssen wir allerdings alle Zustände eliminieren, die vom Startzustand aus nicht erreichbar sind.

Beispiel 4.22 Betrachten Sie die Tabelle in Abbildung 4.9, in der die Äquivalenz und Unterscheidbarkeit der Zustände aus Abbildung 4.8 dargestellt sind. Die Aufteilung in äquivalente Blöcke ergibt ($\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\}$). Beachten Sie, dass die drei Paare äquivalenter Zustände jeweils zusammen einen Block bilden, während die einzelnen Zustände, die von allen anderen unterscheidbar sind, für sich genommen einen Block darstellen. ■

Satz 4.23 Die Äquivalenz von Zuständen ist transitiv. Das heißt, wenn in einem DEA $A = (Q, \Sigma, \delta, q_0, F)$ die Zustände p und q und zudem die Zustände q und r äquivalent sind, dann müssen auch p und r äquivalent sein.

Beweis: Beachten Sie, dass es sich bei der Transitivität um eine Eigenschaft handelt, die wir von jeder »Äquivalenz« fordern. Allerdings wird eine Beziehung nicht dadurch transitiv, dass wir sie »Äquivalenz« nennen, sondern wir müssen beweisen, dass diese Bezeichnung gerechtfertigt ist.

Angenommen, die Zustandspaare $\{p, q\}$ und $\{q, r\}$ seien äquivalent, das Paar $\{p, r\}$ sei jedoch unterscheidbar. Dann muss es eine Eingabezeichenreihe w geben, derart dass genau einer der Zustände $\hat{\delta}(p, w)$ und $\hat{\delta}(r, w)$ einen akzeptierenden Zustand repräsentiert.

Nun prüfen wir, ob $\hat{\delta}(p, w)$ und deshalb wegen der Äquivalenz $\{p, q\}$ auch $\hat{\delta}(q, w)$ einen akzeptierenden Zustand repräsentiert. Falls dem so ist, dann ist das Paar $\{q, r\}$ unterscheidbar, da $\hat{\delta}(q, w)$ einen akzeptierenden Zustand darstellt, $\hat{\delta}(r, w)$ dagegen nicht. Repräsentiert dagegen $\hat{\delta}(p, w)$ und mithin auch $\hat{\delta}(q, w)$ keinen akzeptierenden Zustand, dann ist $\{q, r\}$ aus ähnlichen Gründen unterscheidbar. Wir schließen daher durch Widerspruch, dass das Zustandspaar $\{p, r\}$ nicht unterscheidbar ist, und daher ist dieses Paar äquivalent. ■

Wir können mithilfe von Satz 4.23 den Algorithmus zur Partitionierung der Zustände rechtfertigen. Konstruiere für jeden Zustand q einen Block, der aus q und all den

Zuständen besteht, die mit q äquivalent sind. Wir müssen nun zeigen, dass die resultierenden Blöcke eine Unterteilung (auch *Partition* genannt) darstellen, d. h. kein Zustand gehört zwei Blöcken an.

Zuerst ist festzustellen, dass alle Zustände in einem Block zueinander äquivalent sind. Das heißt, wenn p und r zwei Zustände im Block der zu q äquivalenten Zustände sind, dann sind nach Satz 4.23 p und r zueinander äquivalent.

Dass zwei verschiedene Blöcke B und C, die durch die Zustände s und t erzeugt seien, disjunkt sind, ergibt sich durch eine Widerspruchsannahme. Wären B und C nicht disjunkt, dann gäbe es einen Zustand p in B und C sowie einen Zustand q, der o.B.d.A. in B, aber nicht in C liegt. Dann ist p äquivalent zu s und t, und q ist nur zu s, aber nicht zu t äquivalent. Das steht in Widerspruch zu Satz 4.23, wonach q äquivalent zu p, p äquivalent zu t und deshalb auch q äquivalent zu t ist. Zwei Blöcke B und C sind also entweder identisch oder disjunkt. Die verschiedenen Blöcke stellen somit eine Partitionierung der Zustandsmenge dar. Wir beschließen die Analyse mit:

Satz 4.24 Wenn wir für jeden Zustand q eines DEA einen Block erstellen, der aus q und allen mit q äquivalenten Zuständen besteht, dann bilden die verschiedenen Blöcke eine Partition der Zustandsmenge[6]. Das heißt, jeder Zustand gehört genau einem Block an. Alle Elemente eines Blocks sind zueinander äquivalent, und keine zwei aus unterschiedlichen Blöcken gewählten Zustände sind äquivalent. ■

Wir sind nun in der Lage, den Algorithmus zur Minimierung eines DEA $A = (Q, \Sigma, \delta, q_0, F)$ prägnant zu formulieren:

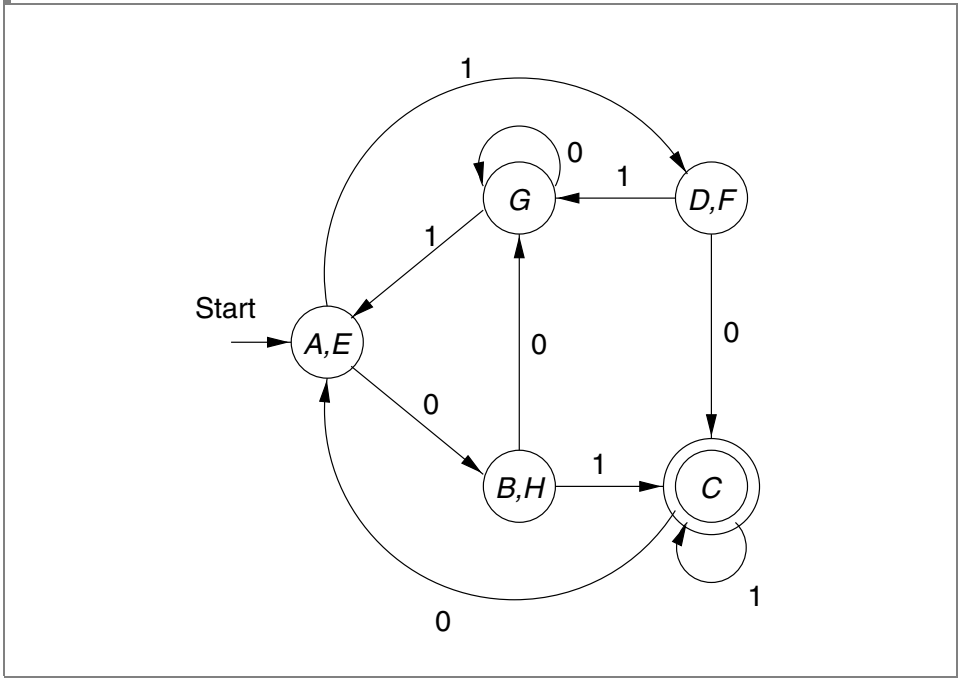
1. Ermittle mithilfe des Table-filling-Algorithmus alle Paare äquivalenter Zustände.
2. Partitioniere die Zustandsmenge Q unter Verwendung der oben beschriebenen Methode in Blöcke, die jeweils alle zu einem Zustand q äquivalenten Zustände enthalten.
3. Konstruiere den äquivalenten Minimal-DEA B unter Verwendung der Blöcke als dessen Zustände. Sei γ die Übergangsfunktion von B. Angenommen, S sei eine Menge äquivalenter Zustände von A und a sei ein Eingabesymbol; dann muss es einen Block T mit Zuständen geben, derart dass für alle Zustände q in S $\delta(q, a)$ ein Element des Blocks T ist. Andernfalls würde das Eingabesymbol a bei den beiden Zuständen p und q aus S einen Übergang in Zustände unterschiedlicher Blöcke bewirken, und diese Zustände wären nach Satz 4.24 unterscheidbar. Dann aber wären p und q nicht äquivalent und dürften nicht beide in S enthalten sein. Infolgedessen können wir unterstellen, dass $\gamma(S, a) = T$. Überdies gilt:
   a) Beim Startzustand von B handelt es sich um den Block, der den Startzustand von A enthält.
   b) Die Menge der akzeptierenden Zustände von B entspricht der Menge von Blöcken, die akzeptierende Zustände von A enthält. Zu beachten ist, dass

6. Sie sollten sich vergegenwärtigen, dass ein- und derselbe Block ausgehend von verschiedenen Zuständen mehrfach gebildet werden kann. Die Partition besteht jedoch aus den *verschiedenen* Blöcken, daher kommt jeder Block nur einmal in der Partition vor.

gilt: Wenn ein Zustand in einem Block akzeptierend ist, dann müssen alle Zustände dieses Blocks akzeptierend sein. Begründen lässt sich dies damit, dass jeder akzeptierende Zustand von jedem nicht akzeptierenden Zustand unterscheidbar ist und daher ein Block äquivalenter Zustände nicht sowohl akzeptierende als auch nicht akzeptierende Zustände enthalten kann.

Beispiel 4.25 Wir wollen nun den DEA aus Abbildung 4.8 minimieren. In Beispiel 4.22 haben wir die Unterteilung der Zustände in Blöcke festgelegt. Abbildung 4.12 zeigt den Minimalautomaten. Dessen fünf Zustände entsprechen den fünf Blöcken äquivalenter Zustände des Automaten aus Abbildung 4.8.

Abbildung 4.12: Mit dem Automaten aus Abbildung 4.8 äquivalenter DEA, der über die minimale Anzahl von Zuständen verfügt



{*A*, *E*} ist der Startzustand, da in Abbildung 4.8 *A* der Startzustand ist. {*C*} ist der einzige akzeptierende Zustand, weil in Abbildung 4.8 *C* der einzige akzeptierende Zustand ist. Beachten Sie, dass die Übergänge in Abbildung 4.12 die Übergänge in Abbildung 4.8 genau widerspiegeln. Beispielsweise enthält Abbildung 4.12 für die Eingabe 0 einen Übergang von {*A*, *E*} nach {*B*, *H*}. Das ist richtig, weil der Automat in Abbildung 4.8 bei der Eingabe 0 von *A* in *B* und von *E* in *H* übergeht. Für die Eingabe 1 wird ein Übergang von {*A*, *E*} nach {*D*, *F*} angegeben. Abbildung 4.8 zeigt, dass bei der Eingabe 1 sowohl von *A* als auch von *E* aus ein Übergang in *F* erfolgt. Daher ist die Wahl des Nachfolgers von {*A*, *E*} für die Eingabe 1 auch korrekt. Beachten Sie, dass die Tatsache, dass es weder von *A* noch von *E* aus einen Übergang in *D* gibt, nicht wichtig ist. Sie können überprüfen, ob alle anderen Übergänge ebenso korrekt sind. ■

4.4.4 Warum minimierte DEAs unschlagbar sind

Angenommen, wir haben einen DEA A und wir minimieren ihn, um unter Verwendung der Unterteilungsmethode von Satz 4.24 einen DEA M zu konstruieren. Der Satz zeigt, dass wir die Zustände von A nicht in weniger Gruppen einteilen können, wenn der resultierende DEA äquivalent sein soll. Könnte es jedoch einen anderen, von A unabhängigen DEA N geben, der dieselbe Sprache wie A und M akzeptiert, aber weniger Zustände als M besitzt? Wir können durch Widerspruch beweisen, dass es keinen solchen DEA N gibt.

Zuerst prüfen wir nach dem in Abschnitt 4.4.1 beschriebenen Verfahren, welche Zustände von M und N unterscheidbar sind, wobei wir M und N so behandeln, als gehe es um einen DEA. Wir können davon ausgehen, dass M und N keine Zustände mit gleichem Namen enthalten, sodass sich die Übergangsfunktion des kombinierten Automaten aus der Vereinigung der Übergangsregeln von M und N ergibt. Zustände des kombinierten DEA sind genau dann akzeptierend, wenn sie in dem ursprünglichen DEA M bzw. N akzeptierend sind.

Die Startzustände von M und N sind nicht unterscheidbar, weil $L(M) = L(N)$. Wenn zudem $\{p, q\}$ nicht unterscheidbar sind, dann sind auch deren Nachfolger für jedes Eingabesymbol nicht unterscheidbar; denn wenn wir die Nachfolger unterscheiden könnten, dann könnten wir auch p von q unterscheiden.

Die Zustände eines NEA minimieren

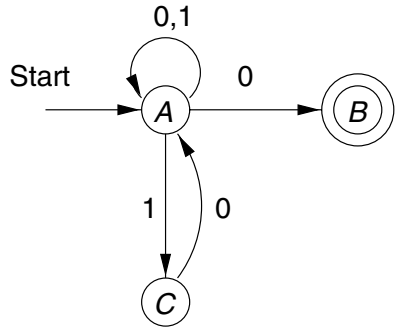
Sie nehmen vielleicht an, dass dieselbe Technik zur Aufteilung der Zustände, mit der die Zustände eines DEA minimiert werden, auch eingesetzt werden könnte, um einen zu einem gegebenen NEA oder DEA äquivalenten Minimal-NEA zu finden. Obwohl es jedoch möglich ist, durch ein erschöpfendes Aufzählungsverfahren einen NEA, der eine gegebene reguläre Sprache akzeptiert, mit so wenigen Zuständen wie möglich zu finden, können wir die Zustände eines für die Sprache gegebenen NEA nicht einfach gruppieren.

Abbildung 4.13 zeigt ein Beispiel. Die drei Zustände sind nicht äquivalent. Sicherlich ist der akzeptierende Zustand *B* von den nicht akzeptierenden Zuständen *A* und *C* unterscheidbar. Die Zustände *A* und *C* sind durch die Eingabe 0 unterscheidbar. Die 0-Nachfolger von *C* bestehen lediglich aus *A*, umfassen also keinen akzeptierenden Zustand, während die 0-Nachfolger von *A* aus {*A*, *B*} bestehen und somit einen akzeptierenden Zustand enthalten. Folglich lässt sich durch die Gruppierung äquivalenter Zustände die Anzahl der Zustände in Abbildung 4.13 nicht reduzieren.

Wir können jedoch einen kleineren NEA *N* für dieselbe Sprache konstruieren, indem wir einfach Zustand *C* entfernen. Man erkennt sofort, dass *N* ebenso wie der ursprüngliche NEA genau alle auf 0 endenden Zeichenreihen akzeptiert.

Weder M noch N können einen nicht erreichbaren Zustand besitzen, da wir diesen sonst eliminieren und einen kleineren DEA für dieselbe Sprache konstruieren könnten. Folglich ist jeder Zustand von M von mindestens einem Zustand von N nicht unterscheidbar. Verständlich wird dies, wenn wir annehmen, p sei ein Zustand von M. Dann gibt es eine Zeichenreihe $a_1a_2 \ldots a_k$, die einen Übergang vom Startzustand von M zum Zustand p bewirkt. Diese Zeichenreihe bewirkt auch einen Übergang vom Startzustand von N zu einem Zustand q. Da wir wissen, dass die Startzustände nicht unterscheidbar sind, wissen wir auch, dass ihre Nachfolger durch das Eingabesymbol a_1

Abbildung 4.13: Ein NEA, der sich nicht über die Zustandsäquivalenz minimieren lässt



nicht unterscheidbar sind. Dann sind auch die Nachfolger dieser Zustände durch das Eingabesymbol a_2 nicht unterscheidbar und so weiter, bis wir schlussfolgern, dass p und q nicht unterscheidbar sind.

Da N weniger Zustände als M besitzt, gibt es zwei verschiedene Zustände in M, die von ein und demselben Zustand in N nicht unterscheidbar und daher auch voneinander nicht unterscheidbar sind. Der Automat M wurde jedoch so entworfen, dass alle seine Zustände voneinander unterscheidbar sind. Hiermit liegt ein Widerspruch vor, und folglich ist die Annahme, dass es N gibt, falsch, d. h. M besitzt in der Tat höchstens so viele Zustände wie ein beliebiger mit A äquivalenter DEA. Formal haben wir bewiesen:

Satz 4.26 Wenn A ein DEA ist und M der DEA, der nach dem in Satz 4.24 beschriebenen Algorithmus aus A konstruiert wird, dann besitzt M höchstens so viele Zustände wie irgendein zu A äquivalenter DEA. ■

Wir können sogar etwas Stärkeres als Satz 4.26 behaupten. Es muss eine 1-zu-1-Entsprechung zwischen den Zuständen irgendeines anderen Minimal-DEA N und dem DEA M geben. Begründen lässt sich dies damit, dass wir oben argumentiert haben, dass jeder Zustand von M äquivalent zu einem Zustand von N sein muss und kein Zustand von M zu zwei Zuständen von N äquivalent sein kann. Wir können in ähnlicher Weise argumentieren, dass kein Zustand von N äquivalent zu zwei Zuständen von M sein kann, obwohl jeder Zustand von N *zu* einem Zustand von M äquivalent sein muss. Folglich ist der zu A äquivalente Minimal-DEA eindeutig, wenn man von einer möglichen Umbenennung der Zustände absieht.

4.4.5 Übungen zum Abschnitt 4.4

* **Übung 4.4.1** Tabelle 4.1 stellt die Übergangstafel eines DEA dar.

a) Bilden Sie die Tafel der unterscheidbaren Zustände für diesen Automaten.

b) Konstruieren Sie den minimalen äquivalenten DEA.

Tabelle 4.1: Ein zu minimierender DEA

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

Übung 4.4.2 Wiederholen Sie Übung 4.4.1 für den durch Tabelle 4.2 beschriebenen DEA.

Tabelle 4.2: Ein weiterer zu minimierender DEA

	0	1
$\rightarrow A$	B	E
B	C	F
$*C$	D	H
D	E	H
E	F	I
$*F$	G	B
G	H	B
H	I	C
$*I$	A	E

!! **Übung 4.4.3** Angenommen, p und q seinen unterscheidbare Zustände eines DEA A mit n Zuständen. Beschreiben Sie als Funktion von n die kleinste Obergrenze für die Länge der kürzesten Zeichenreihe, mit der p von q unterschieden werden kann.

4.5 Zusammenfassung von Kapitel 4

- *Pumping-Lemma*: Wenn eine Sprache regulär ist, dann besitzt jede ausreichend lange Zeichenreihe der Sprache eine nichtleere Teilzeichenreihe, deren beliebig häufige Wiederholung Zeichenreihen ergibt, die auch in der Sprache enthalten sind. Mithilfe dieses Merkmals lässt sich beweisen, dass viele andere Sprachen *nicht* regulär sind.
- *Operationen, die die Regularität einer Sprache erhalten*: Es gibt viele Operationen, deren Anwendung auf reguläre Sprachen wiederum eine reguläre Sprache ergibt. Zu diesen Operationen gehören Vereinigung, Verkettung, Hüllenbildung, Durchschnitt, Komplement, Differenz, Spiegelung, Homomorphismus (Ersetzung der einzelnen Symbole durch eine jeweils zugeordnete Zeichenreihe) und inverser Homomorphismus.
- *Prüfung, ob eine reguläre Sprache leer ist*: Es gibt einen Algorithmus, der anhand einer Repräsentation einer regulären Sprache – wie einem Automaten oder einem regulären Ausdruck – feststellt, ob die beschriebene Sprache die leere Menge ist.
- *Prüfung der Zugehörigkeit zu einer regulären Sprache*: Es gibt einen Algorithmus, der auf der Grundlage einer gegebenen Zeichenreihe und einer Repräsentation einer regulären Sprache feststellt, ob diese Zeichenreihe in der Sprache enthalten ist.
- *Prüfung der Unterscheidbarkeit von Zuständen*: Zwei Zustände eines DEA sind dann unterscheidbar, wenn es eine Eingabezeichenreihe gibt, die bei genau einem dieser beiden Zustände einen Übergang zu einem akzeptierenden Zustand bewirkt. Ausgehend von der Tatsache, dass Zustandspaare, die aus einem akzeptierenden und einem nicht akzeptierenden Zustand bestehen, unterscheidbar sind und durch die Suche nach Zustandspaaren, deren Nachfolger (zu einem Eingabesymbol) unterscheidbar sind, können sämtliche Paare unterscheidbarer Zustände entdeckt werden.
- *Minimierung deterministischer endlicher Automaten*: Wir können die Zustände eines DEA in Gruppen nicht unterscheidbarer Zustände zerlegen. Mitglieder zweier verschiedener Gruppen sind stets unterscheidbar. Wenn wir jede Gruppe durch einen einzelnen Zustand ersetzen, dann erhalten wir einen äquivalenten DEA, der höchstens so viele Zustände wie ein beliebiger DEA für dieselbe Sprache besitzt.

LITERATURANGABEN ZU KAPITEL 4

Abgesehen von den offensichtlichen Abgeschlossenheitseigenschaften regulärer Sprachen – Vereinigung, Verkettung und Sternoperator –, die von Kleene [6] gezeigt wurden, sehen fast alle Ergebnisse über Abgeschlossenheitseigenschaften regulärer Sprachen aus wie ähnliche Ergebnisse über kontextfreie Sprachen (die Sprachklasse, die wir in den nächsten Kapiteln untersuchen werden). So ist das Pumping-Lemma für reguläre Sprachen eine Vereinfachung des entsprechenden Ergebnisses für kontextfreie Sprachen von Bar-Hillel, Perles und Shamir [1]. Aus demselben Aufsatz gehen indirekt verschiedene der anderen hier beschriebenen Abgeschlossenheitseigenschaften hervor. Die Abgeschlossenheit bezüglich inverser Homomorphismen stammt jedoch aus [2].

Die Quotientenoperation, die in Übung 4.2.2 vorgestellt wurde, ist [3] entnommen. Dieser Aufsatz behandelt tatsächlich sogar eine allgemeinere Operation, bei der anstelle →

→ eines einzelnen Symbols *a* eine beliebige reguläre Sprache steht. Die Operationen vom Typ »teilweise Entfernung«, die in Übung 4.2.8 beispielhaft mit der ersten Hälfte der Zeichenreihen einer regulären Sprache ausgeführt werden, wurden erstmals in [8] beschrieben. Seiferas und McNaughton [9] arbeiteten den allgemeinen Fall aus, wann eine solche Operation die Regularität einer Sprache erhält.

Die ursprünglichen Entscheidungsalgorithmen, z. B. zur Bestimmung des Leerseins und der Endlichkeit von regulären Sprachen sowie der Zugehörigkeit zu einer regulären Sprache, stammen aus [7]. Algorithmen zur Minimierung der Zustände eines DEA erscheinen dort und in [5]. Der effizienteste Algorithmus zur Suche nach dem DEA mit der minimalen Anzahl von Zuständen wird in [4] beschrieben.

1. Y. Bar-Hillel, M. Perles und E. Shamir [1961]. »On formal properties of simple phase-structure grammars«, *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14**, S. 143-172.
2. S. Ginsburg und G. Rose [1963]. »Operations which preserve definability in languages«, *J:ACM* **10**:2, S. 175-195.
3. S. Ginsburg und E. H. Spanier [1963]. »Quotients of context-free languages«, *J:ACM* **10**:4, S. 487–492.
4. J. E. Hopcroft. »An *n* log *n* algorithm for minimizing the states in a finite automaton, in Z. Kohavi (Hg.) *The Theory of Machines and Computations*, Academic Press, New York, S. 189–196.
5. D. A. Huffman [1954]. »The synthesis of sequential switching circuits«, J. Franklin Inst. **257**:3-4, S. 161–190 und 275–303.
6. S. C. Kleene [1956]. »Representation of events in nerve nets and finite automata«, in C. E. Shannon und J. McCarthy, *Automata Studies*, Princeton Univ. Press, S. 3–42.
7. E. F. Moore [1956]. »Gedanken experiments on sequential machines«, in C. E. Shannon und J. McCarthy [1956]. *Automata Studies*, Princeton Univ. Press, S. 129–153.
8. R. E. Stearns und J. Hartmanis [1963]. »Regularity-preserving modifications of regular expressions«, *Information and Control* **6**:1, S. 55–69.
9. J. L. Seiferas und R. McNaughton [1976]. »Regularity-preserving modifications«, *Theoretical Computer Science* **2**:2, S. 147–154.

KAPITEL 5

Kontextfreie Grammatiken und Sprachen

Wir wenden unsere Aufmerksamkeit nun von den regulären Sprachen ab und richten sie auf eine größere Klasse von Sprachen, die als »kontextfreie Sprachen« bezeichnet werden. Diese Sprachen besitzen eine natürliche, rekursive Notation, die so genannten »kontextfreien Grammatiken«. Kontextfreie Grammatiken spielen seit den sechziger Jahren eine zentrale Rolle in der Compilertechnologie. Die ehemals zeitaufwändige Implementierung von Parsern (Funktionen, die die Struktur eines Programms erkennen) wurde dank ihnen zu einer Routineaufgabe, die sich an einem Nachmittag erledigen lässt. In den letzten Jahren wurden kontextfreie Grammatiken zur Beschreibung von Dokumentenformaten verwendet und zwar mithilfe so genannter Dokumenttypdefinitionen (DTD), die von XML-Programmierern (XML = eXtensible Markup Language) zum Informationsaustausch im Internet eingesetzt werden.

In diesem Kapitel stellen wir die Notation kontextfreier Grammatiken vor und zeigen, wie diese Grammatiken Sprachen definieren. Wir erörtern den »Parsebaum«, ein Abbild der Struktur, die eine Grammatik den Zeichenreihen einer Sprache auferlegt. Der Parsebaum ist das Ergebnis eines Parsers für ein Programm in einer Programmiersprache und stellt die Art und Weise dar, in der die Struktur von Programmen erfasst wird.

Zudem gibt es eine automatenähnliche Notation, die so genannten »Pushdown-Automaten« oder »Kellerautomaten«, zur Beschreibung genau der kontextfreien Sprachen. Wir stellen die Pushdown-Automaten in Kapitel 6 vor. Obwohl sie weniger wichtig sind als die endlichen Automaten, erweisen sich die Pushdown-Automaten, insbesondere ihre Äquivalenz mit kontextfreien Grammatiken, als hilfreich bei der Untersuchung der Abgeschlossenheits- und Entscheidbarkeitseigenschaften von kontextfreien Sprachen in Kapitel 7.

5.1 Kontextfreie Grammatiken

Wir beginnen mit einer informellen Einführung der Notation kontextfreier Grammatiken. Nachdem wir einige der wichtigsten Anwendungen dieser Grammatiken aufgezeigt haben, werden wir formale Definitionen angeben. Wir zeigen, wie man eine Grammatik formal definiert, und stellen das Verfahren der »Ableitung« vor, womit festgelegt wird, welche Zeichenreihen in der Sprache einer Grammatik enthalten sind.

5.1.1 Ein informelles Beispiel

Lassen Sie uns die Sprache der Palindrome betrachten. Ein *Palindrom* ist eine Zeichenreihe, die vorwärts und rückwärts gelesen identisch ist, z. B. `otto` oder der englische Satz `madamimadam` (»Madam, I' m Adam«, angeblich die ersten Worte, die Eva im Garten Eden hörte). Anders ausgedrückt, die Zeichenreihe w ist genau dann ein Palindrom, wenn gilt $w = w^R$. Der Einfachheit halber wollen wir nur die Palindrome betrachten, die sich mit dem Alphabet {0, 1} beschreiben lassen. Diese Sprache beinhaltet Zeichenreihen wie 0110, 11011 und ε, nicht jedoch 011, 0101.

Es lässt sich leicht beweisen, dass die Sprache L_{pal} der Palindrome aus Nullen und Einsen keine reguläre Sprache ist. Wir tun dies mithilfe des Pumping-Lemmas. Wenn L_{pal} eine reguläre Sprache ist, dann sei n die zugehörige Konstante und $w = 0^n10^n$ ein zu betrachtendes Palindrom. Wenn L_{pal} regulär ist, dann können wir w in $w = xyz$ zerlegen, derart dass y aus einer oder mehr Nullen der ersten Gruppe besteht. Folglich würde die Zeichenreihe xz, die ebenfalls in L_{pal} enthalten sein müsste, wenn L_{pal} regulär ist, weniger Nullen links von der einzelnen Eins als rechts davon aufweisen. Daher kann xz kein Palindrom sein. Damit haben wir die Annahme widerlegt, dass L_{pal} eine reguläre Sprache ist.

Es gibt eine natürliche, rekursive Definition dafür, wann eine aus Nullen und Einsen bestehende Zeichenreihe in L_{pal} enthalten ist. Sie beginnt mit der Induktionsannahme, dass einige wenige offensichtliche Zeichenreihen in L_{pal} enthalten sind, und nutzt dann die Erkenntnis, dass eine Zeichenreihe dann ein Palindrom ist, wenn sie mit dem gleichen Symbol beginnt und endet. Außerdem muss auch die resultierende Zeichenreihe ein Palindrom sein, wenn das erste und das letzte Symbol entfernt werden. Das heißt:

INDUKTIONSBEGINN: ε, 0 und 1 sind Palindrome.

INDUKTIONSSCHRITT: Wenn w ein Palindrom ist, dann sind auch $0w0$ und $1w1$ Palindrome. Eine Zeichenreihe ist nur dann ein aus Nullen und Einsen bestehendes Palindrom, wenn sie sich aus der Induktionsannahme und dem Induktionsschritt ergibt.

Eine kontextfreie Grammatik ist eine formale Notation zur Beschreibung solcher rekursiven Definitionen für Sprachen. Eine Grammatik besteht aus einer oder mehreren Variablen, die Klassen von Zeichenreihen (d. h. Sprachen) repräsentieren. In unserem Beispiel brauchen wir nur eine Variable P, die die Menge der Palindrome repräsentiert, also die Klasse der Zeichenreihen, die die Sprache L_{pal} bilden. Es gibt Regeln, die besagen, wie die Zeichenreihen einer Klasse aufgebaut sind. Zum Aufbau der Zeichenreihen können Symbole des Alphabets, Zeichenreihen, die bereits als einer der Klassen zugehörig bekannt sind, oder beides verwendet werden.

Beispiel 5.1 Tabelle 5.1 enthält die Regeln zur Definition von Palindromen, die hier in der Notation kontextfreier Grammatiken formuliert sind. Sie werden im Abschnitt 5.1.2 erfahren, was diese Regeln im Einzelnen bedeuten.

Die ersten drei Regeln bilden die Induktionsannahme. Aus ihnen geht hervor, dass die Klasse der Palindrome die Zeichenreihen ε, 0 und 1 enthält. Die rechte Seite dieser Regeln (der Teil rechts vom Pfeil) enthält keine Variablen, wobei der Grund dafür ist, dass sie eine Definitionsgrundlage darstellen.

Die letzten beiden Regeln bilden den Induktionsschritt der Definition. Regel 4 besagt beispielsweise: Wenn wir eine beliebige Zeichenreihe w aus der Klasse P nehmen und damit die Zeichenreihe $0w0$ bilden, dann ist die resultierende Zeichenreihe

Tabelle 5.1: Eine kontextfreie Grammatik für Palindrome

1.	$P \to \varepsilon$
2.	$P \to 0$
3.	$P \to 1$
4.	$P \to 0P0$
5.	$P \to 1P1$

auch in P enthalten. Regel 5 besagt analog, dass auch die Zeichenreihe $1w1$ in P enthalten ist. ■

5.1.2 Definition kontextfreier Grammatiken

Die grammatikalische Beschreibung einer Sprache umfasst vier wichtige Komponenten:

1. Es gibt eine endliche Menge von Symbolen, aus denen die Zeichenreihen der Sprache bestehen, die definiert wird. Im Beispiel der Palindrome ist diese Menge {0, 1}. Die Elemente dieser Menge (dieses Alphabets) nennt man Terminale oder terminale Zeichen oder terminale Symbole.
2. Es gibt eine endliche Menge von *Variablen*, die gelegentlich auch als *nichtterminale Zeichen* oder *syntaktische Kategorien* bezeichnet werden. Jede Variable repräsentiert eine Sprache, d. h. eine Menge von Zeichenreihen. In unserem Beispiel verwendeten wir nur eine Variable namens P zur Beschreibung der Klasse der Palindrome über dem Alphabet {0, 1}.
3. Eine dieser Variablen repräsentiert die Sprache, die definiert wird, und wird als *Startsymbol* bezeichnet. Andere Variablen stehen für zusätzliche Klassen von Zeichenreihen, die bei der Definition der Sprache des Startsymbols hilfreich sind. In unserem Beispiel ist P, die einzige Variable, das Startsymbol.
4. Es gibt eine endliche Menge von *Produktionen* oder *Regeln*, die die rekursive Definition der Sprache repräsentieren. Jede Produktion besteht aus:
   a) einer Variablen, die durch die Produktion (teilweise) definiert wird. Diese Variable wird häufig auch als *Kopf* der Produktion bezeichnet.
   b) dem Produktionssymbol $\to$.
   c) einer Zeichenreihe aus null oder mehr terminalen Symbolen und Variablen. Diese Zeichenreihe, die als *Rumpf* der Produktion bezeichnet wird, repräsentiert eine Form, Zeichenreihen der Sprache der im Kopf enthaltenen Variablen zu bilden. Hierbei werden die terminalen Symbole unverändert gelassen und jede Variable im Rumpf durch eine beliebige Zeichenreihe ersetzt, die bekanntermaßen der Sprache dieser Variablen angehört.

   Tabelle 5.1 stellt ein Beispiel für Produktionen dar.

Diese vier Komponenten bilden eine *kontextfreie Grammatik*, auch einfach nur *Grammatik* oder *kfG* genannt. Wir werden eine kfG G durch ihre vier Komponenten beschreiben, das heißt, $G = (V, T, P, S)$, wobei V für die Menge der Variablen, T für die terminalen Symbole, P für die Menge der Produktionen und S für das Startsymbol steht.

Beispiel 5.2 Die Grammatik G_{pal} für die Palindrome wird repräsentiert durch

$$G_{pal} = (\{P\}, \{0, 1\}, A, P)$$

wobei A für die Menge der fünf Produktionen steht, die in Tabelle 5.1 aufgeführt wurden. ■

Beispiel 5.3 Wir wollen eine komplexere kfG untersuchen, die (vereinfachte) Ausdrücke einer typischen Programmiersprache repräsentiert. Wir werden uns auf die Operatoren + und ∗ beschränken, die die Addition und die Multiplikation repräsentieren. Als Argumente lassen wir Bezeichner zu, aber statt die vollständige Menge typischer Bezeichner (ein Buchstabe gefolgt von null oder mehr Buchstaben oder Ziffern) lassen wir nur die Buchstaben a und b sowie die Ziffern 0 und 1 zu. Jeder Bezeichner muss mit einem a oder b beginnen, dem eine beliebige Zeichenreihe aus $\{a, b, 0, 1\}^*$ folgen kann.

Wir brauchen in dieser Grammatik zwei Variablen. Eine Variable, die wir E nennen, repräsentiert Ausdrücke. Sie ist das Startsymbol und repräsentiert die Sprache der Ausdrücke, die wir definieren. Die andere Variable I repräsentiert Bezeichner. Deren Sprache ist sogar regulär; es ist nämlich die Sprache des regulären Ausdrucks

$$(\mathbf{a} + \mathbf{b})(\mathbf{a} + \mathbf{b} + \mathbf{0} + \mathbf{1})^*$$

Wir dürfen in Grammatiken einen regulären Ausdruck allerdings nicht direkt verwenden. Stattdessen verwenden wir eine Menge von Produktionen, die dasselbe wie der reguläre Ausdruck aussagen.

Tabelle 5.2: Eine kontextfreie Grammatik für einfache Ausdrücke

1.	$E \to I$
2.	$E \to E + E$
3.	$E \to E * E$
4.	$E \to (E)$
5.	$I \to a$
6.	$I \to b$
7.	$I \to Ia$
8.	$I \to Ib$
9.	$I \to I0$
10.	$I \to I1$

Die Grammatik für Ausdrücke wird formal wie folgt ausgedrückt: $G = (\{E, I\}, T, P, E)$, wobei T für die Menge der Symbole $\{+, *, (,), a, b, 0, 1)$ und P für die in Tabelle 5.2 dargestellte Menge der Produktionen steht. Wir interpretieren die Produktionen wie folgt:

Regel (1) ist die Basisregel für Ausdrücke. Sie besagt, dass ein einzelner Bezeichner ein Ausdruck sein kann. Die Regeln (2) bis (4) beschreiben die Fälle des Induktionsschritts für Ausdrücke. Regel (2) besagt, dass ein Ausdruck aus zwei durch ein Pluszeichen verbundenen Ausdrücken bestehen kann; Regel (3) besagt dasselbe für das Multiplikationszeichen. Regel (4) besagt, dass ein in runde Klammern gesetzter Ausdruck wiederum ein Ausdruck ist.

Die Regeln (5) bis (10) beschreiben die Bezeichner I. Die Basis besteht aus den Regeln (5) und (6), die besagen, dass a und b Bezeichner sind. Die übrigen vier Regeln bilden den Induktionsschritt. Sie besagen, dass jedem Bezeichner eines der Symbole a, b, 0 oder 1 folgen kann, sodass sich daraus ein neuer Bezeichner ergibt. ■

Kompakte Notation für Produktionen

Es ist vorteilhaft, von einer Produktion als zu ihrer Kopfvariable »gehörig« zu sprechen. Wir werden oft Bemerkungen verwenden wie »die Produktionen für A« oder »A-Produktionen«, wenn wir Produktionen meinen, deren Kopf Variable A ist. Wir könnten die Produktionen für eine Grammatik schreiben, indem wir jede Variable einmal auflisten und dann die Körper der Produktionen für diese Variable auflisten, jeweils getrennt durch einen vertikalen Strich. Das heißt, dass die Produktionen $A \to \alpha_1$, $A \to \alpha_2$, ... , $A \to \alpha_n$ ersetzt werden können durch die Notation $A \to \alpha_1|\alpha_2|...|\alpha_n$. Die Grammatik für Palindrome aus Tabelle 5.1 könnte z.B. folgendermaßen geschrieben werden: $P \to \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$.

5.1.3 Ableitungen mithilfe einer Grammatik

Wir wenden die Produktionen einer kfG an, um zu schließen, dass bestimmte Zeichenreihen in der Sprache einer bestimmten Variablen enthalten sind. Es gibt zwei Ansätze für diese Schlussfolgerung. Der eher konventionelle Ansatz besteht darin, die Regeln vom Rumpf zum Kopf anzuwenden. Das heißt, wir nehmen Zeichenreihen, die bekanntermaßen in den Sprachen der einzelnen Variablen des Rumpfs enthalten sind, verketten sie in der richtigen Reihenfolge mit den terminalen Symbolen, die im Rumpf vorkommen, und schließen, dass die resultierende Zeichenreihe in der Sprache der im Kopf angegebenen Variablen enthalten ist. Wir werden dieses Verfahren im Folgenden *rekursive Inferenz* nennen.

Daneben gibt es einen anderen Ansatz zur Definition der Sprache einer Grammatik, bei dem die Produktionen vom Kopf zum Rumpf angewandt werden. Wir expandieren das Startsymbol mithilfe einer der Produktionen (d. h. einer Produktion, deren Kopf das Startsymbol darstellt). Wir expandieren zudem die resultierende Zeichenreihe, indem wir eine der Variablen durch den Rumpf einer ihrer Produktionen ersetzen, und fahren damit fort, bis wir eine Zeichenreihe ableiten, die ausschließlich aus terminalen Symbolen besteht. Die Sprache der Grammatik umfasst sämtliche Zeichenreihen aus terminalen Symbolen, die sich auf diese Weise bilden lassen. Diese Art der Verwendung von Grammatiken wird als *Ableitung* bezeichnet.

Wir werden mit einem Beispiel für den ersten Ansatz, die rekursive Inferenz, beginnen. Häufig scheint die Verwendung von Grammatiken in Ableitungen jedoch

natürlicher, und wir werden als Nächstes eine Notation zur Beschreibung dieser Ableitungen entwickeln.

Beispiel 5.4 Betrachten wir einige der Schlüsse, die wir mithilfe der Grammatik für Ausdrücke aus Tabelle 5.2 ziehen können. Tabelle 5.3 fasst diese Schlüsse zusammen. Beispielsweise besagt Zeile (*i*), dass wir aus Produktion 5 darauf schließen können, dass die Zeichenreihe *a* in der Sprache für *I* enthalten ist. Die Zeilen (*ii*) bis (*iv*) besagen, dass wir schlussfolgern können, dass *b*00 ein Bezeichner ist, indem wir Produktion 6 einmal (um *b* zu erhalten) und dann Produktion 9 zweimal anwenden (um die beiden Nullen anzuhängen).

Tabelle 5.3: Mithilfe der Grammatik aus Tabelle 5.2 auf Zeichenreihen schließen

	Erschlossene Zeichenreihe	Für die Sprache von	Verwendete Produktion	Verwendete Zeichenreihe(n)
(*i*)	*A*	*I*	5	–
(*ii*)	*B*	*I*	6	–
(*iii*)	*b*0	*I*	9	(*ii*)
(*iv*)	*b*00	*I*	9	(*iii*)
(*v*)	*A*	*E*	1	(*i*)
(*vi*)	*b*00	*E*	1	(*iv*)
(*vii*)	*a* + *b*00	*E*	2	(*v*), (*vi*)
(*viii*)	(*a* + *b*00)	*E*	4	(*vii*)
(*ix*)	*a* ∗ (*a* + *b*00)	*E*	3	(*v*), (*viii*)

In den Zeilen (*v*) und (*vi*) wird Produktion 1 genutzt, um darauf zu schließen, dass die Zeichenreihen *a* und *b*00, die gemäß der Schlussfolgerung aus den Zeilen (*i*) und (*iv*) Bezeichner sind, auch in der Sprache der Variablen *E* enthalten sind, da jeder Bezeichner ein Ausdruck ist. Zeile (*vii*) setzt die Produktion 2 ein, um zu dem Schluss zu kommen, dass die Summe dieser Bezeichner ein Ausdruck ist. In Zeile (*viii*) wird gemäß Produktion 4 darauf geschlossen, dass auch der in Klammern gesetzte Ausdruck aus Zeile (*vii*) ein Ausdruck ist; und in Zeile (*ix*) wird nach Produktion 3 der Bezeichner *a* mit dem Ausdruck multipliziert, den wir in Zeile (*viii*) erschlossen haben. ■

Das Verfahren der Ableitung von Zeichenreihen, indem Produktionen vom Kopf zum Rumpf angewandt werden, erfordert die Definition eines neuen Relationssymbols $\Rightarrow$. Angenommen, $G = (V, T, P, S)$ sei eine kfG. Sei $\alpha A \beta$ eine Zeichenreihe aus terminalen Symbolen und Variablen, wobei A eine Variable ist. Das heißt, α und β sind Zeichenreihen aus $(V \cup T)^*$, und A ist in V enthalten. $A \to \gamma$ sei eine Produktion von G. Dann sagen wir, $\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$. Wenn bekannt ist, dass von G die Rede ist, sagen wir einfach

$\alpha A\beta \Rightarrow \alpha\gamma\beta$. Beachten Sie, dass in einem Ableitungsschritt eine beliebige Variable in der Zeichenreihe durch den Rumpf einer ihrer Produktionen ersetzt wird.

Wir können die Relation $\Rightarrow$ erweitern, sodass sie null, einen oder mehrere Ableitungsschritte repräsentiert, ebenso wie die Übergangsfunktion δ eines endlichen Automaten zu $\hat{\delta}$ erweitert wurde. Bei Ableitungen geben wir durch einen Stern ($*$) »null oder mehr Schritte« an:

INDUKTIONSBEGINN: Für jede Zeichenreihe α, die aus terminalen Symbolen und Variablen besteht, sagen wir, dass $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$. Das heißt, jede Zeichenreihe ergibt eine Ableitung ihrer selbst.

INDUKTIONSSCHRITT: Wenn $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ und $\beta \overset{*}{\underset{G}{\Rightarrow}} \gamma$, dann gilt $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. Das heißt, wenn aus α in null oder mehr Schritten zu β und β in null oder mehr Schritten zu γ werden kann, dann kann α zu γ werden. Anders ausgedrückt, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ bedeutet, dass es eine Folge von Zeichenreihen $\gamma_1, \gamma_2, ..., \gamma_n$ für ein $n \geq 1$ gibt, derart dass

1. $\alpha = \gamma_1$,
2. $\beta = \gamma_n$ und
3. für $i = 1, 2, ..., n\text{-}1$, $\gamma_i \Rightarrow \gamma_{i+1}$ gilt.

Wenn die Grammatik G bekannt ist, dann verwenden wir $\overset{*}{\Rightarrow}$ statt $\overset{*}{\underset{G}{\Rightarrow}}$.

Beispiel 5.5 Die Schlussfolgerung, dass $a * (a + b00)$ in der Sprache der Variablen E enthalten ist, kann durch eine Ableitung dieser Zeichenreihe gezeigt werden, indem mit der Zeichenreihe E begonnen wird. Im Folgenden wird eine mögliche Ableitung dargestellt:

$$E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow$$

$$a * (E) \Rightarrow a * (E + E) \Rightarrow \mathrm{a} * (I + E) \Rightarrow a * (a + E) \Rightarrow$$

$$a * (a + I) \Rightarrow \mathrm{a} * (a + I0) \Rightarrow \mathrm{a} * (a + I00) \Rightarrow a * (a + b00)$$

Im ersten Schritt wird E durch den Rumpf von Produktion 3 ersetzt (aus Tabelle 5.2). Im zweiten Schritt wird Produktion 1 eingesetzt, um die erste Variable E durch I zu ersetzen, und so weiter. Beachten Sie, dass wir uns systematisch an die Strategie gehalten haben, stets die äußerste linke Variable in der Zeichenreihe zu ersetzen. Wir können jedoch in jedem Schritt wählen, welche Variable ersetzt werden soll, und wir können für diese Variable eine beliebige Produktion verwenden. Im zweiten Schritt hätten wir beispielsweise die zweite Variable E unter Verwendung von Produktion 4 durch (E) ersetzen können. In diesem Fall lautete die Zeichenreihe $E * E = \Rightarrow E * (E)$. Wir hätten auch eine Ersetzung wählen können, die nicht die gleiche Zeichenreihe von terminalen Symbolen ergeben würde. Ein einfaches Beispiel hierfür wäre, wenn wir Produktion 2 im ersten Schritt verwendet und somit $E \Rightarrow E + E$ erhalten hätten. Es gibt keine Ersetzungen für die beiden Variablen E auf der rechten Seite, mit denen $E + E$ in $a * (a + b00)$ umgeformt werden würde.

Wir können die Ableitung mithilfe der Beziehung $\overset{*}{\Rightarrow}$ komprimieren. Wir wissen auf Grund der Induktionsannahme, dass $E \overset{*}{\Rightarrow} E$. Durch wiederholte Anwendung des Induktionsschritts erhalten wir $E \overset{*}{\Rightarrow} E * E$, $E \overset{*}{\Rightarrow} I * E$ etc. und schließlich $E \overset{*}{\Rightarrow} a * (a + b00)$.

Die beiden Sichtweisen – rekursive Inferenz und Ableitung – sind äquivalent. Das heißt, man kann genau dann mittels rekursiver Inferenz darauf schließen, dass eine Zeichenreihe w aus terminalen Symbolen in der Sprache einer Variablen A enthalten ist, wenn $A \stackrel{*}{\Rightarrow} w$. Der Beweis dieser Tatsache erfordert jedoch einigen Aufwand, und wir werden erst in Abschnitt 5.2 darauf eingehen. ■

5.1.4 Links- und rechtsseitige Ableitungen

Um die Anzahl der Wahlmöglichkeiten in der Ableitung einer Zeichenreihe zu begrenzen, ist häufig die Forderung hilfreich, dass in jedem Schritt jeweils die äußerste linke Variable durch einen ihrer Produktionsrümpfe ersetzt wird. Eine solche Ableitung wird als *linksseitige* (engl. *leftmost*) *Ableitung* bezeichnet, wir und verwenden die Relationssymbole $\underset{lm}{\Rightarrow}$ und $\underset{lm}{\stackrel{*}{\Rightarrow}}$ zur Angabe eines bzw. mehrerer Schritte einer linksseitigen Ableitung. Falls nicht offensichtlich ist, welche Grammatik verwendet wird, dann kann bei beiden Symbolen auch der Name G unter dem Pfeil angegeben werden.

Analog hierzu kann man fordern, dass in jedem Schritt die äußerste rechte Variable durch einen ihrer Produktionsrümpfe ersetzt wird. Diese Art der Ableitung wird als *rechtsseitige* (engl. *rightmost*) *Ableitung* bezeichnet, und die Symbole $\underset{rm}{\Rightarrow}$ und $\underset{rm}{\stackrel{*}{\Rightarrow}}$ stehen für einen bzw. mehrere Schritte einer rechtsseitigen Ableitung. Auch hier kann der Name der Grammatik unterhalb des Pfeils angegeben werden, wenn nicht klar ist, welche Grammatik verwendet wird.

Notation für kfG-Ableitungen

Es ist eine Reihe von Konventionen zur Benennung von Symbolen im Zusammenhang mit kfGs gebräuchlich. Wir werden uns an die folgenden Konventionen halten:

1. Kleinbuchstaben vom Anfang des Alphabets (*a*, *b* usw.) stehen für terminale Symbole. Wir werden zudem annehmen, dass Ziffern und andere Zeichen wie + und Klammern terminale Symbole sind.
2. Großbuchstaben vom Anfang des Alphabets (*A*, *B* usw.) repräsentieren Variable.
3. Kleinbuchstaben vom Ende des Alphabets (z. B. *w* oder *z*) bezeichnen Zeichenreihen aus terminalen Symbolen. Diese Konvention soll daran erinnern, dass die terminalen Symbole den Eingabesymbolen von Automaten entsprechen.
4. Großbuchstaben vom Ende des Alphabets (wie *X* oder *Y*) stehen für terminale Symbole oder Variable.
5. Griechische Kleinbuchstaben (z. B. α und β) repräsentieren Zeichenreihen, die aus terminalen Symbolen und/oder Variablen bestehen.

Es gibt keine spezielle Notation für Zeichenreihen, die ausschließlich aus Variablen bestehen, da dieses Konzept nicht weiter wichtig ist. Es ist allerdings möglich, dass eine mit einem griechischen Kleinbuchstaben wie α benannte Zeichenreihe ausschließlich Variable enthält.

Beispiel 5.6 Die Ableitung im Beispiel 5.5 war eine linksseitige Ableitung. Daher können wir diese Ableitung auch wie folgt beschreiben:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E \underset{lm}{\Rightarrow}$$
$$a*(E) \underset{lm}{\Rightarrow} a*(E+E) \underset{lm}{\Rightarrow} a*(I+E) \underset{lm}{\Rightarrow} a*(a+E) \underset{lm}{\Rightarrow}$$
$$a*(a+I) \underset{lm}{\Rightarrow} a*(a+I0) \underset{lm}{\Rightarrow} a*(a+I00) \underset{lm}{\Rightarrow} a*(a+b00)$$

Wir können die linksseitige Ableitung auch durch $E \overset{*}{\underset{lm}{\Rightarrow}} a*(a+b00)$ zusammenfassen oder durch Ausdrücke wie $E*E \overset{*}{\underset{lm}{\Rightarrow}} a*(E)$ mehrere Schritte der Ableitung angeben.

Es gibt eine rechtsseitige Ableitung, bei der für jede Variable dieselbe Ersetzung durchgeführt wird, bei der aber diese Ersetzungen in einer anderen Reihenfolge erfolgen. Die rechtsseitige Ableitung lautet:

$$E \underset{rm}{\Rightarrow} E*E \underset{rm}{\Rightarrow} E*(E) \underset{rm}{\Rightarrow} E*(E+E) \underset{rm}{\Rightarrow}$$
$$E*(E+I) \underset{rm}{\Rightarrow} E*(E+I0) \underset{rm}{\Rightarrow} E*(E+I00) \underset{rm}{\Rightarrow} E*(E+b00) \underset{rm}{\Rightarrow}$$
$$E*(I+b00) \underset{rm}{\Rightarrow} E*(a+b00) \underset{rm}{\Rightarrow} I*(a+b00) \underset{rm}{\Rightarrow} a*(a+b00)$$

Diese Ableitung ermöglicht uns den Schluss: $E \overset{*}{\underset{rm}{\Rightarrow}} a*(a+b00)$. ■

Zu jeder Ableitung existiert eine äquivalente linksseitige und eine äquivalente rechtsseitige Ableitung. Das heißt, wenn w eine Zeichenreihe aus terminalen Symbolen ist, dann gilt $A \overset{*}{\Rightarrow} w$ genau dann, wenn $A \overset{*}{\underset{lm}{\Rightarrow}} w$, und $A \overset{*}{\Rightarrow} w$ genau dann, wenn $A \overset{*}{\underset{rm}{\Rightarrow}} w$. Wir werden auch diese Behauptungen in Abschnitt 5.2 beweisen.

5.1.5 Die Sprache einer Grammatik

Wenn $G = (V, T, P, S)$ eine kfG ist, dann umfasst die Sprache $L(G)$ von G die Menge von Zeichenreihen aus terminalen Symbolen, die sich vom Startsymbol ableiten lassen. Das heißt,

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

Wenn eine Sprache L die Sprache einer kontextfreien Grammatik ist, dann wird L als kontextfreie Sprache kfS bezeichnet. Wir haben behauptet, dass die Grammatik aus Tabelle 5.1 die Sprache der Palindrome über dem Alphabet {0, 1} definiert. Daher ist die Menge der Palindrome eine kontextfreie Sprache. Wir können diese Aussage wie folgt beweisen.

Satz 5.7 $L(G_{pal})$, wobei G_{pal} die Grammatik aus Beispiel 5.1 ist, ist die Menge der Palindrome über dem Alphabet {0, 1}.

Beweis: Wir werden beweisen, dass eine Zeichenreihe w aus {0, 1}* genau dann in $L(G_{pal})$ enthalten ist, wenn sie ein Palindrom ist, d. h. $w = w^R$.

(Wenn-Teil) Angenommen, w ist ein Palindrom. Wir zeigen durch Induktion über $|w|$, dass w in $L(G_{pal})$ enthalten ist.

Induktionsbeginn: Wir verwenden die Längen 0 und 1 als Basis. Wenn $|w| = 0$ oder $|w| = 1$, dann ist w gleich ε, 0 oder 1. Da es die Produktionen $P \to \varepsilon$, $P \to 0$ und $P \to 1$ gibt, schließen wir, dass $P \overset{*}{\Rightarrow} w$ in jedem dieser Basisfälle gilt.

Induktionsschritt: Angenommen, $|w| \geq 2$. Da $w = w^R$, muss w mit dem gleichen Symbol beginnen und enden. Das heißt, $w = 0x0$ oder $w = 1x1$. Zudem muss x ein Palindrom sein, das heißt $x = x^R$. Beachten Sie, dass wir die Aussage $|w| \geq 2$ brauchen, um schließen zu können, dass es am linken und rechten Ende von w zwei verschiedene Nullen bzw. Einsen gibt.

Wenn $w = 0x0$, dann wissen wir unter Bezugnahme auf die Induktionsannahme, dass $P \overset{*}{\Rightarrow} z$. Dann gibt es eine Ableitung von w aus P, nämlich $P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$. Wenn $w = 1x1$, dann verwenden wir dieselbe Argumentation, jedoch die Produktion $P \to 1P1$ im ersten Schritt. In beiden Fällen schließen wir, dass w in $L(G_{pal})$ enthalten ist, und vervollständigen den Beweis.

(Genau-dann-Teil) Wir nehmen nun an, dass w in $L(G_{pal})$ enthalten ist, d. h. $P \overset{*}{\Rightarrow} w$. Wir müssen zeigen, dass w ein Palindrom ist. Der Beweis ist ein Induktionsbeweis über die Anzahl der Schritte in der Ableitung von w von P.

Induktionsbeginn: Wenn die Ableitung aus einem Schritt besteht, dann muss sie eine der drei Produktionen verwenden, die kein P im Rumpf enthalten. Das heißt, die Ableitung lautet $P \to \varepsilon$, $P \to 0$ oder $P \to 1$. Da ε, 0 und 1 Palindrome sind, ist die Basisannahme bewiesen.

Induktionsschritt: Angenommen, die Ableitung erfordere $n + 1$ Schritte, ($n \geq 1$) und die Aussage sei wahr für alle Ableitungen mit n Schritten. Das heißt, wenn $P \overset{*}{\Rightarrow} x$ in n Schritten, dann ist x ein Palindrom.

Betrachten Sie eine Ableitung von w in $(n + 1)$ Schritten, die die Form

$$P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$$

oder $P \Rightarrow 1P1 \overset{*}{\Rightarrow} 1x1 = w$ haben muss, da $n + 1$ Schritte mindestens zwei Schritte sein müssen und $P \to 0P0$ sowie $P \to 1P1$ die einzigen beiden Produktionen sind, deren Verwendung mehr als einen Ableitungsschritt zulässt. Beachten Sie, dass in beiden Fällen in n Schritten $P \overset{*}{\Rightarrow} x$.

Nach der Induktionsannahme wissen wir, dass x ein Palindrom ist, d. h. $x = x^R$. Wenn dies wahr ist, dann sind auch $0x0$ und $1x1$ Palindrome. Beispielsweise $(0x0)^R = 0x^R0 = 0x0$. Also ist w ein Palindrom, womit der Beweis vollständig erbracht ist. ■

5.1.6 Satzformen

Ableitungen vom Startsymbol erzeugen Zeichenreihen, die eine besondere Rolle erfüllen. Wir nennen sie »Satzformen«. Das heißt, wenn $G = (V, T, P, S)$ eine kfG ist, dann ist jede Zeichenreihe α aus $(V \cup T)$, für die $S \overset{*}{\Rightarrow} \alpha$ gilt, eine Satzform. Wenn $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, dann ist α eine linksseitige Satzform, und wenn $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$, dann ist α eine rechtsseitige Satzform. Beachten Sie, dass die Sprache $L(G)$ die Satzformen umfasst, die in T enthalten sind; d. h. sie bestehen lediglich aus terminalen Symbolen.

Beispiel 5.8 Betrachten Sie die Grammatik für die Ausdrücke aus Tabelle 5.2. Beispielsweise ist $E * (I + E)$ eine Satzform, da es folgende Ableitung gibt:

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

Diese Ableitung ist aber weder links- noch rechtsseitig, da im letzten Schritt die mittlere Variable E ersetzt wird.

Betrachten Sie $a * E$ mit der linksseitigen Ableitung als Beispiel für eine linksseitige Satzform:

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

Als weiteres Beispiel zeigt die Ableitung

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$$

dass $E * (E + E)$ eine rechtsseitige Satzform ist. ■

Die Form von Beweisen über Grammatiken

Satz 5.7 ist typisch für Beweise, die zeigen, dass eine Grammatik eine bestimmte informell definierte Sprache definiert. Wir entwickeln zuerst eine Induktionsannahme, die aussagt, welche Eigenschaften die von den Variablen abgeleiteten Zeichenreihen haben. In diesem Beispiel gab es nur eine Variable P, und daher mussten wir nur annehmen, dass ihre Zeichenreihen Palindrome seien.

Wir beweisen den »Wenn«-Teil: Wenn eine Zeichenreihe w die informelle Aussage über die Zeichenreihen einer der Variablen A erfüllt, dann zeigen wir $A \overset{*}{\Rightarrow} w$. Für gewöhnlich beweisen wir den »Wenn«-Teil durch Induktion über die Länge von w. Wenn es k Variable gibt, dann umfasst die zu beweisende Induktionsaussage k Teile, die durch eine sich gegenseitig bedingende Induktion bewiesen werden müssen.

Wir müssen außerdem den »Genau-dann«-Teil beweisen, der besagt, wenn $A \overset{*}{\Rightarrow} w$, dann erfüllt w die informelle Aussage über die von der Variablen A abgeleiteten Zeichenreihen. Der Beweis dieses Teils besteht in der Regel in einem Induktionsbeweis über die Anzahl der Ableitungsschritte. Wenn die Grammatik über Produktionen verfügt, in denen zwei oder mehr Variablen in den abgeleiteten Zeichenreihen vorkommen, dann müssen wir eine Ableitung mit n Schritten in verschiedene Teile aufteilen, die jeweils den Ableitungen von den einzelnen Variablen entsprechen. Diese Ableitungen können weniger als n Schritte umfassen. Daher müssen wir einen Induktionsbeweis führen, in dem die Aussage für alle Werte kleiner gleich n vorausgesetzt wird, wie in Abschnitt 1.4.2 erörtert.

5.1.7 Übungen zum Abschnitt 5.1

Übung 5.1.1 Entwerfen Sie kontextfreie Grammatiken für die folgenden Sprachen:

***** a) Die Menge $\{0^n1^n \mid n \geq 1\}$, das heißt, die Menge aller Zeichenreihen, die aus einer oder mehr Nullen gefolgt von der gleichen Anzahl Einsen bestehen

***!** b) Die Menge $\{a^ib^jc^k \mid i \neq j \text{ oder } j \neq k\}$, das heißt, die Menge der Zeichenreihen, die sich aus Symbolen a gefolgt von Symbolen b gefolgt von Symbolen c zusammensetzen, wobei entweder die Anzahl der Symbole a und b oder die der Symbole b und c oder die aller drei Symbole unterschiedlich ist

! c) Die Menge aller Zeichenreihen, die aus den Symbolen a und b bestehen und für keine Zeichenreihe w die Form ww haben

!! d) Die Menge aller Zeichenreihen, die doppelt so viele Nullen wie Einsen enthalten

Übung 5.1.2 Die folgende Grammatik erzeugt die Sprache des regulären Ausdrucks $\mathbf{0}^*\mathbf{1}(\mathbf{0}+\mathbf{1})^*$:

$$\begin{aligned} \mathrm{S} &\to A1B \\ A &\to 0A \mid \varepsilon \\ \mathrm{B} &\to 0B \mid 1B \mid \varepsilon \end{aligned}$$

Geben Sie die linksseitige und die rechtsseitige Ableitung für die folgenden Zeichenreihen an:

a) 00101

b) 1001

c) 00011

! **Übung 5.1.3** Zeigen Sie, dass jede reguläre Sprache eine kontextfreie Sprache ist. *Hinweis*: Konstruieren Sie eine kfG durch Induktion über die Anzahl der Operatoren in dem regulären Ausdruck.

! **Übung 5.1.4** Eine kfG wird als rechtslinear bezeichnet, wenn jeder Produktionsrumpf höchstens eine Variable enthält und diese Variable sich am rechten Ende befindet. Das heißt, alle Produktionen einer rechtslinearen Grammatik haben die Form $A \to wB$ oder $A \to w$, wobei A und B für Variable und w für eine Zeichenreihe aus null oder mehr terminalen Symbolen stehen.

a) Zeigen Sie, dass jede rechtslineare Grammatik eine reguläre Sprache erzeugt. *Hinweis*: Konstruieren Sie einen ε-NEA, der linksseitige Ableitungen simuliert, indem jeweils ein Zustand die einzige Variable in der aktuellen linksseitigen Satzform repräsentiert.

b) Zeigen Sie, dass jede reguläre Sprache eine rechtslineare Grammatik besitzt. *Hinweis*: Beginnen Sie mit einem DEA und lassen Sie die Variablen der Grammatik Zustände repräsentieren.

*! **Übung 5.1.5** Sei $T = \{0, 1, (,), +, *, \emptyset, e\}$. Wir können uns T als die Menge der Symbole vorstellen, die in regulären Ausdrücken über dem Alphabet $\{0, 1\}$ verwendet werden. Der einzige Unterschied besteht darin, dass wir das Symbol e statt ε verwenden, um mögliche Verwechslungen in dem Folgenden zu vermeiden. Ihre Aufgabe besteht darin, eine kfG über der Menge der terminalen Symbole T zu entwerfen, die genau die regulären Ausdrücke über dem Alphabet $\{0, 1\}$ erzeugt.

Übung 5.1.6 Wir haben die Relation $\stackrel{*}{\Rightarrow}$ mit der Basis »$\alpha \Rightarrow \alpha$« und dem Induktionsschritt definiert, der besagt, »$\alpha \stackrel{*}{\Rightarrow} \beta$ und $\beta \Rightarrow \gamma$ impliziert $\alpha \stackrel{*}{\Rightarrow} \gamma$«. Es gibt verschiedene andere Möglichkeiten der Definition von $\stackrel{*}{\Rightarrow}$, die ebenso aussagen: »$\stackrel{*}{\Rightarrow}$ steht für null oder mehr $\Rightarrow$ Schritte«. Beweisen Sie, dass die folgenden Aussagen wahr sind:

a) $\alpha \stackrel{*}{\Rightarrow} \beta$ gilt genau dann, wenn es eine Folge von einer oder mehreren Zeichenreihen

$$\gamma_1, \gamma_2, \ldots, \gamma_n$$

gibt, derart dass $\alpha = \gamma_1$, $\beta = \gamma_n$ und für $i = 1, 2, \ldots, n-1$ $\gamma_i \Rightarrow \gamma_{i+1}$ gilt.

b) Wenn $\alpha \stackrel{*}{\Rightarrow} \beta$ und $\beta \stackrel{*}{\Rightarrow} \gamma$, dann $\alpha \stackrel{*}{\Rightarrow} \gamma$. *Hinweis*: Führen Sie einen Induktionsbeweis über die Anzahl der Schritte in der Ableitung $\beta \stackrel{*}{\Rightarrow} \gamma$.

! Übung 5.1.7 Betrachten Sie die kfG G, die durch folgende Produktionen definiert wird:

$$S \rightarrow aS \mid Sb \mid a \mid b$$

a) Beweisen Sie durch Induktion über der Zeichenreihenlänge, dass keine Zeichenreihe in $L(G)$ die Teilzeichenreihe ba enthält.

b) Beschreiben Sie die Sprache $L(G)$ informell. Begründen Sie Ihre Anwort mithilfe von Teil (a).

!! Übung 5.1.8 Betrachten Sie die kfG G, die durch folgende Produktionen definiert wird:

$$S \rightarrow aSbS \mid bSaS \mid \varepsilon$$

Beweisen Sie, dass $L(G)$ die Menge aller Zeichenreihen beschreibt, die aus gleich vielen Symbolen a und b bestehen.

5.2 Parsebäume

Es gibt eine Baumdarstellung von Ableitungen, die sich als extrem nützlich erwiesen hat. Dieser Baum veranschaulicht genau, wie die Symbole der terminalen Zeichenreihen in Teilzeichenreihen gruppiert werden, die jeweils in der Sprache einer der Variablen der Grammatik enthalten sind. Wichtiger ist vielleicht sogar noch, dass dieser Baum, der bei Anwendung in einem Compiler als »Parsebaum« bezeichnet wird, dort die bevorzugte Datenstruktur zur Repräsentation von Quelltextprogrammen ist. In einem Compiler erleichtert die Baumstruktur des Quelltextprogramms dessen Übersetzung in ausführbaren Code, da dann natürliche, rekursive Funktionen diese Übersetzung ausführen können.

In diesem Abschnitt stellen wir den Parsebaum vor und zeigen, dass Parsebäume in einer engen Beziehung zu Ableitungen und rekursiven Inferenzen stehen. Wir werden später das Phänomen der Mehrdeutigkeit von Grammatiken und Sprachen studieren, das eine wichtige Anwendung der Parsebäume darstellt. Manche Grammatiken lassen zu, dass gewisse terminale Zeichenreihen mehr als einen Parsebaum haben. Eine solche Grammatik ist für Programmiersprachen ungeeignet, weil der Compiler dann nicht mit Sicherheit entscheiden kann, wie der korrekte ausführbare Code für gewisse Programme lautet.

5.2.1 Parsebäume aufbauen

Wir wollen uns im Folgenden auf die Grammatik $G = (V, T, P, S)$ konzentrieren. Bei den *Parsebäumen* für G handelt es sich um Bäume, die die folgenden Bedingungen erfüllen:

1. Jeder innere Knoten ist mit einer in V enthaltenen Variablen beschriftet.
2. Jedes Blatt ist entweder mit einer Variablen, einem terminalen Symbol oder mit ε beschriftet. Falls das Blatt mit ε beschriftet ist, dann muss es der einzige Nachfolger seines Vorgängerknotens sein.
3. Wenn ein innerer Knoten die Beschriftung A und seine Nachfolgerknoten von links nach rechts gesehen die Beschriftungen

$$X_1, X_2, ..., X_k$$

tragen, dann ist $A \to X_1 X_2 ... X_k$ eine in P enthaltene Produktion. Beachten Sie, dass eines der Symbole X nur dann ε sein kann, wenn es die Beschriftung eines einzelnen Nachfolgers von A darstellt und $A \to \varepsilon$ eine Produktion von G ist.

Terminologie im Zusammenhang mit Bäumen

Wir gehen davon aus, dass Sie das Konzept der Baumstruktur kennen und mit den allgemein gebräuchlichen Definitionen in Bezug auf Bäume vertraut sind. Ungeachtet dessen fassen wir diese im Folgenden noch einmal zusammen:

- Bäume sind Sammlungen von *Knoten*, die einander *über- und untergeordnet* sind bzw. eine so genannte *Vorgänger-Nachfolger*-Beziehung haben. Ein Knoten hat höchstens einen Vorgänger, der über dem Knoten dargestellt wird, und null oder mehr Nachfolger, die unterhalb des Knotens dargestellt werden. Über- und untergeordnete Knoten werden durch Linien miteinander verbunden. Abbildung 5.1, Abbildung 5.2 und Abbildung 5.3 sind Beispiele für Bäume.
- Es gibt einen Knoten, die so genannte *Wurzel*, dem kein Knoten übergeordnet ist. Dieser Knoten befindet sich an der Spitze des Baums. Knoten ohne untergeordnete Knoten werden als *Blätter* bezeichnet. Knoten, die keine Blattknoten sind, werden *innere Knoten* genannt.
- Der Nachfolgerknoten eines Nachfolgerknotens ... eines Knotens ist ein *Nachkomme* dieses Knotens. Der Vorgängerknoten eines Vorgängerknotens ... eines Knotens ist ein Vorfahre dieses Knotens. Trivialerweise sind Knoten Vorfahren und Nachkommen ihrer selbst.
- Die untergeordneten Knoten eines Knotens werden »von links nach rechts« angeordnet und so gezeichnet. Wenn sich Knoten *N* links von Knoten *M* befindet, dann werden alle Nachkommen von Knoten *N* als links von den Nachkommen von Knoten *M* befindlich betrachtet.

Beispiel 5.9 Abbildung 5.1 zeigt einen Parsebaum für die Grammatik einfacher Ausdrücke aus Tabelle 5.2. Die Wurzel ist mit der Variablen E beschriftet. Wir sehen, dass an der Wurzel die Produktion $E \to E + E$ verwendet wird, da die untergeordneten Knoten von links nach rechts gesehen die Beschriftungen E, + und E tragen. Am äußersten linken Nachfolger der Wurzel wird die Produktion $E \to I$ verwendet, da dieser Knoten einen untergeordneten Knoten mit der Beschriftung I besitzt. ■

Abbildung 5.1: Parsebaum, der darstellt, wie *I* + *E* von *E* abgeleitet wird

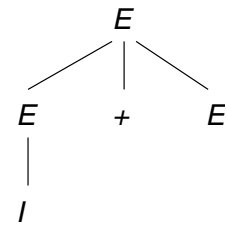
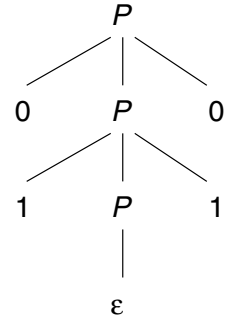


Abbildung 5.2: Parsebaum, der die Ableitung P $\stackrel{*}{\Rightarrow}$ 0110 darstellt



Beispiel 5.10 Abbildung 5.2 zeigt einen Parsebaum für die Grammatik der Palindrome aus Tabelle 5.1. An der Wurzel wird die Produktion $P \to 0P0$ und am mittleren untergeordneten Knoten die Produktion $P \to 1P1$ verwendet. Beachten Sie, dass am Ende die Produktion $P \to \varepsilon$ eingesetzt wird. Ein mit ε beschrifteter Knoten kann in einem Parsebaum nur an dieser Stelle vorkommen, an der der mit dem Produktionskopf beschriftete Knoten nur einen untergeordneten Knoten (mit der Beschriftung ε) besitzt. ■

5.2.2 Der Ergebnis eines Parsebaums

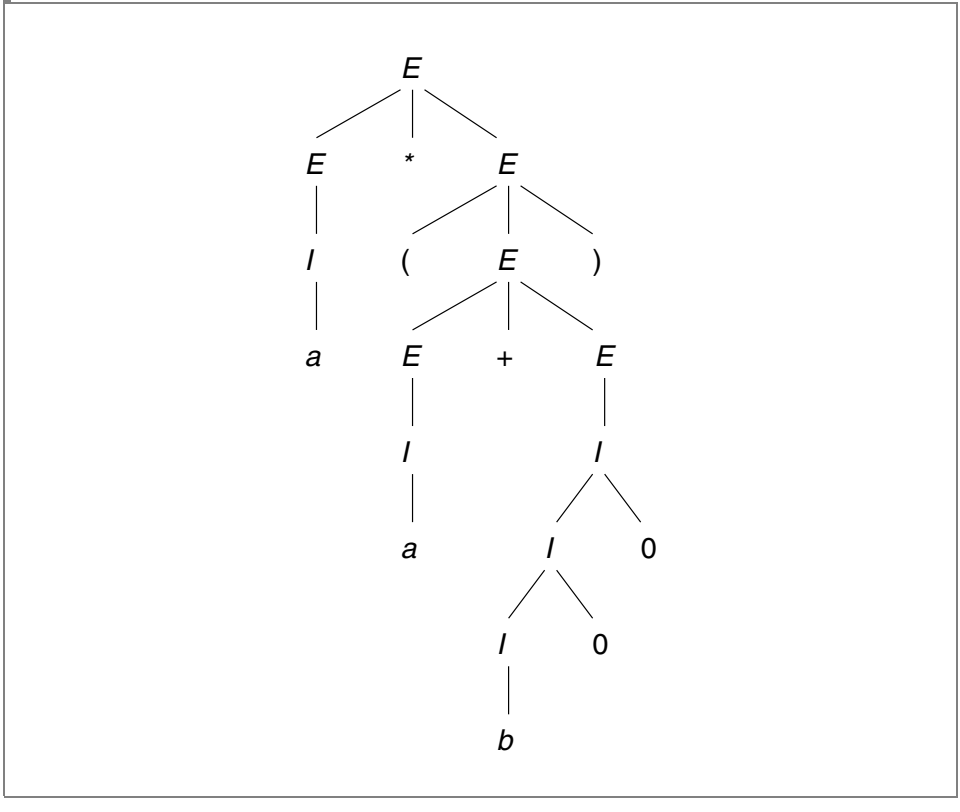
Wenn wir die Blätter eines Parsebaums betrachten und sie von links nach rechts verketten, dann erhalten wir eine Zeichenreihe, die als *Ergebnis* des Parsebaums bezeichnet wird und stets eine Zeichenreihe ist, die von der Variablen an der Wurzel abgeleitet wird. Die Tatsache, dass das Ergebnis von der Wurzel abgeleitet wird, wird in Kürze bewiesen. Von besonderer Bedeutung sind jene Parsebäume, für die gilt:

1. Das Ergebnis ist eine terminale Zeichenreihe. Das heißt, alle Blätter sind entweder mit einem terminalen Symbol oder mit ε beschriftet.
2. Die Wurzel ist mit dem Startsymbol beschriftet.

Dies sind die Parsebäume, deren Ergebnisse Zeichenreihen aus der Sprache der zu Grunde liegenden Grammatik sind. Wir werden zudem beweisen, dass die Sprache einer Grammatik auch beschrieben werden kann als Menge der Ergebnisse der Parsebäume, deren Wurzel das Startsymbol und deren Ergebnis eine terminale Zeichenreihe ist.

Beispiel 5.11 Abbildung 5.3 ist ein Beispiel für einen Baum mit einer terminalen Zeichenreihe als Ergebnis und dem Startsymbol als Wurzel. Dieser Baum basiert auf der Grammatik für einfache Ausdrücke, die wir in Beispiel 5.5 vorgestellt haben. Ergebnis dieses Baums ist die Zeichenreihe $a * (a + b00)$, die wir im Beispiel 5.5 abgeleitet haben. Wie wir später sehen werden, ist dieser spezielle Parsebaum eine Repräsentation dieser Ableitung. ■

Abbildung 5.3: Parsebaum, der zeigt, dass $a * (a + b00)$ in der Sprache unserer Grammatik für Ausdrücke enthalten ist



5.2.3 Inferenz, Ableitungen und Parsebäume

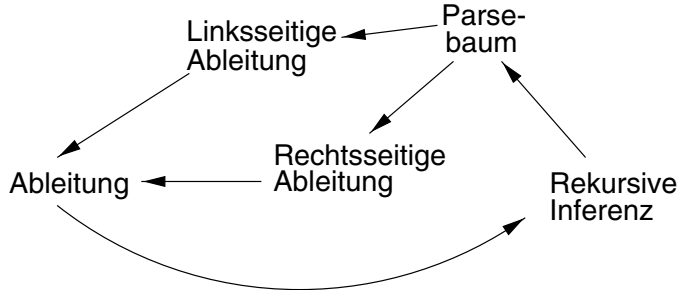
Jedes der bislang vorgestellten verschiedenen Konzepte zur Beschreibung einer Grammatik liefert im Grunde genommen dieselben Fakten über Zeichenreihen. Das heißt, wenn eine Grammatik $G = (V, T, P, S)$ gegeben ist, werden wir zeigen, dass die folgenden Aussagen äquivalent sind:

1. Das rekursive Inferenzverfahren legt fest, dass die terminale Zeichenreihe w in der Sprache einer Variablen A enthalten ist.
2. $A \stackrel{*}{\Rightarrow} w$.
3. $A \stackrel{*}{\underset{lm}{\Rightarrow}} w$.
4. $A \stackrel{*}{\underset{rm}{\Rightarrow}} w$.
5. Es gibt einen Parsebaum mit der Wurzel A, der w ergibt.

Abgesehen von der rekursiven Inferenz, die wir lediglich für terminale Zeichenreihen definiert haben, sind alle anderen Bedingungen – die Existenz von Ableitungen, links- oder rechtsseitigen Ableitungen und Parsebäumen – auch dann äquivalent, wenn w eine Zeichenreihe ist, die auch Variable enthält.

Wir müssen diese Äquivalenzen beweisen und tun dies nach der in Abbildung 5.4 dargestellten Strategie. Das heißt, jeder Pfeil in diesem Diagramm zeigt an, dass wir einen Satz beweisen, der besagt, dass die Zeichenreihe w die Bedingung an der Spitze des Pfeils erfüllt, wenn sie der Bedingung am Anfang des Pfeils genügt. Beispielsweise werden wir in Satz 5.12 zeigen, dass es einen Parsebaum mit der Wurzel A und dem Ergebnis w gibt, wenn durch rekursive Inferenz gezeigt werden kann, dass w in der Sprache von A enthalten ist.

Abbildung 5.4: Beweis der Äquivalenz bestimmter Aussagen über Grammatiken



Beachten Sie, dass zwei dieser Pfeile sehr einfach sind und nicht formal bewiesen werden. Wenn w eine linksseitige Ableitung von A besitzt, dann gibt es für w mit Sicherheit eine Ableitung von A, da eine linksseitige Ableitung eine Ableitung *ist*. Analog gilt, wenn w eine rechtsseitige Ableitung von A besitzt, dann gibt es mit Sicherheit eine Ableitung. Wir werden nun die schwierigeren Schritte dieser Äqivalenz beweisen.

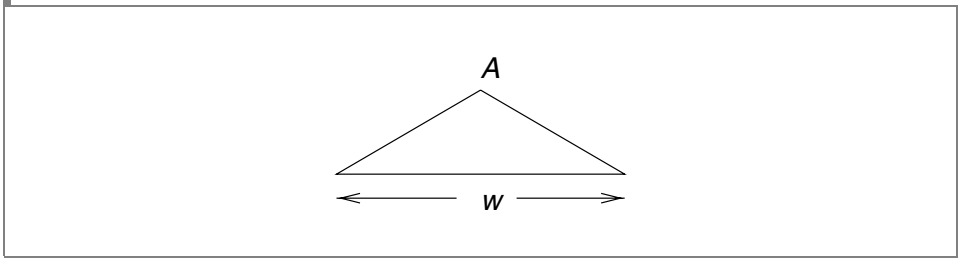
5.2.4 Von Inferenzen zu Bäumen

Satz 5.12 Sei $G = (V, T, P, S)$ eine kfG. Wenn aus dem rekursiven Inferenzverfahren hervorgeht, dass die terminale Zeichenreihe w in der Sprache der Variablen A enthalten ist, dann gibt es einen Parsebaum mit der Wurzel A und dem Ergebnis w.

BEWEIS: Der Beweis besteht aus einer Induktion über die Anzahl von Schritten, die zur Inferenz erforderlich sind, dass die Zeichenreihe w in der Sprache von A enthalten ist.

INDUKTIONSBEGINN: Ein Schritt. In diesem Fall darf nur die Basis des Inferenzverfahrens zum Einsatz gekommen sein. Folglich muss es eine Produktion $A \rightarrow w$ geben. Der in Abbildung 5.5 dargestellte Baum, der an jeder Position für w lediglich über einen Blattknoten verfügt, erfüllt die Bedingung, ein Parsebaum für die Grammatik G zu sein, da er offensichtlich das Ergebnis w und die Wurzel A hat. In dem speziellen Fall, dass $w = \varepsilon$, besitzt der Baum lediglich einen Blattknoten mit der Beschriftung ε und stellt einen zulässigen Parsebaum mit der Wurzel A und dem Ergebnis w dar.

Abbildung 5.5: Der konstruierte Basisfall von Satz 5.12



INDUKTIONSSCHRITT: Angenommen, nach $n + 1$ Inferenzschritten würde auf die Tatsache geschlossen, dass w in der Sprache von A enthalten ist, und die Aussage des Satzes gelte für alle Zeichenreihen x und alle Variablen B, derart dass nach n oder weniger Inferenzschritten auf die Zugehörigkeit von x zur Sprache von B geschlossen wird. Betrachten Sie den letzten Schritt der Inferenz, dass w in der Sprache von A enthalten ist. Diese Inferenz erfordert eine Produktion für A der Form $A \rightarrow X_1X_2 \ldots X_k$, wobei jedes X_i entweder für ein terminales Symbol oder eine Variable steht.

Wir können w aufspalten in $w_1w_2 \ldots w_k$, wobei gilt:

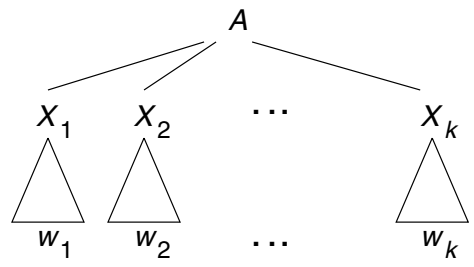
1. Wenn X_i ein terminales Symbol ist, dann ist $w_i = X_i$, d. h. w_i besteht lediglich aus diesem einen terminalen Symbol der Produktion.

2. Ist X_i eine Variable, dann ist w_i eine Zeichenreihe, deren Zugehörigkeit zur Sprache von X zuvor durch das Inferenzverfahren ermittelt wurde. Das heißt, diese Inferenz über w_i erforderte höchstens n der $n + 1$ Schritte, weil der letzte Schritt (die Anwendung der Produktion $A \rightarrow X_1X_2 \ldots X_k$) sicher nicht Teil der Inferenz über w_i ist. Infolgedessen können wir die Induktionsannahme auf w_i und X_i anwenden und darauf schließen, dass es einen Parsebaum mit dem Ergebnis w_i und der Wurzel X_i gibt.

Wir konstruieren dann einen Baum mit der Wurzel A und dem Ergebnis w, wie in Abbildung 5.6 dargestellt. Der Baum besitzt die Wurzel A, der die Knoten $X_1,X_2, \ldots, X_k$ untergeordnet sind. Diese Wahl ist zulässig, da $A \rightarrow X_1X_2 \ldots X_k$ eine Produktion von G ist.

Der Knoten jedes X_i bildet die Wurzel eines Teilbaums, der w_i ergibt. Im Fall (1), in dem X_i ein terminales Symbol ist, handelt es sich bei diesem Teilbaum um einen einfachen Baum mit einem einzigen Knoten mit der Beschriftung X_i. Das heißt, der Teilbaum besteht aus genau diesem einen untergeordneten Knoten der Wurzel. Da im Fall (1) $w_i = X_i$, ist die Bedingung erfüllt, dass w_i das Ergebnis des Teilbaums ist.

Abbildung 5.6: Baum, der im Induktionsschritt des Beweises von Satz 5.12 verwendet wird



Im Fall (2) ist X_i eine Variable. Wir fordern hier unter Berufung auf die Induktionsannahme, dass es einen Baum mit der Wurzel X_i und dem Ergebnis w_i gibt. In Abbildung 5.6 ist dieser Baum jeweils unterhalb des Knotens von X_i dargestellt.

Der auf diese Weise konstruierte Baum besitzt die Wurzel A. Sein Ergebnis wird gebildet, indem die Ergebnisse der Teilbäume von links nach rechts verkettet werden. Die resultierende Zeichenreihe $w_1w_2 \ldots w_k$ ist w. ■

5.2.5 Von Bäumen zu Ableitungen

Wir werden nun zeigen, wie man aus einem Parsebaum eine linksseitige Ableitung konstruiert. Da die Methode zur Konstruktion einer rechtsseitigen Ableitung auf den gleichen Konzepten basiert, werden wir diesen Fall nicht behandeln. Zum Verständnis der Konstruktion von Ableitungen müssen wir zuerst erläutern, wie die Ableitung einer Zeichenreihe aus einer Variablen in eine andere Ableitung eingebettet werden kann. Ein Beispiel soll dies veranschaulichen.

Beispiel 5.13 Lassen Sie uns wieder die in Tabelle 5.2 beschriebene Grammatik für einfache Ausdrücke betrachten. Es lässt sich leicht überprüfen, dass es folgende Ableitung gibt:

$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab$$

Infolgedessen ist für beliebige Zeichenreihen α und β auch wahr, dass

$$\alpha E\beta \Rightarrow \alpha I\beta \Rightarrow \alpha Ib\beta \Rightarrow \alpha ab\beta$$

Begründen lässt sich dies damit, dass wir im Kontext von α und β dieselben Ersetzungen von Produktionsköpfen durch Produktionsrümpfe vornehmen können, wie wir es unabhängig von diesen Zeichenreihen können[1].

Wenn beispielsweise eine Ableitung vorliegt, die mit $E \Rightarrow E + E \Rightarrow E + (E)$ beginnt, dann könnten wir die Ableitung von ab aus der zweiten Variablen E durchführen,

1. In der Tat hat diese Eigenschaft, dass die Ersetzung einer Variablen durch eine Zeichenreihe unabhängig vom Kontext erfolgen kann, zu dem Begriff »kontextfrei« geführt. Es gibt eine mächtigere Klasse von Grammatiken, die so genannten »kontextabhängigen« Grammatiken, bei der Ersetzungen nur dann zulässig sind, wenn bestimmte Zeichenreihen auf der linken und/oder rechten Seite stehen. Kontextabhängige Grammatiken spielen heute in der Praxis allerdings keine große Rolle.

indem wir »E + (« als α und »)« als β behandeln. Diese Ableitung würde dann wie folgt fortgesetzt:

$$E + (E) \Rightarrow E + (I) \Rightarrow E + (Ib) \Rightarrow E + (ab)$$ ■

Wir sind nun in der Lage, einen Satz zu beweisen, nach dem wir einen Parsebaum in eine linksseitige Ableitung umwandeln können. Der Beweis besteht aus einer Induktion über die *Höhe* des Baums, womit die maximale Länge aller Pfade gemeint ist, die von der Wurzel zu einem Blatt führen. Beispielsweise hat der in Abbildung 5.3 dargestellte Baum die Höhe 7. In diesem Baum führt der längste Pfad von der Wurzel zu dem Blatt mit der Beschriftung b. Beachten Sie, dass die Pfadlänge üblicherweise als Anzahl der Kanten und nicht der Knoten angegeben wird und dass daher ein Pfad, der aus einem einzigen Knoten besteht, die Länge 0 hat.

Satz 5.14 Sei $G = (V, T, P, S)$ eine kfG, und nehmen wir an, es gäbe einen Parsebaum mit einer Wurzel, die mit der Variablen A beschriftet ist, und dem Ergebnis w, wobei w in T^* enthalten ist. Dann gibt es eine linksseitige Ableitung $A \overset{*}{\underset{lm}{\Rightarrow}} w$ in der Grammatik G.

Beweis: Wir führen einen Induktionsbeweis über die Höhe des Baums.

Induktionsbeginn: Der Basisfall ist Höhe 1, also die geringste Höhe, die ein Parsebaum mit einer terminalen Zeichenreihe als Ergebnis haben kann. In diesem Fall muss der Baum wie in Abbildung 5.5 aussehen, wobei die Beschriftung der Wurzel A und die der untergeordneten Knoten von links nach rechts gelesen w lautet. Da es sich um einen Parsebaum handelt, muss es eine Produktion $A \rightarrow w$ geben. Folglich ist $A \overset{*}{\underset{lm}{\Rightarrow}} w$ eine einen Schritt umfassende linksseitige Ableitung von w aus A.

Induktionsschritt: Wenn die Höhe des Baums n beträgt, wobei $n > 1$, dann muss der Baum wie in Abbildung 5.6 aussehen. Das heißt, es gibt eine Wurzel mit der Beschriftung A und Nachfolgerknoten, die von links nach rechts mit den Bezeichnungen X_1, X_2, ..., X_k beschriftet sind, wobei X_i für ein terminales Zeichen oder eine Variable stehen kann.

1. Wenn X_i ein terminales Zeichen ist, dann ist w_i als Zeichenreihe zu definieren, die ausschließlich aus X_i besteht.
2. Ist X_i eine Variable, dann muss diese die Wurzel eines Teilbaums bilden, der eine terminale Zeichenreihe zum Ergebnis hat, die wir als w_i bezeichnen. Beachten Sie, dass der Teilbaum in diesem Fall eine geringere Höhe als n hat, sodass die Induktionsannahme gilt. Das heißt, es gibt eine linksseitige Ableitung $X_i \overset{*}{\underset{lm}{\Rightarrow}} w_i$.

Beachten Sie, dass $w = w_1 w_2 \ldots w_k$.

Wir konstruieren wie folgt eine linksseitige Ableitung von w. Wir beginnen mit dem Schritt $A \underset{lm}{\Rightarrow} X_1 X_2 \ldots X_k$. Dann zeigen wir nacheinander für jedes $i = 1, 2, \ldots, k$, dass gilt

$$A \overset{*}{\underset{lm}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

Dieser Beweis besteht eigentlich aus einer weiteren Induktion über i. Wir wissen bereits, dass für die Basis $i = 0$, $A \underset{lm}{\Rightarrow} X_1X_2 \ldots X_k$ gilt. Im Induktionsschritt nehmen wir an, dass

$$A \overset{*}{\underset{lm}{\Rightarrow}} w_1w_2 \cdots w_{i-1}X_iX_{i+1} \cdots X_k$$

a) Wenn X_i ein terminales Zeichen ist, dann ist nichts zu tun. Wir können X_i jedoch im Folgenden als terminale Zeichenreihe w_i betrachten. Daher haben wir bereits

$$A \overset{*}{\underset{lm}{\Rightarrow}} w_1w_2 \cdots w_iX_{i+1}X_{i+2} \cdots X_k$$

b) Wenn X_i eine Variable ist, dann fahren wir mit der Ableitung von w_i aus X_i im Kontext der zu konstruierenden Ableitung fort. Das heißt, wenn die Ableitung wie folgt lautet

$$X_i \underset{lm}{\Rightarrow} \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \cdots \underset{lm}{\Rightarrow} w_i$$

fahren wir fort mit

$$\begin{aligned}
&w_1w_2 \cdots w_{i-1}X_iX_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&w_1w_2 \cdots w_{i-1}\alpha_1X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&w_1w_2 \cdots w_{i-1}\alpha_2X_{i+1} \cdots X_k \underset{lm}{\Rightarrow} \\
&\cdots \\
&w_1w_2 \cdots w_iX_{i+1}X_{i+2} \cdots X_k
\end{aligned}$$

Das Ergebnis ist die Ableitung $A \overset{*}{\underset{lm}{\Rightarrow}} w_1w_2 \ldots w_iX_{i+1} \ldots X_k$.

Sobald $i = k$, besteht das Ergebnis aus einer linksseitigen Ableitung von w aus A. ■

Beispiel 5.15 Wir wollen nun die linksseitige Ableitung für den in Abbildung 5.3 dargestellten Baum konstruieren. Wir werden nur den letzten Schritt zeigen, in dem wir die Ableitung aus dem gesamten Baum aus Ableitungen konstruieren, die den Teilbäumen der Wurzel entsprechen. Das heißt, wir werden voraussetzen, dass wir aus der rekursiven Anwendung der in Satz 5.14 beschriebenen Technik gefolgert haben, dass der Teilbaum, dessen Wurzel der erste Nachfolger des Wurzelknotens ist, die linksseitige Ableitung $E \underset{lm}{\Rightarrow} I \underset{lm}{\Rightarrow} a$ besitzt, während der Teilbaum, dessen Wurzel der dritte Nachfolger der Wurzel ist, folgende linksseitige Ableitung besitzt:

$$\begin{aligned}
&E \underset{lm}{\Rightarrow} (E) \underset{lm}{\Rightarrow} (E+E) \underset{lm}{\Rightarrow} (I+E) \underset{lm}{\Rightarrow} (a+E) \underset{lm}{\Rightarrow} \\
&(a+I) \underset{lm}{\Rightarrow} (a+I0) \underset{lm}{\Rightarrow} (a+I00) \underset{lm}{\Rightarrow} (a+b00)
\end{aligned}$$

Zum Aufbau einer linksseitigen Ableitung für den gesamten Baum beginnen wir mit dem Schritt an der Wurzel $A \underset{lm}{\Rightarrow} E * E$. Dann ersetzen wir die erste Variable E entsprechend ihrer Ableitung, wobei wir jedem Schritt $*E$ nachstellen, um dem allgemeineren

Kontext dieser Ableitung Rechnung zu tragen. Die linksseitige Ableitung lautet nach diesem Schritt:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E$$

Da der Stern (*) in der Produktion, die an der Wurzel verwendet wird, keine Ableitung erfordert, berücksichtigt die obige linksseitige Ableitung bereits die ersten beiden Nachfolger der Wurzel. Wir vervollständigen die linksseitige Ableitung, indem wir die Ableitung von $E \overset{*}{\underset{lm}{\Rightarrow}} (a + b00)$ in einem Kontext verwenden, in dem $a*$ vorangestellt und ε nachgestellt wird. Diese Ableitung kam bereits in Beispiel 5.6 vor; sie lautet:

$$\begin{aligned}
&E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E \underset{lm}{\Rightarrow} \\
&a*(E) \underset{lm}{\Rightarrow} a*(E+E) \underset{lm}{\Rightarrow} a*(I+E) \underset{lm}{\Rightarrow} a*(a+E) \underset{lm}{\Rightarrow} \\
&a*(a+I) \underset{lm}{\Rightarrow} a*(a+I0) \underset{lm}{\Rightarrow} a*(a+I00) \underset{lm}{\Rightarrow} a*(a+b00)
\end{aligned}$$

Ein analoger Satz lässt sich für die Konstruktion einer rechtsseitigen Ableitung aus einem Parsebaum beweisen. Damit haben wir, ohne diesen Beweis auszuführen, den folgenden

Satz 5.16 Sei $G = (V, T, P, S)$ eine kfG, und nehmen wir an, es gäbe einen Parsebaum, dessen Wurzel mit der Variablen A beschriftet ist und der die Zeichenreihe w ergibt, wobei w in T^* enthalten ist; dann gibt es in der Grammatik G eine rechtsseitige Ableitung $A \overset{*}{\underset{rm}{\Rightarrow}} w$.

5.2.6 Von Ableitungen zu rekursiven Inferenzen

Wir schließen nun den in Abbildung 5.4 dargestellten Kreis, indem wir zeigen, dass in allen Fällen, in denen es für eine kontextfreie Grammatik eine Ableitung $A \overset{*}{\Rightarrow} w$ gibt, mit dem rekursiven Inferenzverfahren feststellbar ist, dass w in der Sprache von A enthalten ist. Bevor wir den Satz und den Beweis angeben, wollen wir eine wichtige Beobachtung über Ableitungen festhalten.

Angenommen, es gibt die Ableitung $A \Rightarrow X_1X_2 \ldots X_k \overset{*}{\Rightarrow} w$. Dann können wir w in die Teilstücke $w = w_1w_2 \ldots w_k$ aufspalten, derart dass $X_i \overset{*}{\Rightarrow} w_i$. Zu beachten ist, wenn X_i ein terminales Symbol ist, dann ist $w_i = X_i$ und die Ableitung umfasst null Schritte. Diese Beobachtung ist nicht schwer zu beweisen. Man kann durch Induktion über die Anzahl der Ableitungsschritte zeigen, dass gilt: Wenn $X_1X_2 \ldots X_k \overset{*}{\Rightarrow} \alpha$; dann befinden sich alle Positionen von α, die sich aus der Ableitung aus X_i ergeben, links von all den Positionen, die sich aus der Ableitung aus X_j ergeben, wenn $i < j$.

Wenn X_i eine Variable ist, dann erhalten wir die Ableitung $X_i \overset{*}{\Rightarrow} w_i$, indem wir mit der Ableitung $A \overset{*}{\Rightarrow} w$ beginnen und nacheinander Folgendes eliminieren:

a) Alle Positionen aus den Satzformen, die sich entweder links oder rechts von den von X_i abgeleiteten Positionen befinden

b) Alle Schritte, die für die Ableitung von w_i aus X_i nicht relevant sind.

Ein Beispiel soll dieses Verfahren verdeutlichen.

Beispiel 5.17 Betrachten Sie die folgende Ableitung unter Verwendung der Grammatik aus Tabelle 5.2:

$$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow I * E + E \Rightarrow I * I + E \Rightarrow$$
$$I * I + I \Rightarrow a * I + I \Rightarrow a * b + I \Rightarrow a * b + a$$

Betrachten Sie die dritte Satzform $E * E + E$ und das mittlere E in dieser Form[2].

Wenn wir von $E * E + E$ ausgehen, können wir die Schritte der obigen Ableitung nachvollziehen, wobei wir jedoch die Positionen löschen, die vom Ausdruck $E *$ links vom mittleren E oder vom Ausdruck $+ E$ rechts davon abgeleitet wurden. Die Schritte der Ableitung lauten dann $E, E, I, I, I\ b, b$. Das heißt, im nächsten Schritt wird das mittlere E nicht geändert, im nachfolgenden Schritt wird es in I geändert, in den nächsten beiden Schritten bleibt das I unverändert, im anschließenden Schritt wird es in b geändert, und im letzten Schritt wird nichts mehr verändert, was vom mittleren E abgeleitet wurde.

Wenn wir nur die Schritte nehmen, die das Ergebnis der Ableitung vom mittleren E ändern, dann wird aus der Zeichenreihensequenz $E, E, I, I, I\ b, b$ die Ableitung $E \Rightarrow I \Rightarrow b$. Diese Ableitung beschreibt korrekt, wie das mittlere E während der gesamten Ableitung umgeformt wird. ■

Satz 5.18 Sei $G = (V, T, P, S)$ eine kfG, und nehmen wir an, es gibt eine Ableitung $A \overset{*}{\underset{G}{\Rightarrow}} w$, wobei w in T^* enthalten ist. Dann stellt die Anwendung des rekursiven Inferenzverfahrens auf G fest, dass w in der Sprache der Variablen A enthalten ist.

Beweis: Der Beweis ist eine Induktion über die Länge der Ableitung $A \overset{*}{\Rightarrow} w$.

Induktionsbeginn: Wenn die Ableitung nur einen Schritt umfasst, dann muss $A \rightarrow w$ eine Produktion sein. Da w ausschließlich aus terminalen Symbolen besteht, wird die Tatsache, dass w in der Sprache von A enthalten ist, im Basisteil der rekursiven Inferenz entdeckt.

Induktionsschritt: Angenommen, die Ableitung erfordere $n + 1$ Schritte, und nehmen wir weiter an, die Aussage gelte für jede Ableitung mit n oder weniger Schritten. Wir geben die Ableitung in der Form $A \Rightarrow X_1X_2 \ldots X_k \overset{*}{\Rightarrow} w$ an. Dann können wir, wie weiter oben erörtert, w aufspalten in $w = w_1w_2 \ldots w_k$, wobei gilt:

a) Wenn X_i ein terminales Symbol ist, dann ist $w_i = X_i$.

b) Wenn X_i eine Variable ist, dann ist $X_i \overset{*}{\Rightarrow} w_i$. Da der erste Schritt der Ableitung $A \overset{*}{\Rightarrow} w$ sicherlich nicht Teil der Ableitung $X_i \overset{*}{\Rightarrow} w_i$ ist, wissen wir, dass diese Ableitung n oder weniger Schritte umfasst. Folglich ist die Induktionsannahme anwendbar, und wir wissen, dass durch Inferenz auf die Zugehörigkeit von w_i zur Sprache von X_i geschlossen wird.

Wir verfügen nun über die Produktion $A \rightarrow X_1X_2 \ldots X_k$, wobei w_i entweder gleich X_i oder bekanntermaßen in der Sprache von X_i enthalten ist. Da $w = w_1w_2 \ldots w_k$, haben wir gezeigt, dass durch Inferenz auf die Zugehörigkeit von w zur Sprache von A geschlossen wird. ■

2. Wir sind in unserer Erörterung der Suche nach untergeordneten Ableitungen von umfangreicheren Ableitungen davon ausgegangen, dass wir eine Variable in der zweiten Satzform einer Ableitung bearbeiten würden. Das Konzept lässt sich jedoch auf Variable in jedem Ableitungsschritt übertragen.

5.2.7 Übungen zum Abschnitt 5.2

Übung 5.2.1 Zeichnen Sie für die Grammatik und für die einzelnen Zeichenreihen aus Übung 5.1.2 Parsebäume.

! **Übung 5.2.2** Angenommen, in G sei eine kfG ohne Produktionen mit ε als Rumpf. Wenn w in $L(G)$ enthalten ist, die Länge n hat und w über eine Ableitung in m Schritten erzeugt wird, zeigen Sie, dass w einen Parsebaum mit $n + m$ Knoten besitzt.

! **Übung 5.2.3** Es sollen dieselben Voraussetzungen wie für Übung 5.2.2 gelten, aber wir lassen zu, dass G Produktionen mit ε als Rumpf enthält. Zeigen Sie, dass ein Parsebaum für $w \neq \varepsilon$ höchstens $n + 2m - 1$ Knoten aufweisen kann.

! **Übung 5.2.4** In Abschnitt 5.2.6 haben wir erwähnt, dass gilt: Wenn $X_1X_2 \ldots X_k \stackrel{*}{\Rightarrow} \alpha$, dann befinden sich alle Positionen von α, die sich aus der Ableitung aus X_i ergeben, links von all den Positionen, die sich aus der Ableitung aus X_j ergeben, wenn $i < j$. Beweisen Sie dies. *Hinweis*: Führen Sie einen Induktionsbeweis über die Anzahl der Ableitungsschritte.

5.3 Anwendungen kontextfreier Grammatiken

Kontextfreie Grammatiken waren von N. Chomsky ursprünglich als Mittel zur Beschreibung natürlicher Sprachen gedacht. Diese Erwartung hat sich nicht erfüllt. Jedoch: Nachdem die Anwendung rekursiv definierter Konzepte in der Informatik stark zugenommen hat, wuchs auch der Bedarf an kontextfreien Grammatiken als Möglichkeit, Instanzen dieser Konzepte zu beschreiben. Wir werden im Folgenden zwei dieser Anwendungen skizzieren, und zwar eine alte und eine neue.

1. Grammatiken werden zur Beschreibung von Programmiersprachen eingesetzt. Wichtig ist vor allem, dass es ein mechanisches Verfahren gibt, die Sprachbeschreibung einer kfG in einen Parser umzuwandeln, d. h. die Compilerkomponente, die die Struktur eines Quelltextprogramms erkennt und durch einen Parsebaum repräsentiert. Dies ist eine der frühesten Anwendungen von kfGs. In der Tat stellt diese Anwendungen einen der ersten Bereiche dar, in dem theoretische Konzepte der Informatik ihren Weg in die Praxis fanden.

2. Von der Entwicklung von XML (eXtensible Markup Language) erwartet man sich in weiten Kreisen eine Erleichterung des elektronischen Handels, da die daran Beteiligten gemeinsame Konventionen hinsichtlich des Formats von Aufträgen, Produktbeschreibungen und vielen anderen Dokumententypen vereinbaren und einsetzen können. Ein grundlegender Teil von XML ist die *Dokumenttypdefinition* (*DTD*), bei der es sich im Grunde genommen um eine kontextfreie Grammatik handelt, die alle zulässigen Tags und die Art und Weise, in der Tags verschachtelt werden können, beschreibt. Mit Tags sind die in spitze Klammern gesetzten Schlüsselwörter gemeint, die Sie vielleicht aus HTML kennen, z. B. `<EM>` und `</EM>` zur Hervorhebung des innerhalb dieser Tags stehenden Textes. XML-Tags legen aber nicht die Formatierung des Textes, sondern die Bedeutung des Textes fest. Beispielsweise könnte man eine Zeichenreihe, die als Telefonnummer interpretiert werden soll, zwischen die Tags `<PHONE>` und `</PHONE>` setzen.

5.3.1 Parser

Viele Aspekte von Programmiersprachen haben eine Struktur, die durch reguläre Ausdrücke beschrieben werden kann. Wir haben beispielsweise in Beispiel 3.9 erörtert, wie Bezeichner durch reguläre Ausdrücke dargestellt werden können. Es gibt allerdings auch sehr wichtige Aspekte von typischen Programmiersprachen, die sich nicht alleine durch reguläre Ausdrücke repräsentieren lassen. Es folgen zwei Beispiele hierfür.

Beispiel 5.19 In typischen Sprachen können runde und/oder eckige Klammern ineinander verschachtelt werden, wobei die Anzahl linker und rechter Klammern gleich sein muss. Das heißt, wir müssen in der Lage sein, wenigstens einer linken Klammer die nächste, sich rechts von ihr befindliche rechte Klammer zuzuordnen, die beiden Klammern zu löschen und den Vorgang zu wiederholen, bis keine Klammer übrig bleibt. Wenn sich auf diese Weise alle Klammern entfernen lassen, dann weist die Zeichenreihe die gleiche Anzahl von rechten und linken Klammern in korrekter Verschachtelung auf, andernfalls nicht. Beispiele für Zeichenreihen mit der gleichen Anzahl von linken und rechten Klammern in korrekter Verschachtelung, so genannte ausgewogene Zeichenreihen, sind (()), () (), (() ()) und ε, wogegen)(und (() es nicht sind.

Die Grammatik $G_{bal} = (\{B\}, \{(,)\}, P, B)$ generiert genau alle ausgewogenen Zeichenreihen, wobei P aus folgenden Produktionen besteht:

$$B \to BB \mid (B) \mid \varepsilon$$

Die erste Produktion $B \to BB$ besagt, dass die Verkettung zweier ausgewogener Zeichenreihen wiederum eine ausgewogene Zeichenreihe ergibt. Diese Behauptung leuchtet ein, weil wir die Klammern in den beiden Zeichenreihen unabhängig voneinander behandeln können. Die zweite Produktion $B \to (B)$ besagt, dass wir eine ausgewogene Zeichenreihe in Klammern setzen können und wiederum eine ausgewogene Zeichenreihe erhalten. Auch diese Regel ist vernünftig, weil wir die Klammern der inneren Zeichenreihe einander zuordnen und entfernen können und dann die äußeren Klammern übrig bleiben, die wir ebenso einander zuordnen können. Die dritte Produktion $B \to \varepsilon$ ist die Basis und besagt, dass die leere Zeichenreihe eine ausgewogene Zeichenreihe ist.

Die obige informelle Argumentation sollte uns davon überzeugen, dass G_{bal} nur ausgewogene Zeichenreihen erzeugt. Wir müssen die Umkehrung beweisen, also dass jede ausgewogene Zeichenreihe von dieser Grammatik erzeugt wird. Der Induktionsbeweis über die Länge einer ausgewogenen Zeichenreihe ist nicht schwer und bleibt dem Leser als Übung überlassen.

Wir haben erwähnt, dass die Menge der Zeichenreihen mit der gleichen Anzahl von linken und rechten Klammern keine reguläre Sprache ist, und wir werden dies nun beweisen. Wäre $L(G_{bal})$ regulär, dann gäbe es nach dem Pumping-Lemma für reguläre Sprachen eine Konstante n für diese Sprache. Betrachten Sie die ausgewogene Zeichenreihe $w = (^n)^n$, das heißt, n linke runde Klammern gefolgt von n rechten runden Klammern. Wenn wir w nach dem Pumping-Lemma in $w = xyz$ aufspalten, dann besteht y nur aus linken Klammern, und folglich repräsentiert xz mehr rechte Klammern als linke Klammern. Diese Zeichenreihe ist nicht ausgewogen und widerspricht damit der Annahme, dass die Sprache der ausgewogenen Zeichenreihen regulär ist. ■

Programmiersprachen bestehen natürlich nicht nur aus Klammern, aber Klammern sind ein zentraler Bestandteil von arithmetischen und regulären Ausdrücken. Die Grammatik aus Tabelle 5.2 ist für die Struktur arithmetischer Ausdrücke typisch, obwohl nur zwei Operationen verwendet werden (Plus- und Multiplikationszeichen) und die Struktur von Bezeichnern detailliert beschrieben wird, die besser von einer lexikalischen Analysekomponente des Compilers behandelt werden würden, wie wir in Abschnitt 3.3.2 erwähnt haben. Die durch Tabelle 5.2 beschriebene Sprache ist jedenfalls nicht regulär. Entsprechend dieser Grammatik ist $({}^n a)^n$ ein zulässiger Ausdruck. Wir können mithilfe des Pumping-Lemmas zeigen, dass eine Zeichenreihe, von der einige linke Klammern entfernt und das *a* und die rechten Klammern unverändert gelassen werden, wenn die Sprache regulär wäre, auch einen zulässigen Ausdruck darstellen würde, was aber nicht der Fall ist.

Es gibt zahlreiche Aspekte typischer Programmiersprachen, die sich wie ausgewogene Zeichenreihen verhalten. Für gewöhnlich kommen Klammern in allen möglichen Arten von Ausdrücken vor. Der Beginn und das Ende von Quelltextblöcken, wie z. B. `begin` und `end` in Pascal oder die geschweiften Klammern in C, sind hierfür Beispiele. Das heißt, in einem C-Programm müssen geschweifte Klammern stets in ausgewogener Form verwendet werden.

Gelegentlich tritt ein verwandtes Muster auf, in dem eine zusätzliche linke »Klammer« in einem Klammernpaar vorkommen kann. Ein Beispiel hierfür ist die Behandlung von `if` und `else` in C. Eine `if`-Anweisung kann mit und ohne `else`-Klausel vorkommen. Eine Grammatik, die die möglichen Sequenzen von `if` und `else` (hier durch *i* und *e* angegeben) repräsentiert, sieht wie folgt aus:

$$S \to \varepsilon \mid SS \mid iS \mid iSe$$

Beispielsweise sind *ieie*, *iie* und *iei* möglichen Folgen von `if`- und `else`-Klauseln, und jede dieser Zeichenreihen wird von der oben angegebenen Grammatik generiert. Beispiele für unzulässige Folgen, die von dieser Grammatik nicht generiert werden, sind *ei* und *ieeii*.

Ein einfacher Test (dessen Richtigkeit zu beweisen, wir dem Leser als Übung überlassen), mit dem sich feststellen lässt, ob eine Folge von `if`- und `else`-Klauseln von der Grammatik erzeugt wird, besteht darin, die einzelnen *e* nacheinander von links nach rechts zu betrachten. Suchen Sie nach dem ersten *i* links von dem *e*, das Sie gerade untersuchen. Wenn kein *i* vorhanden ist, dann ist die Zeichenreihe nicht in der Sprache enthalten. Gibt es ein solches *i*, dann löschen Sie es zusammen mit dem untersuchten *e*. Gibt es kein weiteres *e*, hat die Zeichenreihe den Test bestanden und ist in der Sprache enthalten. Sind weitere *e* vorhanden, untersuchen Sie das nächste *e* in der beschriebenen Weise.

Beispiel 5.20 Betrachten Sie die Zeichenreihe *iee*. Das erste *e* wird dem nächsten *i* links von ihm zugeordnet. Die beiden Zeichen werden entfernt, sodass das Zeichen *e* übrig bleibt. Da es noch ein *e* gibt, betrachten wir dieses. Es ist allerdings kein *i* mehr links von ihm vorhanden, und daher scheitert der Test; *iee* ist nicht in der Sprache enthalten. Beachten Sie, dass dieser Schluss gültig ist, da ein C-Programm nicht mehr `else`-Klauseln als `if`-Anweisungen enthalten kann.

Betrachten Sie als weiteres Beispiel *iieie*. Wenn wir das erste *e* dem *i* zu seiner Linken zuordnen, dann bleibt die Zeichenreihe *iie* übrig. Wenn wir das letzte *e* dem nächsten *i* zu seiner Linken zuordnen, bleibt das Zeichen *i*. Nun ist kein weiteres *e* vorhanden, und der Test ist erfolgreich; d. h. die Zeichenreihe ist in der Sprache enthal-

ten. Auch dieser Schluss ist vernünftig, weil die Zeichenreihe *iieie* einem C-Programm entspricht, das wie das Programmfragment in Listing 5.7 strukturiert ist. Der Zuordnungsalgorithmus gibt uns (und dem C-Compiler) sogar darüber Aufschluss, welches *if* einem bestimmten *else* zuzuordnen ist. Dies ist eine unabdingbare Information, wenn der Compiler den vom Programmierer beabsichtigten Kontrollfluss für das Programm sicherstellen soll. ■

Listing 5.7: Eine **if-else**-Struktur; die beiden **else**-Klauseln gehören zu den vorstehenden **if**-Anweisungen, und die erste **if**-Anweisung enthält keine **else**-Klausel

```
if (Bedingung) {
    ...
    if (Bedingung) Anweisung;
    else Anweisung;
    ...
    if (Bedingung) Anweisung;
    else Anweisung;
    ...
}
```

5.3.2 Der YACC-Parsergenerator

Die Erzeugung eines Parsers (Funktion, die aus Quelltextprogrammen Parsebäume erstellt) wurde in dem Befehl YACC institutionalisiert, der in allen Unix-Systemen vorhanden ist. Dem Befehl YACC wird eine kfG als Eingabe übergeben, und zwar in einer Notation, die sich von der hier verwendeten nur in Details unterscheidet. Jeder Produktion ist eine *Aktion* zugeordnet, wobei es sich um ein C-Quelltextfragment handelt, das ausgeführt wird, sobald der Knoten des Parsebaums erzeugt wird, der (zusammen mit seinen Nachfolgern) dieser Produktion entspricht. Normalerweise handelt es sich bei der Aktion um Quelltext zur Erzeugung dieses Knotens, obwohl der Baum in einigen YACC-Anwendungen gar nicht wirklich konstruiert wird und die Aktion etwas anderes ausführt, wie z. B. Objektcode zu erzeugen.

Beispiel 5.21 Listing 5.8 ist ein Beispiel einer kfG in der YACC-Notation. Die Grammatik entspricht der aus Tabelle 5.2. Wir haben die Aktionen nicht explizit aufgeführt, sondern nur die erforderlichen geschweiften Klammern und ihre Position in der YACC-Eingabe angegeben.
Beachten Sie die folgenden Entsprechungen zwischen der YACC-Notation für Grammatiken und unserer Notation:

- Der Doppelpunkt wird als Produktionssymbol verwendet und entspricht somit unserem Pfeil $\rightarrow$.
- Alle Produktionen mit einem gegebenen Kopf werden zusammengefasst und ihre Rümpfe durch den vertikalen Strich voneinander getrennt. Wir lassen diese Konvention optional zu.
- Die Liste der Rümpfe zu einem gegebenen Kopf wird mit einem Semikolon beendet. Wir haben kein Terminierungssymbol verwendet.

Listing 5.8: Beispiel für eine Grammatik in YACC-Notation

```
Exp : Id              {...}
    | Exp '+' Exp     {...}
    | Exp '*' Exp     {...}
    | '(' Exp ')'     {...}
    ;
Id  : 'a'             {...}
    | 'b'             {...}
    | Id 'a'          {...}
    | Id 'b'          {...}
    | Id '0'          {...}
    | Id '1'          {...}
    ;
```

- Terminale Zeichenreihen werden in halbe Anführungszeichen gesetzt. Obwohl wir dies nicht gezeigt haben, erlaubt YACC es den Benutzern, auch symbolische terminale Zeichen zu definieren. Diese terminalen Zeichen werden von der lexikalischen Analysekomponente im Quelltext erkannt und durch einen Rückgabewert dem Parser angezeigt.
- Nicht in Anführungszeichen gesetzte Zeichenreihen von Buchstaben und Ziffern sind Variablennamen. Wir haben dieses Merkmal genutzt, um unseren Variablen suggestive Namen – `Exp` und `Id` – zu geben, hätten aber auch *E* und *I* verwenden können. ■

5.3.3 Markup-Sprachen

Wir werden als Nächstes eine Familie von »Sprachen« betrachten, die als *Markup-Sprachen* oder *Kennzeichnungssprachen* bezeichnet werden. Die »Zeichenreihen« dieser Sprachen sind Dokumente, die bestimmte Markierungen (so genannte *Tags*) enthalten. Diese Tags sagen etwas über die Semantik der verschiedenen Zeichenreihen innerhalb des Dokuments aus.

HTML (HyperText Markup Language) ist die Markup-Sprache, die Ihnen wahrscheinlich am vertrautesten ist. Diese Sprache hat zwei Hauptfunktionen: die Erstellung von Verknüpfungen zwischen Dokumenten und die Beschreibung des Formats (»Aussehen«) eines Dokuments. Wir werden die Struktur von HTML vereinfacht darstellen; aus den folgenden Beispielen sollten jedoch die Struktur und die Art und Weise hervorgehen, in der eine kfG zur Beschreibung zulässiger HTML-Dokumente und als Richtlinie zur Verarbeitung von Dokumenten (d. h. Ausgabe auf einem Monitor oder Drucker) eingesetzt werden kann.

Beispiel 5.22 Listing 5.9 (a) zeigt ein Textstück, das aus einer Liste von Dingen besteht, und Listing 5.3 (b) zeigt dessen Darstellung in HTML. Beachten Sie an Listing 5.9 (b), dass HTML aus gewöhnlichem Text besteht, der mit Tags durchsetzt ist. Zusammengehörige öffnende und schließende Tags haben für eine Zeichenreihe *x* die Form <*x*> und </*x*>[3]. Beispielsweise kommen in dem HTML-Text die zusammengehörigen Tags `<EM>` und `</EM>` vor, die anzeigen, dass der dazwischenstehende Text her-

vorgehoben, d. h. in Kursivschrift oder einer anderen geeigneten Schrift gesetzt werden soll. Der Quelltext enthält zudem die Tags <OL> und </OL>, die die Erstellung einer nummerierten Liste bewirken.

Listing 5.9: Ein HTML-Dokument und dessen Erscheinungsbild in der Ausgabe

Dinge, die ich *hasse*:
1. Verschimmeltes Brot.
2. Leute, die in der Überholspur zu langsam fahren.

a) So wird der Text angezeigt

```
<P>Dinge, die ich <EM>hasse</EM>:
<OL>
<LI>Verschimmeltes Brot.
<LI>Leute, die in der Überholspur zu langsam fahren.
</OL>
```

b) HTML-Quelltext

Der HTML-Quelltext enthält auch zwei Beispiele für allein stehende Tags: <P> und <LI>, die zur Einleitung eines Absatzes bzw. der Listeneinträge dienen. In HTML ist es zulässig, ja sogar erwünscht, dass für diese Tags die zugehörigen abschließenden Tags </P> bzw. </LI> am Ende eines Absatzes bzw. der Liste angegeben werden, aber es ist nicht unbedingt erforderlich. Wir haben die abschließenden Tags weggelassen, damit die Beispielgrammatik für HTML, die wir entwickeln werden, etwas komplizierter wird. ■

Es gibt eine Reihe von Zeichenreihenklassen in einem HTML-Dokument. Wir werden nicht versuchen, sie hier alle aufzulisten, im Folgenden finden Sie jedoch einen Überblick über die Klassen, die für das Verständnis von Text unabdingbar sind, wie die aus Beispiel 5.22. Wir führen für jede Klasse eine Variable mit einem beschreibenden Namen ein.

1. *Text* ist jede Zeichenreihe aus Zeichen, die wörtlich interpretiert werden kann; d. h. sie enthält keine Tags. In Listing 5.9 ist »Verschimmeltes Brot« ein Beispiel für ein *Textelement*.
2. *Char* ist jede Zeichenreihe, die aus einem einzigen in HTML-Texten zulässigen Zeichen besteht. Beachten Sie, dass Leerzeichen Zeichen vom Typ *Char* sind.
3. *Doc* repräsentiert Dokumente, die Folgen von »Elementen« darstellen. Wir definieren Elemente als Nächstes, und diese Definition ist verschränkt rekursiv in Bezug auf die Definition von *Doc*.

3. Einige Anfangstags <*x*> enthalten mehr Informationen als lediglich den Namen *x* des Tags. Wir werden diese Möglichkeit in Beispielen allerdings nicht berücksichtigen.

4. *Element* ist entweder eine Zeichenreihe vom Typ *Text* oder ein Paar zusammengehöriger Tags, die ein Dokument einschließen, oder ein einleitendes, allein stehendes Tag, dem ein Dokument folgt.
5. *ListItem* ist das Tag `<LI>` gefolgt von einem Dokument, das aus einem einzelnen Listeneintrag besteht.
6. *List* steht für eine Folge von null oder mehr Listeneinträgen.

Tabelle 5.4: Teil einer HTML-Grammatik

1.	*Char*	→	*a* | *A* | ...
2.	*Text*	→	ε | *Char Text* ...
3.	*Doc*	→	ε | *Element Doc* ...
4.	*Element*	→	*Text* | `<EM>` *Doc* `</EM>` | `<P>` *Doc* | `<OL>` *List* `</OL>` | …
5.	*ListItem*	→	`<LI>` *Doc*
6.	*List*	→	ε | *ListItem List* ...

Tabelle 5.4 ist eine kfG, die so viel von der Struktur der Sprache beschreibt, wie wir behandelt haben. In Zeile (1) heißt es, dass ein Zeichen (*Char*) »a« oder »A« oder jedes andere im HTML-Zeichensatz zulässige Zeichen sein kann. Zeile (2) besagt in Form von zwei Produktionen, dass es sich bei *Text* um die leere Zeichenreihe oder um ein zulässiges Zeichen gefolgt von weiterem Text handelt. Anders ausgedrückt, *Text* besteht aus null oder mehr Zeichen. Beachten Sie, dass < und > keine zulässigen Zeichen sind, obwohl sie durch die Sequenzen `&lt;` und `&gt;` dargestellt werden können. Folglich kann man nicht versehentlich Tags in Text einfügen.

Zeile (3) besagt, dass ein Dokument aus einer Folge von null oder mehr Elementen besteht. Ein Element ist wiederum, wie wir in Zeile (4) erfahren, Text, ein hervorgehobenes Dokument, ein Absatzanfang gefolgt von einem Dokument oder eine Liste. Wir haben zudem angezeigt, dass es weitere Produktionen für *Element* gibt, die anderen Arten von Tags in HTML zugehörig sind. In Zeile (5) erfahren wir, dass sich ein *ListItem* aus dem Tag `<LI>` gefolgt von einem beliebigen Dokument zusammensetzt. Zeile (6) besagt, dass eine Liste aus einer Folge von null oder mehr *ListItems* besteht.

Einige Aspekte von HTML erfordern nicht die Mächtigkeit kontextfreier Grammatiken; reguläre Ausdrücke sind adäquat. Beispielsweise besagen die Zeilen (1) und (2) in Tabelle 5.4 einfach, dass *Text* dieselbe Sprache wie der reguläre Ausdruck (**a** + **A** + ...)* repräsentiert. Einige Aspekte *erfordern* jedoch die Mächtigkeit von kfGs. Jedes Paar zusammengehöriger Tags, das den Anfang und das Ende eines Elements kennzeichnet, z. B. `<EM>` und `</EM>`, ähnelt den Klammernpaaren, die, wie wir wissen, keine reguläre Sprache darstellen.

5.3.4 XML und Dokumenttypdefinitionen

Die Tatsache, dass HTML von einer Grammatik beschrieben wird, ist an und für sich nicht bemerkenswert. Im Grunde genommen können alle Programmiersprachen durch ihre eigenen kfGs beschrieben werden, und daher wäre es verwunderlicher, wenn wir HTML *nicht* auf diese Weise beschreiben könnten. Wenn wir jedoch eine andere bedeutende Markup-Sprache betrachten, nämlich XML (eXtensible Markup Language), stellen wir fest, dass kfGs in der Verwendung dieser Sprache eine sehr wichtige Rolle spielen.

XML dient nicht zur Beschreibung der Formatierung eines Dokuments; dies ist die Aufgabe von HTML. XML versucht stattdessen, die »Semantik« des Textes zu beschreiben. Der Text »12 Maple St.« sieht z. B. wie eine englische Adressangabe aus, ist es aber tatsächlich eine Adresse? In XML würden Tags einen Ausdruck umgeben, der eine Adresse darstellt, beispielsweise:

```
<ADDR>12 Maple St.</ADDR>
```

Es ist allerdings nicht sofort erkennbar, dass `<ADDR>` für die Adresse eines Gebäudes steht. Wenn das Dokument Speicherzuweisung zum Thema hätte, dann würden wir erwarten, dass sich das Tag `<ADDR>` auf eine Speicheradresse bezieht. Zur Erläuterung der verschiedenen Tags und der Strukturen, die zwischen zusammengehörigen Paaren dieser Tags vorkommen können, erwartet man von Personen mit einem gemeinsamen Interesse, dass sie Standards in Form einer DTD (Dokumenttypdefinition) entwickeln.

Eine DTD ist im Grunde genommen eine kontextfreie Grammatik, die über eine eigene Notation zur Beschreibung der Variablen und Produktionen verfügt. Im nächsten Beispiel werden wir eine einfache DTD und etwas von der Sprache vorstellen, die zur Beschreibung von DTDs dient. Die DTD-Sprache hat selbst eine kontextfreie Grammatik, aber wir sind nicht an der Beschreibung dieser Grammatik interessiert. Die Sprache zur Beschreibung von DTDs ist im Grunde genommen eine kfG-Notation, und uns interessiert, wie kfGs in dieser Sprache ausgedrückt werden.

Eine DTD hat folgende Form:

```
<!DOCTYPE Name-der-DTD [
      Liste der Elementdefinitionen
]>
```

Eine Elementdefinition sieht wiederum so aus:

```
<!ELEMENT Elementname (Beschreibung des Elements)>
```

Elementbeschreibungen sind im Grunde genommen reguläre Ausdrücke. Grundlagen dieser Ausdrücke sind:

1. Andere Elementnamen, die die Tatsache widerspiegeln, dass Elemente eines Typs in Elementen anderen Typs vorkommen können. Ebenso wie in HTML kann beispielsweise eine Liste hervorgehobenen Text enthalten.

2. Der spezielle Term `#PCDATA` steht für einen Text, der keine XML-Tags beinhaltet. Dieser Term hat die gleiche Bedeutung wie die Variable *Text* aus dem Beispiel 5.22.

Zulässige Operatoren sind:

1. | steht für Vereinigung, ebenso wie in der Unix-Notation für reguläre Ausdrücke, die in Abschnitt 3.3.1 erörtert wurde.
2. Ein Komma steht für Verkettung.
3. Drei Varianten des Operators zur Hüllenbildung, wie in Abschnitt 3.3.1 beschrieben. Es handelt sich um die Operatoren *, der gewöhnliche Operator mit der Bedeutung »null oder mehr Vorkommen von«, + mit der Bedeutung »ein oder mehr Vorkommen von« und ? mit der Bedeutung »null oder ein Vorkommen von«.

Operatoren und ihre Argumente können durch Klammern gruppiert werden; ansonsten gilt die übliche Rangfolge der Operatoren für reguläre Ausdrücke.

Beispiel 5.23 Stellen wir uns einmal vor, Computerhändler würden sich zusammentun und eine Standard-DTD erstellen, mit der sie Beschreibungen der von ihnen vertriebenen PCs im Internet veröffentlichen können. Jede Beschreibung eines PC umfasst eine Modellnummer, Details zu den Leistungsmerkmalen des Modells, z. B. die Menge an RAM, Anzahl und Größe der Datenträger etc. Listing 5.10 zeigt eine sehr einfache, hypothetische DTD für PCs:

Listing 5.10: DTD für PCs

```
<!DOCTYPE PcSpecs [
<!ELEMENT PCS(PC*)>
<!ELEMENT PC(MODEL, PRICE, PROCESSOR, RAM, DISK+)>
<!ELEMENT MODEL (#PCDATA)>
<!ELEMENT PRICE (#PCDATA)>
<!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
<!ELEMENT MANF (#PCDATA)>
<!ELEMENT MODEL (#PCDATA)>
<!ELEMENT SPEED (#PCDATA)>
<!ELEMENT RAM (#PCDATA)>
<!ELEMENT DISK (HARDDISK | CD | DVD)>
<!ELEMENT HARDDISK (MANF, MODEL, SIZE)>
<!ELEMENT SIZE (#PCDATA)>
<!ELEMENT CD (SPEED)>
<!ELEMENT DVD (SPEED)>
]>
```

Der Name der DTD lautet `PcSpecs`. Das erste Element, das mit dem Startsymbol einer kfG vergleichbar ist, lautet `PCS` (Liste mit PC-Spezifikationen). Seine Definition `PC*` besagt, dass eine `PCS`-Liste aus null oder mehr `PC`-Einträgen besteht.

Anschließend folgt die Definition eines `PC`-Elements. Es besteht aus der Verkettung von fünf Dingen. Bei den ersten vier handelt es sich um andere Elemente, die das Modell, den Preis, den Prozessortyp und den Arbeitsspeicher des PC repräsentieren. Jedes dieser Elemente muss einmal in der angegebenen Reihenfolge vorkommen, da

das Komma eine Verkettung angibt. Der letzte Bestandteil DISK+ besagt, dass für einen PC ein oder mehr DISK-Einträge vorliegen müssen.

Bei vielen dieser Bestandteile handelt es sich um einfachen Text: MODEL, PRICE und RAM sind von diesem Typ. Das Element PROCESSOR hat jedoch eine komplexere Struktur. Aus seiner Definition geht hervor, dass es aus den Elementen MANF, MODEL und SPEED besteht, bei denen es sich um einfachen Text handelt.

Das Element DISK ist am kompliziertesten. Erstens kann es sich um eine Festplatte (HARDDISK), ein CD- oder ein DVD-Laufwerk handeln, da die Regel für das Element DISK eine ODER-Verknüpfung drei anderer Elemente darstellt. Das Element HARDDISK besitzt wiederum eine Struktur, die die Elemente MANF, MODEL und SIZE umfasst, während die Elemente CD und DVD lediglich durch das Element SPEED näher beschrieben werden.

Listing 5.11 enthält ein Beispiel für ein XML-Dokument, das der DTD aus Listing 5.10 entspricht. Beachten Sie, dass jedes Element dieses Dokuments durch ein Tag, das den Namen dieses Elements enthält, eingeleitet und wie in HTML durch ein zugehöriges Tag mit vorangestelltem Schrägstrich abgeschlossen wird. So finden wir in Listing 5.11 auf der äußersten Ebene die Tags <PCS> und </PCS>. Zwischen diesen Tags befindet sich eine Liste mit Einträgen, die jeweils einen PC dieses Herstellers repräsentieren. Wir haben hier nur einen solchen Eintrag angeführt.

Listing 5.11: Teil eines Dokuments, das die Struktur der in Listing 5.10 angegebenen DTD aufweist

```
<PCS>
    <PC>
        <MODEL>4560</MODEL>
        <PRICE>EUR2295</PRICE>
        <PROCESSOR>
            <MANF>Intel</MANF>
            <MODEL>Pentium</MODEL>
            <SPEED>800MHz</SPEED>
        </PROCESSOR>
        <RAM>256</RAM>
        <DISK><HARDDISK>
            <MANF>Maxtor</MANF>
            <MODEL>Diamond</MODEL>
            <SIZE>30,5GB</SIZE>
        </HARDDISK>
        <DISK><CD>
            <SPEED>32x</SPEED>
        </CD></DISK>
    </PC>
    <PC>
        ...
    </PC>
</PCS>
```

Dieser <PC>-Eintrag enthält die Beschreibung eines PC, der über die Modellnummer 4560, den Preis EUR 2295, den Prozessortyp Intel Pentium 800 MHz, 256 RAM, eine Festplatte vom Typ Maxtor Diamond mit 30,5 Gbyte sowie ein 32-fach-CD-ROM-Laufwerk verfügt. Wichtig ist hier allerdings nicht, dass wir dem Dokument diese Informationen entnehmen können, sondern dass ein Programm das Dokument lesen und anhand der Grammatik der in Listing 5.10 dargestellten DTD die Zahlen und Bezeichnungen richtig interpretieren kann. ■

Ihnen ist vielleicht aufgefallen, dass die Regeln für DTD-Elemente wie in Listing 5.10 nicht ganz den Produktionen kontextfreier Grammatiken entsprechen. Viele dieser Regeln haben die korrekte Form. Zum Beispiel entspricht die Definition

```
<!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
```

folgender Produktion:

$$Processor \rightarrow Manf\ Model\ Speed$$

Allerdings weist die Regel

```
<!ELEMENT DISK (HARDDISK | CD | DVD)>
```

keine Definition von DISK auf, die dem Rumpf einer Produktion entspricht.

In diesem Fall ist die Umformung einfach: Wir können diese Regel so interpretieren, als handele es sich um drei Produktionen, wobei der senkrechte Strich dieselbe Bedeutung hat wie in unserer Kurzschreibweise für Produktionen mit demselben Kopf. Diese Regel ist somit äquivalent mit den folgenden drei Produktionen:

$$Disk \rightarrow HardDisk \mid CD \mid DVD$$

Am schwierigsten ist die Regel

```
<!ELEMENT PC(MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

zu interpretieren, deren »Rumpf« einen Operator zur Hüllenbildung enthält. Die Lösung besteht darin, DISK+ durch eine neue Variable zu ersetzen, die wir hier *Disks* nennen wollen und die mithilfe von zwei Produktionen eine oder mehrere Instanzen der Variablen *Disk* erzeugt. Die äquivalenten Produktionen lauten:

$$Pc \rightarrow Model\ Price\ Processor\ Ram\ Disks$$
$$Disks \rightarrow Disk \mid Disk\ Disks$$

Es gibt eine allgemeine Technik zur Umwandlung einer kfG mit regulären Ausdrücken als Produktionsrümpfen in eine gewöhnliche kfG. Wir werden dieses Konzept informell vorstellen. Es bleibt Ihnen überlassen, sowohl die Bedeutung von kfGs mit regulären Ausdrücken als Produktionen als auch den Beweis zu formalisieren, dass die Umformung keine anderen als kontextfreie Sprachen ergibt. Wir zeigen durch Induktion, wie man eine Produktion mit einem regulären Ausdruck als Rumpf in eine Sammlung äquivalenter gewöhnlicher Produktionen umwandelt. Die Induktion erfolgt über die Größe des Ausdrucks im Produktionsrumpf.

Induktionsbeginn: Wenn der Rumpf eine Verkettung von Elementen enthält, dann liegt die Produktion bereits in der für kfGs zulässigen Form vor, und daher ist nichts auszuführen.

INDUKTIONSSCHRITT: Andernfalls sind, abhängig vom letzten Operator im Produktionsrumpf, fünf Fälle zu unterscheiden:

1. Die Produktion hat die Form $A \to E_1, E_2$, wobei E_1 und E_2 in der DTD-Sprache zulässige Ausdrücke sind. Dies ist der Fall der Verkettung. Wir führen zwei neue Variablen B und C ein, die nirgendwo sonst in der Grammatik vorkommen. Dann ersetzen wir $A \to E_1, E_2$ durch die Produktionen

$$\begin{aligned} A &\to BC \\ B &\to E_1 \\ E &\to E_2 \end{aligned}$$

   Die erste Produktion $A \to BC$ ist in kfGs zulässig. Die letzen beiden sind möglicherweise zulässig oder auch nicht. Ihre Rümpfe sind jedoch kürzer als der Rumpf der ursprünglichen Produktion, daher können wir sie per Induktion in das kfG-Format umwandeln.

2. Die Produktion hat die Form $A \to E_1 \mid E_2$. Im Fall dieses Vereinigungsoperators ersetzen wir die Produktion durch das folgende Paar von Produktionen:

$$\begin{aligned} A &\to E_1 \\ A &\to E_2 \end{aligned}$$

   Auch diese Produktionen sind möglicherweise keine zulässigen kfG-Produktionen, deren Rümpfe jedoch kürzer als der Rumpf der ursprünglichen Produktion sind. Wir können die Regeln daher rekursiv anwenden und diese Produktionen schließlich in das kfG-Format umwandeln.

3. Die Produktion hat die Form $A \to (E_1)^*$. Wir führen die neue Variable B ein, die nirgendwo sonst vorkommt, und ersetzen diese Produktion durch:

$$\begin{aligned} A &\to BA \\ A &\to B \\ B &\to E_1 \end{aligned}$$

4. Die Produktion hat die Form $A \to (E_1)^+$. Wir führen die neue Variable B ein, die nirgendwo sonst vorkommt, und ersetzen diese Produktion durch:

$$\begin{aligned} A &\to BA \\ A &\to \varepsilon \\ B &\to E_1 \end{aligned}$$

5. Die Produktion hat die Form $A \to (E_1)?$. Wir ersetzen diese Produktion durch:

$$\begin{aligned} A &\to \varepsilon \\ A &\to E_1 \end{aligned}$$

Beispiel 5.24 Sehen wir uns nun an, wie man folgende DTD-Regel

```
<!ELEMENT PC(MODEL, PRICE, PROCESSOR, RAM, DISK+)>
```

in zulässige kfG-Produktionen überführt. Erstens können wir den Rumpf dieser Regel als Verkettung zweier Ausdrücke betrachten, deren erster die Liste `MODEL`, `PRICE`, `PROCESSOR`, `RAM` und deren zweiter das Element `DISK+` ist. Wenn wir für diese beiden Teilausdrücke die Variablen A und B einführen, können wir folgende Produktionen erstellen:

$$Pc \to AB$$
$$A \to \textit{Model Price Processor Ram}$$
$$B \to \textit{Disk+}$$

Nur die letzte dieser drei Produktionen liegt nicht in zulässiger Form vor. Wir führen eine weitere Variable C ein und die Produktionen:

$$B \to CB \mid C$$
$$C \to \textit{Disk}$$

Allerdings, in diesem speziellen Fall, in dem der Ausdruck zur Ableitung von A nur eine Verkettung von Variablen und *Disk* eine einzelne Variable ist, brauchen wir die Variablen A und C eigentlich gar nicht. Wir könnten stattdessen die folgenden Produktionen verwenden:

$$Pc \to \textit{Model Price Processor Ram } B$$
$$B \to \textit{Disk } B \mid \textit{Disk}$$

■

5.3.5 Übungen zum Abschnitt 5.3

Übung 5.3.1 Beweisen Sie, dass eine Zeichenreihe aus Klammern, die in dem in Beispiel 5.19 dargestellten Sinn ausgewogen ist, von der Grammatik $B \to BB \mid (B) \mid \varepsilon$ erzeugt wird. *Hinweis*: Führen Sie einen Induktionsbeweis über die Länge der Zeichenreihe.

* **Übung 5.3.2** Betrachten Sie die Menge aller ausgewogenen Zeichenreihen aus runden und eckigen Klammern. Ein Beispiel für diese Art von Zeichenreihen ist das folgende. Wenn wir Ausdrücke der Programmiersprache C nehmen, in denen Operanden mit runden Klammern gruppiert, Argumente von Funktionsaufrufen in runde Klammern und Array-Indizes in eckige Klammern gesetzt werden, und alles außer den Klammern daraus löschen, erhalten wir ausgewogene Zeichenreihen aus runden und eckigen Klammern. Zum Beispiel wird aus der Zeichenreihe

```
f(a[i]*(b[i][j], c[g(x)]), d[i])
```

die ausgewogene Klammerzeichenreihe `([]([][][()])[])`. Entwerfen Sie eine Grammatik für alle ausgewogenen Zeichenreihen aus runden und eckigen Klammern.

! **Übung 5.3.3** In Abschnitt 5.3.1 betrachteten wir die Grammatik

$$S \to \varepsilon \mid SS \mid iS \mid iSe$$

und behaupteten, dass wir die Zugehörigkeit einer Zeichenreihe zur Sprache L dieser Grammatik bestimmen könnten, indem wir die nachfolgend beschriebenen Schritte wiederholt ausführen, wobei wir mit einer Zeichenreihe w beginnen. Die Zeichenreihe w ändert sich während der Wiederholungen.

1. Wenn die aktuelle Zeichenreihe mit e beginnt, dann ist die Zeichenreihe w nicht in L enthalten.
2. Enthält die aktuelle Zeichenreihe kein e (wobei sie i enthalten kann), dann ist sie in L enthalten.

3. Andernfalls löschen wir das erste *e* und das unmittelbar links davon stehende *i*. Dann wiederholen wir diese drei Schritte mit der neuen Zeichenreihe.

Beweisen Sie, dass dieses Verfahren zur korrekten Identifizierung der in L enthaltenen Zeichenreihen führt.

Übung 5.3.4 Fügen Sie die folgenden Formatdefinitionen der HTML-Grammatik aus Tabelle 5.4 hinzu:

* a) Listeneinträge müssen mit dem abschließenden Tag `</LI>` beendet werden.

b) Ein Element kann eine geordnete Liste ebenso wie eine ungeordnete Liste darstellen. Ungeordnete Listen werden in die Tags `<UL>` und `</UL>` eingeschlossen.

! c) Ein Element kann eine Tabelle beschreiben. Tabellen werden in die Tags `<TABLE>` und `</TABLE>` eingeschlossen. Innerhalb dieser Tags können sich eine oder mehrere Tabellenzeilen befinden, die jeweils in die Tags `<TR>` und `</TR>` eingeschlossen werden. Die erste Zeile enthält den Tabellenkopf, der über eines oder mehrere Felder verfügt, denen jeweils das Tag `<TH>` vorangestellt ist (wir nehmen an, dass für diese Tags keine abschließenden Tags vorhanden sind, obwohl sie abgeschlossen werden sollten). In den nachfolgenden Zeilen wird den einzelnen Feldern das Tag `<TD>` vorangestellt.

Listing 5.12: Eine DTD für Kurse

```
<!DOCTYPE CourseSpecs [
    <!ELEMENT COURSES (COURSE+)>
    <!ELEMENT COURSE (CNAME, PROF, STUDENT*, TA?)>
    <!ELEMENT CNAME (#PCDATA)>
    <!ELEMENT STUDENT (#PCDATA)>
    <!ELEMENT TA (#PCDATA)>
]>
```

5.4 Mehrdeutigkeit von Grammatiken und Sprachen

Wie wir gesehen haben, verlassen sich Anwendungen von kfGs häufig darauf, dass die Grammatik die Struktur von Dateien beschreibt. In Abschnitt 5.3 wurde beispielsweise gezeigt, wie Grammatiken eingesetzt werden können, um Programme und Dokumente zu strukturieren. Es wurde stillschweigend unterstellt, die Grammatik würde die Struktur jeder Zeichenreihe ihrer Sprache eindeutig bestimmen. Wie wir aber noch sehen werden, definiert nicht jede Grammatik eine eindeutige Struktur.

Wenn eine Grammatik keine eindeutigen Strukturen beschreibt, dann ist es gelegentlich möglich, die Grammatik so abzuändern, dass die Struktur für jede in der Sprache enthaltene Zeichenreihe eindeutig ist. Leider ist dies nicht immer möglich. Das heißt, es gibt einige kontextfreie Sprachen, die »inhärent mehrdeutig« sind. Jede Grammatik für eine solche Sprache ordnet dann wenigstens einer Zeichenreihe der Sprache mehr als eine Struktur zu.

5.4.1 Mehrdeutige Grammatiken

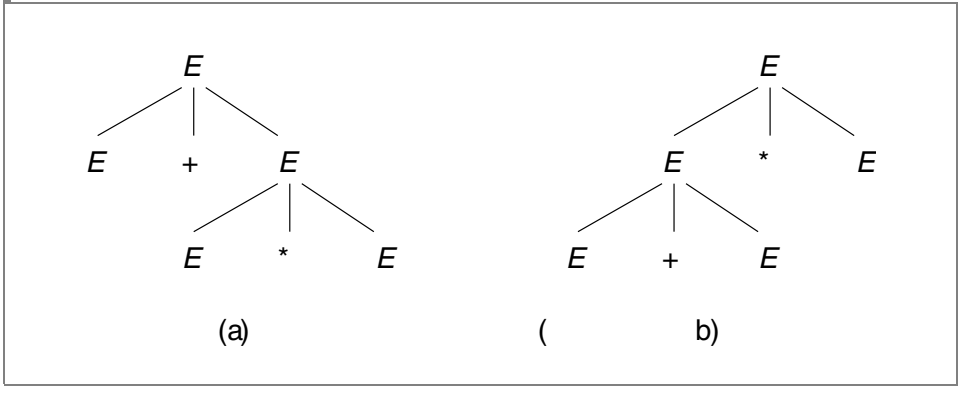
Lassen Sie uns zu unserem Beispiel der Grammatik für Ausdrücke aus Tabelle 5.2 zurückkehren. Diese Grammatik ermöglicht es uns, Ausdrücke mit einer beliebigen Folge der Operatoren $*$ und $+$ zu erzeugen, und die Produktionen $E \to E + E \mid E * E$ ermöglichen es uns, diese Ausdrücke in jeder beliebigen Reihenfolge zu erzeugen.

Beispiel 5.25 Betrachten Sie z. B. die Satzform $E + E * E$. Sie hat zwei Ableitungen von E:

1. $E \Rightarrow E + E \Rightarrow E + E * E$
2. $E \Rightarrow E * E \Rightarrow E + E * E$

Beachten Sie, dass in der Ableitung (1) das zweite E durch $E * E$ ersetzt wird, während in der Ableitung (2) das erste E durch $E + E$ ersetzt wird. Abbildung 5.13 zeigt die beiden Parsebäume dieser Ableitungen, die eine unterschiedliche Struktur haben.

Abbildung 5.13: Zwei Parsebäume mit dem gleichen Ergebnis



Diese beiden Ableitungen weisen signifikante Unterschiede auf. Was die Struktur der Ausdrücke betrifft, besagt Ableitung (1), dass der zweite und der dritte Ausdruck miteinander multipliziert werden und das Ergebnis zum ersten Ausdruck addiert wird, während Ableitung (2) besagt, dass die ersten beiden Ausdrücke addiert werden und das Ergebnis mit dem dritten Ausdruck multipliziert wird. Konkreter ausgedrückt besagt die erste Ableitung, dass z. B. $1 + 2 * 3$ in der Form $1 + (2 * 3) = 7$ gruppiert werden sollte, während die zweite Ableitung für den gleichen Ausdruck die Gruppierung $(1 + 2) * 3 = 9$ nahe legt. Offensichtlich entspricht nur die erste Ableitung unserer Vorstellung von der korrekten Gruppierung arithmetischer Ausdrücke.

Da die Grammatik aus Tabelle 5.2 jeder Zeichenreihe aus terminalen Symbolen, die abgeleitet wird, indem die drei in $E + E * E$ enthaltenen Ausdrücke durch Bezeichner ersetzt werden, zwei unterschiedliche Strukturen zuordnet, wird klar, dass sich diese Grammatik nicht zur Angabe einer eindeutigen Struktur eignet. Sie kann Zeichenreihen korrekt als arithmetische Ausdrücke, aber auch nicht korrekt gruppieren. Um diese Grammatik in einem Compiler einsetzen zu können, müssten wir sie so abändern, dass sie nur die korrekte Gruppierung erzeugt. ■

Andererseits impliziert das bloße Vorhandensein verschiedener Ableitungen (im Gegensatz zu verschiedenen Parsebäumen) für eine Zeichenreihe nicht, dass die Grammatik mangelhaft ist. Das folgende Beispiel soll dies verdeutlichen.

Beispiel 5.26 Bei Verwendung der gleichen Grammatik für einfache Ausdrücke stellen wir fest, dass die Zeichenreihe $a + b$ viele verschiedene Ableitungen besitzt. Zwei Beispiele sind:

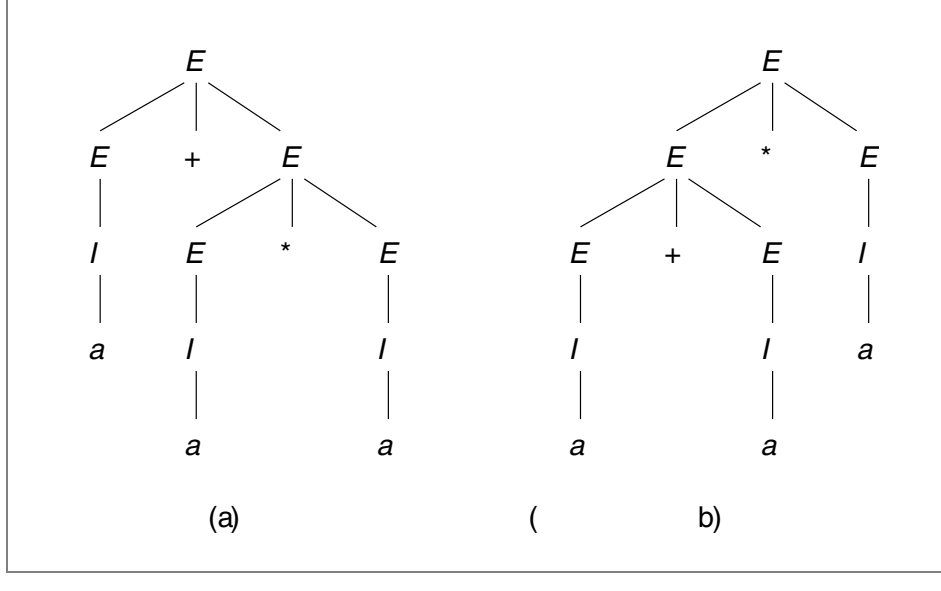
1. $E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
2. $E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

Diese Ableitungen ergeben allerdings dieselbe Struktur. Beide besagen, dass a und b Bezeichner sind, deren Werte addiert werden sollen. Beide Ableitungen erzeugen bei Anwendung der Konstruktion der Sätze 5.18 und 5.12 sogar den gleichen Parsebaum. ■

Diese beiden Beispiele legen nahe, dass nicht die Vielzahl von Ableitungen Mehrdeutigkeit verursacht, sondern das Vorhandensein von zwei oder mehr Parsebäumen. Folglich sagen wir, eine kfG $G = (V, T, P, S)$ ist *mehrdeutig*, wenn T^* mindestens eine Zeichenreihe w enthält, für die zwei verschiedene Parsebäume gefunden werden können, die jeweils die Wurzel S und das Ergebnis w besitzen. Wenn eine Grammatik für jede Zeichenreihe höchstens einen Parsebaum erzeugt, dann ist die Grammatik *eindeutig*.

So führte etwa das Beispiel 5.25 die Mehrdeutigkeit der Grammatik aus Tabelle 5.2 vor Augen. Wir müssen lediglich zeigen, dass die in Abbildung 5.13 dargestellten Bäume vervollständigt werden können, sodass sie terminale Zeichenreihen ergeben. Abbildung 5.14 zeigt ein Beispiel für diese Vervollständigung.

Abbildung 5.14: Bäume mit dem Ergebnis $a + a * a$, die die Mehrdeutigkeit der Grammatik für einfache Ausdrücke demonstrieren



5.4.2 Mehrdeutigkeit aus Grammatiken tilgen

In einer perfekten Welt wären wir in der Lage, einen Algorithmus anzugeben, mit dem sich die Mehrdeutigkeit von Grammatiken entfernen ließe, ähnlich wie wir in Abschnitt 4.4 einen Algorithmus vorstellen konnten, mit dem unnötige Zustände aus einem endlichen Automaten entfernt werden können. Wie wir in Abschnitt 9.5.2. zeigen werden, ist es aber erstaunlicherweise sogar so, dass es nicht einmal einen Algorithmus gibt, mit dem sich feststellen lässt, ob eine kontextfreie Grammatik mehrdeutig ist. Wir werden in Abschnitt 5.4.4 sogar sehen, dass es kontextfreie Sprachen gibt, die ausschließlich mehrdeutige Grammatiken besitzen. Bei diesen Sprachen ist es unmöglich, die Mehrdeutigkeit zu tilgen.

Auflösung von Mehrdeutigkeiten in YACC

Wenn die von uns verwendete Grammatik für einfache Ausdrücke mehrdeutig ist, dann stellt sich die Frage, ob das in Listing 5.8 dargestellte YACC-Beispielprogramm realistisch ist. Es ist zwar wahr, dass die zu Grunde liegende Grammatik mehrdeutig ist, aber ein Großteil der Stärke des YACC-Parsergenerators ergibt sich daraus, dass er dem Benutzer einfache Mechanismen zur Verfügung stellt, mit denen sich viele der häufigsten Ursachen von Mehrdeutigkeiten auflösen lassen. Im Fall der Grammatik für einfache Ausdrücke genügt es, darauf zu bestehen, dass

a) $*$ Vorrang vor + hat. Das heißt, die Operanden von $*$ müssen gruppiert werden, bevor unmittelbar links oder rechts stehende Operanden von + gruppiert werden.

b) Sowohl $*$ als auch + sind linksassoziativ. Das heißt, Folgen von Ausdrücken, die durch $*$ miteinander verbunden sind, werden von links nach rechts gruppiert, und das Gleiche gilt für Ausdrücke, die durch + miteinander verbunden sind.

YACC ermöglicht uns die Angabe der Auswertungsreihenfolge von Operatoren, indem wir Operatoren in der Reihenfolge von der niedrigsten zur höchsten Priorität auflisten. Technisch gesehen wirkt sich die Priorität eines Operators auf die Verwendung jeder Produktion aus, in der dieser Operator das äußerste rechte terminale Symbol des Produktionsrumpfs bildet. Mit den Schlüsselwörtern `%left` und `%right` können Operatoren zudem als links- bzw. rechtsassoziativ deklariert werden. Beispielsweise würden wir die folgenden Anweisungen vor die Grammatik in Listing 5.8 einfügen, um anzugeben, dass die Operatoren $*$ und + linksassoziativ sind und dass $*$ Vorrang vor $*$ hat.

```
%left '+'
%left '*'
```

Glücklicherweise ist dieses Problem in der Praxis nicht so ernst. Für die Arten von Konstrukten, die in üblichen Programmiersprachen vorkommen, gibt es bekannte Techniken zur Auflösung von Mehrdeutigkeiten. Die mit der Grammatik für einfache Ausdrücke verbundene Problemstellung ist recht typisch, und wir werden die Auflösung ihrer Mehrdeutigkeit als wichtiges Beispiel untersuchen.

Zuerst wollen wir aber darauf hinweisen, dass in der Grammatik aus Tabelle 5.2 zwei Ursachen für Mehrdeutigkeiten gegeben sind:

1. Die Auswertungsreihenfolge der Operatoren wird nicht beachtet. Obwohl in Abbildung 5.13 (a) der Operator $*$ richtigerweise vor dem Operator + gruppiert wird, stellt auch Abbildung 5.13 (b) einen gültigen Parsebaum dar, in

dem + vor ∗ gruppiert wird. Wir müssen erzwingen, dass lediglich die Struktur von Abbildung 5.13 (a) in einer eindeutigen Grammatik zulässig ist.

2. Eine Folge identischer Operatoren kann sowohl von links nach rechts als auch von rechts nach links gruppiert werden. Wenn wir in Abbildung 5.13 die Multiplikationszeichen durch Pluszeichen ersetzen würden, dann würden sich für die Zeichenreihe $E + E + E$ auch zwei verschiedene Parsebäume ergeben. Da Addition und Multiplikation assoziativ sind, ist es gleichgültig, ob wir sie von links nach rechts oder von rechts nach links gruppieren. Um Mehrdeutigkeiten zu eliminieren, müssen wir jedoch eine Richtung auswählen. Der konventionelle Ansatz besteht darin, die Gruppierung von links nach rechts vorzuschreiben, sodass die Struktur von Abbildung 5.13 (b) die einzig korrekte Gruppierung der beiden Pluszeichen ist.

Man kann eine bestimmte Auswertungsreihenfolge erzwingen, indem man verschiedene Variablen einführt, die jeweils Ausdrücke mit der gleichen »Bindungsstärke« repräsentieren. Im Einzelnen gilt:

1. Ein *Faktor* ist ein Ausdruck, der sich weder mit dem Operator + noch mit dem Operator ∗ aufspalten lässt. In unserer Ausdruckssprache gibt es lediglich folgende Faktoren:

   a) Bezeichner. Es ist nicht möglich, die Buchstaben eines Bezeichners durch Hinzufügen eines Operators zu trennen.

   b) Jeder in Klammern gesetzte Ausdruck, ungeachtet dessen Inhalt. Klammern sollen verhindern, dass der Ausdruck, der innerhalb der Klammern steht, Operand eines Operators außerhalb der Klammern wird.

2. Ein *Term* ist ein Ausdruck, der nicht durch den Operator + aufgegliedert werden kann. In unserem Beispiel, in dem + und ∗ die einzigen Operatoren sind, ist ein Term ein Produkt mit einem oder mehreren Faktoren. Der Term $a * b$ kann z. B. »aufgegliedert« werden, indem wir die Linksassoziativität nutzen und ihm den Ausdruck $a1 *$ voranstellen. Das heißt, $a1 * a * b$ wird dann in der Form $(a1 * a) * b$ gruppiert, wodurch der ursprüngliche Ausdruck $a * b$ aufgespalten wird. Der Ausdruck $a * b$ lässt sich allerdings nicht durch Hinzufügen eines Terms der Form $a1 +$ oder $+ a1$ aufspalten. Die korrekte Gruppierung des Ausdrucks $a1 + a * b$ lautet $a1 + (a * b)$ und die des Ausdrucks $a * b + a1$ lautet $(a * b) + a1$.

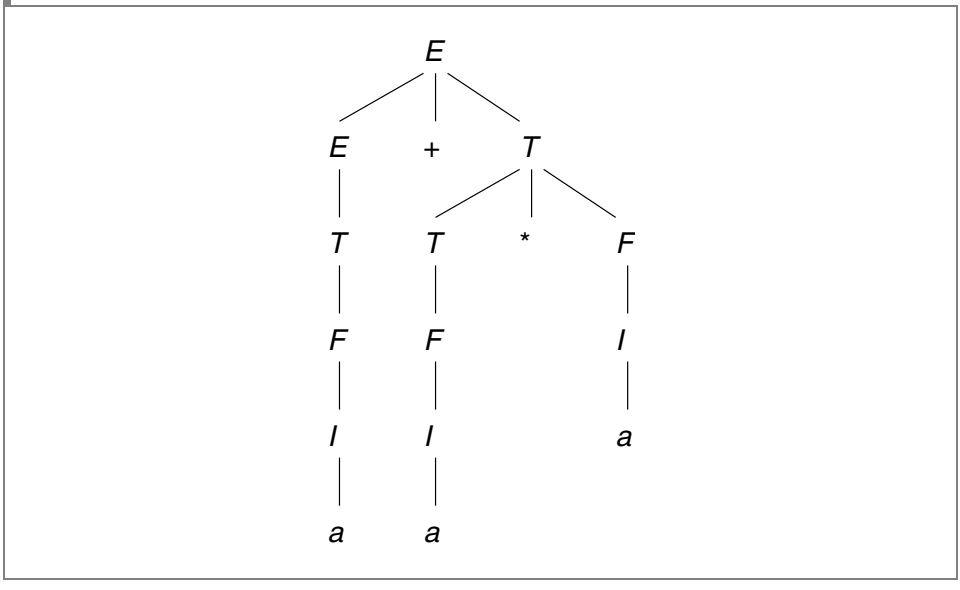
3. Mit *Ausdruck* ist fortan jeder mögliche Ausdruck gemeint, einschließlich solcher Ausdrücke, die sich durch Voranstellen oder Anhängen der Operatoren + und ∗ aufspalten lassen. In unserem Beispiel ist ein Ausdruck folglich die Summe von einem oder mehreren Termen.

Tabelle 5.5: Eine eindeutige Grammatik für einfache Ausdrücke

I	$\rightarrow$	$a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
F	$\rightarrow$	$I \mid (E)$
T	$\rightarrow$	$F \mid T * F$
E	$\rightarrow$	$T \mid E + T$

Beispiel 5.27 Tabelle 5.5 zeigt eine eindeutige Grammatik, die die gleiche Sprache generiert wie die Grammatik aus Tabelle 5.2. Stellen Sie sich *F*, *T* und *E* als die Variablen vor, deren Sprachen die Faktoren, Terme und Ausdrücke gemäß den obigen Definitionen sind. Diese Grammatik lässt beispielsweise nur einen Parsebaum für die Zeichenreihe $a + a * a$ zu; dieser Baum ist in Abbildung 5.15 dargestellt.

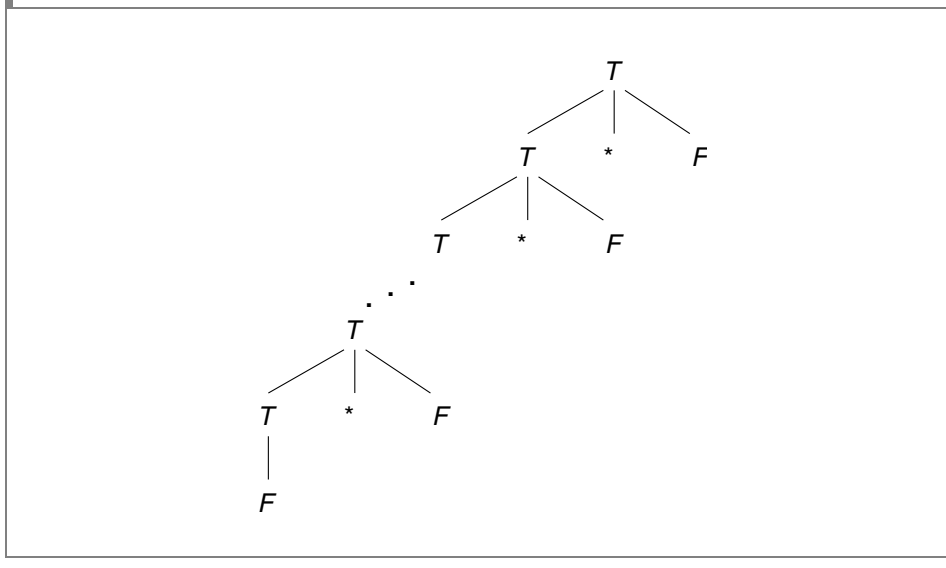
Abbildung 5.15: Der einzige Parsebaum für $a + a * a$



Die Tatsache, dass diese Grammatik eindeutig ist, mag nicht unbedingt offensichtlich sein. Es folgen die verschiedenen Beobachtungen, die erklären, warum keine Zeichenreihe der Sprache zwei verschiedene Parsebäume besitzen kann:

- Jede von *T* abgeleitete Zeichenreihe, ein Term, muss aus einer Folge von einem oder mehreren Faktoren bestehen, die durch den Operator * verbunden sind. Ein Faktor ist gemäß unserer Definition und wie aus den Produktionen für *F* in Tabelle 5.5 hervorgeht entweder ein einzelner Bezeichner oder jeder in Klammern gesetzte Ausdruck.
- Wegen der Form der beiden Produktionen für *T* besteht der einzige Parsebaum für eine Folge von Faktoren in dem Parsebaum, der $f_1 * f_2 * ... * f_n$ für $n > 1$ in einen Term $f_1 * f_2 * ... * f_{n-1}$ und einen Faktor f_n aufspaltet. Dies ist deshalb so, weil *F* Ausdrücke wie $f_{n-1} * f_n$ nicht ableiten kann, ohne sie in Klammern zu setzen. Folglich ist es nicht möglich, dass *F* beim Einsatz der Produktion $T \rightarrow T * F$ irgendetwas anderes als den letzten Faktor ableitet. Das heißt, der Parsebaum für einen Term kann nur wie der in Abbildung 5.16 aussehen.
- Analog hierzu ist ein Ausdruck eine Folge von durch + miteinander verbundenen Termen. Wenn wir die Produktion $E \rightarrow E + T$ zur Ableitung von $t_1 + t_2 + ... + t_n$ verwenden, dann darf das *T* lediglich t_n ableiten, und das *E* im Produktionsrumpf leitet $t_1 + t_2 + ... + t_{n-1}$ ab. Der Grund ist auch hier wieder, dass *T* nicht die Summe von zwei oder mehr Termen ableiten kann, ohne diese in Klammern zu setzen.

Abbildung 5.16: Die Form aller Parsebäume für einen Term



5.4.3 Linksseitige Ableitungen als Möglichkeit zur Beschreibung von Mehrdeutigkeit

Während Ableitungen nicht notwendigerweise eindeutig sind, auch wenn die Grammatik eindeutig ist, stellt sich heraus, dass in einer eindeutigen Grammatik linksseitige und rechtsseitige Ableitungen eindeutig sind. Wir betrachten hier lediglich die linksseitigen Ableitungen und geben das Ergebnis für rechtsseitige Ableitungen an.

Beispiel 5.28 Betrachten Sie als Beispiel die beiden Parsebäume in Abbildung 5.14, die dasselbe Ergebnis haben. Wenn wir daraus linksseitige Ableitungen konstruieren, dann erhalten wir die folgenden linksseitigen Ableitungen von Baum (a) bzw. (b):

a) $$E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} I+E \underset{lm}{\Rightarrow} a+E \underset{lm}{\Rightarrow} a+E*E \underset{lm}{\Rightarrow} a+I*E \underset{lm}{\Rightarrow} a+a*E \underset{lm}{\Rightarrow} a+a*I \underset{lm}{\Rightarrow} a+a*\boldsymbol{a}$$

b) $$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} E+E*E \underset{lm}{\Rightarrow} I+E*E \underset{lm}{\Rightarrow} a+E*E \underset{lm}{\Rightarrow} a+I*E \underset{lm}{\Rightarrow} a+a*E \underset{lm}{\Rightarrow} a+a*I \underset{lm}{\Rightarrow} \boldsymbol{a}+a*a$$

Beachten Sie, dass sich diese beiden linksseitigen Ableitungen unterscheiden. Dieses Beispiel beweist den Satz zwar nicht, illustriert aber, dass die Unterschiedlichkeit der Bäume unterschiedliche Schritte in der linksseitigen Ableitung erzwingt. ■

Satz 5.29 Für jede Grammatik $G = (V, T, P, S)$ und jede Zeichenreihe w in T^* gilt, dass w genau dann zwei verschiedene Parsebäume besitzt, wenn w über zwei unterschiedliche linksseitige Ableitungen aus S verfügt.

Beweis: (Nur-wenn-Teil) Wenn wir die Konstruktion einer linksseitigen Ableitung aus einem Parsebaum im Beweis von Satz 5.14 untersuchen, dann sehen wir, dass an all den Stellen, an denen die beiden Parsebäume zuerst einen Knoten besitzen, an dem verschiedene Produktionen verwendet werden, auch die konstruierten linksseitigen Ableitungen verschiedene Produktionen verwenden und folglich verschiedene Ableitungen darstellen.

(Wenn-Teil) Obwohl wir bislang noch keine direkte Konstruktion eines Parsebaums aus einer linksseitigen Ableitung beschrieben haben, ist das Konzept nicht schwer zu verstehen. Man beginne mit der Konstruktion eines Baums, der lediglich über die Wurzel mit der Beschriftung S verfügt. Überprüfe die Ableitung Schritt für Schritt. In jedem Schritt wird eine Variable ersetzt, und diese Variable entspricht dem äußersten linken Knoten des zu konstruierenden Baums, der keine Nachfolgerknoten, aber eine Variable in seiner Beschriftung besitzt. Aus der in diesem Schritt der linksseitigen Ableitung verwendeten Produktion wird ermittelt, welche Nachfolgerknoten dieser Knoten haben sollte. Falls es zwei verschiedene linksseitige Ableitungen gibt, dann erhalten die konstruierten Knoten in dem ersten Schritt, an dem sich diese Ableitungen unterscheiden, verschiedene Listen von Nachfolgerknoten, und dieser Unterschied stellt sicher, dass die Parsebäume verschieden sind. ■

5.4.4 Inhärente Mehrdeutigkeit

Eine kontextfreie Sprache L wird als inhärent mehrdeutig bezeichnet, wenn sämtliche Grammatiken dieser Sprache mehrdeutig sind. Wenn auch nur eine Grammatik für L eindeutig ist, dann ist L eine eindeutige Sprache. Wir haben z. B. festgestellt, dass die Sprache der Ausdrücke, die von der in Tabelle 5.2 dargestellten Grammatik erzeugt wird, eigentlich eindeutig ist. Obwohl diese Grammatik mehrdeutig ist, gibt es eine andere Grammatik für dieselbe Sprache, die nicht mehrdeutig ist, nämlich die Grammatik aus Tabelle 5.5.

Wir werden hier nicht beweisen, dass es inhärent mehrdeutige Sprachen gibt. Stattdessen diskutieren wir ein Beispiel für eine Sprache, deren inhärente Mehrdeutigkeit bewiesen werden kann, und wir werden informell erläutern, warum jede Grammatik für diese Sprache mehrdeutig sein muss. Bei der fraglichen Sprache handelt es sich um

$$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

Das heißt, L besteht aus Zeichenreihen in $\mathbf{a}^+\mathbf{b}^+\mathbf{c}^+\mathbf{d}^+$, derart dass eine der beiden folgenden Aussagen gilt:

1. Die Anzahl der Symbole a und b ist gleich und die Anzahl der Symbole c und d ist gleich. Oder:
2. Die Anzahl der Symbole a und d ist gleich und die Anzahl der Symbole b und c ist gleich.

L ist eine kontextfreie Sprache. Die Grammatik für L ist in Tabelle 5.6 dargestellt. Die beiden Arten der in L enthaltenen Zeichenreihen werden durch zwei verschiedene Grupppen von Produktionen erzeugt.

Diese Grammatik ist mehrdeutig. So gibt es beispielsweise für die Zeichenreihe *aabbccdd* zwei verschiedene linksseitige Ableitungen:

1. $S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$

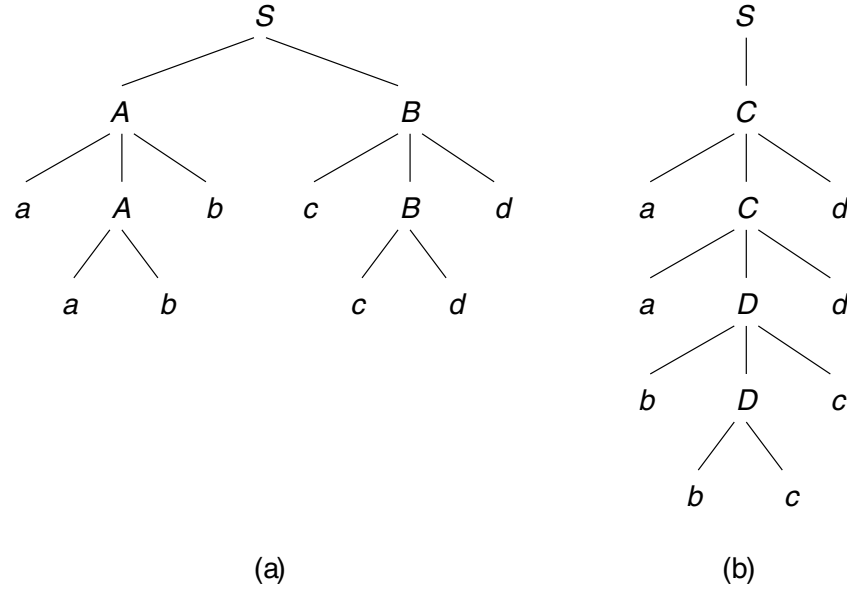
2. $S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$

und die beiden in Abbildung 5.17 dargestellten Parsebäume.

Tabelle 5.6: Grammatik für eine inhärent mehrdeutige Sprache

S	$\rightarrow$	$AB \mid C$
A	$\rightarrow$	$aAb \mid ab$
B	$\rightarrow$	$cBd \mid cd$
C	$\rightarrow$	$aCd \mid aDd$
D	$\rightarrow$	$bDc \mid bc$

Abbildung 5.17: Zwei Parsebäume für *aabbccdd*



Der Beweis, dass alle Grammatiken für L mehrdeutig sein müssen, ist kompliziert. Er lässt sich in groben Zügen wie folgt beschreiben. Wir müssen argumentieren, dass mit Ausnahme einer endlichen Anzahl alle Zeichenreihen, die über gleich viele Symbole a, b, c und d verfügen, auf zwei verschiedene Arten generiert werden müssen: Bei einem Verfahren müssen die Anzahl der generierten Symbole a und b und die Anzahl der generierten Symbole c und d gleich sein, und beim zweiten Verfahren müssen die

Anzahl der generierten Symbole a und d sowie die Anzahl der generierten Symbole b und c gleich sein.

Die einzige Möglichkeit zur Erzeugung von Zeichenreihen, die über die gleiche Anzahl der Symbole a und b verfügen, besteht in der Verwendung einer Variablen wie A in der Grammatik aus Tabelle 5.6. Natürlich gibt es Varianten, aber diese Varianten unterscheiden sich nicht grundsätzlich. Zum Beispiel:

- Einige kleine Zeichenreihen lassen sich vermeiden, indem die Basisproduktion $A \to ab$ beispielsweise in $A \to aaabbb$ abgeändert wird.
- Wir könnten es so einrichten, dass die Variable A ihre Aufgabe gemeinsam mit anderen Variablen erledigt, z. B. indem wir die Variablen A_1 und A_2 und folgende Produktionen einführen: $A_1 \to aA_2b \mid ab$; $A_2 \to aA_1b \mid ab$.
- Wir könnten es auch so einrichten, dass die Anzahl der von A erzeugten Symbole a und b nicht gleich ist, sondern sich um eine endliche Zahl unterscheidet. Wir könnten beispielsweise mit einer Produktion wie $S \to AbB$ beginnen und dann unter Verwendung der Produktion $A \to aAb \mid a$ eine um eins höhere Anzahl von Symbolen a als Symbole b generieren.

Wir können allerdings in keinem Fall vermeiden, dass es Mechanismen zur Erzeugung gleich vieler a und b geben muss.

Analog können wir argumentieren, dass es eine Variable wie B geben muss, die die gleiche Anzahl von Symbolen c und d erzeugt. Zudem müssen Variablen, die die Rolle von C (gleiche Anzahl von Symbolen a und d erzeugen) und D (gleiche Anzahl von Symbolen b und c erzeugen) übernehmen, in der Grammatik verfügbar sein. Wenn diese Argumentation formalisiert wird, beweist sie, dass die Grundgrammatik, gleichgültig, welche Modifikationen daran vorgenommen werden, stets zumindest *einige* Zeichenreihen der Form $a^nb^nc^nd^n$ auf zwei Arten generiert, wie es durch die Grammatik aus Tabelle 5.6 geschieht.

5.4.5 Übungen zum Abschnitt 5.4

* **Übung 5.4.1** Betrachten Sie die Grammatik

$$S \to aS \mid aSbS \mid \varepsilon$$

Diese Grammatik ist mehrdeutig. Zeigen Sie im Einzelnen, dass für die Zeichenreihe aab jeweils zwei Exemplare existieren von

a) Parsebäumen

b) linksseitigen Ableitungen

c) rechtsseitigen Ableitungen

! **Übung 5.4.2** Beweisen Sie, dass die Grammatik aus Übung 5.4.1 genau all jene Zeichenreihen aus den Symbolen a und b generiert, die so geformt sind, dass jedes Präfix mindestens so viele a wie b enthält.

*! **Übung 5.4.3** Finden Sie eine Grammatik für die Sprache aus Übung 5.4.1, die nicht mehrdeutig ist.

!! **Übung 5.4.4** Einige Zeichenreihen aus den Symbolen a und b verfügen in der Grammatik aus Übung 5.4.1 über einen eindeutigen Parsebaum. Formulieren Sie einen effizienten Test, mit dem sich feststellen lässt, ob es sich bei einer gegebenen Zeichenreihe um eine dieser Zeichenreihen handelt. Der Test »alle Parsebäume ausprobieren, um festzustellen, wie viele die gegebene Zeichenreihe ergeben« ist nicht ausreichend effizient.

! **Übung 5.4.5** Diese Frage betrifft die Grammatik aus Übung 5.1.2, die wir hier nochmals darstellen:

$$S \to A1B$$
$$A \to 0A \mid \varepsilon$$
$$B \to 0B \mid 1B \mid \varepsilon$$

a) Zeigen Sie, dass diese Grammatik eindeutig ist.

b) Finden Sie eine Grammatik für dieselbe Sprache, die mehrdeutig ist, und demonstrieren Sie deren Mehrdeutigkeit.

*! **Übung 5.4.6** Ist Ihre Grammatik aus Übung 5.1.5. eindeutig? Falls nicht, formulieren Sie sie um, sodass sie nicht mehrdeutig ist.

Übung 5.4.7 Die folgende Grammatik erzeugt *Präfix* -Ausdrücke mit den Operanden x und y sowie den binären Operatoren +, – und ∗:

$$E \to +EE \mid *EE \mid -EE \mid x \mid y$$

a) Finden Sie links- und rechtsseitige Ableitungen sowie einen Ableitungsbaum für die Zeichenreihe `+*-xyxy`.

! b) Beweisen Sie, dass diese Grammatik eindeutig ist.

5.5 Zusammenfassung von Kapitel 5

- *Kontextfreie Grammatiken (kfG)*: Eine kfG ist ein Verfahren zur Beschreibung von Sprachen durch rekursive Regeln, die als Produktionen bezeichnet werden. Eine kfG besteht aus einer Menge von Variablen, einer Menge von terminalen Symbolen und einem Startsymbol sowie den Produktionen. Jede Produktion setzt sich aus einer Kopfvariablen und einem Rumpf zusammen, der eine Zeichenreihe aus einer oder mehreren Variablen und/oder terminalen Symbolen umfasst.
- *Ableitungen und Sprachen*: Ausgehend vom Startsymbol leiten wir terminale Zeichenreihen ab, indem wir wiederholt eine Variable durch den Rumpf einer Produktion ersetzen, die diese Variable im Kopf enthält. Die Sprache der kfG besteht aus der Menge terminaler Zeichenreihen, die auf diese Weise abgeleitet werden können, und wird als kontextfreie Sprache bezeichnet.
- *Links- und rechtsseitige Ableitungen*: Wenn wir stets die äußerste linke (bzw. äußerste rechte) Variable einer Zeichenreihe ersetzen, dann wird die resultierende Ableitung als linksseitige bzw. rechtsseitige Ableitung bezeichnet. Jede Zeichenreihe der Sprache einer kfG besitzt mindestens eine linksseitige und mindestens eine rechtsseitige Ableitung.

- *Satzformen*: Jeder Schritt einer Ableitung repräsentiert eine Zeichenreihe aus Variablen und/oder terminalen Symbolen. Wir bezeichnen eine solche Zeichenreihe als Satzform. Wenn die Ableitung linksseitig (bzw. rechtsseitig) ist, dann ist diese Zeichenreihe eine linksseitige (bzw. rechtsseitige) Satzform.
- *Parsebäume*: Ein Parsebaum ist ein Baum, der die wichtigsten Bestandteile einer Ableitung zeigt. Innere Knoten werden mit den Variablen und Blattknoten mit den terminalen Symbolen oder ε beschriftet. Zu jedem inneren Knoten muss eine Produktion vorhanden sein, derart dass der Kopf der Produktion die Beschriftung des Knotens bildet und die Nachfolgerknoten von links nach rechts gelesen den Rumpf der Produktion ergeben.
- *Äquivalenz von Parsebäumen und Ableitungen*: Eine terminale Zeichenreihe ist in der Sprache einer Grammatik genau dann enthalten, wenn sie das Ergebnis mindestens eines Parsebaums ist. Folglich stellt das Vorhandensein von linksseitigen Ableitungen, rechtsseitigen Ableitungen und Parsebäumen äquivalente Bedingungen dar, die jeweils genau die in der Sprache einer kfG enthaltenen Zeichenreihen definieren.
- *Mehrdeutige Grammatiken*: Bei einigen kfGs ist es möglich, dass eine terminale Zeichenreihe über mehrere Parsebäume, mehrere linksseitige Ableitungen und mehrere rechtsseitige Ableitungen verfügt. Eine solche Grammatik wird mehrdeutig genannt.
- *Eliminierung von Mehrdeutigkeit*: Für viele nützliche Grammatiken, wie z. B. Grammatiken, die die Struktur von in einer typischen Programmiersprache geschriebenen Programmen beschreiben, lässt sich eine eindeutige Grammatik finden, die dieselbe Sprache erzeugt. Leider ist diese eindeutige Grammatik häufig viel komplexer als die einfache mehrdeutige Grammatik der Sprache. Es gibt zudem einige (meist recht künstliche) kontextfreie Sprachen, die als inhärent mehrdeutig bezeichnet werden, da jede Grammatik dieser Sprache mehrdeutig ist.
- *Parser*: Kontextfreie Grammatiken stellen das grundlegende Konzept für die Implementierung von Compilern und andere Verarbeitungsprogramme von Programmiersprachen dar. Dienstprogramme wie YACC erhalten eine kfG als Eingabe und erzeugen einen Parser, d. h. die Komponente eines Compilers, die die Struktur der Programme erschließt.
- *Dokumenttypdefinition (DTD)*: Der sich ausbildende XML-Standard für den Datenaustausch über Internetdokumente besitzt eine Notation, die so genannte DTD, zur Beschreibung der Struktur solcher Dokumente, die auf dem verschachtelten Einfügen semantischer Tags in die Dokumente beruht. Die DTD ist im Grunde genommen eine kontextfreie Grammatik, deren Sprache jeweils Klassen gleichartiger Dokumente beschreibt.

LITERATURANGABEN ZU KAPITEL 5

Kontextfreie Grammatiken wurden erstmals von Chomsky [4] als Beschreibungsmethode für natürliche Sprachen vorgeschlagen. Ein ähnliches Konzept wurde kurz danach zur Beschreibung von Computersprachen eingesetzt – Fortran von Backus [2] und Algol von Naur [7]. Infolgedessen werden kontextfreie Grammatiken gelegentlich auch als »Grammatiken in Backus-Naur-Form« bezeichnet.

Die Mehrdeutigkeit von Grammatiken wurde von Cantor [3] und Floyd [5] etwa zur selben Zeit als Problem erkannt. Gross [6] geht als Erster auf die inhärente Mehrdeutigkeit ein.

Anwendungen von kfG in Compilern finden Sie in [1]. DTDs werden in der Beschreibung des XML-Standards definiert [8].

1. A. V. Aho, R. Sethi und J. D. Ullman [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA.
2. J. W. Backus [1959]. »The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference, *Proc. Intl. Conf. on Information Processing*, UNESCO, S. 125–132.
3. D. C. Cantor [1962]. »On the ambiguity problem of Backus systems«, *J. ACM* **9**:4, S. 477–479.
4. N. Chomsky [1956]. »Three models for the description of language«, *IRE Trans. on Information Theory* **2**:3, S. 113–124.
5. R. W. Floyd [1962]. »On ambiguity in phrase-structure languages«, *Comm. ACM* **5**:10, S. 526–534.
6. M. Gross [1964]. »Inherent ambiguity of minimal linear grammars«, *Information and Control* **7**:3, S. 366–368.
7. P. Naur et al. [1960]. »Report on the algorithmic language ALGOL 60«, *Comm. ACM* **3**:5, S. 299–314. Siehe auch *Comm. ACM* **6**:1 (1963), S. 1–17.
8. World-Wide-Web Consortium, `http://www.w3.org/TR/REC-xml` (1998).

KAPITEL 6

Pushdown-Automaten

Es gibt einen Automatentyp, der kontextfreie Sprachen definiert. Dieser Automat, der so genannte »Pushdown-Automat« (PDA) oder Kellerautomat[1], ist eine Erweiterung der nichtdeterministischen endlichen Automaten mit ε-Übergängen, die eine Möglichkeit zur Definition regulärer Sprachen darstellen. Bei dem Pushdown-Automaten handelt es sich im Grunde genommen um einen ε-NEA, dem ein Stack hinzugefügt wurde. Der Stack kann gelesen werden und es gibt die pop- und die push-Operation. Alle drei Operationen können nur zuoberst im Stack ausgeführt werden.

In diesem Kapitel definieren wir zwei verschiedene Varianten des Pushdown-Automaten: einen Automaten, der wie endliche Automaten durch den Übergang in einen akzeptierenden Zustand akzeptiert, und einen zweiten Automatentyp, der akzeptiert, indem er ungeachtet seines aktuellen Zustands den Stack leert. Wir zeigen, dass diese beiden Varianten genau die kontextfreien Sprachen akzeptieren, das heißt, Grammatiken können in äquivalente Pushdown-Automaten umgewandelt werden und umgekehrt. Wir werden zudem kurz die Unterklasse der Pushdown-Automaten betrachten, die deterministisch ist. Diese Automaten akzeptieren alle regulären Sprachen, aber lediglich eine echte Teilmenge der kontextfreien Sprachen. Da sie den Mechanismen des Parsers eines typischen Compilers stark ähneln, ist die Beobachtung wichtig, welche Sprachkonstrukte von deterministischen Pushdown-Automaten erkannt werden können und welche nicht.

6.1 Definition der Pushdown-Automaten

In diesem Abschnitt stellen wir den Pushdown-Automaten zuerst informell und dann als formales Konstrukt vor.

6.1.1 Informelle Einführung

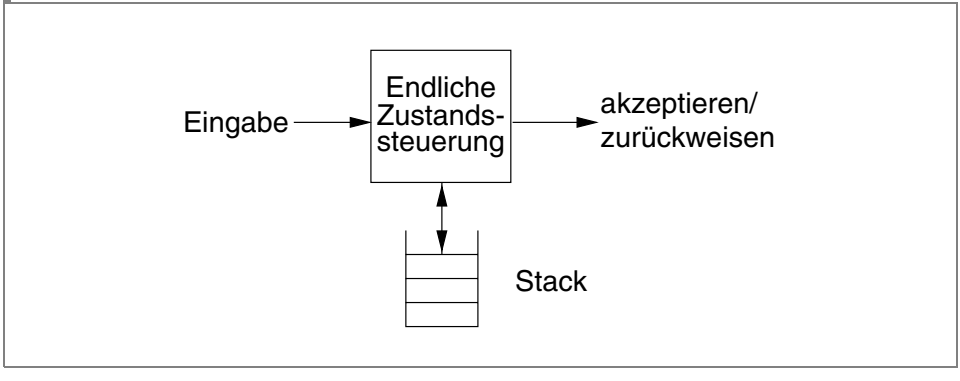
Der Pushdown-Automat ist im Grunde genommen ein nichtdeterministischer endlicher Automat mit ε-Übergängen, der über ein zusätzliches Leistungsmerkmal verfügt: einen Stack, auf dem er eine Zeichenreihe von »Stacksymbolen« speichern kann. Das Vorhandensein dieses Stacks bedeutet, dass sich der Pushdown-Automat im Gegensatz zum endlichen Automaten eine unendliche Menge von Informationen

1. In der deutschen Fachliteratur wird die Bezeichnung Kellerautomat bevorzugt verwendet. Wir bleiben im Folgenden aber bei der englischen Bezeichnung Pushdown-Automat, nachdem auch diese Bezeichnung in der deutschen Literatur akzeptiert ist.

»merken« kann. Im Gegensatz zu einem allgemeinen Computer, der ebenfalls über die Fähigkeit verfügt, sich beliebig große Mengen von Informationen zu merken, kann der Pushdown-Automat lediglich über ein LIFO-Verfahren auf die Informationen auf seinem Stack zugreifen.

Infolgedessen gibt es Sprachen, die von einem Computerprogramm erkannt werden könnten, aber für einen Pushdown-Automaten nicht erkennbar sind. In der Tat erkennen Pushdown-Automaten lediglich genau alle kontextfreien Sprachen. Obwohl es viele kontextfreie Sprachen gibt, von denen wir einige gesehen haben, die nicht regulär sind, existieren auch einige einfach zu beschreibende Sprachen, die nicht kontextfrei sind (siehe Abschnitt 7.2). Ein Beispiel für eine nicht kontextfreie Sprache ist $\{0^n1^n2^n \mid n \geq 1\}$, die Menge der Zeichenreihen, die aus gleich großen Gruppen der Symbole 0, 1 und 2 besteht.

Abbildung 6.1: Ein Pushdown-Automat ist im Grunde genommen ein endlicher Automat mit einer Stack-Datenstruktur



Wir können den Pushdown-Automaten informell als Gerät betrachten, das der Darstellung in Abbildung 6.1 gleicht. Eine »endliche Zustandssteuerung« liest nacheinander die einzelnen Symbole der Eingabe. Der Pushdown-Automat kann das Symbol, das sich oben auf dem Stack befindet, beobachten und seinen Zustandsübergang abhängig machen von seinem aktuellen Zustand, dem Eingabesymbol und dem Symbol am oberen Ende des Stacks. Zudem kann er »spontane« Übergänge durchführen, indem er statt des Eingabesymbols ε als Eingabe verwendet. Der Pushdown-Automat führt im Rahmen eines Übergangs Folgendes aus:

1. Er entnimmt der Eingabe das Symbol, das er für den Zustandsübergang verwendet. Wird ε als Eingabe genommen, dann wird kein Eingabesymbol verbraucht.
2. Er geht in einen anderen Zustand über, der mit dem aktuellen Zustand identisch sein kann.
3. Er ersetzt das Symbol am oberen Ende des Stacks durch eine beliebige Zeichenreihe. Die Zeichenreihe kann ε sein; das entspricht einer pop-Operation. Sie kann dasselbe Symbol sein, das aktuell zuoberst im Stack steht, d.h., der Stack bleibt unverändert. Sie kann auch das oberste Stacksymbol durch ein anderes ersetzen; das entspricht einer pop-Operation, gefolgt von einer push-

Operation. Schließlich kann das oberste Stacksymbol durch eine Zeichenreihe einer Länge n > 1 ersetzt werden; das entspricht einer pop-Operation, gefolgt von n push-Operationen.

Beispiel 6.1 Wir wollen folgende Sprache betrachten:

$$L_{wwr} = \{ww^R \mid w \text{ ist in } (\mathbf{0} + \mathbf{1})^* \text{ enthalten}\}$$

Diese Sprache besteht aus Palindromen mit einer geraden Anzahl von Zeichen über dem Alphabet {0, 1}. Es ist eine kontextfreie Sprache, die von der Grammatik aus Tabelle 5.1 erzeugt wird, die um die Produktionen $P \to 0$ und $P \to 1$ verkürzt wurde.

Wir können wie folgt einen informellen Pushdown-Automaten entwerfen, der die Sprache L_{wwr} akzeptiert[2].

1. Wir beginnen in einem Zustand q_0, der die »Annahme« repräsentiert, dass die Mitte der Zeichenreihe noch nicht erreicht worden ist, d. h., dass das Ende der Zeichenreihe w, dem die Umkehrung von w folgt, noch nicht gelesen wurde. Solange sich der Automat im Zustand q_0 befindet, kann er Symbole lesen und auf dem Stack speichern, indem er nacheinander jeweils eine Kopie der einzelnen Eingabesymbole oben auf dem Stack ablegt.

2. Wir können jederzeit annehmen, dass die Mitte, d. h. das Ende von w, erreicht worden ist. w ist dann auf dem Stack, wobei sich das Ende von w am oberen Ende des Stacks und der Anfang von w am unteren Ende des Stacks befindet. Wir zeigen diese Annahme an, indem wir spontan in den Zustand q_1 übergehen. Da der Automat nichtdeterministisch ist, gehen wir aber gleichzeitig von der Annahme aus, dass das Ende von w noch nicht erreicht ist, d.h., der Automat behält auch den Zustand q_0 bei und fährt damit fort, Eingabesymbole zu lesen und sie auf dem Stack abzulegen.

3. Sobald der Zustand q_1 erreicht wurde, vergleichen wir die Eingabesymbole mit dem obersten Symbol im Stack. Stimmen diese Symbole überein, dann ist das Eingabesymbol verarbeitet und das oberste Symbol wird vom Stack entfernt. Stimmen sie nicht überein, dann war die Annahme falsch, dass das Ende der Zeichenreihe w erreicht worden war und w^R folgt. Dieser Zweig endet, aber es können noch andere Zweige des nichtdeterministischen Automaten vorhanden sein und schließlich zur Akzeptanz führen.

4. Wenn der Stack leer ist, wurde eine Eingabe w gefolgt von w^R gelesen. Wir akzeptieren die Eingabe, die bis zu diesem Zeitpunkt gelesen worden ist. ■

6.1.2 Formale Definition des Pushdown-Automaten

Unsere formale Notation für einen *Pushdown-Automaten* (PDA) umfasst sieben Komponenten. Wir geben die Spezifikation eines PDA P wie folgt an:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

2. Wir könnten ebenso einen Pushdown-Automaten für L_{pal} entwerfen, also die Sprache, deren Grammatik in Tabelle 5.1 dargestellt ist. L_{wwr} ist jedoch etwas einfacher und erlaubt es uns, uns auf die wichtigen Konzepte im Zusammenhang mit Pushdown-Automaten zu konzentrieren.

Diese Komponenten haben die folgende Bedeutung:

Q: eine endliche Menge von *Zuständen* , vergleichbar mit den Zuständen eines endlichen Automaten.

Σ: eine endliche Menge von *Eingabesymbolen*, die ebenfalls mit der entsprechenden Komponente eines endlichen Automaten vergleichbar ist.

Γ: ein endliches *Stackalphabet*. Diese Komponente, für die es keine Entsprechung in endlichen Automaten gibt, besteht aus der Menge der Eingabesymbole, die auf dem Stack abgelegt werden dürfen.

δ: die *Übergangsfunktion* . Wie bei endlichen Automaten, bestimmt die Übergangsfunktion δ das Verhalten des Automaten. Formal ausgedrückt, wird δ ein Tripel als Argument übergeben, $\delta(q, a, X)$, wobei

1. q ein in Q enthaltener Zustand ist.
2. a entweder ein Eingabesymbol aus Σ oder $a = \varepsilon$, die leere Zeichenreihe, ist, die nicht als Eingabesymbol verstanden wird.
3. X ein Stacksymbol, also ein Element von Γ ist.

Die Ausgabe von δ besteht aus einer endlichen Menge von Paaren (p, γ), wobei p für den neuen Zustand und γ für die Zeichenreihe der Stacksymbole steht, die X auf dem oberen Ende des Stacks ersetzt. Wenn z. B. $\gamma = \varepsilon$, dann wird eine pop-Operation ausgeführt, d.h., das oberste Stacksymbol wird gelöscht; wenn $\gamma = X$, dann bleibt der Stack unverändert, und wenn $\gamma = YZ$, dann wird X durch Z ersetzt und Y wird zuoberst auf dem Stack abgelegt.

q_0: der *Startzustand*. Der PDA befindet sich in diesem Zustand, bevor ein Zustandsübergang erfolgt ist.

Z_0: das *Startsymbol*. Anfangs enthält der Stack des PDA lediglich eine Instanz dieses Symbols.

F: die Menge der *akzeptierenden Zustände* oder *Endzustände* .

Kein »Durcheinander«

Es kann verschiedene Paare geben, die in manchen Situationen für einen PDA in Frage kommen. Nehmen wir beispielsweise an $\delta(q, a, X) = \{(p, YZ), (r, \varepsilon)\}$. Wenn wir eine Bewegung des PDA darstellen, müssen wir die Paare als Einheit behandeln; wir können nicht aus einem Paar einen Zustand und aus einem anderen Paar die auf dem Stack zu ersetzende Zeichenreihe wählen. Wenn sich der PDA im Zustand q befindet, X das oberste Stacksymbol ist und die Eingabe a gelesen wird, könnten wir in den Zustand p wechseln und X durch YZ ersetzen oder wir könnten in den Zustand r wechseln und X vom Stack entfernen. Dagegen ist es nicht möglich, in den Zustand p zu wechseln und X vom Stack zu entfernen oder in den Zustand r zu wechseln und X durch YZ zu ersetzen.

Beispiel 6.2: Wir wollen einen PDA P entwerfen, der die Sprache L_{wwr} aus Beispiel 6.1 akzeptiert. Einige Details, die wir zur Verwaltung des Stacks benötigen, sind in jenem Beispiel nicht gegeben. Wir werden das Stacksymbol Z_0 verwenden, um das

untere Ende des Stacks zu markieren. Dieses Symbol muss gegeben sein, damit noch etwas auf dem Stack ist, das einen Übergang in den akzeptierenden Zustand q_2 ermöglicht, nachdem wir w vom Stack entfernt und erkannt haben, dass die Eingabe ww^R gelesen worden ist. Somit kann unser PDA für L_{wwr} wie folgt beschrieben werden:

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

wobei δ durch die folgenden Regeln definiert wird:

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0\ Z_0)\}$ und $\delta(q_0, 1, Z_0) = \{(q_0, 1\ Z_0)\}$. Eine dieser Regeln trifft anfangs zu, wenn sich der Automat im Zustand q_0 und das Startsymbol Z_0 sich oben auf dem Stack befindet. Wir lesen das erste Eingabezeichen und legen es auf dem Stack ab, wobei Z_0 darunter auf dem Stack verbleibt und das untere Ende des Stacks markiert.

2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$ und $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. Diese vier Regeln ermöglichen es uns, den Zustand q_0 beizubehalten und Eingaben zu lesen, wobei die gelesenen Eingabesymbole auf dem Stack abgelegt und die auf dem Stack vorhandenen Symbole nicht verändert werden.

3. $\delta(q_0, \varepsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \varepsilon, 0) = \{(q_1, 0)\}$, $\delta(q_0, \varepsilon, 1) = \{(q_1, 1)\}$. Diese Regeln erlauben es P, vom Zustand q_0 spontan (mit der Eingabe ε) in den Zustand q_1 zu wechseln, ohne das Symbol am oberen Ende des Stacks zu verändern.

4. $\delta(q_1, 0, 0) = \{(q_1, \varepsilon)\}$ und $\delta(q_1, 1, 1) = \{(q_1, \varepsilon)\}$. Wenn sich der Automat im Zustand q_1 befindet, dann kann er ein Eingabesymbol mit dem Symbol am oberen Ende des Stacks vergleichen und bei Übereinstimmung das oberste Symbol vom Stack löschen.

5. $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$. Wenn der Automat die Stackanfangsmarkierung Z_0 erreicht und sich im Zustand q_1 befindet, dann wurde eine Eingabe der Form ww^R gelesen. Der Automat geht in den Zustand q_2 über und akzeptiert. ■

6.1.3 Eine grafische Notation für PDAs

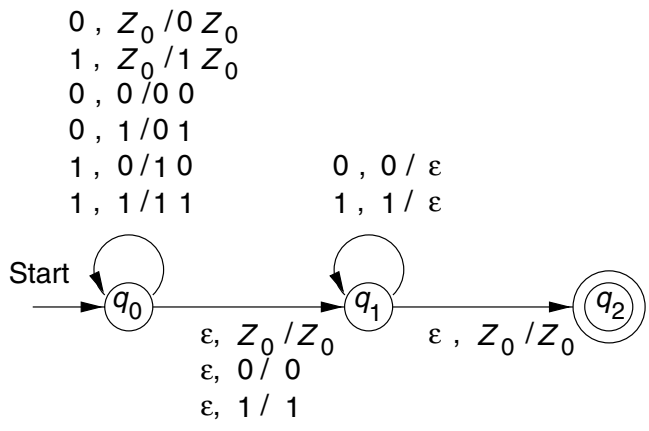
Es ist teilweise recht schwierig, eine Liste mit δ-Fakten wie in Beispiel 6.2 nachzuvollziehen. Unter Umständen kann ein Diagramm, das eine verallgemeinerte Form des Übergangsdiagramms für endliche Automaten ist, bestimmte Aspekte des Verhaltens eines gegebenen PDA verständlicher darstellen. Wir werden daher ein *Übergangsdiagramm* für PDAs vorstellen und fortan verwenden, in dem

a) die Knoten den Zuständen des PDA entsprechen.

b) ein Pfeil mit der Beschriftung *Start* den Startzustand anzeigt und akzeptierende Zustände durch eine doppelte Kreislinie gekennzeichnet sind, wie bei endlichen Automaten.

c) die Pfeile im folgenden Sinn die Übergänge des PDA widerspiegeln. Ein Pfeil mit der Beschriftung $a, X/\alpha$, der vom Zustand q zum Zustand p führt, bedeutet, dass $\delta(q, a, X)$ das Paar (p, α) – und möglicherweise andere Paare – enthält. Das heißt, aus der Beschriftung des Pfeils geht hervor, welches Eingabesymbol verwendet wird und wie das obere Ende des Stacks beim Übergang von q in p verändert wird.

Abbildung 6.2: Darstellung eines PDA als verallgemeinertes Übergangsdiagramm



Das Diagramm gibt nur darüber keinen Aufschluss, welches Stacksymbol das Startsymbol ist. Gemäß Konvention ist Z_0 das Startsymbol, sofern nicht anders angegeben.

Beispiel 6.3 Der PDA aus Beispiel 6.2 wird durch das Diagramm in Abbildung 6.2 repräsentiert. ■

6.1.4 Beschreibung der Konfiguration eines PDA

Bislang haben wir nur eine informelle Vorstellung davon, wie ein PDA »rechnet«. Der PDA wechselt in Reaktion auf Eingabesymbole (oder ε) von einer Konfiguration zur nächsten Konfiguration. Im Gegensatz zu den endlichen Automaten, bei denen lediglich der Zustand (neben dem Eingabesymbol) für einen Übergang von Bedeutung ist, umfasst die Konfiguration eines PDA sowohl den Zustand als auch den Inhalt des Stacks. Da der Stack beliebig groß sein kann, ist er häufig der wichtigere Teil der Gesamtkonfiguration eines PDA. Es ist zudem hilfreich, den verbleibenden Teil der Eingabe als Teil der Konfiguration darzustellen.

Daher werden wir die Konfiguration eines PDA durch ein Tripel (q, w, γ) darstellen, wobei

1. q für den Zustand steht.
2. w die verbleibende Eingabe repräsentiert.
3. γ für den Inhalt des Stacks steht.

Gemäß Konvention stellen wir das obere Ende des Stacks am linken Ende von γ und das untere Ende des Stacks am rechten Ende von γ dar. Ein solches Tripel wird als eine Konfiguration des PDA bezeichnet.
Für endliche Automaten genügte die $\hat{\delta}$-Notation, um Folgen von Konfigurationen zu beschreiben, da die Konfiguration hier lediglich aus dem Zustand besteht. Für PDAs ist jedoch eine Notation erforderlich, die Änderungen des Zustands, der Eingabe und

des Stacks repräsentiert. Daher verwenden wir folgende Notation zur Verkettung von Konfigurationen, die eine oder mehrere Bewegungen eines PDA repräsentieren.

Sei $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ein PDA. Wir definieren $\vdash_P$ oder einfach $\vdash$, wenn die Bezugnahme auf P klar ist, wie folgt. Angenommen, $\delta(q, a, X)$ enthält (p, α). Dann gilt für alle Zeichenreihen w aus Σ^* und β aus Γ^*:

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

Diese Bewegung spiegelt die Vorstellung wider, dass der Automat vom Zustand q in den Zustand p übergehen kann, indem er das Symbol a (das ε sein kann) aus der Eingabe einliest und X auf dem Stack durch α ersetzt. Beachten Sie, dass die restliche Eingabe w und der restliche Inhalt des Stacks β die Aktion des PDA nicht beeinflussen. Diese Daten werden einfach mitgeführt, weil sie womöglich spätere Ereignisse beeinflussen.

Wir verwenden das Symbol $\vdash_P^*$ oder $\vdash^*$, wenn die Bezugnahme auf P klar ist, um eine oder mehrere Bewegungen des PDA zu repräsentieren. Das heißt:

INDUKTIONSBEGINN: $I \vdash^* I$ für jede Konfiguration I.

INDUKTIONSSCHRITT: $I \vdash^* J$, wenn es eine Konfiguration K gibt, derart dass $I \vdash K$ und $K \vdash^* J$.

Das heißt, $I \vdash^* J$, wenn es eine Folge von Konfigurationen $K_1, K_2, ..., K_n$ gibt, derart dass $I = K_1$, $J = K_n$ und für alle $i = 1, 2, ..., n\text{-}1$ gilt $K_1 \vdash K_{i+1}$.

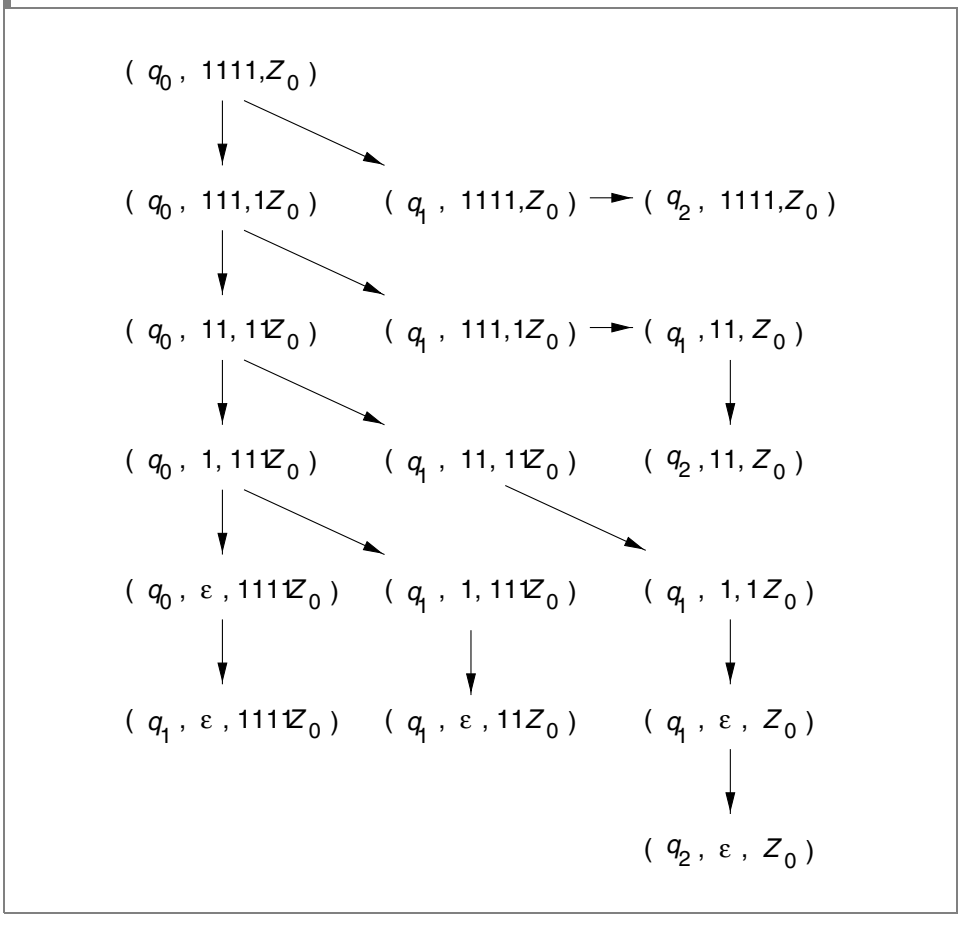
Beispiel 6.4 Betrachten wir das Verhalten des PDA aus Beispiel 6.2 bei der Eingabe 1111. Da q_0 der Startzustand und Z_0 das Startsymbol im Stack ist, lautet die anfängliche Konfiguration $(q_0, 1111, Z_0)$. Bei dieser Eingabe kann der PDA mehrmals falsche Annahmen treffen. Die gesamte Folge von Konfigurationen, die der PDA von der anfänglichen Konfiguration $(q_0, 1111, Z_0)$ ausgehend erreichen kann, ist in Abbildung 6.3 dargestellt. Die Relation $\vdash$ wird durch Pfeile repräsentiert.

Von der anfänglichen Konfiguration aus gibt es zwei Bewegungsmöglichkeiten. Die erste geht von der Annahme aus, dass die Mitte noch nicht erreicht worden ist, und führt zur Konfiguration $(q_0, 111, 1Z_0)$. Hier wurde eine 1 aus der Eingabe entfernt und auf dem Stack abgelegt.

Die zweite Bewegungsmöglichkeit von der anfänglichen Konfiguration aus basiert auf der Annahme, dass die Mitte erreicht worden ist. Ohne Zeichen aus der Eingabe zu entnehmen, geht der PDA in den Zustand q_1 über und erreicht damit die Konfiguration $(q_1, 1111, Z_0)$. Da der PDA im Zustand q_1 akzeptieren kann, wenn er Z_0 als oberstes Symbol auf dem Stack vorfindet, wechselt der PDA von dort zur Konfiguration $(q_2, 1111, Z_0)$. Diese Konfiguration ist tatsächlich keine akzeptierende Konfiguration, da nicht alle Zeichen der Eingabe verbraucht wurden. Hätte es sich bei der Eingabe um ε statt um 1111 gehandelt, hätte dieselbe Folge von Bewegungen zur Konfiguration (q_2, ε, Z_0) geführt, womit gezeigt wäre, dass ε akzeptiert wird.

Der PDA kann auch annehmen, dass er nach dem Lesen einer 1 bereits die Mitte erreicht hat, das heißt, wenn er die Konfiguration $(q_0\ 111, 1Z_0)$ hat. Auch diese Annahme führt nicht zur Akzeptanz, da nicht die gesamte Eingabe verbraucht werden kann. Die korrekte Annahme, nämlich dass die Mitte nach dem Lesen von zwei Einsen erreicht worden ist, ergibt die Konfigurationsfolge $(q_0, 1111, Z_0) \vdash (q_0, 111, 1Z_0) \vdash (q_0, 11, 11Z_0) \vdash (q_1, 11, 11Z_0) \vdash (q_1, 1, 1Z_0) \vdash (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$. ■

Abbildung 6.3: Konfigurationen des PDA aus Beispiel 6.2 bei der Eingabe 1111



Es gibt drei wichtige Grundregeln bezüglich Konfigurationen und deren Übergängen, die für die weiteren Überlegungen zu PDAs benötigt werden:

1. Wenn eine Folge von Konfigurationen (*Berechnung*) für einen PDA P zulässig ist, dann ist auch die Berechnung zulässig, die sich daraus ergibt, dass die gleiche zusätzliche Eingabezeichenreihe in jeder Konfiguration an das Ende der Eingabe (zweite Komponente) angehängt wird.
2. Wenn eine Berechnung für einen PDA P zulässig ist, dann ist auch die Berechnung zulässig, die sich daraus ergibt, dass die gleichen zusätzlichen Stacksymbole in jeder Konfiguration am unteren Ende des Stacks hinzugefügt werden.
3. Wenn eine Berechnung für einen PDA P zulässig ist und ein Teil der Eingabe nicht verbraucht worden ist, dann können wir in jeder Konfiguration diesen Teil aus der Eingabe entfernen, ohne dass die resultierende Berechnung unzulässig wird.

Intuitiv können Daten, die der P nie zu Gesicht bekommt, seine Berechnungen nicht beeinflussen. Wir formalisieren Punkt (1) und (2) in einem einzigen Satz.

Konventionen der Schreibweise für PDAs

Wir werden weiterhin Konventionen hinsichtlich der Verwendung von Symbolen verwenden, die wir für endliche Automaten und Grammatiken eingeführt haben. Bei der Übernahme der Notation ist es hilfreich, sich klar zu machen, dass die Stacksymbole eine ähnliche Rolle spielen wie die Vereinigung der terminalen Symbole und Variablen in einer kontextfreien Grammatik. Folglich definieren wir:

1. Symbole des Eingabealphabets werden durch Kleinbuchstaben vom Anfang des Alphabets, z. B. *a*, *b*, repräsentiert.
2. Zustände werden für gewöhnlich durch *q* und *p* oder andere im Alphabet benachbarte Kleinbuchstaben dargestellt.
3. Zeichenreihen aus Eingabesymbolen werden durch Kleinbuchstaben wie *w* und *z* vom Ende des Alphabets angegeben.
4. Stacksymbole werden durch Großbuchstaben wie X oder Y vom Ende des Alphabets repräsentiert.
5. Zeichenreihen aus Stacksymbolen werden durch griechische Buchstaben wie α oder γ dargestellt.

Satz 6.5 Wenn $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ein PDA ist und $(q, x, \alpha) \overset{*}{\vdash}_P (p, y, \beta)$, dann gilt für jede Zeichenreihe w aus Σ^* und jede Zeichenreihe γ aus Γ^*, dass

$$(q, xw, \alpha\gamma) \overset{*}{\vdash}_P (p, yw, \beta\gamma)$$

Beachten Sie, dass eine formale Aussage der oben beschriebenen Grundregel (1) vorliegt, wenn $\gamma = \varepsilon$, und dass eine formale Aussage der Grundregel (2) vorliegt, wenn $w = \varepsilon$.

Beweis: Der Beweis ist eine sehr einfache Induktion über die Anzahl der Schritte in der Folge von Konfigurationen, die $(q, xw, \alpha\gamma)$ in $(p, yw, \beta\gamma)$ überführen. Jede Bewegung in der Folge $(q, xw, \alpha\gamma) \overset{*}{\vdash}_P (p, yw, \beta\gamma)$ wird durch die Übergänge von P gerechtfertigt, ohne w und/oder γ in irgendeiner Weise zu verwenden. Daher ist auch dann jede Bewegung noch gerechtfertigt, wenn sich diese Zeichenreihen jeweils an der Eingabe bzw. am unteren Ende des Stacks befinden. ■

Beachten Sie, dass die Umkehrung des Satzes falsch ist. Es gibt Dinge, die der PDA tun kann, indem er einige Symbole von γ verwendet und sie dann wieder auf den Stack zurückschreibt, zu denen er nicht in der Lage wäre, wenn er γ nie betrachten würde. Wie Grundregel (3) besagt, können wir jedoch ungenutzte Eingabesymbole entfernen, da es für einen PDA nicht möglich ist, Eingabesymbole zu verbrauchen

und diese Symbole dann in der Eingabe wiederherzustellen. Wir formulieren Grundregel (3) wie folgt formal:

Satz 6.6 Wenn $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ein PDA ist und

$$(q, xw, \alpha) \overset{*}{\vdash_P} (p, yw, \beta),$$

dann gilt $(q, x, \alpha) \overset{*}{\vdash_P} (p, y, \beta)$. ■

Konfigurationen für endliche Automaten

Sie mögen sich vielleicht fragen, warum wir für endliche Automaten keine Notation wie die Konfigurationen, die wir für PDAs verwenden, eingeführt haben. Obwohl endliche Automaten keinen Stack besitzen, könnten wir Paare der Form (q, w), wobei q für einen Zustand und w für die restliche Eingabe steht, als Konfiguration eines endlichen Automaten verwenden.

Wir hätten dies zwar tun können, würden damit aber nicht mehr Informationen aus der Erreichbarkeit von Konfigurationen gewinnen als mit der $\hat{\delta}$-Notation. Das heißt, für endliche Automaten können wir zeigen, dass genau dann $\hat{\delta}(q, w) = p$ gilt, wenn $(q, wx) \vdash^* (p, x)$ für alle Zeichenreihen x. Die Tatsache, dass x beliebig gewählt werden kann, ohne dass diese Wahl das Verhalten des endlichen Automaten beeinflusst, ist ein Satz, analog zu den Sätzen 6.5 und 6.6.

6.1.5 Übungen zum Abschnitt 6.1

Übung 6.1.1 Angenommen der PDA $P = (\{q, p\}, \{0, 1\}, \{Z_0, X\}, \delta, q, Z_0, \{p\})$ besitzt die folgende Übergangsfunktion:

1. $\delta(q, 0, Z_0) = \{(q, XZ_0)\}$.
2. $\delta(q, 0, X) = \{(q, XX)\}$.
3. $\delta(q, 1, X) = \{(q, X)\}$.
4. $\delta(q, \varepsilon, X) = \{(p, \varepsilon)\}$.
5. $\delta(p, \varepsilon, X) = \{(p, \varepsilon)\}$.
6. $\delta(p, 1, X) = \{(p, XX)\}$.
7. $\delta(p, 1, Z_0) = \{(p, \varepsilon)\}$.

Zeigen Sie, ausgehend von der Anfangskonfiguration (q, w, Z_0), alle mit folgenden Eingaben erreichbaren Konfigurationen:

* a) 01.

b) 0011.

c) 010.

6.2 Die Sprachen der Pushdown-Automaten

Wir haben unterstellt, dass ein PDA seine Eingabe akzeptiert, indem er sie verbraucht und in einen akzeptierenden Zustand übergeht. Wir nennen diesen Ansatz »Akzeptanz durch Endzustand«. Es gibt einen zweiten Ansatz zur Definition der Sprache eines PDA, der wichtige Anwendungen besitzt. Wir können für jeden PDA die Sprache auch durch »Akzeptanz durch leeren Stack« definieren, d. h. die Menge der Zeichenreihen, die den PDA veranlassen, seinen Stack zu leeren.

Diese beiden Methoden sind äquivalent in dem Sinn, dass es für eine Sprache L genau dann einen PDA gibt, der sie durch Übergang in seinen Endzustand akzeptiert, wenn es für L einen PDA gibt, der sie durch das Leeren seines Stacks akzeptiert. Bei einem gegebenen PDA P unterscheiden sich jedoch für gewöhnlich die Sprachen, die P durch Endzustand und durch leeren Stack akzeptiert. Wir werden in diesem Abschnitt zeigen, wie man einen PDA, der L durch Endzustand akzeptiert, in einen PDA umwandelt, der L durch leeren Stack akzeptiert, und umgekehrt.

6.2.1 Akzeptanz durch Endzustand

Sei $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ein PDA. Dann ist die *Sprache* $L(P)$*, die von* P *durch Endzustand akzeptiert* wird,

$$\{w \mid (q_0, w, Z_0) \overset{*}{\vdash_P} (q, \varepsilon, \beta)\}$$

für einen Zustand q in F und eine Stackzeichenreihe α. Das heißt, ausgehend von der anfänglichen Konfiguration, in der w die Eingabezeichenreihe darstellt, liest P w aus der Eingabe und geht in einen akzeptierenden Zustand über. Der Inhalt des Stacks ist zu diesem Zeitpunkt irrelevant.

Beispiel 6.7 Wir haben behauptet, der PDA aus Beispiel 6.2 akzeptiere die Sprache L_{wwr}, also die Sprache der in $\{0, 1\}^*$ enthaltenen Zeichenreihen, die die Form ww^R haben. Lassen Sie uns überprüfen, ob diese Aussage wahr ist. Der Beweis ist eine Genaudann-wenn-Aussage: Der PDA P aus dem Beispiel 6.2 akzeptiert eine Zeichenreihe x durch Endzustand genau dann, wenn x die Form ww^R hat.

(Wenn-Teil) Dieser Teil des Beweises ist einfach. Wir müssen lediglich die Berechnungen von P zeigen, die zur Akzeptanz führen. Wenn $x = ww^R$, dann können wir Folgendes feststellen:

$$(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \overset{*}{\vdash} (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$$

Das heißt, der PDA hat die Möglichkeit, die Zeichenreihe w aus seiner Eingabe einzulesen und sie in umgekehrter Reihenfolge auf dem Stack zu speichern. Anschließend geht er spontan in den Zustand q_1 über, vergleicht die Zeichenreihe w^R in der Eingabe mit der identischen Zeichenreihe auf seinem Stack und geht schließlich spontan in den Zustand q_2 über.

(Nur-dann-Teil) Dieser Teil ist schwieriger. Wir stellen zuerst fest, dass der PDA nur aus dem Zustand q_1 heraus in den akzeptierenden Zustand q_2 übergehen kann und zwar nur dann, wenn dabei Z_0 das oberste Stacksymbol bildet. Außerdem beginnt jede akzeptierende Berechnung von P im Zustand q_0, führt zu einem Übergang nach q_1 und kehrt danach nie zu q_0 zurück. Somit genügt es, die Bedingungen für x zu finden, derart dass $(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \varepsilon, Z_0)$. Damit erhalten wir genau diejenigen

Zeichenreihen x, die P durch Erreichen seines Endzustands akzeptiert. Wir werden nun durch Induktion über $|x|$ eine etwas allgemeinere Aussage beweisen:

- Wenn $(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \alpha)$, dann hat x die Form ww^R.

Induktionsbeginn: Wenn $x = \varepsilon$, dann hat x die Form ww^R (mit $w = \varepsilon$). Somit ist die Konklusion wahr und folglich ist die Aussage wahr. Beachten Sie, dass wir nicht argumentieren müssen, dass die Hypothese $(q_0, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \alpha)$ wahr ist, obwohl dies der Fall ist.

Induktionsschritt: Angenommen, $x = a_1, a_2, \ldots a_n$ für $n > 0$. Ausgehend von der Konfiguration (q_0, x, α) kann P zwei Bewegungen ausführen:

1. $(q_0, x, \alpha) \vdash (q_1, x, \alpha)$. Wenn sich der PDA P im Zustand q_1 befindet, kann er nur jeweils ein Symbol vom Stack entfernen. P muss also mit jedem gelesenen Eingabesymbol ein Symbol vom Stack entfernen, wobei zunächst $|x| > 0$. Wenn also $(q_1, x, \alpha) \overset{*}{\vdash} (q_1, \varepsilon, \beta)$, dann ist β kürzer als α und kann nicht gleich α sein.

2. $(q_0, a_1a_2 \ldots a_n, \alpha) \vdash (q_0, a_2 \ldots a_n, a_1\alpha)$. Eine Bewegungsfolge kann jetzt nur dann mit (q_1, x, α) enden, wenn mit der letzten Bewegung ein Symbol vom Stack entfernt wird:

   $$(q_1, a_n, a_1\alpha) \vdash (q_1, \varepsilon, \alpha)$$

   In diesem Fall muss $a_1 = a_n$ sein. Wir wissen zudem, dass

   $$(q_0, a_2 \ldots a_n, a_1\alpha) \overset{*}{\vdash} (q_1, a_n, a_1\alpha)$$

   Nach Satz 6.6 können wir das Symbol a_n vom Ende der Eingabe entfernen, da es nicht verwendet wird. Also gilt

   $$(q_0, a_2 \ldots a_{n-1}, a_1\alpha) \overset{*}{\vdash} (q_1, \varepsilon, a_1\alpha)$$

   Da die Eingabe für diese Folge kürzer als n ist, können wir die Induktionshypothese anwenden und schließen, dass $a_2 \ldots a_{n-1}$ die Form yy^R hat, wobei y beliebig ist. Da $x = a_1yy^Ra_n$ und da wir wissen, dass $a_1 = a_n$, schließen wir, dass x die Form ww^R hat, wobei $w = a_1y$.

Dies ist das Kernstück des Beweises, dass x nur dann durch Endzustand akzeptiert werden kann, wenn x für ein w gleich ww^R ist. Somit ist der »Nur-dann-Teil« des Beweises abgeschlossen, aus dem zusammen mit dem zuvor bewiesenen »Wenn-Teil« hervorgeht, dass P genau die in L_{wwr} enthaltenen Zeichenreihen durch Endzustand akzeptiert. ■

6.2.2 Akzeptanz durch leeren Stack

Für jeden PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ definieren wir zudem

$$N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)\}$$

für einen beliebigen Zustand q. Das heißt, $N(P)$ ist die Menge der Eingabezeichenreihen w, die P einlesen kann und bei der er gleichzeitig seinen Stack leeren kann[3].

3. In $N(P)$ steht das N für »Null-Stack«, ein Synonym für »leerer Stack«.

Beispiel 6.8 Der PDA P aus dem Beispiel 6.2 leert seinen Stack nie, daher ist $N(P) = \emptyset$. Eine kleine Modifikation wird es P jedoch erlauben, L_{wwr} sowohl durch das Leeren des Stacks als auch durch den Endzustand zu akzeptieren. Statt des Übergangs $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$ verwenden wir $\delta(q_1, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}$. Jetzt kann P das letzte Symbol von seinem Stack entfernen, während er akzeptiert, und $L(P) = N(P) = L_{wwr}$. ■

Da die Menge der akzeptierenden Zustände im Fall der Akzeptanz durch leeren Stack irrelevant ist, lassen wir gelegentlich die letzte (siebte) Komponente in der Spezifikation eines PDA P weg, wenn uns lediglich die Sprache interessiert, die P durch Leeren seines Stacks akzeptiert. Wir geben P dann als Sextupel $(Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ an.

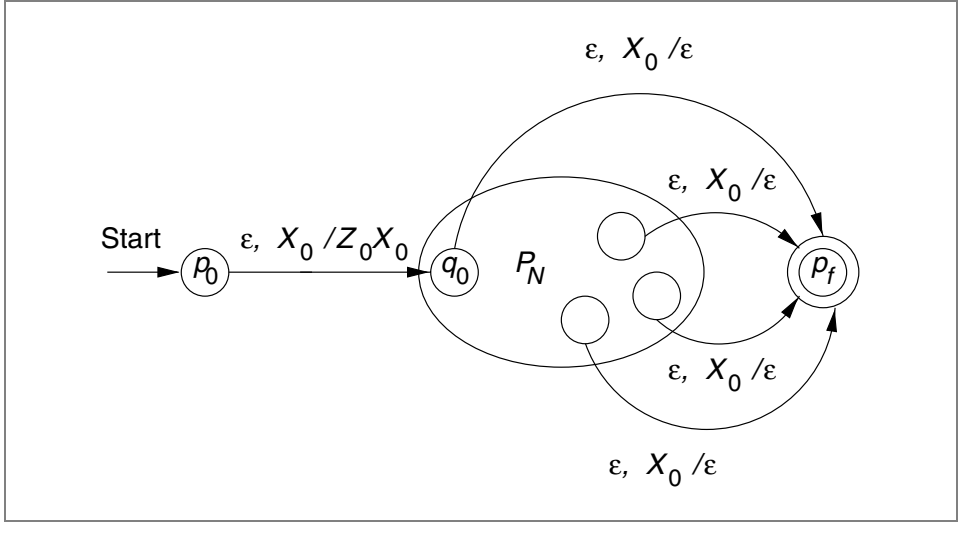
6.2.3 Vom leeren Stack zum Endzustand

Wir werden zeigen, dass es sich bei den Klassen von Sprachen, die für einen PDA P $L(P)$ sind, um die gleiche Klasse von Sprachen handelt, die $N(P)$ für einen im Allgemeinen anderen PDA P sind. Diese Klasse ist auch genau die Klasse der kontextfreien Sprachen, wie wir in Abschnitt 6.3 sehen werden. Unsere erste Konstruktion zeigt, wie man aus einem PDA P_N, der eine Sprache L durch Leeren seines Stacks akzeptiert, einen PDA P_F konstruiert, der L durch Endzustand akzeptiert.

Satz 6.9 Wenn für einen PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$ die Sprache $L = N(P_N)$, dann gibt es einen PDA P_F, derart dass $L = L(P_F)$.

Beweis: Der diesem Beweis zu Grunde liegende Gedankengang ist in Abbildung 6.4 dargestellt. Wir verwenden ein neues Symbol X_0, das kein Symbol aus Γ sein darf. X_0 ist sowohl das Startsymbol von P_F als auch eine Markierung am unteren Ende des Stacks, die uns mitteilt, wann P_N einen leeren Stack erreicht hat. Das heißt, wenn P_F X_0 als oberstes Symbol auf dem Stack vorfindet, dann weiß der Automat, dass P_N bei der gleichen Eingabe seinen Stack leeren würde.

Abbildung 6.4: P_F simuliert P_N und akzeptiert, wenn P_N seinen Stack leert



Wir brauchen auch einen neuen Startzustand p_0, dessen einzige Funktion darin besteht, das Startsymbol Z_0 von P_N auf dem Stack abzulegen und in den Startzustand q_0 von P_N zu wechseln. Dann simuliert P_F P_N so lange, bis der Stack von P_N leer ist, was P_F daran erkennt, dass er X_0 als oberstes Symbol auf dem Stack vorfindet. Schließlich brauchen wir einen weiteren Zustand p_f, der den akzeptierenden Zustand von P_F repräsentiert. Dieser PDA geht in den Zustand p_f über, sobald er erkennt, dass P_N seinen Stack geleert hat.

Die Spezifikation von P_F lautet wie folgt:

$$P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$$

wobei δ_F definiert wird durch

1. $\delta_F(p_0, \varepsilon, X_0) = \{(q_0, Z_0\, X_0)\}$. Von seinem Startzustand aus geht P_F unmittelbar in den Startzustand von P_N über, wobei er das Startsymbol Z_0 auf dem Stack ablegt.
2. Für alle Zustände q aus Q, Eingabesymbole a aus Σ oder $a = \varepsilon$ und Stacksymbole Y aus Γ enthält $\delta_F(q, a, Y)$ alle Paare aus $\delta_N(q, a, Y)$.
3. Zusätzlich zur Regel (2) enthält $\delta_F(q, \varepsilon, X_0)$ für jeden Zustand q aus Q (p_f, ε).

Wir müssen zeigen, dass w genau dann in $L(P_F)$ enthalten ist, wenn w in $N(P_N)$ enthalten ist.

(Wenn-Teil) Gegeben ist die Aussage, dass für einen Zustand $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \varepsilon)$. Dank Satz 6.5 können wir X_0 am unteren Ende des Stacks einfügen und schließen, dass $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, X_0)$. Da nach der oben genannten Regel (2) P_F sämtliche Bewegungen von P_N ausführt, können wir zudem folgern, dass $(q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, X_0)$. Wenn wir diese Bewegungsfolge mit der ersten und der letzten Bewegung kombinieren, die sich aus den Regeln (1) und (3) ergeben, dann erhalten wir:

$$(p_0, w, X_0) \underset{P_F}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_F}{\vdash}} (q, \varepsilon, X_0) \underset{P_F}{\vdash} (p_f, \varepsilon, \varepsilon)$$

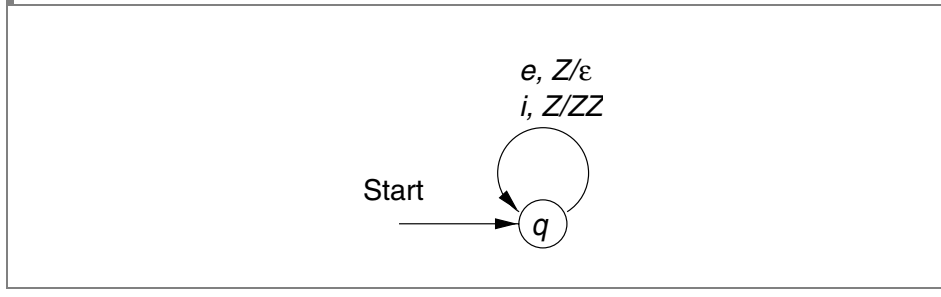
Folglich akzeptiert P_F die Zeichenreihe w durch Endzustand.

(Nur-wenn-Teil) Die Umkehrung erfordert lediglich, dass wir erkennen, dass die zusätzlichen Übergänge nach den Regeln (1) und (3) uns sehr beschränkte Möglichkeiten geben, w durch Endzustand zu akzeptieren. Wir müssen Regel (3) im letzten Schritt anwenden und wir können diese Regel nur verwenden, wenn der Stack von P_F lediglich X_0 enthält. Das Symbol X_0 kommt an keiner anderen Stelle des Stacks vor als an der untersten Position. Überdies wird Regel (1) nur im ersten Schritt angewandt und sie *muss* in diesem Schritt angewandt werden.

Folglich muss jede Berechnung von P_F, die zur Akzeptanz von w führt, wie die Folge (6.1) aussehen. Zudem muss die Mitte der Berechnung (alle Schritte außer dem ersten und dem letzten) auch eine Berechnung von P_N sein, wobei sich X_0 unten im Stack befinden muss. Begründet wird dies dadurch, dass, abgesehen vom ersten und letzten Schritt, P_F keine Übergänge ausführen kann, die nicht ebenso Übergänge von P_N sind, und dass X_0 dabei nur dann als oberstes Element im Stack auftreten kann, wenn die Berechnung im nächsten Schritt endet. Wir schließen daraus, dass $(q_0, w, Z_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, \varepsilon)$. Das heißt, w ist in $N(P_N)$ enthalten. ■

Beispiel 6.10 Wir wollen einen PDA entwerfen, der Folgen von `if`- und `else`-Wortsymbolen eines C-Programms verarbeitet, wobei *i* für `if` und *e* für `else` steht. Sie wissen aus Abschnitt 5.3.1, dass es problematisch ist, wenn in einem Präfix mehr `else` als `if` vorhanden sind, da es dann nicht möglich ist, jedes `else` einem voranstehenden `if` zuzuordnen. Daher werden wir das Stacksymbol *Z* einsetzen, um die Differenz zwischen der Anzahl der bereits gelesenen *i* und der Anzahl der *e* zu verzeichnen. Dieser einfache PDA, der nur über einen Zustand verfügt, ist im Übergangsdiagramm aus Abbildung 6.5 dargestellt.

Abbildung 6.5: PDA, der `if`/`else`-Fehler durch Leeren des Stacks akzeptiert



Wir legen ein weiteres *Z* auf dem Stack ab, wenn ein *i* gelesen wird, und entfernen ein *Z* vom Stack, wenn ein *e* gelesen wird. Da wir mit einem *Z* auf dem Stack beginnen, halten wir uns tatsächlich an die Regel, dass *n*-1 mehr *i* als *e* gelesen wurden, wenn der Stack Z^n enthält. Wenn der Stack leer ist, dann lag eine um eins höhere Anzahl von *e* als von *i* vor, und die bis dahin gelesene Eingabe wurde damit zum ersten Mal unzulässig. Unser PDA akzeptiert diese Art von Zeichenreihen durch einen leeren Stack. Die formale Spezifikation von P_N lautet:

$$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$$

wobei δ_N definiert wird durch:

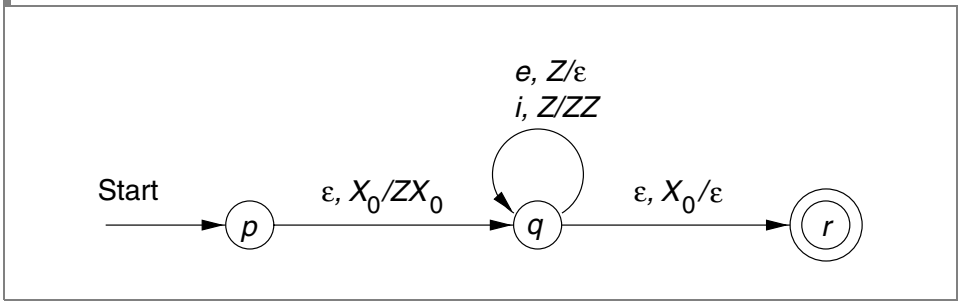
1. $\delta_N(q, i, Z) = \{(q, ZZ)\}$. Diese Regel bewirkt, dass ein *Z* auf dem Stack abgelegt wird, wenn ein *i* gelesen wird.
2. $\delta_N(q, e, Z) = \{(q, \varepsilon)\}$. Diese Regel bewirkt, dass ein *Z* vom Stack entfernt wird, wenn ein *e* gelesen wird.

Lassen Sie uns nun aus P_N einen PDA P_F konstruieren, der dieselbe Sprache durch Endzustand akzeptiert. Das Übergangsdiagramm für diesen PDA P_F ist in Abbildung 6.6[4] dargestellt. Wir führen einen neuen Startzustand *p* und einen akzeptierenden Zustand *r* ein. Wir werden X_0 als Stackanfangsmarkierung verwenden. Die formale Definition von P_F lautet:

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

4. Es ist nicht von Belang, dass wir hier die Zustände *p* und *q* verwenden, während in Satz 6.9 von den Zuständen p_0 und p_i die Rede ist. Die Namen von Zuständen können beliebig gewählt werden.

Abbildung 6.6: Konstruktion eines PDA, der durch Endzustand die Sprache des PDA aus Abbildung 6.5 akzeptiert



wobei δ_F sich wie folgt zusammensetzt:

1. $\delta_F(p, \varepsilon, X_0) = \{(q, ZX_0)\}$. Diese Regel bewirkt, dass P_F mit der Simulation von P_N beginnt, wobei X_0 die Stackanfangsmarkierung darstellt.
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$. Diese Regel bewirkt, dass ein Z auf dem Stack abgelegt wird, wenn ein i gelesen wird, entsprechend dem Verhalten von P_N.
3. $\delta_F(q, e, Z) = \{(q, \varepsilon)\}$. Diese Regel bewirkt, dass ein Z vom Stack entfernt wird, wenn ein e gelesen wird; auch diese Regel simuliert P_N.
4. $\delta_F(q, \varepsilon, X_0) = \{(r, \varepsilon, \varepsilon)\}$. Diese Regel bedeutet, dass P_F akzeptiert, wenn der simulierte Automat P_N seinen Stack geleert hat. ■

6.2.4 Vom Endzustand zum leeren Stack

Lassen Sie uns nun in die entgegengesetzte Richtung gehen: Wir nehmen einen PDA P_F, der eine Sprache L durch Endzustand akzeptiert, und konstruieren einen PDA P_N, der die Sprache L durch Leeren seines Stacks akzeptiert. Die Konstruktion ist einfach und in Abbildung 6.7 dargestellt. Wir fügen zu jedem akzeptierenden Zustand von P_F einen Übergang für die Eingabe ε zu einem neuen Zustand p hinzu. Wenn sich P_N im Zustand p befindet, dann leert er seinen Stack und liest kein weiteres Eingabesymbol ein. Daher wird P_N seinen Stack nach dem Einlesen von w leeren, wenn P_F nach dem Einlesen der Eingabe w in einen akzeptierenden Zustand übergeht.

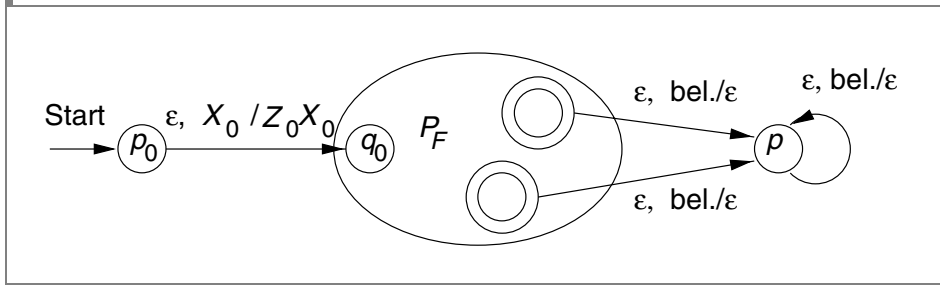
Um zu verhindern, dass eine Situation simuliert wird, in der P_F zufällig seinen Stack leert, ohne zu akzeptieren, muss P_N eine Stackanfangsmarkierung X_0 verwenden. Diese Markierung ist das Startsymbol von P_N und wie bei der Konstruktion von Satz 6.9 muss P_N in einem neuen Zustand p_0 beginnen, dessen einzige Funktion darin besteht, das Startsymbol von P_F auf dem Stack abzulegen und zum Startzustand von P_F zu wechseln. Die Konstruktion ist in Abbildung 6.7 skizziert und wird im nächsten Satz formalisiert.

Satz 6.11 Sei L die Sprache $L(P_F)$ für einen PDA P_F $(Q, \Sigma, \Gamma, \delta_F, p_0, Z_0, F)$. Dann gibt es einen PDA P_N, derart dass $L = N(P_N)$.

Beweis: Die Konstruktion entspricht der Darstellung in Abbildung 6.7. Sei

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

Abbildung 6.7: P_N simuliert P_F und leert seinen Stack genau dann, wenn P_F in einen akzeptierenden Zustand übergeht



wobei δ_N definiert wird durch:

1. $\delta_N(p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$. Wir beginnen, indem wir das Startsymbol von P_F auf dem Stack ablegen und in den Startzustand von P_F wechseln.
2. Für alle Zustände q aus Q, Eingabesymbole a aus Σ oder $a = \varepsilon$ und Y aus Γ gilt, dass $\delta_N(q, a, Y)$ jedes Paar enthält, das in $\delta_F(q, a, Y)$ enthalten ist. Das heißt, P_N simuliert P_F.
3. Für alle akzeptierenden Zustände q aus F und Stacksymbole Y aus Γ oder $Y = X_0$ gilt, dass $\delta_N(q, \varepsilon, Y)$ das Paar (p, ε) enthält. Nach dieser Regel kann, sobald P_F akzeptiert, P_N damit beginnen, seinen Stack zu leeren, ohne weitere Eingabesymbole einzulesen.
4. Für alle Stacksymbole Y aus Γ oder $Y = X_0$ ist $\delta_N(p, \varepsilon, Y) = \{(p, \varepsilon)\}$. Wenn P_N den Zustand p erreicht hat, was nur nach dem Akzeptieren von P_F der Fall sein kann, dann nimmt P_N so lange Symbole von seinem Stack, bis der Stack leer ist, und liest keine weiteren Eingabesymbole ein.

Wir müssen nun beweisen, dass w in $N(P_N)$ genau dann enthalten ist, wenn w in $L(P_F)$ enthalten ist. Der Gedankengang ist ähnlich wie beim Beweis von Satz 6.9. Der »Wenn-Teil« ist eine direkte Simulation und der »Nur-wenn-Teil« erfordert, das wir die beschränkte Zahl der Dinge untersuchen, die der konstruierte PDA P_N ausführen kann.

(Wenn-Teil) Angenommen, für einen akzeptierenden Zustand q und eine Stackzeichenreihe α gelte $(q_0, w, Z_0) \overset{*}{\vdash}_{P_F} (q, \varepsilon, \alpha)$. Auf Grund der Tatsache, dass jeder Zustandsübergang von P_F eine Bewegung von P_N ist, und durch Anwendung von Satz 6.5, wonach X_0 am unteren Ende auf dem Stack gehalten werden kann, wissen wir, dass $(q_0, w, Z_0X_0) \overset{*}{\vdash}_{P_N} (q, \varepsilon, \alpha X_0)$. Somit kann P_N Folgendes ausführen:

$$(p_0, w, X_0) \vdash_{P_N} (q_0, w, Z_0X_0) \overset{*}{\vdash}_{P_N} (q, \varepsilon, \alpha X_0) \overset{*}{\vdash}_{P_N} (p, \varepsilon, \varepsilon)$$

Die erste Bewegung folgt aus Regel (1) in der Konstruktion von P_N, während die letzte Bewegungsfolge den Regeln (3) und (4) entspricht. Somit wird w von P_N durch Leeren des Stacks akzeptiert.

(Nur-wenn-Teil) Der PDA P_N kann den Stack nur leeren, indem er in den Zustand p übergeht, da sich X_0 am unteren Ende des Stacks befindet und P_F für X_0 keine Bewegungen definiert. In den Zustand p kann P_N wiederum nur wechseln, wenn der simu-

lierte PDA P_F einen akzeptierenden Zustand annimmt. Die erste Bewegung von P_N entspricht mit Sicherheit der nach Regel (1) festgelegten Bewegung. Folglich sieht jede akzeptierende Berechnung von P_N wie folgt aus:

$$(p_0, w, X_0) \vdash_{P_N} (q_0, w, Z_0X_0) \overset{*}{\vdash}_{P_N} (q, \varepsilon, \alpha X_0) \overset{*}{\vdash}_{P_N} (p, \varepsilon, \varepsilon)$$

wobei q ein akzeptierender Zustand von P_F ist.

Überdies sind alle Bewegungen zwischen den Konfigurationen (q_0, w, Z_0X_0) und $(q, \varepsilon, \alpha X_0)$ Bewegungen von P_F. Insbesondere war X_0 vor dem Erreichen der Konfiguration $(q, \varepsilon, \alpha X_0)$ nie das oberste Stacksymbol[5]. Somit schließen wir, dass dieselbe Berechnung in P_F vorkommen kann, ohne dass sich X_0 auf dem Stack befindet; d. h. $(q_0, w, Z_0) \overset{*}{\vdash}_{P_F} (q, \varepsilon, \alpha)$. Daraus folgt, dass P_F w durch Endzustand akzeptiert, und daher ist w in $L(P_F)$ enthalten. ■

6.2.5 Übungen zum Abschnitt 6.2

Übung 6.2.1 Entwerfen Sie einen PDA für jede der folgenden Sprachen. Die PDA$_n$ können durch Endzustand oder Leeren des Stacks akzeptieren, je nachdem, was bequemer ist.

* a) $\{0^n1^n \mid n \geq 1\}$

b) Die Menge aller Zeichenreihen aus Nullen und Einsen, derart dass kein Präfix mehr Einsen als Nullen enthält.

c) Die Menge aller Zeichenreihen aus Nullen und Einsen mit der gleichen Anzahl von Nullen und Einsen.

! Übung 6.2.2 Entwerfen Sie einen PDA für jede der folgenden Sprachen:

* a) $\{a^ib^jc^k \mid i = j$ oder $j = k\}$. Beachten Sie, dass sich diese Sprache von der aus Übung 5.1.1 (b) unterscheidet.

b) Die Menge aller Zeichenreihen mit doppelt so vielen Nullen wie Einsen.

!! Übung 6.2.3 Entwerfen Sie einen PDA für jede der folgenden Sprachen:

a) $\{a^ib^jc^k \mid i \neq j$ oder $j \neq k\}$.

b) Die Menge aller Zeichenreihen aus den Symbolen a und b, die für kein w die Form ww haben.

***! Übung 6.2.4** Sei P ein PDA, der durch Leeren des Stacks die Sprache $L = N(P)$ akzeptiert, und nehmen wir an, ε sei nicht in L enthalten. Beschreiben Sie, wie der PDA P abgeändert werden könnte, sodass er $L \cup \{\varepsilon\}$ durch Leeren seines Stacks akzeptiert.

5. Natürlich könnte α auch ε sein. In diesem Fall leert P_F seinen Stack zu dem Zeitpunkt, in dem er akzeptiert.

* **Übung 6.2.5** Der PDA $P = (\{q_0, q_1, q_2, q_3, f\}, \{a, b\}, \{Z_0, A, B\}, \delta, q_0, Z_0, \{f\})$ verfügt über die folgenden Regeln zu Definition von δ:

$\delta(q_0, a, Z_0) = (q_1, AAZ_0)$	$\delta(q_0, b, Z_0) = (q_2, BZ_0)$	$\delta(q_0, \varepsilon, Z_0) = (f, \varepsilon)$
$\delta(q_1, a, A) = (q_1, AAA)$	$\delta(q_1, b, A) = (q_1, \varepsilon)$	$\delta(q_1, \varepsilon, Z_0) = (q_0, Z_0)$
$\delta(q_2, a, B) = (q_3, \varepsilon)$	$\delta(q_2, b, B) = (q_2, BB)$	$\delta(q_2, \varepsilon, Z_0) = (q_0, Z_0)$
$\delta(q_3, \varepsilon, B) = (q_2, \varepsilon)$	$\delta(q_3, \varepsilon, Z_0) = (q_1, AZ_0)$	

Beachten Sie, dass wir diesmal die Mengenklammern weggelassen haben, da jede Menge nur ein Paar aus Q x Γ enthält.

* a) Geben Sie eine Berechnung (eine Folge von Konfigurationen) an, die zeigt, dass die Zeichenreihe *bab* in $L(P)$ enthalten ist.

b) Geben Sie eine Berechnung an, die zeigt, dass die Zeichenreihe *abb* in $L(P)$ enthalten ist.

c) Geben Sie den Inhalt des Stacks an, nachdem P b^7a^4 aus der Eingabe gelesen hat.

! d) Beschreiben Sie informell $L(P)$.

Übung 6.2.6 Betrachten Sie den PDA aus Übung 6.1.1.

a) Wandeln Sie P in einen anderen PDA P_1 um, der durch Leeren seines Stacks dieselbe Sprache wie P durch Endzustand akzeptiert, d. h. $N(P_1) = L(P)$.

b) Finden Sie einen PDA P_2, sodass $L(P_2) = N(P)$; d. h. P_2 akzeptiert durch Endzustand die Sprache, die P durch Leeren seines Stacks akzeptiert.

! **Übung 6.2.7** Zeigen Sie Folgendes: Wenn P ein PDA ist, dann gibt es einen PDA P_2 mit nur zwei Stacksymbolen, sodass $L(P_2) = L(P)$. *Hinweis*: Kodieren Sie das Stackalphabet von P in Binärnotation.

*! **Übung 6.2.8** Ein PDA wird als *beschränkt* bezeichnet, wenn er bei jedem Übergang die Höhe des Stacks höchstens um ein Symbol erhöhen kann. Das heißt, wenn (p, γ) in $\delta(q, a, Z)$ ist, muss gelten $\gamma \leq 2$. Zeigen Sie, dass es für einen PDA P einen beschränkten PDA P_3 gibt, sodass $L(P) = L(P_3)$.

6.3 Äquivalenz von Pushdown-Automaten und kontextfreien Grammatiken

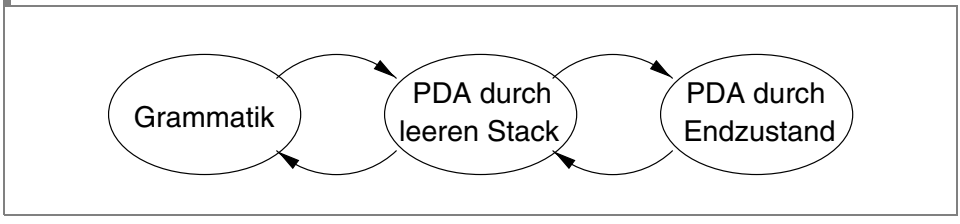
Wir werden nun demonstrieren, dass es sich bei den von PDAs definierten Sprachen genau um die kontextfreien Sprachen handelt. Unsere Strategie ist in Abbildung 6.8 dargestellt. Unser Ziel besteht darin, zu beweisen, dass es sich bei den folgenden drei Klassen von Sprachen um dieselbe Klasse handelt.

1. Die kontextfreien Sprachen, d. h. Sprachen, die durch kontextfreie Grammatiken definiert werden.

2. Die Sprachen, die von einem PDA durch Endzustand akzeptiert werden.
3. Die Sprachen, die von einem PDA durch Leeren des Stacks akzeptiert werden.

Wir haben bereits gezeigt, dass (2) und (3) gleich sind. Am einfachsten lässt sich als Nächstes zeigen, dass (1) und (3) gleich sind, was die Äquivalenz aller drei Sprachen impliziert.

Abbildung 6.8: Aufbau der Konstruktion zum Beweis der Äquivalenz der drei Methoden zur Definition kontextfreier Sprachen



6.3.1 Von Grammatiken zu Pushdown-Automaten

Auf der Grundlage einer gegebenen kontextfreien Grammatik G konstruieren wir einen PDA, der die linksseitigen Ableitungen von G simuliert. Jede linksseitige Satzform, die keine terminale Zeichenreihe ist, kann in der Form $xA\alpha$ angegeben werden, wobei A für die erste Variable von links, x für eine terminale Variable und α für die Zeichenreihe aus terminalen Symbolen und Variablen rechts von A steht. Wir bezeichnen $A\alpha$ als Rumpf der linksseitigen Satzform. Wenn eine linksseitige Satzform nur aus terminalen Zeichenreihen besteht, dann ist ihr Rumpf gleich ε.

Hinter der Konstruktion eines PDA aus einer Grammatik steckt die Absicht, den PDA die Folge von linksseitigen Satzformen simulieren zu lassen, die die Grammatik zur Erzeugung einer gegebenen terminalen Zeichenreihe w verwendet. Der Rumpf jeder Satzform $xA\alpha$ erscheint auf dem Stack, wobei A an oberster Stelle steht. Zu diesem Zeitpunkt wird x dadurch »repräsentiert«, dass x aus der Eingabe gelesen wurde, sodass in der Eingabe der Teil von w übrig bleibt, der dem Präfix x folgt. Das heißt, wenn $w = xy$, dann bleibt noch y in der Eingabe.

Angenommen, der PDA ist in der Konfiguration $(q, y, A\alpha)$, die die linksseitige Satzform $xA\alpha$ repräsentiert. Er »rät«, welche Produktion aus A eingesetzt werden soll, angenommen $A \to \beta$. Die Bewegung des PDA besteht darin, A auf dem Stack durch β zu ersetzen und damit in die Konfiguration $(q, y, \beta\alpha)$ überzugehen. Beachten Sie, dass dieser PDA nur einen Zustand q hat.

Möglicherweise ist $(q, y, \beta\alpha)$ keine Repräsentation einer linksseitigen Satzform, weil β ein Präfix aus terminalen Symbolen besitzen kann. Es kann sogar sein, dass β überhaupt keine Variablen enthält und dass α ein Präfix aus terminalen Symbolen besitzt. Alle terminalen Symbole, die sich am Anfang von $\beta\alpha$ befinden, müssen entfernt werden, damit die nächste Variable am oberen Ende des Stacks sichtbar wird. Diese terminalen Symbole werden mit den nächsten Eingabesymbolen verglichen, um sicherzustellen, dass unsere (geratenen) Annahmen über die linksseitige Ableitung der Eingabezeichenreihe w korrekt sind; sind sie es nicht, dann endet der betreffende Zweig des PDA, ohne zu akzeptieren.

Wenn wir auf diese Weise die linksseitige Ableitung von w erraten können, dann werden wir schließlich die linksseitige Satzform w ganz eingelesen haben. Zu diesem Zeitpunkt wurden alle Symbole auf dem Stack entweder ersetzt (falls es sich um Variablen handelt) oder mit den Eingabesymbolen verglichen (falls es sich um terminale Symbole handelt) und gegebenenfalls entfernt. Der Stack ist leer und wir akzeptieren durch Leeren des Stacks.

Diese informelle Konstruktion lässt sich wie folgt präzisieren. Sei $G = (V, T, Q, S)$ eine kfG. Konstruiere den PDA P, der $L(G)$ durch Leeren des Stacks akzeptiert, wie folgt:

$$P = (\{q\}, T, V \cup T, \delta, q, S)$$

wobei die Übergangsfunktion δ definiert wird durch:

1. Für jede Variable A $\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \to \beta$ ist eine Produktion aus $Q\}$.
2. Für jedes terminale Symbol a, $\delta(q, a, a) = \{(q, \varepsilon)\}$.

Beispiel 6.12 Wir wollen die Grammatik für einfache Ausdrücke aus Tabelle 5.2 in einen PDA umwandeln. Wie Sie wissen, ist diese Grammatik wie folgt definiert:

$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$E \to I \mid E * E \mid E + E \mid (E)$$

Die Menge der terminalen Symbole für den PDA lautet $\{a, b, 0, 1, (,), +, *\}$. Diese acht Symbole sowie die Symbole I und E bilden das Stackalphabet. Die Übergangsfunktion dieses PDA ist definiert durch:

a) $\delta(q, \varepsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}$.

b) $\delta(q, \varepsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, (E))\}$.

c) $\delta(q, a, a) = \{(q, \varepsilon)\}$; $\delta(q, b, b) = \{(q, \varepsilon)\}$; $\delta(q, 0, 0) = \{(q, \varepsilon)\}$; $\delta(q, 1, 1) = \{(q, \varepsilon)\}$; $\delta(q, (, () = \{(q, \varepsilon)\}$; $\delta(q,),)) = \{(q, \varepsilon)\}$; $\delta(q, +, +) = \{(q, \varepsilon)\}$; $\delta(q, *, *) = \{(q, \varepsilon)\}$.

Beachten Sie, dass sich (a) und (b) aus Regel (1) ergeben, während sich die acht Übergänge von (c) aus Regel (2) ergeben. Zudem ist δ leer, soweit in (a) bis (c) nicht definiert. ■

Satz 6.13 Wenn der PDA P auf die oben beschriebene Weise aus der kfG G konstruiert wird, dann gilt $N(P) = L(G)$.

Beweis: Wir werden beweisen, dass w genau dann in $N(P)$ enthalten ist, wenn w in $L(G)$ enthalten ist.

(Wenn-Teil) Angenommen, w ist in $L(G)$ enthalten. Dann hat w eine linksseitige Ableitung

$$S = \gamma_1 \underset{lm}{\Rightarrow} \gamma_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} \gamma_n = w$$

Wir zeigen durch Induktion über i, dass $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, wobei y_i und α_i eine Repräsentation der linksseitigen Satzform γ_i darstellen. Das heißt, sei α_i der Rumpf von γ_i und $\gamma_i = x_i\alpha_i$. Dann ist y_i die Zeichenreihe, für die gilt $x_iy_i = w$, d. h., sie repräsentiert das, was übrig bleibt, wenn x_i aus der Eingabe gelesen wurde.

Induktionsbeginn: Für $i = 1$ gilt $\gamma_i = S$. Somit ist $x_i = \varepsilon$ und $y_i = w$. Da $(q, w, S) \overset{*}{\vdash} (q, w, S)$ nach 0 Bewegungen, ist der Induktionsbeginn bewiesen.

Induktionsschritt: Wir betrachten nun den Fall der zweiten Satzform und nachfolgender linksseitigen Satzformen. Wir nehmen an, dass

$$(q,w,S)\overset{*}{\vdash}(q,y_i,\alpha_i)$$

und beweisen $(q, w, S) \overset{*}{\vdash} (q, y_{i+1}, \alpha_{i+1})$. Da α_i der Rumpf einer linksseitigen Satzform ist, beginnt er mit einer Variablen A. Die linksseitige Ableitung $\gamma_i \Rightarrow \gamma_{i+1}$ erfordert überdies, dass A durch den Rumpf einer seiner Produktionen ersetzt wird, angenommen durch β. Regel (1) der Konstruktion von P ermöglicht es uns, A auf dem Stack durch β zu ersetzen, und nach Regel (2) können wir dann die terminalen Symbole, die sich oben auf dem Stack befinden, mit den nächsten Eingabesymbolen abgleichen. Infolgedessen erreichen wir die Konfiguration $(q, y_{i+1}, \alpha_{i+1})$, die die nächste linksseitige Satzform γ_{i+1} repräsentiert.

Um diesen Beweis zu vervollständigen, stellen wir fest, dass $\alpha_n = \varepsilon$, da der Rumpf von $\gamma_n = w$ leer ist. Folglich ist $(q, w, S) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)$, womit bewiesen ist, dass P die Zeichenreihe w durch Leeren des Stacks akzeptiert.

(Nur-wenn-Teil) Wir müssen etwas Allgemeineres beweisen: Wenn P eine Folge von Bewegungen ausführt, die im Endeffekt bewirken, dass eine Variable A vom oberen Ende des Stacks entfernt wird, ohne dass der Stack unterhalb von A benutzt wird, dann leitet A in G die Zeichenreihe ab, die während dieses Vorgangs aus der Eingabe eingelesen worden ist. Genauer ausgedrückt:

- Wenn $(q, x, A) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$, dann $A \overset{*}{\underset{G}{\Rightarrow}} x$.

Beweisen lässt sich dies, durch eine Induktion über die Anzahl der von P ausgeführten Bewegungen.

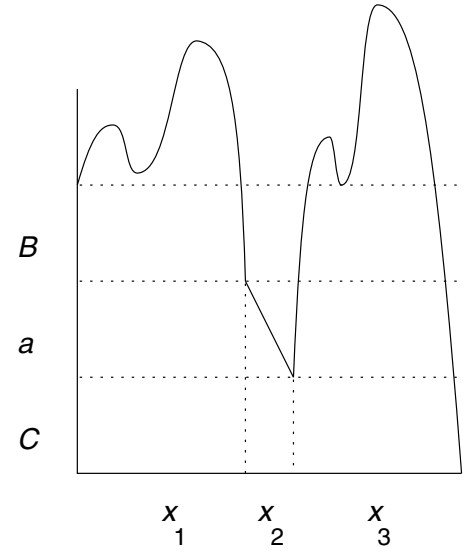
Induktionsbeginn: eine Bewegung. Die einzige Möglichkeit besteht darin, dass $A \to \varepsilon$ eine Produktion von G ist und dass diese Produktion vom PDA P in einer Regel vom Typ (1) verwendet wird. In diesem Fall ist $x = \varepsilon$ und wir wissen, dass $A \Rightarrow \varepsilon$.

Induktionsschritt: Angenommen, P führt n Bewegungen aus, wobei $n > 1$. Die erste Bewegung muss vom Typ (1) sein, wobei A am oberen Ende des Stacks durch einen seiner Produktionsrümpfe ersetzt wird. Begründet wird dies dadurch, dass eine Regel vom Typ (2) nur dann angewandt werden kann, wenn das oberste Stacksymbol ein terminales Symbol ist. Angenommen, es wird die Produktion $A \to Y_1Y_2 \ldots Y_k$ verwendet, wobei Y_i entweder ein terminales Symbol oder eine Variable ist.

Die nächsten n - 1 Bewegungen von P müssen bewirken, dass x aus der Eingabe eingelesen wird und dass die einzelnen Y_1, Y_2 etc. nacheinander vom Stack entfernt werden. Wir können x in $x_1x_2 \ldots x_k$ aufteilen, wobei x_1 den Teil der Eingabe darstellt, der eingelesen wird, bis Y_1 und seine Derivate vom Stack entfernt sind (d. h., bis der Stack erstmals α wird, wenn er zunächst $Y_1\alpha$ war). x_2 repräsentiert dann den nächsten Teil der Eingabe, der eingelesen wird, während Y_2 vom Stack entfernt wird, etc.

Abbildung 6.9 stellt dar, wie die Eingabe x aufgeteilt wird und welche Auswirkungen diese Aufteilung auf den Stack hat. Wir unterstellen hier, dass β gleich BaC ist, und teilen x in die drei Teile $x_1x_2x_3$ auf, wobei $x_2 = a$. Beachten Sie, dass im Allgemeinen gilt: Wenn Y_i ein terminales Symbol ist, dann muss x_i dieses terminale Symbol repräsentieren.

Abbildung 6.9: Der PDA *P* liest *x* aus der Eingabe ein und entfernt *BaC* von seinem Stack



Formal können wir die Schlussfolgerung ziehen, dass $(q, x_i x_{i+1} \ldots x_k, Y_i) \overset{*}{\vdash} (q, x_{i+1} \ldots x_k, \varepsilon)$ für alle $i = 1, 2, \ldots, k$. Überdies kann keine dieser Folgen mehr als $n - 1$ Bewegungen umfassen, sodass die Induktionshypothese gilt, wenn Y_i eine Variable ist. Das heißt, wir können die Schlussfolgerung ziehen, dass $Y_i \overset{*}{\Rightarrow} x_i$.

Wenn Y_i ein terminales Symbol ist, dann darf nur eine Bewegung ausgeführt werden und darin wird das Zeichen $x_i = Y_i$ abgeglichen und x_i gelesen sowie Y_i vom Stack entfernt. Wieder können wir darauf schließen, dass $Y_i \overset{*}{\Rightarrow} x_i$. Dieses Mal sind null Schritte erforderlich. Damit liegt die Ableitung vor:

$$A \Rightarrow Y_1 Y_2 \ldots Y_k \overset{*}{\Rightarrow} x_1 Y_2 \ldots Y_k \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} x_1 x_2 \ldots x_k$$

Das heißt: $A \overset{*}{\Rightarrow} x$.

Um den Beweis zu vervollständigen, sei $A = S$ und $x = w$. Da wir davon ausgehen, dass w in $N(P)$ enthalten ist, wissen wir, dass $(q, w, S) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)$. Nach dem Ergebnis unseres Induktionsbeweises gilt dann $S \overset{*}{\Rightarrow} w$; d. h., w ist in $L(G)$ enthalten. ■

6.3.2 Von PDAs zu Grammatiken

Zur Vervollständigung der Beweise der Äquivalenz zeigen wir nun, dass wir für jeden PDA *P* eine kontextfreie Grammatik *G* finden können, deren Sprache mit der Sprache identisch ist, die *P* durch Leeren seines Stacks akzeptiert. Der Beweis beruht auf dem Gedankengang, dass der entscheidende Vorgang in der Verarbeitung einer gegebenen Eingabe durch einen PDA darin besteht, dass der PDA ein Symbol von seinem Stack entfernt, während er Eingabesymbole einliest. Da ein PDA seinen Zustand ändern kann, während er Symbole vom Stack entfernt, müssen wir auch vermerken, in welchen Zustand der PDA übergeht, wenn er eine Ebene von Symbolen aus seinem Stack entfernt.

Abbildung 6.10: Ein PDA führt eine Folge von Bewegungen aus, die im Endeffekt bewirken, dass ein Symbol vom Stack entfernt wird

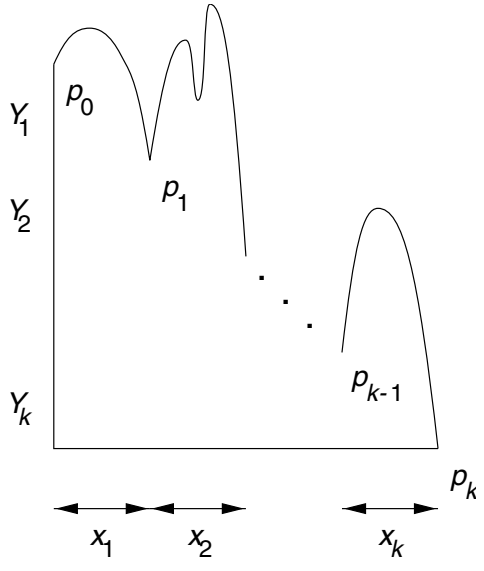


Abbildung 6.10 zeigt, wie die Folge von Symbolen Y_1, Y_2, ..., Y_k vom Stack entfernt wird. Während Y_1 entfernt wird, wird die Eingabe x_1 gelesen. Betont sei, dass dieses »Entfernen« das Ergebnis von (möglicherweise) mehreren Bewegungen ist. Beispielsweise kann die erste Bewegung Y_1 in ein anderes Symbol Z ändern. Die nächste Bewegung kann Z durch UV ersetzen, nachfolgende Bewegungen können dazu führen, dass U vom Stack entfernt wird, und weitere Bewegungen entfernen V vom Stack. Im Endergebnis wurde Y_1 durch nichts ersetzt, d. h., Y_1 wurde vom Stack entfernt und alle bis dahin eingelesenen Eingabesymbole bilden x_1.

Wir zeigen in Abbildung 6.10 zudem die Zustandsänderung. Wir nehmen an, dass der PDA im Zustand p_0 startet, wobei Y_1 das oberste Stacksymbol darstellt. Nach den Bewegungen, mit denen Y_1 vom Stack entfernt wird, befindet sich der PDA im Zustand p_1. Anschließend wird (im Endergebnis) Y_2 vom Stack entfernt, während die Eingabezeichenreihe x_2 gelesen wird, und der PDA geht – möglicherweise nach mehreren Bewegungen – in den Zustand p_2 über, wobei Y_2 vom Stack entfernt worden ist. Diese Berechnung wird so lange fortgesetzt, bis alle Symbole vom Stack entfernt worden sind.

Wir verwenden in unserer Konstruktion einer äquivalenten Grammatik Variablen, die jeweils ein »Ereignis« repräsentieren, das sich wie folgt zusammensetzt:

1. Ein Symbol X wurde im Endergebnis vom Stack entfernt, ohne ersetzt zu werden, und
2. eine Zustandsänderung von einem anfänglichen Zustand p zu q, wenn X auf dem Stack durch ε ersetzt worden ist.

Wir repräsentieren eine solche Variable durch das zusammengesetzte Symbol $[pXq]$. Denken Sie daran, dass diese Zeichenreihe unsere Schreibweise für *eine* Variable ist und nicht fünf Symbole der Grammatik darstellt. Die formale Konstruktion wird durch folgenden Satz beschrieben.

Satz 6.14 Sei $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ ein PDA. Dann gibt es eine kontextfreie Grammatik G, derart dass $L(G) = N(P)$.

Beweis: Wir werden $G = (V, \Sigma, R, S)$ konstruieren, wobei die Menge der Variablen V sich zusammensetzt aus:

1. dem speziellen Symbol S, dem Startsymbol, und
2. allen Symbolen der Form $[pXq]$, wobei p und q in Q enthaltene Zustände sind und X ein Stacksymbol aus Γ ist.

Die Produktionen von G lauten wie folgt:

a) Für alle Zustände p verfügt G über die Produktion $S \rightarrow [q_0 Z_0 p]$. Rufen Sie sich in Erinnerung, was wir beabsichtigen: Ein Symbol wie $[q_0 Z_0 p]$ soll all diejenigen Zeichenreihen w erzeugen, die bewirken, dass der PDA P Z_0 von seinem Stack entfernt, während er vom Zustand q_0 in den Zustand p übergeht. Das heißt: $(q_0 w Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$. Wenn dem so ist, dann besagen diese Produktionen, dass das Startsymbol S alle Zeichenreihen w erzeugt, die P dazu veranlassen, seinen Stack zu leeren, wenn er mit seiner Anfangskonfiguration beginnt.

b) Angenommen, $\delta(q, a, X)$ enthält das Paar $(r, Y_1 Y_2 ... Y_k)$, wobei

   1. a entweder ein Symbol aus Σ oder $a = \varepsilon$ ist.
   2. k kann eine beliebige Zahl einschließlich Null sein, in welchem Fall das Paar dann (r, ε) lautet.

   Dann verfügt G für alle Listen von Zuständen $r_1, r_2, ..., r_k$ über die Produktion

   $$[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]$$

   Diese Produktion besagt, dass eine Möglichkeit, X vom Stack zu entfernen und vom Zustand q in den Zustand r_k zu wechseln, darin besteht, ein Symbol a (das ε sein kann) zu lesen, eine Eingabe zu lesen, um Y_1 vom Stack zu entfernen, während vom Zustand r in den Zustand r_1 gewechselt wird, dann eine weitere Eingabe zu lesen, die Y_2 vom Stack entfernt und vom Zustand r_1 in den Zustand r_2 wechselt usw.

Wir werden nun beweisen, dass die informelle Interpretation der Variablen $[qXp]$ korrekt ist:

- $[qXp] \overset{*}{\Rightarrow} w$ genau dann, wenn $(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$.

(Wenn-Teil) Angenommen $(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$. Wir werden $[qXp] \overset{*}{\Rightarrow} w$ durch Induktion über die Anzahl der Bewegungen des PDA zeigen.

Induktionsbeginn: Ein Schritt. Dann muss (p, ε) in $\delta(q, w, X)$ enthalten sein und w ist entweder ein einzelnes Symbol oder ε. Gemäß der Konstruktion von G ist $[qXp] \rightarrow w$ eine Produktion, daher gilt $[qXp] \Rightarrow w$.

INDUKTIONSSCHRITT: Angenommen, die Sequenz $(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$ erfordert n Schritte und $n > 1$. Der erste Schritt muss folgendermaßen aussehen:

$$(q, w, X) \vdash (r_0, x, Y_1Y_2 \ldots Y_k) \overset{*}{\vdash} (p, \varepsilon, \varepsilon),$$

wobei $w = ax$ für ein a, das entweder ε oder ein Symbol aus Σ ist. Daraus folgt, dass das Paar $(r_0, Y_1Y_2 \ldots Y_k)$ in $\delta(q, a, X)$ enthalten sein muss. Nach der Konstruktion von G muss es zudem eine Produktion der Form $[qXr_k] \rightarrow a[r_0Y_1r_1][r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]$ geben, wobei

1. $r_k = p$ und
2. $r_1, r_2, \ldots, r_{k-1}$ Zustände aus Q sind.

Wie in Abbildung 6.10 dargestellt, können wir insbesondere feststellen, dass jedes der Symbole $Y_1, Y_2, \ldots, Y_k$ nacheinander vom Stack entfernt wird, und wir können p_i als den Zustand des PDA wählen, den dieser beim Entfernen von Y_i innehat, für $i = 1, 2, \ldots, k\text{-}1$. Sei $x = w_1w_2 \ldots w_k$, wobei w_i die Eingabe repräsentiert, die eingelesen wird, während Y_i vom Stack entfernt wird. Dann wissen wir, dass $(r_{i-1}, w_i, Y_i) \overset{*}{\vdash} (r_i, \varepsilon, \varepsilon)$.

Da keine dieser Bewegungsfolgen n Schritte umfassen kann, gilt die Induktionshypothese für sie. Wir schließen daraus, dass $[r_{i-1}Y_ir_i] \overset{*}{\Rightarrow} w_i$. Wir können diese Ableitungen mit der Produktion verbinden, die zuerst verwendet wurde, und daraus schließen

$$\begin{aligned}
&[qXr_k] \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2]\cdots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} \\
&aw_1[r_1Y_2r_2][r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} \\
&aw_1w_2[r_2Y_3r_3]\cdots[r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} \\
&\cdots \\
&aw_1w_2\cdots w_k = w
\end{aligned}$$

wobei $r_k = p$.

(Nur-wenn-Teil) Der Beweis ist eine Induktion über die Anzahl der Ableitungsschritte.

INDUKTIONSBEGINN: Ein Schritt: Dann muss $[qXp] \rightarrow w$ eine Produktion sein. Es kann diese Produktion nur geben, wenn es einen Übergang in P gibt, in dem X vom Stack entfernt und vom Zustand q zum Zustand p gewechselt wird. Das heißt, (p, ε) muss in $\delta(q, a, X)$ enthalten sein und a = w. Dann ist $(q, w, X) \vdash (p, \varepsilon, \varepsilon)$.

INDUKTIONSSCHRITT: Angenommen, $[qXp] \overset{*}{\Rightarrow} w$ in n Schritten, wobei $n > 1$. Betrachten Sie explizit die erste Satzform, die wie folgt aussehen muss:

$$[qXr_k) \Rightarrow a[r_0Y_1r_1][r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k] \overset{*}{\Rightarrow} w$$

wobei $r_k = p$. Diese Produktion muss sich aus der Tatsache ergeben, dass $r_0, Y_1Y_2 \ldots Y_k)$ in $\delta(q, a, X)$ enthalten ist.

Wir können w aufteilen in $aw_1w_2 \ldots w_k$, derart dass $[r_{i-1}Y_ir_i] \overset{*}{\Rightarrow} w_i$ für alle $i = 1,2, \ldots, r_k$. Nach der Induktionshypothese wissen wir, dass für alle i gilt

$$(r_{i-1}, w_i, Y_i) \overset{*}{\vdash} (r_i, \varepsilon, \varepsilon)$$

Wenn wir Satz 6.5 verwenden, um die korrekten Zeichenreihen hinter w_i in der Eingabe und unterhalb von Y_i auf dem Stack einzufügen, dann wissen wir, dass

$$(r_{i-1}, w_i w_{i+1} \ldots w_k, Y_i Y_{i+1} \ldots Y_k) \overset{*}{\vdash} (r_i, w_{i+1} \ldots w_k, Y_{i+1} \ldots Y_k)$$

Wenn wir diese Sequenzen zusammenfügen, dann sehen wir, dass

$$(q, aw_1 w_2 \ldots w_k, X) \vdash (r_0, w_1 w_2 \ldots w_k, Y_1 Y_2 \ldots Y_k) \overset{*}{\vdash}$$

$$(r_1, w_2 w_3 \ldots w_k, Y_2 Y_3 \ldots Y_k) \overset{*}{\vdash} (r_2, w_3 \ldots w_k, Y_3 \ldots Y_k) \overset{*}{\vdash} \ldots \overset{*}{\vdash} (r_k, \varepsilon, \varepsilon)$$

Da $r_k = p$, haben wir gezeigt, dass $(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$.

Wir vervollständigen diesen Beweis wie folgt. $S \overset{*}{\Rightarrow} w$ genau dann, wenn für ein p gilt $[q_0 Z_0 p] \overset{*}{\Rightarrow} w$, auf Grund der Art und Weise, in der die Regeln für das Startsymbol S konstruiert werden. Wir haben also gerade bewiesen, dass $[q_0 Z_0 p] \overset{*}{\Rightarrow} w$ genau dann, wenn $(q, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$, d. h. genau dann, wenn P x durch Leeren seines Stacks akzeptiert. Folglich gilt $L(G) = N(P)$. ■

Beispiel 6.15 Wir wollen den PDA $P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$ aus Beispiel 6.10 in eine Grammatik umwandeln. Rufen Sie sich in Erinnerung, dass P_N alle Zeichenreihen akzeptiert, die einen Verstoß gegen die Regel darstellen, dass es möglich sein muss, jedem *e* (`else`) ein voranstehendes *i* (`if`) zuzuordnen. Da P_N nur über einen Zustand und ein Stacksymbol verfügt, ist diese Konstruktion besonders einfach. Die Grammatik G besitzt nur zwei Variablen:

a) Das Startsymbol S, das in jeder Grammatik nach der mit Satz 6.14 gegebenen Methode konstruiert wird, und

b) $[qZq]$, das einzige Tripel, das sich aus den Zuständen und Stacksymbolen von P_N bilden lässt.

Die Produktionen der Grammatik G lauten wie folgt:

1. Für S existiert lediglich die Produktion $S \to [qZq]$. Wenn der PDA jedoch n Zustände besitzen würde, dann gäbe es n Produktionen dieser Art, da jeder dieser n Zustände den letzten Zustand bilden könnte. Der erste Zustand müsste der Startzustand sein und das Stacksymbol müsste dem Startsymbol entsprechen, wie es unsere Produktion zeigt.

2. Aus der Tatsache, dass $\delta_N(q, i, Z)$ das Paar (q, ZZ) enthält, ergibt sich die Produktion $[qZq] \to i[qZq][qZq]$. In diesem einfachen Beispiel gibt es nur eine Produktion. Gäbe es allerdings n Zustände, dann würde diese eine Regel n^2 Produktionen erzeugen, da es sich sowohl bei den beiden mittleren Zuständen des Produktionsrumpfs als auch bei dem letzten Zustand des Produktionskopfs und -rumpfs um jeden beliebigen Zustand handeln könnte. Das heißt, wenn p und r beliebige Zustände des PDA wären, dann würde sich die Produktion $[qZp] \to i[qZr][rZp]$ ergeben.

3. Aus der Tatsache, dass $\delta_N(q, e, Z)$ das Paar (q, ε) enthält, folgt die Produktion

$$[qZq] \to e$$

Beachten Sie, dass in diesem Fall die Liste der Stacksymbole, durch die Z ersetzt wird, leer ist, sodass der Rumpf als einziges Symbol das Eingabesymbol enthält, das die Bewegung bewirkt hat.

Wir können nun das Tripel $[qZq]$ durch ein weniger kompliziertes Symbol, sagen wir A, ersetzen. Wenn wir dies tun, besteht die gesamte Grammatik aus den folgenden Produktionen:

$$S \to A$$
$$A \to iAA \mid e$$

Da wir feststellen können, dass A und S genau dieselben Zeichenreihen ableiten, können wir sie als identisch betrachten und die gesamte Grammatik wie folgt formulieren:

$$G = (\{S\}, \{i, e\}, \{S \to iSS \mid e\}, S)$$ ■

6.3.3 Übungen zum Abschnitt 6.3

* **Übung 6.3.1** Wandeln Sie die Grammatik

$$S \to 0S1 \mid A\#$$
$$S \to 1A0 \mid S \mid \varepsilon$$

in einen PDA um, der dieselbe Sprache durch Leeren seines Stacks akzeptiert.

Übung 6.3.2 Wandeln Sie die Grammatik

$$S \to aAA$$
$$S \to aS \mid bS \mid a$$

in einen PDA um, der dieselbe Sprache durch Leeren seines Stacks akzeptiert.

* **Übung 6.3.3** Wandeln Sie den PDA $P = (\{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$ in eine kontextfreie Grammatik um, wobei δ definiert ist durch

1. $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$.
2. $\delta(q, 1, X) = \{(q, XX)\}$.
3. $\delta(q, 0, X) = \{(p, X)\}$.
4. $\delta(q, \varepsilon, X) = \{(q, \varepsilon)\}$.
5. $\delta(p, 1, X) = \{(p, \varepsilon)\}$.
6. $\delta(p, 0, Z_0) = \{(q, Z_0)\}$.

Übung 6.3.4 Wandeln Sie den PDA aus Übung 6.1.1 in eine kontextfreie Grammatik um.

Übung 6.3.5 Unten sind einige kontextfreie Grammatiken aufgeführt. Entwerfen Sie für jede einen PDA, der die Sprache durch Leeren seines Stacks akzeptiert. Es steht Ihnen frei, zuerst eine Grammatik für die Sprache zu konstruieren und diese dann in einen PDA umzuwandeln.

a) $\{a^n b^m c^{2(n+m)} \mid n \geq 0, m \geq 0\}$.

b) $\{a^i b^j c^{k)} \mid i = 2j$ oder $j = 2k\}$.

! c) $\{0^n 1^m \mid n \leq m \leq 2n\}$.

*! **Übung 6.3.6** Zeigen Sie, dass es für einen PDA P einen PDA P_1 mit nur einem Zustand gibt, sodass gilt: $N(P_1) = N(P)$.

! **Übung 6.3.7** Angenommen, es ist ein PDA mit s Zuständen, t Stacksymbolen und Regeln gegeben, in denen keine Zeichenreihe, die ein Stacksymbol ersetzt, länger als u ist. Geben Sie eine scharfe Obergrenze für die Anzahl von Variablen der kfG an, die wir nach der in Abschnitt 6.3.2. beschriebenen Methode für diesen PDA konstruieren können.

6.4 Deterministische Pushdown-Automaten

Obwohl PDAs definitionsgemäß nichtdeterministisch sein können, ist die Teilklasse der deterministischen PDAs recht wichtig. Insbesondere Parser verhalten sich wie deterministische PDAs. Daher ist die Klasse der Sprachen, die von diesen Automaten akzeptiert wird, von Interesse, weil sie uns Aufschluss über die Arten von Konstrukten gibt, die sich für die Verwendung in Programmiersprachen eignen. In diesem Abschnitt definieren wir deterministische PDAs und untersuchen einige der Dinge, die diese Automaten leisten bzw. nicht leisten können.

6.4.1 Definition des deterministischen Pushdown-Automaten

Ein PDA ist deterministisch, wenn in keiner Situation mehrere Bewegungen möglich sind. PDAs kennen zwei Arten der Auswahl von Bewegungsmöglichkeiten: Wenn $\delta(q, a, X)$ mehr als ein Paar enthält, dann ist der PDA mit Sicherheit nichtdeterministisch, weil eines dieser Paare zur Bestimmung der nächsten Bewegung gewählt werden kann. Selbst wenn jedoch $\delta(q, a, X)$ stets nur ein einzelnes Paar enthält, bestünde im Allgemeinen immer noch die Wahlmöglichkeit zwischen einem Eingabesymbol und einer ε-Bewegung. Wir definieren daher, dass ein PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ genau dann ein *deterministischer* PDA (kurz DPDA) ist, wenn die folgenden Bedingungen erfüllt sind:

1. $\delta(q, a, X)$ besitzt für jedes q aus Q, a aus Σ oder $a = \varepsilon$ und X aus Γ höchstens ein Element.
2. Wenn $\delta(q, a, X)$ für ein a aus Σ nicht leer ist, dann muss $\delta(q, \varepsilon, X)$ leer sein.

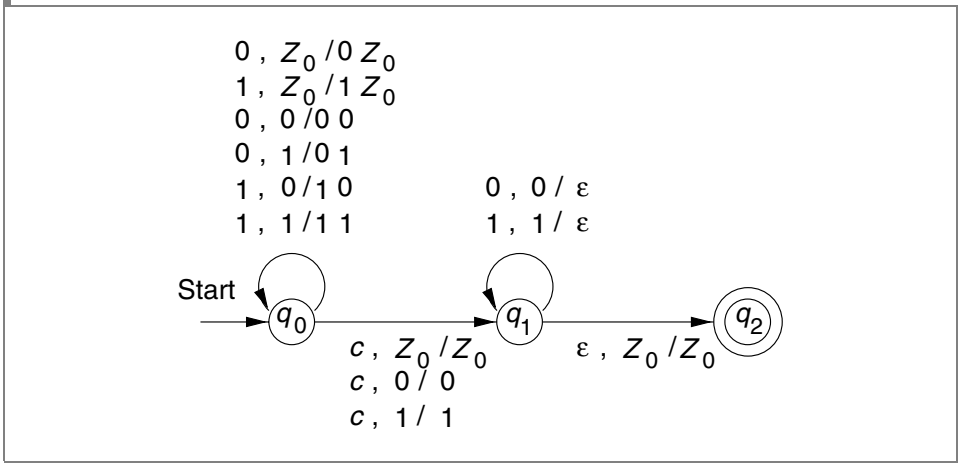
Beispiel 6.16 Es stellt sich heraus, dass die Sprache L_{wwr} aus Beispiel 6.2 eine kontextfreie Sprache ist, für die kein DPDA existiert. Wenn wir jedoch eine »Mittelmarkierung« c in der Mitte einfügen, dann wird die Sprache für einen DPDA erkennbar. Das heißt, ein DPDA kann die Sprache $L_{wcwr} = \{wcw^R \mid w$ ist in $(\mathbf{0} + \mathbf{1})^*$ enthalten$\}$ erkennen.

Die Strategie des DPDA besteht darin, so lange Nullen und Einsen auf seinem Stack zu speichern, bis er die Markierung c liest. Dann wechselt er in einen anderen Zustand, in dem er Eingabesymbole mit Stacksymbolen vergleicht und bei Übereinstimmung das Symbol vom Stack entfernt. Falls die Symbole nicht übereinstimmen, endet die Berechnung ohne Akzeptanz, da die Eingabe nicht die Form wcw^R haben

kann. Wenn der Stack bis zu dem Symbol geleert werden kann, das den Stackanfang markiert, dann akzeptiert der DPDA die Eingabe.

Der Gedankengang ähnelt dem PDA aus Abbildung 6.2. Dieser PDA ist allerdings nichtdeterministisch, weil er im Zustand q_0 stets die Wahlmöglichkeit hat, das nächste Eingabesymbol auf den Stack zu legen oder eine ε-Bewegung in den Zustand q_1 auszuführen; er muss also raten, ob er die Mitte erreicht hat. Der DPDA für die Sprache L_{wcwr} wird im Übergangsdiagramm von Abbildung 6.11 dargestellt.

Abbildung 6.11: Deterministischer PDA, der die Sprache L_{wcwr} akzeptiert



Dieser PDA ist offensichtlich deterministisch. Er hat in einem Zustand nie mehrere Bewegungsmöglichkeiten, wenn ein Eingabesymbol und ein Stacksymbol gegeben sind. Was die Wahl zwischen einem echten Eingabesymbol und ε betrifft, so führt er lediglich dann eine ε-Bewegung vom Zustand q_1 in den Zustand q_2 aus, wenn Z_0 das oberste Stacksymbol bildet. Im Zustand q_1 sind allerdings keine anderen Bewegungen möglich, wenn Z_0 das oberste Stacksymbol ist. ■

6.4.2 Reguläre Sprachen und deterministische PDAs

DPDAs akzeptieren eine Klasse von Sprachen, die zwischen den regulären Sprachen und den kontextfreien Sprachen liegt. Wir werden zuerst beweisen, dass die Sprachen von DPDAs alle regulären Sprachen umfassen.

Satz 6.17 Wenn L eine reguläre Sprache ist, dann gibt es einen DPDA P mit $L = L(P)$.

Beweis: Im Grunde genommen kann ein DPDA einen deterministischen endlichen Automaten simulieren. Der PDA behält ein Stacksymbol Z_0 auf seinem Stack, weil ein PDA einen Stack besitzen muss, ignoriert den Stack aber eigentlich und arbeitet lediglich mit seinem Zustand. Formal ausgedrückt, sei $A = (Q, \Sigma, \delta_A, q_0, F)$ ein DFA. Wir konstruieren einen DPDA

$$P = (Q, \Sigma, \{Z_0\}, \delta_P, q_0, Z_0, F)$$

indem $\delta_P(q, a, Z_0) = \{(p, Z_0)\}$ für alle Zustände p und q aus Q definiert wird, derart dass $\delta_A(q, a) = p$.

Wir behaupten, dass $(q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, Z_0)$ genau dann, wenn $\hat{\delta}_A(q_0, w) = p$. Das heißt, P simuliert A über seinen Zustand. Beide Richtungen lassen sich durch einfache Induktionsbeweise über $|w|$ zeigen und wir überlassen die Ausführung dieser Beweise dem Leser. Da sowohl A als auch P akzeptieren, indem sie einen der Zustände aus F annehmen, schließen wir darauf, dass ihre Sprachen identisch sind. ■

Wenn ein DPDA durch Leeren des Stacks akzeptieren soll, dann werden wir erkennen, dass wir lediglich über eine sehr beschränkte Fähigkeit zur Spracherkennung verfügen. Angenommen, eine Sprache L habe die *Präfixeigenschaft* , wenn es keine zwei verschiedene Zeichenreihen x und y in L gibt, derart dass x ein Präfix von y ist.

Beispiel 6.18 Die Sprache L_{wcwr} aus dem Beispiel besitzt die Präfixeigenschaft. Das heißt, es kann keine zwei Zeichenreihen wcw^R und xcx^R geben, wobei eine ein Präfix der anderen ist, sofern nicht $x = w$. Zur Erläuterung nehmen wir an, wcw^R sei ein Präfix von xcx^R und $w \neq x$. Dann muss w kürzer sein als x. Daher muss das c in wcw^R an einer Position stehen, an der in xcx^R eine Null oder Eins steht, und dies ist eine Position im ersten x. Dies widerspricht der Annahme, dass wcw^R ein Präfix von xcx^R ist.

Andererseits gibt es einige sehr einfache Sprachen, die nicht über die Präfixeigenschaft verfügen. Betrachten Sie beispielsweise {0}*, d. h. die Menge aller Zeichenreihen aus Nullen. Offensichtlich gibt es in dieser Sprache Paare von Zeichenreihen, bei denen eine Zeichenreihe ein Präfix der anderen ist, sodass diese Sprache nicht über die Präfixeigenschaft verfügt. In der Tat ist hier bei *jedem* Paar von Zeichenreihen eine Zeichenreihe ein Präfix der anderen. Allerdings ist diese Eigenschaft stärker als nötig, um zu zeigen, dass hier die Präfixeigenschaft nicht gilt. ■

Beachten Sie, dass die Sprache {0}* eine reguläre Sprache ist. Folglich gilt nicht einmal, dass jede reguläre Sprache $N(P)$ für einen DPDA P ist. Wir überlassen den Beweis des folgenden Satzes einer Übung.

Satz 6.19 Eine Sprache L ist genau dann $N(P)$ für einen DPDA P, wenn L die Präfixeigenschaft besitzt und für einen DPDA P' $L = L(P')$ ist. ■

6.4.3 DPDAs und kontextfreie Sprachen

Wir haben bereits gezeigt, dass ein DPDA Sprachen wie L_{wcwr} akzeptieren kann, die nicht regulär sind. Um zu zeigen, dass diese Sprache nicht regulär ist, nehmen wir an, sie sei regulär, und verwenden das Pumping-Lemma. Wenn n die Konstante des Pumping-Lemma ist, dann betrachten wir die Zeichenreihe $w = 0^n c 0^n$, die in L_{wcwr} enthalten ist. Wenn wir diese Zeichenreihe jedoch »aufpumpen«, dann muss sich die Länge der ersten Guppe von Nullen ändern, sodass wir Zeichenreihen in L_{wcwr} erhalten, deren »Mittelmarkierung« sich nicht in der Mitte befindet. Da diese Zeichenreihen nicht in L_{wcwr} enthalten sind, liegt ein Widerspruch vor und wir schließen daraus, dass L_{wcwr} nicht regulär ist.

Andererseits gibt es kontextfreie Sprachen wie L_{wwr}, die für keinen DPDA P $L(P)$ sein können. Der formale Beweis ist kompliziert, eine intuitive Erklärung ist jedoch leicht verständlich. Wenn P ein DPDA ist, der die Sprache L_{wwr} akzeptiert, dann muss er bei Eingabe einer Folge von Nullen diese auf dem Stack speichern oder etwas Entsprechendes tun, um eine beliebige Anzahl von Nullen zu zählen. Er könnte beispielsweise jeweils ein X speichern, nachdem er zwei Nullen gelesen hat und durch seinen Zustand anzeigen, ob es sich um eine gerade oder eine ungerade Zahl gehandelt hat.

Angenommen, P hat n Nullen gelesen und liest dann 110^n. Er muss überprüfen, ob nach 11 n Nullen folgen und zu diesem Zweck muss er seinen Stack leeren[6]. Nun hat P 0^n110^n gelesen. Wenn er als Nächstes eine identische Zeichenreihe liest, dann muss er akzeptieren, weil die vollständige Eingabe die Form ww^R hat, mit $w = 0^n110^n$. Liest er jedoch 0^m110^m für ein $m \neq n$, dann darf P *nicht* akzeptieren. Da sein Stack leer ist, weiß er nicht mehr, welche ganze Zahl n war, und kann L_{wwr} daher nicht korrekt erkennen. Unsere Schlussfolgerung lautet:

- Die Sprachen, die von DPDAs korrekt durch Endzustand akzeptiert werden, umfassen die regulären Sprachen, sind jedoch eine echte Teilmenge der kontextfreien Sprachen.

6.4.4 DPDAs und mehrdeutige Grammatiken

Wir können die Mächtigkeit der DPDA weiter ausloten, indem wir feststellen, dass alle von ihnen akzeptierten Sprachen über eindeutige Grammatiken verfügen. Leider entsprechen die DPDA-Sprachen nicht genau der Teilmenge der kontextfreien Sprachen, die nicht inhärent mehrdeutig sind. Beispielsweise hat L_{wwr} eine eindeutige Grammatik

$$S \rightarrow 0S0 \mid 1S1 \mid \varepsilon.$$

obwohl es keine DPDA-Sprache ist. Die folgenden Sätze präzisieren den oben genannten Punkt.

Satz 6.20 Wenn für einen DPDA P $L = N(P)$, dann hat L eine eindeutige kontextfreie Grammatik.

Beweis: Wir behaupten, dass die Konstruktion aus Satz 6.14 eine eindeutige kontextfreie Grammatik ergibt, wenn sie auf einen deterministischen PDA angewandt wird. Rufen Sie sich zuerst aus Satz 5.29 in Erinnerung, dass nur gezeigt werden muss, dass eine Grammatik eindeutige linksseitige Ableitungen besitzt, um die Eindeutigkeit von G zu beweisen.

Angenommen, P akzeptiert w durch Leeren seines Stacks. Da der PDA deterministisch ist, führt er eine eindeutige Folge von Bewegungen aus und kann keine Bewegung mehr ausführen, sobald sein Stack leer ist. Wenn wir diese Folge von Bewegungen kennen, dann können wir die in der linksseitigen Ableitung verwendeten Produktionen ermitteln, mit der G w ableitet. Es stehen nie mehrere Regeln für P zur Wahl der zu verwendenden Produktion zur Verfügung. Allerdings kann eine Regel von P, sagen wir $\delta(q, a\ X) = \{(r, Y_1Y_2 \ldots Y_k)\}$, viele Produktionen von G verursachen, wobei die Positionen jeweils mit den verschiedenen Zuständen von P besetzt sind, die dieser annimmt, nachdem er $Y_1Y_2 \ldots Y_{k-1}$ vom Stack entfernt hat. Weil P deterministisch ist, kann nur eine dieser Folgen das Verhalten von P korrekt beschreiben und daher führt nur eine dieser Produktionen zur Ableitung von w. ■

Wir können allerdings mehr beweisen: Sogar alle Sprachen, die DPDAs durch Endzustand akzeptieren, haben eine eindeutige Grammatik. Da wir nur wissen, wie man ausgehend von PDAs, die durch Leeren des Stacks akzeptieren, eine Grammatik kon-

6. Diese Aussage stellt den intuitiven Teil dar, der einen formalen Beweis erfordert. Kann P auf irgendeine andere Weise gleich große Blöcke von Nullen vergleichen?

struiert, müssen wir die betreffende Sprache so abändern, dass sie die Präfixeigenschaft aufweist, und dann die resultierende Grammatik so modifizieren, dass sie die ursprüngliche Sprache erzeugt. Wir tun dies mithilfe eines »Endemarkierungssymbols«.

Satz 6.21 Wenn für einen DPDA P $L = L(P)$, dann hat L eine eindeutige kontextfreie Grammatik.

Beweis: Sei \$ ein »Endemarkierungssymbol«, das in den Zeichenreihen von L nicht vorkommt, und sei $L' = L\$$. Das heißt, die Zeichenreihen von L' sind die Zeichenreihen von L, an die jeweils das Symbol \$ angehängt wurde. Damit verfügt L' mit Sicherheit über die Präfixeigenschaft und nach Satz 6.19 gilt $L' = N(P')$ für einen DPDA P'[7]. Nach Satz 6.20 gibt es eine eindeutige Grammatik G', die die Sprache $N(P')$, also L', generiert.

Nun konstruieren wir aus G' eine Grammatik G, derart dass $L(G) = L$. Hierzu müssen wir lediglich die Endmarkierung \$ aus den Zeichenreihen entfernen. Daher behandeln wir \$ als eine Variable von G und führen die Produktion $\$ \to \varepsilon$ ein. Ansonsten sind die Produktionen von G' und G identisch. Da $L(G') = L'$, folgt daraus, dass $L(G) = L$.

Wir behaupten, dass G eindeutig ist. Im Beweis unterscheiden sich die linksseitigen Ableitungen von G und G' nur dadurch, dass die Ableitungen von G einen letzten Schritt aufweisen, in dem \$ durch ε ersetzt wird. Wenn eine terminale Zeichenreihe w in G nun zwei linksseitige Ableitungen hätte, dann hätte $w\$$ zwei linksseitige Ableitungen in G'. Da wir wissen, dass G' eindeutig ist, ist auch G eindeutig. ■

6.4.5 Übungen zum Abschnitt 6.4

Übung 6.4.1 Geben Sie für jeden der folgenden PDAs an, ob er deterministisch ist. Zeigen Sie entweder, ob er der Definition eines DPDA entspricht, oder finden Sie eine Regel, die dagegen verstößt.

a) Der PDA aus Beispiel 6.2.

* b) Der PDA aus Übung 6.1.1.

c) Der PDA aus Übung 6.3.3.

Übung 6.4.2 Geben Sie für jede der folgenden Sprachen einen deterministischen PDA an:

a) $\{0^n1^m \mid n \leq m\}$.

b) $\{0^n1^m \mid n \geq m\}$.

c) $\{0^n1^m0^n \mid n$ und m beliebig$\}$.

---

7. Der Beweis von Satz 6.19 erscheint in Übung 6.4.3, es ist jedoch leicht erkennbar, wie sich P' aus P konstruieren lässt. Wir fügen einen neuen Zustand q hinzu, den P' annimmt, sobald P einen akzeptierenden Zustand innehat und das nächste Eingabesymbol \$ lautet. Im Zustand q entfernt P' alle Symbole vom Stack. Zudem muss P' eine eigene Stackanfangsmarkierung besitzen, um zu verhindern, dass er während der Simulation von P versehentlich seinen Stack leert.

Übung 6.4.3 Wir können Satz 6.19 in drei Teilen beweisen:

* d) Zeigen Sie, dass gilt: Wenn $L = N(P)$ für einen DPDA P, dann hat L die Präfixeigenschaft.

! e) Zeigen Sie, dass gilt: Wenn $L = N(P)$ für einen DPDA P, dann gibt es einen DPDA P', derart dass $L = L(P')$.

!* f) Zeigen Sie, dass gilt: Wenn L über die Präfixeigenschaft verfügt und $L(P')$ für einen DPDA P' ist, dann gibt es einen DPDA P, derart dass $L = N(P)$.

!! **Übung 6.4.4** Zeigen Sie, dass die Sprache

$$L = \{0^n1^n \mid n \geq 1\} \cup \{0^n1^{2n} \mid n \geq 1\}$$

eine kontextfreie Sprache ist, die von keinem DPDA akzeptiert wird. *Hinweis*: Zeigen Sie, dass es zwei Zeichenreihen der Form 0^n1^n mit unterschiedlichen Werten von n, sagen wir n_1 und n_2, geben muss, die einen hypothetischen DPDA für L dazu veranlassen, nach dem Lesen jeder der beiden Zeichenreihen dieselbe Konfiguration anzunehmen. Der DPDA muss dazu fast alles vom Stack entfernen, was er beim Lesen der Nullen dort gespeichert hat, damit er prüfen kann, ob er die gleiche Anzahl von Einsen gelesen hat. Folglich kann der DPDA nicht ermitteln, ob er nach dem weiteren Einlesen von n_1 Einsen oder nach dem weiteren Einlesen von n_2 Einsen akzeptieren soll.

6.5 Zusammenfassung von Kapitel 6

- *Pushdown-Automaten (PDA)*: Ein PDA ist ein nichtdeterministischer endlicher Automat mit einem Stack, der zum Speichern von Zeichenreihen beliebiger Länge benutzt werden kann. Der Stack kann nur am oberen Ende gelesen und verändert werden.
- *Bewegungen von Pushdown-Automaten*: Ein PDA entscheidet auf der Grundlage seines aktuellen Zustands, des nächsten Eingabesymbols und des obersten Stacksymbols, welche Bewegung er als Nächstes ausführt. Er kann auch unabhängig vom Eingabesymbol und ohne dieses Symbol der Eingabe zu entnehmen, eine Bewegung ausführen. Nachdem der PDA nichtdeterministisch ist, kann er dabei die Wahl zwischen endlich vielen Bewegungsmöglichkeiten haben, von denen jede einen neuen Zustand und eine Zeichenreihe von Stacksymbolen angibt, durch die das aktuelle Symbol am oberen Ende des Stacks ersetzt wird.
- *Akzeptanz durch Pushdown-Automaten*: Ein PDA kann auf zweierlei Weise Akzeptanz signalisieren: durch den Übergang in einen akzeptierenden Zustand oder durch Leeren seines Stacks. Diese Methoden sind äquivalent in dem Sinn, dass jede Sprache, die durch die eine Methode akzeptiert wird, auch durch die andere (im Allgemeinen durch einen anderen PDA) akzeptiert wird.
- *Konfigurationen (ID, Instantaneous Description)*: Wir verwenden eine Konfiguration, die sich aus dem Zustand, der verbleibenden Eingabe und dem Stackinhalt zusammensetzt, um die aktuelle Situation eines PDA zu beschreiben. Eine Übergangsfunktion $\vdash$ zwischen Konfigurationen repräsentiert einzelne Bewegungen eines PDA.

- *Pushdown-Automaten und Grammatiken*: Bei den von PDAs durch Endzustand oder leeren Stack akzeptierten Sprachen handelt es sich genau um die kontextfreien Sprachen.
- *Deterministische Pushdown-Automaten*: Ein PDA ist deterministisch, wenn er zu einem gegebenen Zustand, einem Eingabesymbol (einschließlich ε) und einem Stacksymbol höchstens eine Bewegungsmöglichkeit hat. Zudem können deterministische PDAs nie zwischen einer Bewegung, die in Reaktion auf ein echtes Eingabesymbol erfolgt, und einer ε-Bewegung wählen.
- *Akzeptanz durch deterministische Pushdown-Automaten*: Die beiden Akzeptanzmethoden – Endzustand und leerer Stack – unterscheiden sich bei DPDAs. Bei den durch Leeren des Stacks akzeptierten Sprachen handelt es sich genau um die durch Endzustand akzeptierten Sprachen, die die Präfixeigenschaft haben: Keine Zeichenreihe der Sprache ist ein Präfix einer anderen Zeichenreihe der Sprache.
- *Die von DPDAs akzeptierten Sprachen*: Alle regulären Sprachen werden von DPDAs (durch Endzustand) akzeptiert und es gibt nicht reguläre Sprachen, die von DPDAs akzeptiert werden. Die Sprachen der DPDAs sind kontextfreie Sprachen, die eindeutige Grammatiken besitzen. Damit liegen die Sprachen der DPDAs echt zwischen den regulären Sprachen und den kontextfreien Sprachen.

LITERATURANGABEN ZU KAPITEL 6

Das Konzept der Pushdown-Automaten haben Oettinger [4] und Schutzenberger [5] unabhängig voneinander entwickelt. Die Äquivalenz zwischen Pushdown-Automaten und kontextfreien Sprachen war ebenso Ergebnis voneinander unabhängiger Entdeckungen. Sie wurde 1961 in einem Report des MIT von N. Chomski beschrieben, wurde aber zuerst von Evey [1] publiziert.

Der deterministische PDA wurde zuerst von Fischer [2] und Schutzenberger [5] eingeführt. Er gewann später als Modell für Parser an Bedeutung. Insbesondere [3] führt »LR(*k*)-Grammatiken« ein, eine Teilklasse von kfGs, die genau die DPDA-Sprachen erzeugen. Die LR(*k*)-Grammatiken bilden ihrerseits die Grundlage für YACC, das in Abschnitt 5.3.2 erörterte Programm zur Generierung von Parsern.

1. J. Evey [1963]. »Application of pushdown store machines«, *Proc. Fall Joint Computer Conference*, AFIPS Press, Montvale, NJ, S. 215-227.
2. P. C. Fischer [1963], »On computability by certain classes of restricted Turing machines«, Proc. *Fourth Annl. Symposium on Switching Circuit Theory and Logical Design*, S. 23-62.
3. D. E. Knutz [1965], »On the translation of languages from left to right«, *Information and Control* **8**:6, S. 607-639.
4. A. G. Oettinger [1961]. »Automatic syntactic analysis and the pushdown store«, *Proc. Symposia on Applied Math*. **12**, American Mathematical Society, Providence, RI.
5. M. P. Schutzenberger [1963]. »On context-free languages and pushdown automata«, *Information and Control* **6**:3, S. 246-264.

KAPITEL 7

Eigenschaften kontextfreier Sprachen

Wir werden unser Studium der kontextfreien Sprachen (kfS) nun abschließen, in dem wir einige ihrer Eigenschaften kennen lernen. Unsere erste Aufgabe besteht in der Vereinfachung kontextfreier Grammatiken. Diese Vereinfachungen erleichtern den Beweis verschiedener Eigenschaften von kfS, da wir dann wissen, dass eine Sprache in dem Fall eine kfS ist, wenn sie eine Grammatik besitzt, die eine spezielle Form hat.

Wir beweisen dann ein »Pumping-Lemma« für kfS. Dieser Satz verfolgt dieselbe Absicht wie Satz 4.1 für reguläre Sprachen und kann zum Beweis verwendet werden, dass eine Sprache nicht kontextfrei ist. Als Nächstes betrachten wir die Arten von Eigenschaften, die wir in Kapitel 4 an regulären Sprachen studiert haben: Eigenschaften der Abgeschlossenheit und der Entscheidbarkeit. Wir werden sehen, dass einige, aber nicht alle Eigenschaften der Abgeschlossenheit, die reguläre Sprachen besitzen, auch kfS-eigen sind. Ebenso können einige Fragen zu kfS durch Algorithmen entschieden werden, die eine Verallgemeinerung der Tests darstellen, die wir für reguläre Sprachen entwickelt haben. Es gibt jedoch bestimmte Fragen zu kontextfreien Sprachen, die wir nicht beantworten können.

7.1 Normalformen kontextfreier Grammatiken

Dieser Abschnitt soll zeigen, dass jede kontextfreie Sprache (ohne ε) von einer kontextfreien Grammatik (kfG) erzeugt wird, in der alle Produktionen die Gestalt $A \to BC$ oder $A \to a$ haben, wobei A, B und C Variablen sind und a eine terminale Zeichenreihe ist. Diese Form wird *Chomsky-Normalform* genannt. Zur Bildung der Normalform müssen wir erst einige vorläufige Vereinfachungen vornehmen, die für sich genommen in verschiedener Hinsicht hilfreich sind:

1. Wir müssen *unnütze Symbole* eliminieren, d. h. jene Variablen und Symbole, die in keiner Ableitung einer terminalen Zeichenreihe vom Startsymbol vorkommen.
2. Wir müssen *ε-Produktionen* eliminieren, also Produktionen der Form $A \to \varepsilon$ für eine Variable A.
3. Wir müssen *Einheitsproduktionen* (engl. *unit production*) eliminieren, also Produktionen der Form $A \to B$ für die Variablen A und B.

7.1.1 Eliminierung unnützer Symbole

Wir nennen ein Symbol X für eine Grammatik $G = (V, T, P, S)$ *nützlich* , wenn es eine Ableitung der Form $S \stackrel{*}{\Rightarrow} \alpha X \beta \stackrel{*}{\Rightarrow} w$ gibt, wobei w in T^* enthalten ist. Beachten Sie, dass X in V oder T sein kann, und die Satzform $\alpha X \beta$ kann die erste oder letzte in der Ableitung sein. Wenn X nicht nützlich ist, dann bezeichnen wir es als *unnütz* . Offensichtlich wirkt sich das Entfernen unnützer Symbole aus der Grammatik nicht auf die Sprache aus, die von der Grammatik generiert wird, und daher können wir alle unnützen Symbole suchen und eliminieren.

Unser Ansatz bezüglich des Eliminierens unnützer Symbole setzt beim Identifizieren der beiden Eigenschaften an, die ein nützliches Symbol haben muss:

1. Wir bezeichnen X als *erzeugend*, wenn $X \stackrel{*}{\Rightarrow} w$ für eine terminale Zeichenreihe w. Beachten Sie, dass jedes terminale Zeichen erzeugend ist, da w diese terminale Zeichenreihe selbst sein kann, die dann in null Schritten abgeleitet wird.
2. Wir bezeichnen X als *erreichbar*, wenn es für ein α und ein β eine Ableitung S $\stackrel{*}{\Rightarrow} \alpha X \beta$ gibt.

Sicher ist ein nützliches Symbol sowohl erzeugend als auch erreichbar. Wenn wir zuerst die Symbole eliminieren, die nicht erzeugend sind, und dann aus der verbleibenden Grammatik die Symbole eliminieren, die nicht erreichbar sind, dann sollten – wie wir beweisen werden – lediglich nützliche Symbole übrig bleiben.

Beispiel 7.1 Betrachten Sie die Grammatik

$$S \to AB \mid a$$
$$A \to b$$

Abgesehen von B, sind alle Symbole erzeugend; a und b erzeugen sich selbst; S erzeugt A und A erzeugt b. Wenn wir B eliminieren, dann müssen wir die Produktion $S \to AB$ eliminieren, sodass folgende Grammatik übrig bleibt:

$$S \to a$$
$$A \to b$$

Wir stellen nun fest, dass nur S und a von S aus erreichbar sind. Wenn wir A und b eliminieren, bleibt lediglich die Produktion $S \to a$ übrig. Diese Produktion stellt für sich genommen eine Grammatik dar, deren Sprache aus $\{a\}$ besteht, ebenso wie die Sprache der ursprünglichen Grammatik.

Wenn wir aber zuerst die Erreichbarkeit der Symbole überprüfen, dann stellt sich heraus, dass alle Symbole der Grammatik

$$S \to AB \mid a$$
$$A \to b$$

erreichbar sind. Wenn wir dann das Symbol B eliminieren, weil es nicht erzeugend ist, dann bleibt eine Grammatik übrig, die noch immer unnütze Symbole aufweist, nämlich A und b. ■

Satz 7.2 Sei $G = (V, T, P, S)$ eine kfG, und nehmen wir an, dass $L(G) \neq \emptyset$, d. h. G generiert mindestens eine Zeichenreihe. Sei $G_1 = (V_1, T_1, P_1, S)$ die Grammatik, die wir mit folgenden Schritten erhalten:

1. Zuerst eliminieren wir Symbole, die nichts erzeugen, und alle Produktionen, die eines oder mehrere solcher Symbole enthalten. Sei $G_2 = (V_2, T_2, P_2, S)$ diese neue Grammatik. Beachten Sie, dass S erzeugend sein muss, da wir annehmen, dass $L(G)$ mindestens eine Zeichenreihe umfasst, und daher wurde S nicht eliminiert.
2. Zweitens eliminieren wir alle Symbole, die in der Grammatik G_2 nicht erreichbar sind.

Dann enthält G_1 keine unnützen Symbole und $L(G_1) = L(G)$.

Beweis: Angenommen, X ist ein Symbol, das in der Grammatik bleibt, d. h. X ist in $V_1 \cup T_1$ enthalten. Wir wissen, dass $X \overset{*}{\underset{G}{\Rightarrow}} w$ für eine Zeichenreihe w aus T^*. Zudem ist jedes Symbol, das in dieser Ableitung von w aus X verwendet wird, auch erzeugend. Folglich gilt $X \overset{*}{\underset{G_2}{\Rightarrow}} w$.

Da X im zweiten Schritt nicht eliminiert worden ist, wissen wir, dass es α und β geben muss, derart dass $S \overset{*}{\underset{G_2}{\Rightarrow}} \alpha X \beta$. Zudem ist jedes in dieser Ableitung verwendete Symbol erreichbar, und daher gilt $S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta$.

Wir wissen, dass jedes Symbol in $\alpha X \beta$ erreichbar ist, und wir wissen zudem, dass alle diese Symbole in $V_2 \cup T_2$ enthalten sind. Daher ist jedes dieser Symbole erzeugend in G_2. Die Ableitung einer terminalen Zeichenreihe, sagen wir $\alpha X \beta \overset{*}{\underset{G_2}{\Rightarrow}} xwz$, enthält nur Symbole, die von S aus erreichbar sind, weil sie von in $\alpha X \beta$ enthaltenen Symbolen erreicht werden. Folglich ist diese Ableitung auch eine Ableitung in G_1, d. h.

$$S \overset{*}{\underset{G_1}{\Rightarrow}} \alpha X \beta \overset{*}{\underset{G_1}{\Rightarrow}} xwy$$

Wir schließen daraus, dass X in G_1 nützlich ist. Da X ein zufällig gewähltes Symbol von G_1 ist, schließen wir, dass G_1 keine unnützen Symbole enthält.

Schließlich müssen wir zeigen, dass $L(G_1) = L(G)$. Wie gewöhnlich beweisen wir die Gleichheit zweier Mengen dadurch, dass wir zeigen, dass jede in der anderen enthalten ist.

$L(G_1) \subseteq L(G)$: Da wir nur Symbole und Produktionen aus G entfernt haben, um G_1 zu erhalten, folgt daraus, dass $L(G_1) \subseteq L(G)$.

$L(G) \subseteq L(G_1)$: Wir müssen beweisen, dass gilt: Wenn w in $L(G)$ enthalten ist, dann ist w auch in $L(G_1)$ enthalten. Wenn w in $L(G)$ enthalten ist, dann gilt $S \overset{*}{\underset{G}{\Rightarrow}} w$. Jedes Symbol dieser Ableitung ist offensichtlich sowohl erreichbar als auch erzeugend, und somit ist es auch eine Ableitung von G_1. Das heißt, $S \overset{*}{\underset{G_1}{\Rightarrow}} w$, und folglich ist w in $L(G_1)$ enthalten. ■

7.1.2 Berechnung der erzeugenden und erreichbaren Symbole

Zwei Punkte müssen noch geklärt werden. Wie berechnen wir die Menge der erzeugenden Symbole einer Grammatik und wie die Menge der erreichbaren Symbole einer Grammatik? Mit dem Algorithmus, den wir zur Beantwortung dieser Fragen einsetzen, wird versucht, Symbole dieses Typs zu erkennen. Wir werden zeigen, dass ein

Symbol nicht von diesem Typ ist, wenn es mit den induktiven Konstruktionen dieser Mengen nicht als erzeugend bzw. als erreichbar erkannt wird.
Sei $G = (V, T, P, S)$ eine Grammatik. Zur Berechnung der erzeugenden Symbole setzen wir folgende Induktion ein.

Induktionsbeginn: Jedes Symbol aus T ist offensichtlich erzeugend; es erzeugt sich selbst.

Induktionsschritt: Angenommen, es gibt eine Produktion $A \to \alpha$, und von jedem Symbol in α ist bereits bekannt, dass es erzeugend ist. Dann ist A erzeugend. Beachten Sie, dass diese Regel auch für Fälle gilt, in denen $\alpha = \varepsilon$. Alle Variablen, die ε als einen Produktionsrumpf haben, sind mit Sicherheit erzeugend.

Beispiel 7.3 Betrachten Sie die Grammatik aus Beispiel 7.1. Nach Induktionsbeginn sind a und b erzeugend. Im Induktionsschritt können wir unter Verwendung der Produktion $A \to b$ schließen, dass A erzeugend ist, und wir können unter Verwendung der Produktion S $\to a$ schließen, dass S erzeugend ist. Damit ist die Induktion vollständig. Wir können die Produktion $S \to AB$ nicht einsetzen, weil B nicht als erzeugendes Symbol festgestellt wurde. Folglich besteht die Menge der erzeugenden Symbole aus $\{a, b, A, S\}$. ■

Satz 7.4 Der obige Algorithmus findet genau alle erzeugenden Symbole von G.

Beweis: Eine Richtung des Beweises besteht aus einer einfachen Induktion über die Reihenfolge, in der Symbole zur Menge der erzeugenden Symbole hinzugefügt werden, die zeigt, dass jedes hinzugefügte Symbol tatsächlich erzeugend ist. Wir überlassen dem Leser diesen Teil des Beweises.

Für die andere Richtung des Beweises nehmen wir an, X sei ein erzeugendes Symbol, z. B. $X \overset{*}{\underset{G}{\Rightarrow}} w$. Wir beweisen durch Induktion über die Länge dieser Ableitung, dass X als erzeugendes Symbol erkannt wird.

Induktionsbeginn: Null Schritte. In diesem Fall ist X ein terminales Zeichen, und X wird am Induktionsbeginn ermittelt.

Induktionsschritt: Wenn die Ableitung n Schritte mit $n > 0$ umfasst, dann ist X eine Variable. Angenommen, die Ableitung lautet $X \Rightarrow \alpha \overset{*}{\Rightarrow} w$; das heißt, zuerst wird die Produktion $X \Rightarrow \alpha$ verwendet. Jedes Symbol von α leitet eine terminale Zeichenreihe ab, die Teil von w ist, und diese Ableitung muss weniger als n Schritte erfordern. Nach der Induktionshypothese erweist sich jedes Symbol von α als erzeugend. Der Induktionsschritt des Algorithmus erlaubt es uns, unter Verwendung der Produktion $X \to \alpha$ zu folgern, dass X ein erzeugendes Symbol ist.
Nun wollen wir den induktiven Algorithmus betrachten, mit dem wir für die Grammatik $G = (V, T, P, S)$ die Menge der erreichbaren Symbole ermitteln. Wir können auch hier wieder zeigen, dass jedes Symbol, das wir mit diesem Algorithmus nicht finden, in der Tat nicht erreichbar ist.

Induktionsbeginn: S ist mit Sicherheit erreichbar.

Induktionsschritt: Angenommen, wir haben erkannt, dass eine Variable A erreichbar ist. Dann gilt für alle Produktionen mit dem Kopf A, dass auch alle im Rumpf dieser Produktionen enthaltenen Symbole erreichbar sind.

Beispiel 7.5 Wir beginnen wieder mit der Grammatik aus Beispiel 7.1. Nach Induktionsbeginn ist S erreichbar. Da S über die Produktionsrümpfe AB und a verfügt, schließen wir daraus, dass A, B und a erreichbar sind. B besitzt keine Produktionen, aber für A gibt es $A \to b$. Wir folgern daher, dass b erreichbar ist. Nun können keine weiteren Symbole der Menge der erreichbaren Symbole hinzugefügt werden, die lautet $\{S, A, B, a, b\}$. ■

Satz 7.6 Der oben beschriebene Algorithmus findet genau alle erreichbaren Symbole von G.

Beweis: Dieser Beweis besteht aus einem weiteren Paar einfacher Induktionen ähnlich dem Beweis von Satz 7.4. Wir überlassen diesen Beweis dem Leser als Übung. ■

7.1.3 ε-Produktionen eliminieren

Wir werden nun zeigen, dass ε-Produktionen zwar in vielen Problemen des Grammatikentwurfs hilfreich, aber nicht unabdingbar sind. Natürlich ist es ohne eine Produktion mit einem ε-Rumpf unmöglich, die leere Zeichenreihe als Element einer Sprache zu erzeugen. Daher zeigen wir eigentlich Folgendes: Wenn eine Sprache eine kontextfreie Grammatik besitzt, dann hat L – $\{\varepsilon\}$ eine kfG ohne ε-Produktionen. Falls ε nicht in L enthalten ist, dann ist $L = L - \{\varepsilon\}$ und besitzt eine kfG ohne ε-Produktionen.

Wir verfolgen die Strategie, zuerst zu ermitteln, welche Variablen eliminierbar sind. Eine Variable A ist *eliminierbar*, wenn $A \stackrel{+}{\Rightarrow} \varepsilon$. Wenn A eliminierbar ist, dann leitet (möglicherweise) jedes Vorkommen von A in einem Produktionsrumpf, wie z. B. in $B \to CAD$, ε ab. Wir fertigen zwei Versionen dieser Produktion an: eine ohne A im Produktionsrumpf ($B \to CD$), die dem Fall entspricht, in dem A zur Ableitung von ε dient, und eine andere Version mit A im Produktionsrumpf ($B \to CAD$). Wenn wir die Version mit A im Produktionsrumpf verwenden, dann können wir allerdings nicht zulassen, dass A ε ableitet. Dies stellt kein Problem dar, da wir einfach alle Produktionen, deren Rumpf ε ist, eliminieren und damit verhindern, dass irgendeine Variable ε ableitet.

Sei $G = (V, T, P, S)$ eine kontextfreie Grammatik. Wir können mithilfe des folgenden iterativen Algorithmus alle eliminierbaren Symbole von G finden. Anschließend werden wir zeigen, dass es keine eliminierbaren Symbole gibt, die nicht von diesem Algorithmus erkannt werden.

Induktionsbeginn: Wenn $A \to \varepsilon$ eine Produktion von G ist, dann ist A eliminierbar.

Induktionsschritt: Wenn es eine Produktion $B \to C_1C_2 \ldots C_k$ gibt, wobei jedes C_i eliminierbar ist, dann ist B eliminierbar. Beachten Sie, dass jedes C_i eine Variable sein muss, weil terminale Zeichen nicht eliminierbar sind. Daher müssen wir nur Produktionen betrachten, deren Rumpf ausschließlich aus Variablen besteht.ε

Satz 7.7 In jeder Grammatik G sind nur die Symbole eliminierbar, die von dem oben beschriebenen Algorithmus gefunden werden.ε

Beweis: Der »Wenn-Teil« der implizierten Aussage »A ist genau dann eliminierbar, wenn der Algorithmus A als eliminierbar identifiziert« wird bewiesen, indem wir durch eine einfache Induktion über die Reihenfolge, in der diese Symbole erkannt werden, zeigen, dass jedes dieser Symbole tatsächlich ε ableitet. Für den »Nur-wenn-

Teil« können wir einen Induktionsbeweis über die Länge der kürzesten Ableitung $A \overset{*}{\Rightarrow} \varepsilon$ führen.

INDUKTIONSBEGINN: Ein Schritt. Dann muss $A \to \varepsilon$ eine Produktion sein, und A wird im Induktionsbeginnteil des Algorithmus erkannt.

INDUKTIONSSCHRITT: Angenommen, $A \overset{*}{\Rightarrow} \varepsilon$ in n Schritten, wobei $n > 1$. Der erste Schritt muss die Form $A \to C_1C_2 \ldots C_k \overset{*}{\Rightarrow} \varepsilon$ haben, wobei jedes C_i in einer Folge von weniger als n Schritten ε ableitet. Nach der Induktionshypothese wird jedes C_i von dem Algorithmus als eliminierbar erkannt. Folglich wird nach dem Induktionsschritt A dank der Produktion $A \to C_1C_2 \ldots C_k$ als eliminierbar erkannt. ■

Wir beschreiben nun die Konstruktion einer Grammatik ohne ε-Produktionen. Sei $G = (V, T, P, S)$ eine kontextfreie Grammatik. Zunächst werden alle eliminierbaren Symbole von G ermittelt. Wir konstruieren eine neue Grammatik $G_1 = (V_1, T_1, P_1, S)$, deren Menge von Produktionen P_1 wie folgt bestimmt wird.ε

Für jede Produktion $A \to X_1X_2 \ldots X_k$ von P, wobei $k \geq 1$, nehmen wir an, dass m der k X_i eliminierbare Symbole sind. Die neue Grammatik G_1 wird über 2^m Versionen dieser Produktion verfügen, wobei die eliminierbaren X_i in allen möglichen Kombinationen vorhanden sind bzw. fehlen. Es gibt eine Ausnahme: Wenn $m = k$, d. h. wenn alle Symbole eliminierbar sind, dann wird der Fall nicht berücksichtigt, in dem alle X_i fehlen. Beachten Sie zudem, dass eine möglicherweise in P vorhandene Produktion der Form $A \to \varepsilon$ nicht in P_1 übernommen wird.

Beispiel 7.8 Betrachten Sie die Grammatik

$$S \to AB$$
$$A \to aAA \mid \varepsilon$$
$$B \to bBB \mid \varepsilon$$

Zuerst ermitteln wir die eliminierbaren Symbole. A und B sind sofort eliminierbar, weil sie über Produktionen mit ε als Produktionsrumpf verfügen. Wir stellen dann fest, dass S eliminierbar ist, weil der Rumpf der Produktion $S \to AB$ ausschließlich aus eliminierbaren Symbolen besteht. Daher sind alle drei Variablen eliminierbar.ε

Konstruieren wir nun die Produktionen der Grammatik G_1. Als Erstes betrachten wir $S \to AB$. Alle Symbole im Produktionsrumpf sind eliminierbar, und daher können wir vier Versionen formulieren, in denen A und B unabhängig voneinander vorhanden sind oder fehlen. Wir dürfen allerdings nicht alle Symbole weglassen, und daher können wir nur drei der vier Versionen der Produktion angeben:

$$S \to AB \mid A \mid B$$

Als Nächstes betrachten wir die Produktion $A \to aAA$. An der zweiten und dritten Position befinden sich eliminierbare Symbole, und somit sind wieder vier Versionen dieser Produktion mit gegebenem bzw. fehlendem A möglich. In diesem Fall sind alle vier Versionen zulässig, da das nicht eliminierbare Symbol a in jedem Fall vorhanden ist. Damit erhalten wir die vier Produktionen:ε

$$A \to aAA \mid aA \mid aA \mid a$$

Beachten Sie, dass die beiden mittleren Produktionen übereinstimmen, da es gleichgültig ist, welches A eliminiert wird, wenn wir beschließen, eines der beiden A zu eliminieren. Die endgültige Grammatik G_1 besitzt daher nur drei Produktionen für A.

Analog ergibt die Produktion B für G_1:

$$B \to bBB \mid bB \mid b$$

Die beiden ε-Produktionen von G werden nicht in G_1 übernommen. Folglich setzt sich die Grammatik G_1 aus folgenden Produktionen zusammen:ε

$$S \to AB \mid A \mid B$$
$$A \to aAA \mid aA \mid a$$
$$B \to bBB \mid bB \mid b$$ ■

Wir beschließen unser Studium der Eliminierung von ε-Produktionen mit dem Beweis, dass die oben beschriebene Konstruktion die Sprache nur insoweit verändert, als dass ε entfernt wird, wenn es in der Sprache von G enthalten ist. Da die Konstruktion offensichtlich ε-Produktionen eliminiert, erhalten wir damit den vollständigen Beweis für die Behauptung, dass es für jede kfG G eine Grammatik G_1 ohne ε-Produktionen gibt, derart dass

$$L(G_1) = L(G) - \{\varepsilon\}$$

Satz 7.9 Wenn die Grammatik G_1 mit der oben beschriebenen Konstruktion zur Eliminierung von ε-Produktionen aus G gebildet wird, dann gilt $L(G_1) = L(G) - \{\varepsilon\}$.

Beweis: Wir müssen zeigen, dass w mit $w \neq \varepsilon$ genau dann in $L(G_1)$ enthalten ist, wenn w in $L(G)$ enthalten ist. Wie so oft ist es einfacher, eine allgemeinere Aussage zu beweisen. In diesem Fall müssen wir die terminalen Zeichenreihen betrachten, die von jeder Variablen erzeugt werden, obwohl hier eigentlich nur das von Belang ist, was das Startsymbol S generiert. Somit werden wir Folgendes beweisen:

■ $A \overset{*}{\underset{G_1}{\Rightarrow}} w$ genau dann, wenn $A \overset{*}{\underset{G_1}{\Rightarrow}} w$ und $w \neq \varepsilon$.

Der Beweis ist in jedem Fall eine Induktion über die Länge der Ableitung.

(Nur-wenn-Teil) Angenommen, $A \overset{*}{\underset{G_1}{\Rightarrow}} w$ sei wahr. Dann gilt mit Sicherheit $w \neq \varepsilon$, weil G_1 keine ε-Produktionen enthält. Wir müssen durch Induktion über die Länge der Ableitung zeigen, dass $A \overset{*}{\underset{G}{\Rightarrow}} w$.

Induktionsbeginn: Ein Schritt. In diesem Fall muss G_1 eine Produktion $A \to w$ enthalten. Aus der Konstruktion von G_1 geht hervor, dass G eine Produktion $A \to \alpha$ enthält, derart dass die terminale Teilzeichenreihe von α gleich w ist und die in α enthaltenen Variablen eliminierbar sind. Dann gibt es in G die Ableitung $A \overset{*}{\Rightarrow} \alpha \overset{*}{\underset{G}{\Rightarrow}} w$, wobei im zweiten und in nachfolgenden Schritten, sofern vorhanden, von den in α enthaltenen Variablen ε abgeleitet wird.

Induktionsschritt: Angenommen, die Ableitung erfordert $n > 1$ Schritte. Dann sieht die Ableitung wie folgt aus: $A \overset{*}{\underset{G_1}{\Rightarrow}} X_1X_2 \ldots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$. Die erste Produktion ergibt sich aus einer Produktion $A \to_G Y_1Y_2 \ldots Y_m$, wobei $X_1X_2 \ldots X_k$ eine Teilzeichenreihe von $Y_1Y_2 \ldots Y_m$ ist und die Y_i der restlichen Teilzeichenreihe eliminierbare Variablen sind. Wir können w zudem aufteilen in $w_1w_2 \ldots w_k$, wobei $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ für $i = 1, 2, \ldots, k$. Wenn X_i ein terminales Zeichen ist, dann gilt $w_i = X_i$, und wenn X_i eine Variable ist, dann erfordert die

Ableitung $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ weniger als n Schritte. Nach der Induktionshypothese können wir auf $X_i \overset{*}{\underset{G_1}{\Rightarrow}} w_i$ schließen.

Wir konstruieren nun wie folgt eine entsprechende Ableitung in G:

$$A \underset{G}{\Rightarrow} Y_1 Y_2 \cdots Y_m \overset{*}{\underset{G}{\Rightarrow}} X_1 X_2 \cdots X_k \overset{*}{\underset{G}{\Rightarrow}} w_1 w_2 \cdots w_k = w$$

Der erste Schritt besteht in der Anwendung der Produktion $A \to Y_1 Y_2 \ldots Y_m$, von der wir wissen, dass sie in G existiert. Die nächste Gruppe von Schritten repräsentiert die Ableitung von ε von den einzelnen Y_j, die keinem X_i entsprechen. Die letzte Gruppe von Schritten besteht in der Ableitung der w_i von den X_i, die nach der Induktionshypothese vorhanden sind.ε

(Wenn-Teil) Angenommen, $A \overset{*}{\underset{G}{\Rightarrow}} w$ und $w \neq \varepsilon$. Wir zeigen durch Induktion über die Länge n der Ableitung, dass $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

Induktionsbeginn: Ein Schritt: In diesem Fall ist $A \to w$ eine Produktion von G. Da $w \neq \varepsilon$, ist diese Produktion ebenfalls eine Produktion von G_1, und somit gilt $A \overset{*}{\underset{G_1}{\Rightarrow}} w$.

Induktionsschritt: Angenommen, die Ableitung erfordert $n > 1$ Schritte. Dann sieht die Ableitung wie folgt aus: $A \underset{G}{\Rightarrow} Y_1 Y_2 \ldots Y_m \overset{*}{\underset{G}{\Rightarrow}} w$. Wir können w aufteilen in $w_1 w_2 \ldots w_m$, sodass $Y_i \overset{*}{\underset{G}{\Rightarrow}} w_i$ für $i = 1, 2, \ldots, m$. Seien $X_1 X_2 \ldots X_k$ der Reihe nach jene Y_j, derart dass $w_j \neq \varepsilon$. Es muss $k \geq 1$ sein, da $w \neq \varepsilon$. Folglich ist $A \to X_1 X_2 \ldots X_k$ eine Produktion von G_1.

Wir behaupten, dass $X_1 X_2 \ldots X_k \overset{*}{\underset{G}{\Rightarrow}} w$, da nur die Y_j nicht durch X_i repräsentiert werden, die eliminierbar sind und daher nichts zur Ableitung von w beitragen. Da jede der Ableitungen $Y_j \overset{*}{\underset{G}{\Rightarrow}} w_j$ weniger als n Schritte erfordert, können wir die Induktionshypothese anwenden und schließen, dass $Y_j \overset{*}{\underset{G_1}{\Rightarrow}} w_j$, wenn $w_j \neq \varepsilon$.

Folglich gilt $A \underset{G_1}{\Rightarrow} X_1 X_2 \ldots X_k \overset{*}{\underset{G_1}{\Rightarrow}} w$.

Wir vervollständigen den Beweis nun folgendermaßen. Wir wissen, dass w genau dann in $L(G_1)$ enthalten ist, wenn $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. Wenn wir im obigen Beweis $A = S$ setzen, dann wissen wir, dass w genau dann in $L(G_1)$ enthalten ist, wenn $S \overset{*}{\underset{G_1}{\Rightarrow}} w$ und $w \neq \varepsilon$. Das heißt, w ist genau dann in $L(G_1)$ enthalten, wenn w in $L(G)$ enthalten ist und $w \neq \varepsilon$. ■

7.1.4 Einheitsproduktionen eliminieren

Als Einheitsproduktion bezeichnen wir eine Produktion der Form $A \to B$, wobei A und B Variablen sind. Diese Produktionen können nützlich sein. So zeigt Beispiel 5.27 beispielsweise, wie wir mithilfe der Produktionen $E \to T$ und $T \to F$ eine eindeutige Grammatik für einfache Ausdrücke erstellen können:

$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$F \to I \mid (E)$$
$$T \to F \mid T * F$$
$$E \to T \mid E + T$$

Einheitsproduktionen können bestimmte Beweise jedoch komplizieren, und sie erfordern überdies zusätzliche Ableitungsschritte, die aus technischer Sicht nicht notwendig sind. Beispielsweise können wir die Produktion $E \to T$ durch die beiden Produktionen $E \to F \mid T * F$ ersetzen. Mit diesen Änderungen werden die Einheitsproduktionen nicht eliminiert, da wir die Einheitsproduktion $E \to F$ einge-

führt haben, die zuvor nicht Teil der Grammatik war. Durch die Erweiterung von $E \to F$ durch die beiden Produktionen für F erhalten wir $E \to I \mid (E) \mid T * F$. Wir haben noch immer eine Einheitsproduktion, nämlich $E \to I$. Wenn wir die Variable I jedoch auf alle sechs möglichen Arten erweitern, erhalten wir

$$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \mid (E) \mid T * F$$

Jetzt ist die Einheitsproduktion für E eliminiert. Beachten Sie, dass $E \to a$ *keine* Einheitsproduktion ist, da das Symbol im Produktionsrumpf ein terminales Zeichen und keine Variable ist. Gemäß der Definition von Einheitsproduktionen muss der Rumpf eine Variable sein.

Die oben beschriebene Technik – Einheitsproduktionen so lange erweitern, bis sie verschwunden sind – funktioniert häufig. Sie kann allerdings scheitern, wenn zirkuläre Einheitsproduktionen gegeben sind, wie $A \to B$, $B \to C$ und $C \to A$. Eine Technik, die mit Sicherheit stets funktioniert, besteht darin, unter Verwendung einer Folge von Einheitsproduktionen zuerst all die Paare von Variablen A und B zu suchen, derart dass $A \overset{*}{\Rightarrow} B$. Beachten Sie, dass die Ableitung $A \overset{*}{\Rightarrow} B$ auch dann wahr sein kann, wenn darin keine Einheitsproduktion auftritt. Beispielsweise können die Produktionen $A \to BC$ und $C \to \varepsilon$ vorliegen.

Sobald wir diese Paare ermittelt haben, können wir jede Folge von Ableitungsschritten, in denen $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \ldots \Rightarrow B_n \Rightarrow \alpha$, durch eine Produktion ersetzen, in der die Produktion $B_n \to \alpha$, die keine Einheitsproduktion darstellt, direkt auf A angewandt wird, d. h. $A \to \alpha$. Nachfolgend wird zuerst die induktive Konstruktion der Paare (A, B) beschrieben, derart dass $A \overset{*}{\Rightarrow} B$ ausschließlich Einheitsproduktionen verwendet. Ein solches Paar wird hier als Einheitspaar (engl. *unit pair*) bezeichnet.

INDUKTIONSBEGINN: (A, A) ist ein Einheitspaar für die Variable A, das heißt, $A \overset{*}{\Rightarrow} A$ in null Schritten.

INDUKTIONSSCHRITT: Angenommen, wir haben ermittelt, dass (A, B) ein Einheitspaar ist, und $B \to C$ ist eine Produktion, wobei C eine Variable ist. Dann ist (A, C) ein Einheitspaar.

Beispiel 7.10 Betrachten Sie die Grammatik für einfache Ausdrücke aus Beispiel 5.27, die wir weiter oben nochmals angegeben haben. Nach Induktionsbeginn erhalten wir die Einheitspaare (E, E), (T, T), (F, F) und (I, I). Für den Induktionsschritt können wir auf Folgendes schließen:

1. (E, E) und die Produktion $E \to T$ ergeben das Einheitspaar (E, T).
2. (E, T) und die Produktion $T \to F$ ergeben das Einheitspaar (E, F).
3. (E, F) und die Produktion $F \to I$ ergeben das Einheitspaar (E, I).
4. (T, T) und die Produktion $T \to F$ ergeben das Einheitspaar (T, F).
5. (T, F) und die Produktion $F \to I$ ergeben das Einheitspaar (T, I).
6. (F, F) und die Produktion $F \to I$ ergeben das Einheitspaar (F, I).

Wir können auf keine weiteren Paare schließen, und in der Tat repräsentieren diese zehn Paare alle Ableitungen, in denen ausschließlich Einheitsproduktionen verwendet werden. ■

Das Entwicklungsmuster sollte Ihnen nun bekannt vorkommen. Es gibt einen einfachen Beweis dafür, dass der von uns vorgeschlagene Algorithmus alle gewünschten Paare ermittelt. Wir setzen dann unsere Kenntnis dieser Paare ein, um Einheitsproduktionen aus der Grammatik zu entfernen und zu zeigen, dass die beiden Grammatiken dieselbe Sprache beschreiben.

Satz 7.11 Der oben beschriebene Algorithmus findet genau alle Einheitspaare einer kfG G.

Beweis: In der einen Richtung besteht der Beweis aus einer einfachen Induktion über die Reihenfolge, in der die Paare erkannt werden, die zeigt, dass gilt: Wenn (A, B) als Einheitspaar erkannt wird, dann gibt es eine Ableitung $A \overset{*}{\underset{G}{\Rightarrow}} B$, in der nur Einheitsproduktionen verwendet werden. Wir überlassen Ihnen diesen Teil des Beweises.

In der anderen Richtung nehmen wir an, dass es eine Ableitung $A \overset{*}{\underset{G}{\Rightarrow}} B$ gibt, in der nur Einheitsproduktionen verwendet werden. Wir können durch Induktion über die Länge der Ableitung zeigen, dass das Paar (A, B) gefunden wird.

Induktionsbeginn: Null Schritte. In diesem Fall ist $A = B$, und das Paar (A, B) wird im Induktionsbeginnschritt hinzugefügt.

Induktionsschritt: Angenommen, $A \overset{*}{\Rightarrow} B$ in n Schritten, wobei $n > 0$ und in jedem Schritt eine Einheitsproduktion angewandt wird. Dann sieht die Ableitung wie folgt aus:

$$A \overset{*}{\Rightarrow} C \rightarrow B$$

Die Ableitung $A \overset{*}{\Rightarrow} C$ erfordert $n - 1$ Schritte, und somit erkennen wir nach der Induktionshypothese das Paar (A, C). Im Induktionsschritt des Algorithmus wird dann das Paar (A, C) mit der Produktion $C \rightarrow B$ kombiniert, sodass auf das Paar (A, B) geschlossen wird. ■

Zur Eliminierung von Einheitsproduktionen gehen wir folgendermaßen vor. Aus einer gegebenen kfG $G = (V, T, P, S)$ konstruieren wir die kfG $G_1 = (V_1, T_1, P_1, S)$:

1. Wir suchen alle Einheitspaare von G.
2. Für jedes Einheitspaar (A, B) fügen wir P_1 alle Produktionen $A \rightarrow \alpha$ hinzu, wobei $B \rightarrow \alpha$ eine Produktion aus P ist, die keine Einheitsproduktion darstellt. Beachten Sie, dass $A = B$ möglich ist. P_1 enthält damit auch alle Produktionen aus P, die keine Einheitsproduktionen sind.

Beispiel 7.12 Lassen Sie uns mit Beispiel 7.10 fortfahren, in dem Schritt (1) der oben beschriebenen Konstruktion für die Grammatik aus Beispiel 5.27 ausgeführt wurde. Tabelle 7.1 fasst Schritt (2) des Algorithmus zusammen, in welchem wir die neue Menge von Produktionen erstellen, indem wir das erste Element eines Paars als Kopf und alle Rümpfe von Produktionen des zweiten Elements, die keine Einheitsproduktionen sind (die wir im Folgenden »gewöhnliche Produktionen« nennen werden), als Produktionsrümpfe einsetzen.

Die resultierende Grammatik sieht folgendermaßen aus:

$$E \to E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$T \to T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$F \to (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$ ■

Tabelle 7.1: Grammatik, die im Schritt (2) des Algorithmus zur Eliminierung von Einheitsproduktionen konstruiert wird

Paar	Produktionen
(E, E)	$E \to E + T$
(E, T)	$E \to T * F$
(E, F)	$E \to (E)$
(E, I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \to T * F$
(T, F)	$T \to (E)$
(T, I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \to (E)$
(F, I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Satz 7.13 Wenn die Grammatik G_1 nach dem oben beschriebenen Algorithmus zur Eliminierung von Einheitsproduktionen aus G konstruiert wird, dann gilt $L(G_1) = L(G)$.

Beweis: Wir zeigen, dass w genau dann in $L(G)$ enthalten ist, wenn w in $L(G_1)$ enthalten ist.

(Wenn-Teil) Angenommen, $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. Da jede Produktion von G_1 äquivalent ist zu einer Folge von null oder mehr Einheitsproduktionen in G, denen eine gewöhnliche Produktion von G folgt, wissen wir, dass $\alpha \overset{*}{\underset{G_1}{\Rightarrow}} \beta$ die Ableitung $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ impliziert. Das heißt, jeder Schritt einer Ableitung in G_1 kann durch einen oder mehrere Ableitungsschritte in G ersetzt werden. Wenn wir diese Folge von Schritten zusammenfassen, schließen wir, dass $S \overset{*}{\underset{G}{\Rightarrow}} w$.

(Nur-wenn-Teil) Nehmen wir nun an, dass w in $L(G)$ enthalten ist. Nach der in Abschnitt 5.2 beschriebenen Äquivalenz wissen wir, dass w eine linksseitige Ableitung besitzt, d. h. $S \underset{lm}{\Rightarrow} w$. Wenn eine Einheitsproduktion in einer linksseitigen Ableitung verwendet wird, dann stellt die Variable im Produktionsrumpf die äußerste linke Variable dar und wird sofort ersetzt. Folglich kann die linksseitige Ableitung in der Grammatik G in eine Folge von Schritten aufgeteilt werden, in denen jeweils eine gewöhnliche Produktion auf null oder mehr Einheitsproduktionen folgt. Beachten Sie, dass jede gewöhnliche Produktion, der keine Einheitsproduktion vorangeht, für sich genommen einen »Schritt« darstellt. Jeder dieser Schritte kann durch eine Produktion von G_1 ausgeführt werden, weil die Konstruktion von G_1 genau solche Produktionen

erstellt hat, die null oder mehr Einheitsproduktionen gefolgt von einer gewöhnlichen Produktion widerspiegeln. Folglich gilt: $S \overset{*}{\underset{G_1}{\Rightarrow}} w$. ■

Wir können nun die verschiedenen bislang beschriebenen Vereinfachungen zusammenfassen. Wir möchten eine beliebige kfG G in eine äquivalente kfG umwandeln, die keine unnützen Symbole, keine ε-Produktionen und keine Einheitsproduktionen besitzt. In der Konstruktion muss die Reihenfolge der Schritte mit Bedacht gewählt werden. Sicher ist folgende Reihenfolge:

1. Eliminierung der ε-Produktionen
2. Eliminierung der Einheitsproduktionen
3. Eliminierung der unnützen Symbole

Sie sollten beachten, dass wir ebenso wie in Abschnitt 7.1.1, wo wir die beiden Schritte in der richtigen Reihenfolge ausführen mussten, da das Ergebnis sonst unnütze Symbole hätte aufweisen können, die drei oben genannten Schritte in der richtigen Reihenfolge ausführen müssen, da sonst möglicherweise nicht alle Eliminationen korrekt erfolgen.

Satz 7.14 Wenn G eine kfG ist, die eine Sprache erzeugt, die mindestens eine von ε abweichende Zeichenreihe enthält, dann gibt es eine andere kfG G_1, derart dass $L(G_1) = L(G) - \{\varepsilon\}$, und G_1 besitzt keine ε-Produktionen, keine Einheitsproduktionen und keine unnützen Symbole.

Beweis: Wir beginnen mit der Eliminierung von ε-Produktionen nach der in Abschnitt 7.1.3 beschriebenen Methode. Wenn wir dann die Einheitsproduktionen nach der in Abschnitt 7.1.4 beschriebenen Methode eliminieren, führen wir keine neuen ε-Produktionen ein, da jeder Rumpf einer neuen Produktion mit einem Rumpf einer alten Produktion identisch ist. Schließlich eliminieren wir unnütze Symbole nach der in Abschnitt 7.1.1 beschriebenen Methode. Da mit dieser Transformation nur Produktionen und Symbole entfernt und keine neue Produktionen eingeführt werden, wird die resultierende Grammatik weder ε-Produktionen noch Einheitsproduktionen aufweisen.

7.1.5 Chomsky-Normalform

Wir beenden unser Studium der Grammatikvereinfachungen, indem wir zeigen, dass jede nichtleere kfG ohne ε eine Grammatik G besitzt, in der alle Produktionen in einer von zwei einfachen Formen vorliegen:

1. $A \to BC$, wobei A, B und C jeweils Variablen sind, oder
2. $A \to a$, wobei A eine Variable und a ein terminales Symbol ist.

Zudem enthält G keine unnützen Symbole. Eine solche Grammatik wird als Grammatik in *Chomsky-Normalform* oder *CNF* bezeichnet[1].

1. N. Chomsky ist der Linguist, der als Erster kontextfreie Grammatiken als Mittel zur Beschreibung natürlicher Sprachen vorschlug und bewies, dass jede kfG in diese Form gebracht werden kann. Interessanterweise scheint die CNF in Anwendungen der natürlichsprachlichen Linguistik keine bedeutende Rolle zu spielen. Dagegen werden wir verschiedene andere Anwendungen vorstellen, wie z. B. einen effizienten Test zur Feststellung der Zugehörigkeit einer Zeichenreihe zu einer kontextfreien Sprache (Abschnitt 7.4.4).

Um eine Grammatik in CNF zu bringen, beginnen wir mit einer Grammatik, die die Beschränkungen von Satz 7.14 erfüllt; d. h. die Grammatik besitzt weder ε-Produktionen noch Einheitsproduktionen noch unnütze Symbole. Jede Produktion einer solchen Grammatik hat entweder die Form $A \to a$, die bereits in der CNF zulässig ist, oder einen Rumpf mit einer Länge von 2 oder mehr. Unsere Aufgabe besteht nun darin,

a) dafür zu sorgen, dass alle Produktionsrümpfe mit der Länge zwei oder mehr nur Variablen enthalten.

b) Produktionsrümpfe mit einer Länge von 3 oder mehr in eine aufeinander aufbauende Folge von Produktionen aufzuschlüsseln, die jeweils einen Rumpf aus zwei Variablen besitzen.

Die Konstruktion für Aufgabe (a) sieht folgendermaßen aus. Für jedes terminale Symbole a, das in einem Produktionsrumpf der Länge 2 oder mehr vorkommt, erstellen wir eine neue Variable A mit der einzigen Produktion $A \to a$. Wir ersetzen alle Vorkommen von a in jedem Produktionsrumpf der Länge 2 oder mehr durch A. Jetzt verfügt jede Produktion über einen Rumpf, der entweder aus einem einzigen terminalen Symbol oder mindestens zwei Variablen und keinen terminalen Symbolen besteht.

Für Schritt (b) müssen wir Produktionen der Form $A \to B_1B_2 \ldots B_k$ mit $k \geq 3$ in Gruppen von Produktionen aufteilen, deren Rumpf jeweils zwei Variablen umfasst. Wir führen $k-2$ neue Variablen C_1, C_2, ..., C_{k-2} ein. Die ursprüngliche Produktion wird ersetzt durch $k-1$ Produktionen der Form

$$A \to B_1C_1,\ C_1 \to B_2C_2 \ldots C_{k-3} \to B_{k-2}C_{k-2},\ C_{k-2} \to B_{k-1}B_k$$

Beispiel 7.15 Wir wollen die Grammatik aus Beispiel 7.12 in CNF bringen. Für Aufgabe (a) ist festzustellen, dass die Grammatik acht terminale Symbole – a, b, 0, 1, +, *, (,) – enthält, die jeweils in einem Produktionsrumpf vorkommen, der nicht nur aus einem einzelnen terminalen Symbol besteht. Wir müssen für diese terminalen Symbole daher acht neue Variablen und acht Produktionen einführen, in denen die jeweils neue Variable durch das entsprechende terminale Symbol ersetzt wird. Wir verwenden die Anfangsbuchstaben der englischen Namen der Symbole als Variablen und erhalten damit:

$$A \to a\ B \to b\ Z \to 0\ O \to 1$$
$$P \to +\ M \to *\ L \to (\ R \to)$$

Wenn wir diese Produktionen einführen und alle terminalen Symbole in den Produktionsrümpfen, die nicht nur aus einem terminalen Symbol bestehen, durch die entsprechende Variable ersetzen, erhalten wir die in Tabelle 7.2 dargestellte Grammatik.

Abgesehen von den Produktionsrümpfen der Länge 3 (*EPT*, *TMF* und *LER*), liegen nun alle Produktionen in Chomsky-Normalform vor. Einige dieser Produktionsrümpfe kommen in mehreren Produktionen vor. Wir können durch die Einführung nur einer neuen Variablen in einem Schritt auf diese Mehrfachvorkommen eingehen. Wir führen für *EPT* die neue Variable C_1 ein und ersetzen die Produktion $E \to EPT$, in der *EPT* das einzige Mal vorkommt, durch die Produktionen $E \to EC_1$ und $C_1 \to PT$.

Für *TMF* führen wir die neue Variable C_2 ein. Die beiden Produktionen $E \to TMF$ und $T \to TMF$, in denen dieser Produktionsrumpf verwendet wird, werden durch $E \to TC_2$, $T \to TC_2$ und $C_2 \to MF$ ersetzt. Dann führen wir für *LER* die neue Variable C_3 ein und ersetzen die drei Produktionen $E \to LER$, $T \to LER$ und F $\to LER$ durch $E \to LC_3$,

Tabelle 7.2: Alle Produktionsrümpfe wurden so umgeformt, dass sie entweder aus einem einzigen terminalen Symbol oder aus mehreren Variablen bestehen

E	$\rightarrow$	$EPT \mid TMF \mid LER \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
T	$\rightarrow$	$TMF \mid LER \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
F	$\rightarrow$	$LER \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
I	$\rightarrow$	$a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
A	$\rightarrow$	a
B	$\rightarrow$	b
Z	$\rightarrow$	0
O	$\rightarrow$	1
P	$\rightarrow$	+
M	$\rightarrow$	*
L	$\rightarrow$	(
R	$\rightarrow$	)

$T \rightarrow C_3$, $F \rightarrow LC_3$ und $C_3 \rightarrow ER$. Die sich daraus ergebende Grammatik, die in CNF vorliegt, ist in Tabelle 7.3 dargestellt. ■

Tabelle 7.3: Grammatik, in der alle Rümpfe aus einem terminalen Symbol oder zwei Variablen bestehen

E	$\rightarrow$	$EC_1 \mid TC_2 \mid LC_3 \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
T	$\rightarrow$	$TC_2 \mid LC_3 \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
F	$\rightarrow$	$LC_3 \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
I	$\rightarrow$	$a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
A	$\rightarrow$	a
B	$\rightarrow$	b
Z	$\rightarrow$	0
O	$\rightarrow$	1
P	$\rightarrow$	+
M	$\rightarrow$	*
L	$\rightarrow$	(
R	$\rightarrow$	)
C_1	$\rightarrow$	PT
C_2	$\rightarrow$	MF
C_3	$\rightarrow$	ER

Satz 7.16 Wenn G eine kfG ist, deren Sprache mindestens eine von ε abweichende Zeichenreihe enthält, dann gibt es eine Grammatik G_1 in Chomsky-Normalform, derart dass $L(G_1) = L(G) - \{\varepsilon\}$.

Beweis: Nach Satz 7.14 können wir eine kfG G_2 finden, derart dass $L(G_2) = L(G) - \{\varepsilon\}$ und dass G_2 weder unnütze Symbole noch ε-Produktionen noch Einheitsproduktionen besitzt. Die Konstruktion, mit der G_2 in die CNF-Grammatik G_1 umgeformt wird, ändert die Produktionen so, dass jede Produktion von G_1 durch eine oder mehrere Produktionen von G_2 simuliert werden kann. Umgekehrt haben die in G_2 eingeführten Variablen jeweils nur eine Produktion, sodass sie nur in der vorgesehenen Weise verwendet werden können. Formal ausgedrückt: Wir beweisen, dass w genau dann in $L(G_2)$ enthalten ist, wenn w in $L(G_1)$ enthalten ist.

(Nur-wenn-Teil) Wenn w eine Ableitung in G_2 hat, dann lässt sich jede der verwendeten Produktionen, die wir $A \to X_1X_2 \ldots X_k$ nennen wollen, durch eine Folge von Produktionen von G_1 ersetzen. Das heißt, ein Schritt in der Ableitung in G_2 wird zu einem oder mehreren Schritten in der Ableitung von w unter Verwendung der Produktionen von G_1. Wenn ein X_i ein terminales Symbol ist, dann wissen wir, dass G_1 über eine entsprechende Variable B_i und eine Produktion $B_i \to X_i$ verfügt. Wenn $k > 2$ ist, dann besitzt G_1 eine Produktion der Form $A \to B_1C_1$, $C \to B_2C_2$ etc., wobei B_1 entweder für die für das terminale Symbol X_i eingeführte Variable oder für X_i selbst steht, falls X_i eine Variable ist. Diese Produktionen simulieren in G_1 einen Ableitungsschritt von G_2, in dem die Produktion $A \to X_1X_2 \ldots X_k$ verwendet wird. Wir schließen daraus, dass G_1 eine Ableitung von w enthält und dass w somit in $L(G_1)$ enthalten ist.

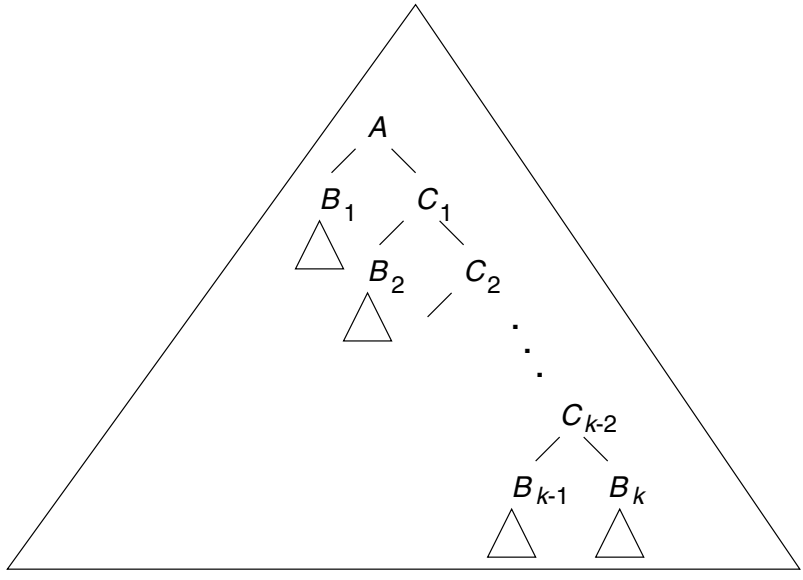
(Wenn-Teil) Angenommen, w ist in $L(G_1)$ enthalten. Dann gibt es einen Parsebaum in G_1 mit S an der Wurzel und dem Ergebnis w. Wir wandeln diesen Parsebaum in einen Parsebaum von G_2 um, der ebenso über die Wurzel S und das Ergebnis w verfügt.

Zuerst »revidieren« wir Teil (b) der CNF-Konstruktion. Das heißt, angenommen es gibt einen Knoten mit der Beschriftung A, der zwei untergeordnete Knoten mit der Bezeichnung B_1 und C_1 besitzt, wobei C_1 eine der Variablen ist, die in Teil (b) eingeführt wurden, dann muss dieser Teil des Parsebaums wie Abbildung 7.1 (a) aussehen. Das heißt, weil jede dieser neu eingeführten Variablen lediglich über eine Produktion verfügt, können sie nur in einer bestimmten Weise im Parsebaum erscheinen, und alle Variablen, die für die Produktion $A \to B_1B_2 \ldots B_k$ eingeführt wurden, müssen zusammen erscheinen, wie in der Abbildung gezeigt.

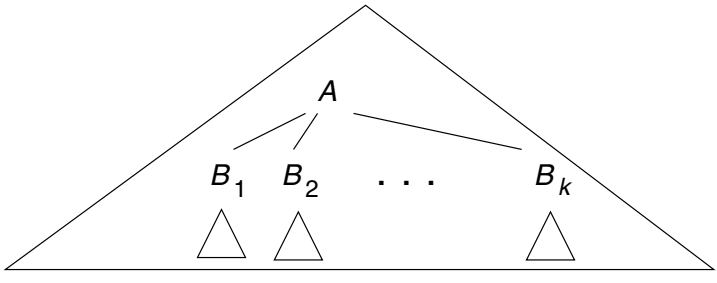
Eine jede dieser Gruppen von Knoten im Parsebaum kann durch die Produktion ersetzt werden, die sie repräsentiert. Die Umformung des Parsebaums wird in Abbildung 7.1. (b) dargestellt.

Der resultierende Parsebaum ist nicht notwendigerweise ein Parsebaum von G_2. Dies liegt darin begründet, dass in Schritt (a) der CNF-Konstruktion andere Variablen eingeführt wurden, die einzelne terminale Symbole ableiten. Wir können diese Variablen jedoch im aktuellen Parsebaum identifizieren und einen Knoten, der mit einer solchen Variablen A beschriftet ist, sowie dessen einzigen untergeordneten Knoten mit der Beschriftung a durch einen einzigen Knoten mit der Beschriftung a ersetzen. Nun stellt jeder innere Knoten des Parsebaums eine Produktion von G_2 dar. Da w das Ergebnis eines Parsebaums in G_2 ist, schließen wir daraus, dass w in $L(G_2)$ enthalten ist. ■

Abbildung 7.1: Ein Parsebaum in G_1 muss eingeführte Variablen in einer speziellen Weise verwenden



(a)



(b)

Greibach-Normalform

Es gibt eine weitere interessante Normalform für Grammatiken, die wir nicht beweisen werden[2]. Jede Sprache, die mindestens eine von ε abweichende Zeichenreihe umfasst, ist $L(G)$ für eine Grammatik G, deren Produktionen alle die Form $A \to a\alpha$ haben, wobei a für ein terminales Symbol und α für eine Zeichenreihe aus null oder mehr Variablen steht. Es ist kompliziert, eine Grammatik in diese Form zu bringen, auch wenn wir diese Aufgabe dadurch vereinfachen, dass wir beispielsweise mit einer Grammatik in Chomsky-Normalform beginnen. Grob gesagt, erweitern wir dann die erste Variable jedes nichtterminalen Produktionskörpers, bis wir ein terminales Symbol am Anfang des Produktionskörpers erhalten. Da es jedoch zirkuläre Produktionen geben kann, in welchem Fall wir diese Situation nie erreichen, ist es notwendig, das Verfahren völlig anders zu steuern. Die zu diesem Zweck eingefügten neuen Produktionen ändern zwar nichts an der erzeugten Sprache, sie ändern jedoch im Allgemeinen die Ableitungsstrukturen der Zeichenreihen in dieser Sprache, wodurch die Anwendbarkeit in der Praxis in gewissen Fällen eingeschränkt wird.

Diese Form, die nach Sheila Greibach, die als Erste ein Verfahren zur Konstruktion einer solchen Grammatik beschrieb, als Greibach-Normalform bezeichnet wird, hat verschiedene interessante Konsequenzen. Da mit jedem Einsatz einer Produktion genau ein terminales Symbol in eine Satzform eingeführt wird, besitzt eine Zeichenreihe der Länge n eine Ableitung mit genau n Schritten. Wenn wir die PDA-Konstruktion aus Satz 6.13 auf eine Grammatik in Greibach-Normalform anwenden, dann erhalten wir einen PDA ohne ε-Regeln, womit gezeigt wird, dass es stets möglich ist, ε-Übergänge eines PDA zu eliminieren.

7.1.6 Übungen zum Abschnitt 7.1

* **Übung 7.1.1** Finden Sie eine äquivalente Grammatik für

$$S \to AB \mid CA$$
$$A \to a$$
$$B \to BC \mid AB$$
$$C \to aB \mid b$$

die keine unnützen Symbole enthält.

* **Übung 7.1.2** Beginnen Sie mit folgender Grammatik:

$$S \to ASB \mid \varepsilon$$
$$A \to aAS \mid a$$
$$B \to SbS \mid A \mid bb$$

a) Enthält diese Grammatik unnütze Symbole? Falls ja, eliminieren Sie diese.

b) Eliminieren Sie ε-Produktionen.

c) Eliminieren Sie Einheitsproduktionen.

d) Bringen Sie die Grammatik in Chomsky-Normalform.

2. Siehe hierzu Übung 7.1.11.

Übung 7.1.3 Wiederholen Sie Übung 7.1.2 für die folgende Grammatik:

$$S \to 0A0 \mid 1B1 \mid BB$$
$$A \to C$$
$$B \to S \mid A$$
$$\mathrm{C} \to \mathrm{S} \mid \varepsilon$$

Übung 7.1.4 Wiederholen Sie Übung 7.1.2 für die folgende Grammatik:

$$S \to AAA \mid B$$
$$A \to aA \mid B$$
$$B \to \varepsilon$$

Übung 7.1.5 Wiederholen Sie Übung 7.1.2 für die folgende Grammatik:

$$S \to aAa \mid bBb \mid \varepsilon$$
$$A \to C \mid a$$
$$B \to C \mid b$$
$$C \to CDE \mid \varepsilon$$
$$\mathrm{D} \to A \mid B \mid ab$$

Übung 7.1.6 Entwerfen Sie eine CNF-Grammatik für die Menge der Zeichenreihen mit einer ausgewogenen Anzahl von linken und rechten Klammern. Sie müssen nicht mit einer Grammatik, die nicht in CNF vorliegt, beginnen.

!! **Übung 7.1.7** Angenommen, G sei eine kfG mit p Produktionen und kein Produktionsrumpf sei länger als n. Zeigen Sie, dass gilt: Wenn $\mathrm{A} \overset{*}{\underset{G}{\Rightarrow}} \varepsilon$, dann lässt sich ε aus A in nicht mehr als $(n^p - 1)/(n - 1)$ Schritten ableiten. Wie weit kann man sich dieser Obergrenze tatsächlich annähern?

! **Übung 7.1.8** Angenommen, es liegt eine Grammatik G mit n Produktionen vor, die keine ε-Produktionen enthalten, und wir wollen diese Grammatik in CNF transformieren.

a) Zeigen Sie, dass die CNF-Grammatik höchstens $O(n^2)$ Produktionen umfasst.

b) Zeigen Sie, dass diese CNF-Grammatik eine Anzahl von Produktionen besitzen kann, die zu n^2 proportional ist. *Hinweis*: Betrachten Sie die Konstruktion, mit der Einheitsproduktionen eliminiert werden.

Übung 7.1.9 Formulieren Sie die Induktionsbeweise, die erforderlich sind, um die Beweise der folgenden Sätze zu vervollständigen:

a) Den Teil von Satz 7.4, in dem wir zeigen, dass die erkannten Symbole in der Tat erzeugend sind

b) Beide Richtungen von Satz 7.6, in dem wir zeigen, dass der Algorithmus zur Erkennung erreichbarer Symbole aus Abschnitt 7.1.2 korrekt ist

c) Den Teil von Satz 7.11, in dem wir zeigen, dass es sich bei allen erkannten Paaren tatsächlich um Einheitspaare handelt

*! **Übung 7.1.10** Ist es möglich, für jede kontextfreie Sprache ohne ε eine Grammatik zu finden, deren Produktion entweder die Form $A \to BCD$ (d. h. der Rumpf besteht aus drei Variablen) oder $A \to a$ (d. h. der Rumpf besteht aus einem einzigen terminalen Symbol) haben? Beweisen Sie, dass dies möglich ist, oder führen Sie ein Gegenbeispiel an.

Übung 7.1.11 In dieser Übung werden wir zeigen, dass es für jede kontextfreie Sprache, die mindestens eine von ε abweichende Zeichenreihe enthält, eine kfG in Greibach-Normalform gibt, die $L - \{\varepsilon\}$ erzeugt. Rufen Sie sich in Erinnerung, dass bei einer Grammatik in Greibach-Normalform (GNF) jeder Produktionsrumpf mit einem terminalen Symbol beginnt. Zur Konstruktion wird eine Reihe von Lemmata und andere Konstruktionen eingesetzt.

a) Angenommen, eine kfG G enthält die Produktion $A \to \alpha B \beta$ und alle Produktionen für B lauten $B \to \gamma_1 \mid \gamma_2 \mid ... \mid \gamma_n$. Wenn wir $A \to \alpha B \beta$ durch alle Produktionen ersetzen, die wir erhalten, wenn wir für B einen Rumpf einer B-Produktion einsetzen, d. h. $A \to \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid ... \mid \alpha\gamma_n\beta$, dann erzeugt die resultierende Grammatik dieselbe Sprache wie G.

Gehen Sie im Folgenden davon aus, dass die Grammatik G für L in Chomsky-Normalform vorliegt und dass die Variablen mit $A_1, A_2, ..., A_k$ bezeichnet werden.

*! b) Zeigen Sie, dass wir durch wiederholtes Durchführen der Umformung aus Teil (a) G in eine äquivalente Grammatik umwandeln können, in der jeder Produktionsrumpf für A_i entweder mit einem terminalen Symbol oder mit A_j, wobei $j \geq i$, beginnt. In jedem Fall sind alle dem ersten nachfolgenden Symbole im Produktionsrumpf Variablen.

! c) Angenommen, G_1 sei die Grammatik, die wir erhalten, wenn wir Schritt (b) mit G ausführen. Angenommen, A_i sei eine beliebige Variable und $A_i \to A_i\alpha_1 \mid ... \mid A_i\alpha_m$ repräsentiere alle A_i-Produktionen, deren Rumpf mit A_i beginnt. Seien

$$A_i \to \beta_1 \mid ... \mid \beta_p$$

alle anderen A_i-Produktionen. Beachten Sie, dass jedes β_j entweder mit einem terminalen Symbol oder mit einer Variablen mit einem Index, der größer als j ist, beginnen muss. Wir führen eine neue Variable B_i ein und ersetzen die erste Gruppe von m Produktionen durch

$$A_i \to \beta_1 B_i \mid ... \mid \beta_p B_i$$
$$B_i \to \alpha_1 \mathrm{B}_i \mid \alpha_1 \mid \ldots \mid \alpha_m B_i \mid \alpha_m$$

Beweisen Sie, dass die resultierende Grammatik dieselbe Sprache wie G und G_1 erzeugt.

*! d) Sei G_2 die aus Schritt (c) resultierende Grammatik. Beachten Sie, dass alle A_i-Produktionen einen Rumpf haben, der entweder mit einem terminalen Symbol oder mit A_j, wobei $j > i$, beginnt. Zudem verfügen alle B_i-Produktionen über einen Rumpf, der mit einem terminalen Symbol oder mit einem A_j beginnt. Beweisen Sie, dass G_2 eine äquivalente Grammatik in GNF besitzt. *Hinweis*: Bearbeiten Sie unter Verwendung von Teil (a) zuerst die Produktionen für A_k, dann die für A_{k-1} etc. bis zu A_1. Bearbeiten Sie anschließend wieder unter Verwendung von Teil (a) die B_i-Produktionen in beliebiger Reihenfolge.

Übung 7.1.12 Verwenden Sie die Konstruktion aus Übung 7.1.11, um folgende Grammatik in GNF zu transformieren:

$$S \rightarrow AA \mid 0$$
$$A \rightarrow SS \mid 1$$

7.2 Das Pumping-Lemma für kontextfreie Sprachen

Wir werden nun ein Hilfsmittel für den Nachweis entwickeln, dass bestimmte Sprachen nicht kontextfrei sind. Der Satz, der als »Pumping-Lemma für kontextfreie Sprachen« bezeichnet wird, besagt, dass es in jeder ausreichend langen Zeichenreihe einer kontextfreien Sprache möglich ist, zwei Teilzeichenreihen zu finden, die simultan i-mal ($i \geq 0$) wiederholt werden können, und als Ergebnis eine Zeichenreihe zu erhalten, die ebenso in der Sprache enthalten ist.

Wir können diesem Satz das analoge Pumping-Lemma für reguläre Sprache aus Satz 4.1 gegenüberstellen, das besagt, dass es stets möglich ist, eine Zeichenreihe zu finden, die i-mal wiederholt werden kann. Der Unterschied wird deutlich, wenn wir eine Sprache wie $L = \{0^n 1^n \mid n \geq 1\}$ betrachten. Wir können zeigen, dass diese Sprache nicht regulär ist, indem wir für n eine Zahl wählen und eine Teilzeichenreihe aus Nullen wiederholen, da wir dann eine Zeichenreihe erhalten, die aus mehr Nullen als Einsen besteht. Das Pumping-Lemma für kontextfreie Sprachen besagt jedoch, dass wir zwei Zeichenreihen für das »Pumpen« finden können. Daher sind wir möglicherweise gezwungen, eine Zeichenreihe aus Nullen und aus Einsen zu wählen und zu wiederholen und damit nur in L enthaltene Zeichenreihen zu generieren. Da L eine kontextfreie Sprache ist, muss das auch so sein, denn wir sollten gar nicht in der Lage sein, mit dem Pumping-Lemma für kontextfreie Sprachen Zeichenreihen zu bilden, die nicht in L enthalten sind.

7.2.1 Die Größe von Parsebäumen

Unser erster Schritt in der Ableitung des Pumping-Lemmas für kontextfreie Sprachen besteht darin, die Form und Größe von Parsebäumen zu untersuchen. Eine Anwendung der CNF besteht darin, Parsebäume in Binärbäume umzuwandeln. Diese Bäume besitzen einige nützliche Eigenschaften, von denen wir hier eine nutzen werden.

Satz 7.17 Angenommen, es liegt ein Parsebaum entsprechend einer Grammatik in Chomsky-Normalform $G = (V, T, P, S)$ vor, und angenommen, dieser Parsebaum ergibt die terminale Zeichenreihe w. Wenn der längste Pfad die Länge n hat, dann gilt $|w| \leq 2^{n-1}$.

Beweis: Der Beweis ist eine einfache Induktion über n.

Induktionsbeginn: $n = 1$. Sie erinnern sich, dass die Länge eines Pfads in einem Baum der Anzahl von Kanten entspricht, d. h. um eins geringer als die Anzahl der Knoten ist. Folglich besteht ein Ableitungsbaum für G mit einem Pfad mit der maximalen Länge 1 lediglich aus der Wurzel und einem Blatt, das mit einem terminalen Zeichen beschriftet ist. Daher ist $|w| = 1$. Da in diesem Fall $2^{n-1} = 2^0 = 1$, haben wir den Induktionsbeginn bewiesen.

Induktionsschritt: Angenommen, der längste Pfad hätte die Länge n und $n > 1$. An der Wurzel des Baums muss eine Produktion der Form $A \rightarrow BC$ eingesetzt werden, da $n >$

1; d. h. der Baum könnte nicht mit einer Produktion aufgebaut werden, deren Rumpf ein terminales Zeichen ist. Kein Pfad in den Teilbäumen, deren Wurzel B bzw. C bildet, kann eine Länge haben, die größer als $n - 1$ ist. Nach der Induktionshypothese haben diese beiden Teilbäume Ergebnisse einer Länge von höchstens 2^{n-2}. Das Ergebnis des gesamten Baums besteht aus der Verkettung dieser beiden Ergebnisse und hat daher eine Länge von höchstens $2^{n-2} + 2^{n-2} = 2^{n-1}$. Damit ist der Induktionsschritt bewiesen.

7.2.2 Aussage des Pumping-Lemmas

Das Pumping-Lemma für kontextfreie Sprache ist dem Pumping-Lemma für reguläre Sprachen recht ähnlich. Wir teilen hier jedoch jede in der kfL L enthaltene Zeichenreihe z in fünf Teile auf und »pumpen« jeweils den zweiten und vierten Teil simultan auf.

Satz 5.18 (Pumping-Lemma für kontextfreie Sprachen) Sei L eine kontextfreie Sprache. Dann gibt es eine Konstante n, für die gilt: Wenn z eine Zeichenreihe aus L mit einer Länge $|z|$ von mindestens n ist, dann können wir eine Zerlegung von z angeben mit $z = uvwxy$, für die folgende Bedingungen erfüllt sind:

1. $|vwx| \leq n$. Das heißt, der mittlere Teil ist nicht zu lang.
2. $vx \neq \varepsilon$. Da v und x die Teile sind, die »aufgepumpt« werden, besagt diese Bedingung, dass wenigstens eine der zu wiederholenden Zeichenreihen nicht leer sein darf.
3. Für alle $i \geq 0$ ist uv^iwx^iy in L enthalten. Das heißt, auch wenn die beiden Zeichenreihen v und x beliebig oft wiederholt werden, einschließlich nullmal, ist die sich daraus ergebende Zeichenreihe ein Element von L.

Beweis: Der erste Schritt besteht darin, eine Grammatik G in Chomsky-Normalform für L zu finden. Eine solche Grammatik existiert nicht, falls L die kontextfreie Sprache $\emptyset$ oder $\{\varepsilon\}$ repräsentiert. Wenn $L = \emptyset$, dann ist die Aussage des Satzes, in der von einer Zeichenreihe z in L die Rede ist, auch in diesem Fall gültig, da $\emptyset$ keine Zeichenreihe z enthält. Ist $L = \{\varepsilon\}$, dann gibt es in L für $n = 1$ kein z mit $|z| \geq n$, und die Aussage des Pumping-Lemmas ist gültig.

Nun beginnen wir mit einer CNF-Grammatik $G = (V, T, P, S)$, derart dass $L(G) = L - \{\varepsilon\}$, und nehmen an, G verfüge über m Variablen. Wir wählen $n = 2^m$.[3] Als Nächstes nehmen wir an, z sei aus L mit $|z| = n$. Nach Satz 7.17 muss jeder Parsebaum, dessen längster Pfad höchstens die Länge m hat, ein Ergebnis der Länge $2^{m-1} = n/2$ oder eine geringere Länge haben. Ein solcher Parsebaum kann nicht das Ergebnis z haben, da z zu lang ist. Folglich muss jeder Parsebaum mit dem Ergebnis z einen Pfad besitzen, der mindestens die Länge $m + 1$ hat.

In Abbildung 7.2 ist der längste Pfad in dem Baum für z dargestellt, wobei k mindestens gleich m und der Pfad die Länge $k + 1$ hat. Da $k \geq m$, liegen mindestens $m + 1$ Vorkommen der Variablen $A_0, A_1, ..., A_k$ auf dem Pfad. Da V nur m verschiedene Variablen enthält, muss es sich bei mindestens zwei der letzten $m + 1$ Variablen auf dem Pfad (d. h. A_{k-m} bis einschließlich A_k) um dieselben Variablen handeln. Angenommen, $A_i = A_j$, wobei $k - m \leq i < j \leq k$.

3. Wenn G keine Variable enthält, dann haben alle Zeichenreihen in $L(G)$ die Länge 1. Dann gibt es für $n = 2$ kein z mit $|z| = n$, und die Aussage des Pumping-Lemmas ist gültig, d. h. wir können annehmen, dass $m \geq 1$.

Dann ist es möglich, den Baum wie in Abbildung 7.3 dargestellt aufzuteilen. Die Zeichenreihe w ist das Ergebnis des Teilbaums, der von A_j als Wurzel ausgeht. Bei den Zeichenreihen v und x handelt es sich um die Zeichenreihen, die sich im Ergebnis des großen Baums mit der Wurzel A_i links bzw. rechts von w befinden. Beachten Sie, dass v und x nicht beide gleich ε sein können, da die Grammatik keine Einheitsproduktionen enthält. Schließlich sind u und y diejenigen Teile von z, die sich links bzw. rechts von dem Teilbaum mit der Wurzel A_i befinden.

Abbildung 7.2: Jede ausreichend lange Zeichenreihe in *L* muss in ihrem Parsebaum einen langen Pfad besitzen

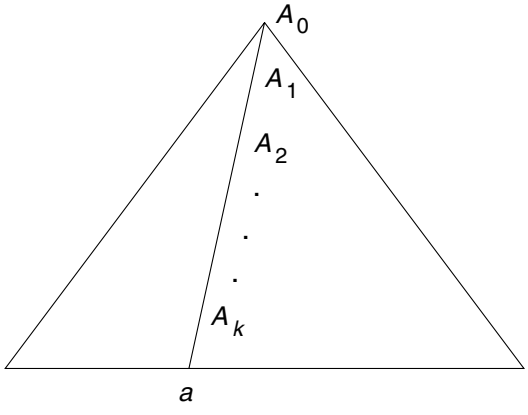
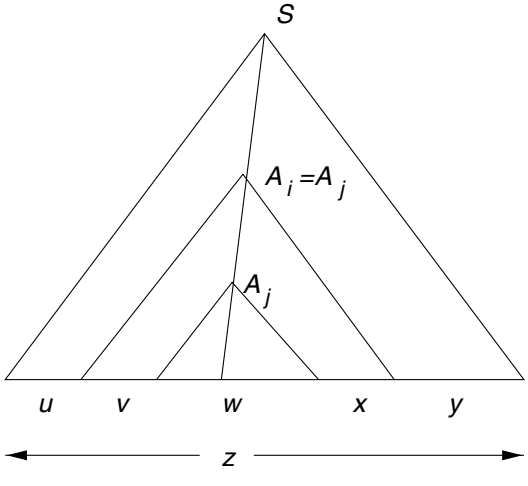
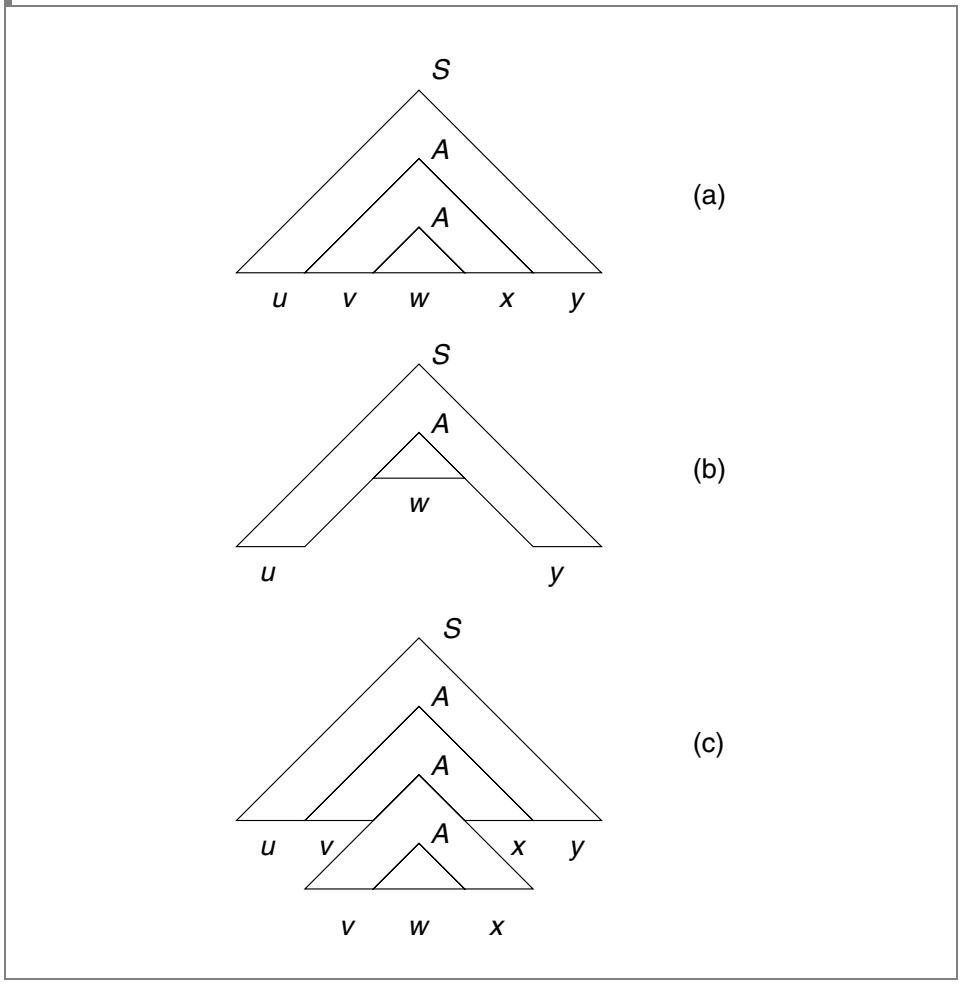


Abbildung 7.3: Aufteilung der Zeichenreihe *w*, sodass das Pumping-Lemma angewandt werden kann



Wenn $A_i = A_j = A$, dann können wir, wie in Abbildung 7.4 gezeigt, neue Parsebäume aus dem ursprünglichen Baum konstruieren. Zuerst können wir den Teilbaum mit der Wurzel A_i und dem Ergebnis vwz durch den Teilbaum mit der Wurzel A_j und dem Ergebnis w ersetzen. Wir können diese Ersetzung vornehmen, weil A_j die Wurzel beider Bäume ist. Der daraus resultierende Baum ist in Abbildung 7.4 (b) dargestellt. Er hat das Ergebnis uwy und entspricht dem Fall $i = 0$ in dem Zeichenreihenmuster uv^iwx^iy.

Abbildung 7.4: Parsebäume, wenn die Zeichenreihen v und x nullmal und zweimal wiederholt werden



Eine andere Möglichkeit ist in Abbildung 7.4 (c) dargestellt. Dort wurde der Teilbaum mit der Wurzel A_j durch den gesamten Teilbaum mit der Wurzel A_i ersetzt. Auch hier ist die Begründung wieder, dass wir einen Baum mit der Wurzel A durch einen anderen mit der gleichen Bezeichnung an der Wurzel ersetzen. Dieser Baum hat das Ergebnis uv^2wx^2y. Würden wir dann den Teilbaum aus Abbildung 7.4 (c) mit dem Ergebnis

w durch den größeren Teilbaum mit dem Ergebnis vwx ersetzen, dann erhielten wir einen Teilbaum mit dem Ergebnis uv^3wx^3y, und wir könnten diesen Vorgang für jeden Exponenten i wiederholen. Folglich weist G für alle Zeichenreihen uv^iwx^iy Parsebäume auf, und damit ist das Pumping-Lemma fast bewiesen.

Zu zeigen bleibt die Bedingung (1), die besagt, dass $|vwx| \leq n$. A_i befindet sich unserer Festlegung nach jedoch nahe dem Ende des Baumes, sodass $k - i \leq m$ gilt. Daher ist der längste Pfad in dem Teilbaum mit der Wurzel A_i nicht größer als $m + 1$. Nach Satz 7.17 hat der Teilbaum mit der Wurzel A_i ein Ergebnis, dessen Länge nicht größer als $2^m = n$ ist. ■

7.2.3 Anwendungen des Pumping-Lemmas für kontextfreie Sprachen

Beachten Sie, dass wir, wie bei dem zuvor beschriebenen Pumping-Lemma für reguläre Sprachen, das Pumping-Lemma als »Spiel für zwei Gegner« betrachten können.

1. Wir wählen eine Sprache L, von der wir zeigen möchten, dass sie keine kontextfreie Sprache ist.
2. Unser »Kontrahent« darf n wählen. Da wir seine Wahl nicht kennen, müssen wir alle möglichen n berücksichtigen.
3. Wir dürfen z wählen und können hierbei n als Parameter verwenden.
4. Unser Kontrahent darf z in $wvwxy$ aufteilen, wobei lediglich die Beschränkungen, dass $|vwx| \leq n$ und $vx \neq \varepsilon$, erfüllt sein müssen.
5. Wir »gewinnen« das Spiel, indem wir i wählen und zeigen, dass uv^iwx^iy nicht in L enthalten ist.

Wir werden nun einige Beispiele für Sprachen vorstellen, von denen wir mithilfe des Pumping-Lemmas beweisen können, dass es keine kontextfreien Sprachen sind. Unser erstes Beispiel zeigt, dass in kontextfreien Sprachen zwei Gruppen von Symbolen verglichen und daraufhin überprüft werden können, ob sie gleich oder verschieden sind, nicht jedoch drei Gruppen.

Beispiel 7.19 Sei L die Sprache $\{0^n1^n2^n \mid n \geq 1\}$. Das heißt, L besteht aus allen in $\mathbf{0}^+\mathbf{1}^+\mathbf{2}^+$ enthaltenen Zeichenreihen, die sich aus der gleichen Anzahl dieser drei Symbole zusammensetzen, z. B. 012, 001122 etc. Angenommen, L sei kontextfrei. Dann gibt es nach dem Pumping-Lemma eine ganze Zahl n.[4] Wir wählen $z = 0^n1^n2^n$.

Angenommen, der »Kontrahent« teilt z in $z = uvwxy$ auf, wobei $|vwx| \leq n$ und $vx \neq \varepsilon$. Wir wissen, dass vwx nicht Nullen und Zweien enthalten kann, da die letzte Null und die erste Zwei $n + 1$ Positionen voneinander entfernt sind. Wir werden zeigen, dass L eine Zeichenreihe enthält, die bekanntermaßen nicht in L enthalten ist, und beweisen damit durch Widerspruch, dass L keine kontextfreie Sprache ist. Folgende Fälle sind zu unterscheiden:

1. vwx enthält keine Zweien. Dann besteht vx nur aus Nullen und Einsen und verfügt über mindestens eines dieser Symbole. In diesem Fall enthält uwy, das

4. Beachten Sie, dass es sich bei diesem n um die Konstante handelt, die vom Pumping-Lemma gegeben wird, und nicht um die lokale Variable n aus der Definition von L.

nach dem Pumping-Lemma in L enthalten sein müsste, n Zweien und weniger als n Nullen oder weniger als n Einsen oder sowohl weniger als n Nullen als auch weniger als n Einsen. Die Zeichenreihe entspricht damit nicht der Definition von L, und wir schließen daraus, dass L in diesem Fall keine kontextfreie Sprache ist.

2. vwx enthält keine Nullen. In diesem Fall besteht uwy aus n Nullen und weniger als n Einsen oder weniger als n Zweien. Daher kann diese Zeichenreihe nicht in L enthalten sein.

Gleich, welcher Fall gilt, wir können schließen, dass L eine Zeichenreihe enthält, die definitionsgemäß nicht in L enthalten ist. Dank dieses Widerspruchs können wir darauf schließen, dass unsere Annahme falsch war: L ist keine kontextfreie Sprache. ■

Kontextfreie Sprachen können zudem nicht zwei Paare vergleichen, die aus der gleichen Anzahl von Symbolen bestehen, die durchmischt sind. Dieser Gedanke wird im folgenden Beispiel präzisiert, indem mithilfe des Pumping-Lemma bewiesen wird, dass eine Sprache dieser Art nicht kontextfrei ist.

Beispiel 7.20 Sei L die Sprache $\{0^i1^j2^i3^j \mid i \geq 1 \text{ und } j \geq 1\}$. Wenn L kontextfrei ist, dann sei n die Konstante für L und $z = 0^n1^n2^n3^n$. Wir können z aufteilen in $z = uvwxy$, sodass die üblichen Bedingungen $|vwx| \leq n$ und $vx \neq \varepsilon$ erfüllt sind. Dann ist die Zeichenreihe vwx entweder die Teilzeichenreihe eines Symbols oder sie erstreckt sich über zwei benachbarte Symbole.

Wenn vwx nur ein Symbol (n-mal) enthält, dann besitzt uwy jeweils n von drei verschiedenen Symbole und weniger als n vom vierten Symbol. Folglich kann diese Zeichenreihe nicht in L enthalten sein. Wenn sich vwx über zwei Symbole erstreckt, z. B. die Einsen und Zweien, dann fehlen in uwy entweder einige Einsen oder einige Zweien oder beides. Angenommen, es fehlen Einsen. Da die Zeichenreihe n Dreien enthält, kann sie nicht in L enthalten sein. Wenn uwy weniger Zweien enthält, dann kann die Zeichenreihe nicht in L enthalten sein, weil sie n Nullen besitzt. Wir haben damit die Annahme widerlegt, dass L eine kontextfreie Sprache sei, und schließen daraus, dass L keine kontextfreie Sprache ist. ■

Als letztes Beispiel zeigen wir, dass kontextfreie Sprachen nicht die Gleichheit bzw. Ungleichheit von zwei Zeichenreihen beliebiger Länge feststellen können, wenn die Zeichenreihen über einem Alphabet mit mehr als einem Symbol gebildet werden. Eine Implikation dieser Beobachtung ist nebenbei gesagt, dass Grammatiken keinen geeigneten Mechanismus bieten, um die Einhaltung bestimmter »semantischer« Beschränkungen einer Programmiersprache zu erzwingen, wie z. B. die allgemeine Forderung, dass Bezeichner vor ihrer Verwendung deklariert werden müssen. In der Praxis werden andere Mechanismen verwendet, z. B. eine »Symboltabelle«, in der alle deklarierten Bezeichner verzeichnet werden, und wir versuchen erst gar nicht, einen Parser zu entwerfen, der prüft, ob ein Bezeichner vor seiner Verwendung definiert worden ist.

Beispiel 7.21 Sei $L = \{ww \mid w \text{ ist in } \{0, 1\}^* \text{ enthalten}\}$. Das heißt, L besteht aus sich wiederholenden Zeichenreihen, wie z. B. ε, 0101, 00100010 oder 110110. Wenn L eine kontextfreie Sprache ist, dann sei n ihre Pumping-Lemma-Konstante. Betrachten Sie die Zeichenreihe $z = 0^n1^n0^n1^n$. Diese Zeichenreihe ist in L enthalten.

Entsprechend dem Verfahrensmuster der vorhergehenden Beispiele können wird z in $uvwxy$ aufteilen, wobei $|vwx| \leq n$ und $vx \neq \varepsilon$. Wir werden zeigen, dass uwy nicht in L enthalten ist, und somit durch Widerspruch beweisen, dass L keine kontextfreie Sprache ist.

Zuerst stellen wir fest, dass $|uwy| \geq 3n$, wenn $|vwx| \leq n$. Das heißt, wenn uwy eine sich wiederholende Zeichenreihe darstellt, z. B. tt, dann muss t eine Länge von mindestens $3n/2$ haben. Abhängig davon, an welcher Position sich vwx innerhalb von z befindet, sind verschiedene Fälle zu unterscheiden.

1. Angenommen, vwx befindet sich innerhalb der ersten n Nullen. Das heißt, vx soll aus k Nullen bestehen, wobei $k > 0$. Dann beginnt uwy mit $0^{n-k}1^n$. Da $|uwy| = 4n - k$, wissen wir, dass $|t| = 2n - k/2$, wenn $uwy = tt$. Folglich endet t erst nach dem ersten Block von Einsen, d. h. t endet mit 0. uwy endet jedoch mit 1, und folglich kann uwy nicht gleich tt sein.

2. Angenommen, vwx erstreckt sich über den ersten Block von Nullen und reicht in den ersten Block von Einsen. Es ist möglich, dass vx lediglich aus Nullen besteht, wenn $x = \varepsilon$. Dann entspricht die Argumentation für den Beweis, dass uwy nicht die Gestalt tt hat, der von Fall (1). Falls vx mindestens eine Eins enthält, dann muss die Zeichenreihe t, die mindestens die Länge $3n/2$ hat, mit 1^n enden, weil uwy mit 1^n endet. Allerdings gibt es außer dem letzten Block keinen Block mit n Einsen, und daher kann t in uwy nicht wiederholt vorkommen.

3. Ist vwx im ersten Block von Einsen enthalten, dann lässt sich entsprechend dem zweiten Teil von Fall (2) argumentieren, dass uwy nicht in L enthalten sein kann.

4. Angenommen, vwx erstreckt sich über die Fuge des ersten Blocks von Einsen und des zweiten Blocks von Nullen. Falls vx tatsächlich keine Nullen enthält, dann entspricht die Argumentation genau dem Fall, in dem vwx im ersten Block von Einsen enthalten ist. Falls vx mindestens eine Null enthält, dann beginnt uwy mit einem Block von n Nullen, und ebenso beginnt t mit einem Block von n Nullen, wenn $uvw = tt$. Es ist allerdings kein zweiter Block von Nullen in uvw enthalten, der für die zweite Kopie von t erforderlich wäre. Wir schließen daraus, dass auch in diesem Fall uwy nicht in L enthalten ist.

5. In den übrigen Fällen, in denen sich vwx in der zweiten Hälfte von z befindet, ist die Argumentation mit den Fällen symmetrisch, in denen vwx in der ersten Hälfte von z enthalten ist.

Folglich ist uwy in keinem Fall in L enthalten, und wir schließen daraus, dass L keine kontextfreie Sprache ist. ■

7.2.4 Übungen zum Abschnitt 7.2

Übung 7.2.1 Zeigen Sie mithilfe des Pumping-Lemmas für kontextfreie Sprachen, dass jede der folgenden Sprachen nicht kontextfrei ist:

* a) $\{a^i b^j c^k \mid i < j < k\}$.

b) $\{a^n b^n c^i \mid i \leq n\}$.

c) $\{0^p \mid p$ ist eine Primzahl$\}$. *Hinweis*: Orientieren Sie sich am Beispiel 4.3, in dem gezeigt wurde, dass diese Sprache nicht regulär ist.

***!** d) $\{0^i 1^j \mid j = i^2\}$.

! e) $\{a^n b^n c^i \mid n \leq i \leq 2n\}$.

! f) $\{ww^R w \mid w$ ist eine Zeichenreihe aus Nullen und Einsen$\}$, das heißt, die Menge von Zeichenreihen, die aus einer Zeichenreihe w, gefolgt von deren Spiegelung und einem weiteren Exemplar von w bestehen, z. B. 001100001.

! Übung 7.2.2 Wenn wir das Pumping-Lemma auf eine kontextfreie Sprache anzuwenden versuchen, dann »gewinnt der Gegner« und wir können den Beweis nicht abschließen. Zeigen Sie, worin das Problem besteht, wenn wir eine der folgenden Sprachen für L wählen:

a) $\{00, 11\}$

***** b) $\{0^n 1^n \mid n \geq 1\}$

***** c) Die Menge der Palindrome über dem Alphabet $\{0, 1\}$

! Übung 7.2.3 Es gibt eine mächtigere Version des Pumping-Lemmas für kontextfreie Sprachen, die als Ogdens Lemma bezeichnet wird. Sie unterscheidet sich von dem von uns bewiesenen Pumping-Lemma dadurch, dass wir uns auf n »ausgezeichnete« Positionen einer Zeichenreihe z konzentrieren können und dass sichergestellt ist, dass die zu wiederholenden Zeichenreihen zwischen 1 und n *ausgezeichnete* Positionen aufweisen. Dies bietet den Vorteil, dass eine Sprache aus zwei Teilen bestehende Zeichenreihen umfassen kann, wobei das Pumping-Lemma auf einen Teil angewendet werden kann, ohne dass Zeichenreihen erzeugt werden, die nicht zur Sprache gehören, während bei Anwendung auf den anderen Teil *nicht* zur Sprache gehörende Zeichenreihen erzeugt werden. Wir können den Beweis, dass eine Sprache nicht kontextfrei ist, nur dann abschließen, wenn das Pumping-Lemma auf letzteren Teil angewendet wird. Die formale Aussage von Ogdens Lemma lautet: Wenn L eine kontextfreie Sprache ist, dann gibt es eine Konstante n, für die gilt: Wenn z eine beliebige Zeichenreihe mit einer Länge von mindestens n aus L ist, in der wir mindestens n Positionen als *ausgezeichnete* Positionen auswählen, dann können wir z in $z = uvwxy$ zerlegen, derart dass

1. vwx höchstens n ausgezeichnete Positionen umfasst.
2. vx mindestens eine ausgezeichnete Position umfasst.
3. für alle $i \geq 0$ gilt, dass $uv^i wx^i y$ in L enthalten ist.

Beweisen Sie Ogdens Lemma. *Hinweis*: Der Beweis ist im Wesentlichen mit dem für das Pumping-Lemma aus Satz 7.18 identisch, abgesehen davon, dass wir bei der Wahl eines langen Pfads im Parsebaum von z so tun, als gäbe es in z keine Positionen, die nicht ausgezeichnet sind.

*** Übung 7.2.4** Verwenden Sie Ogdens Lemma aus Übung 7.2.3, um den Beweis aus Beispiel 7.21 zu vereinfachen, dass $L = \{ww \mid w$ ist in $\{0, 1\}^*$ enthalten$\}$ keine kontextfreie Sprache ist. *Hinweis*: Die beiden mittleren Blöcke von $z = 0^n 1^n 0^n 1^n$ müssen aus ausgezeichneten Positionen bestehen.

Übung 7.2.5 Verwenden Sie Ogdens Lemma aus Übung 7.2.3, um zu zeigen, dass die folgenden Sprachen keine kontextfreien Sprachen sind:

! a) $\{0^i1^j0^k \mid j = \max(i, k)\}$.

!! b) $\{a^nb^nc^i \mid i \neq n\}$ *Hinweis*: Wenn n die Konstante für Ogdens Lemma ist, betrachten Sie die Zeichenreihe $z = a^nb^nc^{n!}$.

7.3 Abgeschlossenheit kontextfreier Sprachen

Wir werden nun einige der Operationen mit kontextfreien Sprachen betrachten, die mit Sicherheit eine kontextfreie Sprache erzeugen. Viele dieser Eigenschaften der Abgeschlossenheit gleichen den Sätzen für reguläre Sprachen aus dem Abschnitt 4.2. Es gibt jedoch einige Unterschiede.

Erstens stellen wir eine Operation namens Substitution vor, in der wir die einzelnen Symbole der Zeichenreihen einer Sprache jeweils durch eine gesamte Sprache ersetzen. Diese Operation stellt eine Verallgemeinerung der Homomorphismen dar, die wir in Abschnitt 4.2.3 behandelt haben, und ist hilfreich beim Beweis anderer Abgeschlossenheitseigenschaften kontextfreier Sprachen, z. B. für die Operationen der regulären Ausdrücke: Vereinigung, Verkettung und Hüllenbildung. Wir zeigen, dass die kontextfreien Sprachen bezüglich Homomorphismen und inversen Homomorphismen abgeschlossen sind. Im Gegensatz zu regulären Sprachen sind kontextfreie Sprachen bezüglich des Durchschnitts und der Differenz nicht abgeschlossen. Allerdings ist sowohl der Durchschnitt als auch die Differenz aus einer kontextfreien Sprache und einer regulären Sprache stets eine kontextfreie Sprache.

7.3.1 Substitution

Sei Σ ein Alphabet, und nehmen wir an, wir wählen für jedes Symbol a aus Σ eine Sprache L_a. Die gewählten Sprachen können über beliebigen Alphabeten gebildet werden, die nicht notwendigerweise Σ oder einander entsprechen müssen. Mit dieser Wahl von Sprachen wird eine Funktion s (eine *Substitution*) über Σ definiert, und wir werden auf die verschiedenen L_a für die einzelnen Symbole a in der Form $s(a)$ Bezug nehmen.

Wenn $w = a_1a_2 \ldots a_n$ eine Zeichenreihe aus Σ^* ist, dann ist $s(w)$ die Sprache aus allen Zeichenreihen $x_1x_2 \ldots x_n$, derart dass für $i = 1, 2, \ldots, n$ die Zeichenreihe x_i in der Sprache $s(a_i)$ enthalten ist. Anders ausgedrückt, $s(w)$ ist die Verkettung der Sprachen $s(a_1)s(a_2) \ldots s(a_n)$. Wir können die Definition von s auf Sprachen ausweiten: $s(L)$ ist die Vereinigung von $s(w)$ für alle Zeichenreihen w aus L.

Beispiel 7.22 Angenommen $s(0) = \{a^nb^n \mid n \geq 1\}$ und $s(1) = \{aa, bb\}$. Das heißt, s ist eine Substitution über dem Alphabet $\Sigma = \{0, 1\}$. Die Sprache $s(0)$ besteht aus der Menge aller Zeichenreihen mit einem oder mehreren Symbolen a gefolgt von der gleichen Anzahl von Symbolen b, während $s(1)$ die endliche Sprache ist, die aus den beiden Zeichenreihen aa und bb besteht.

Sei $w = 01$. Dann stellt $s(w)$ die Verkettung der Sprachen $s(0)s(1)$ dar. Genauer gesagt, $s(w)$ besteht aus allen Zeichenreihen der Form a^nb^naa sowie a^nb^{n+2}, wobei $n \geq 1$.

Nehmen wir nun an, $L = L(\mathbf{0}^*)$, d. h. die Menge aller Zeichenreihen aus Nullen einschließlich ε. Dann ist $s(L) = (s(0))^*$.

Diese Sprache entspricht der Menge aller Zeichenreihen der Form

$$a^{n_1}b^{n_1}a^{n_2}b^{n_2} \ldots a^{n_k}b^{n_k}$$

für ein $k \geq 0$ und eine Folge ausgewählter positiver ganzer Zahlen n_1, n_2, ..., n_k. Zur Sprache gehören Zeichenreihen wie ε, *aabbaaabbb* und *abaabbabab*. ■

Satz 7.23 Wenn L eine kontextfreie Sprache über dem Alphabet Σ ist und s eine Substitution über Σ, derart dass $s(a)$ für jedes a aus Σ eine kontextfreie Sprache ist, dann ist $s(L)$ eine kontextfreie Sprache.

Beweis: Der Grundgedanke besteht darin, dass wir eine kfG für L nehmen und darin jedes terminale Symbol a durch das Startsymbol einer kfG für die Sprache $s(a)$ ersetzen. Ergebnis ist eine einzige kfG, die $s(L)$ erzeugt. Damit dies funktioniert, müssen jedoch einige Details richtig gewählt werden.

Formal ausgedrückt, wir beginnen mit Grammatiken für jede der relevanten Sprachen, z. B. $G = (V, \Sigma, P, S)$ für L und $G_a = (V_a, T_a, P_a, S_a)$ für jedes a aus Σ. Da wir für Variable beliebige Namen wählen können, stellen wir sicher, dass die Variablenmengen disjunkt sind; d. h. eine Variable A ist höchstens einmal in V und in den V_a enthalten. Diese Namenwahl soll garantieren, dass wir nach dem Zusammenführen der Produktionen der verschiedenen Grammatiken in eine Menge von Produktionen nicht versehentlich die Produktionen von zwei Grammatiken vermischen können und damit Ableitungen erhalten, die keiner Ableitung einer der gegebenen Grammatiken entsprechen.

Wir konstruieren eine neue Grammatik $G' = (V', T', P', S)$ für $s(L)$ wie folgt:

- V' ist die Vereinigung von V und allen V_a für a aus Σ.
- T' ist die Vereinigung aller T_a für a aus Σ.
- P' setzt sich zusammen aus

  allen Produktionen der verschiedenen P_a für a aus Σ.

  den Produktionen von P, wobei jedes Vorkommen des terminalen Symbols a in einem Produktionsrumpf von P durch S_a ersetzt worden ist.

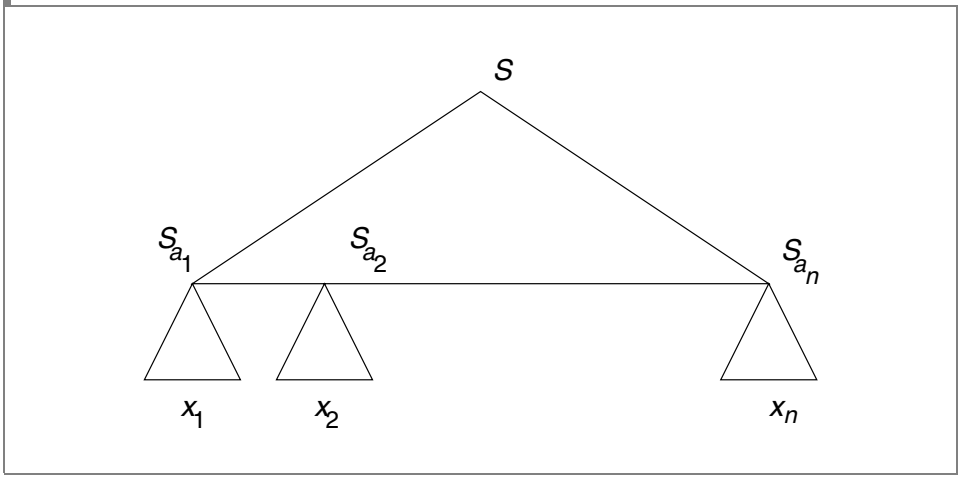
Folglich beginnen alle Parsebäume der Grammatik G' wie die Parsebäume von G; statt aber ein in Σ^* enthaltenes Ergebnis zu generieren, weisen diese Bäume eine Grenzlinie auf, an der alle Knoten Beschriftungen tragen, die S_a für ein a aus Σ entsprechen. Jeder dieser Knoten bildet die Wurzel eines Parsebaums von G_a, dessen Ergebnis eine in der Sprache $s(a)$ enthaltene terminale Zeichenreihe ist. Abbildung 7.5 zeigt einen solchen Parsebaum.

Nun müssen wir beweisen, dass diese Konstruktion in dem Sinn funktioniert, als G' die Sprache $s(L)$ erzeugt. Formal ausgedrückt:

- Eine Zeichenreihe w ist genau dann in $L(G')$ enthalten, wenn w in $s(L)$ enthalten ist.

(Wenn-Teil) Angenommen, w sei in $s(L)$ enthalten. Dann gibt es eine Zeichenreihe $x = a_1a_2 \ldots a_n$ in L und für $i = 1, 2, \ldots n$ die Zeichenreihen x_i in $s(a_i)$, derart dass $w = x_1x_2 \ldots x_n$. Der Teil von G', der sich aus den Produktionen von G ergibt, in denen jedes a durch S_a ersetzt wurde, erzeugt eine Zeichenreihe, die wie x aussieht, aber anstelle jedes a S_a

Abbildung 7.5: Parsebäume von *G'* beginnen mit einem Parsebaum von *G* und enden mit vielen Parsebäumen, die jeweils einer der Grammatiken G_a angehören



enthält. Diese Zeichenreihe lautet $S_{a_1} S_{a_2} \ldots S_{a_n}$. Dieser Teil der Ableitung von w wird durch das obere Dreieck in Abbildung 7.5 veranschaulicht.

Da die Produktionen jeder Grammatik G_a auch Produktionen von G' sind, ist die Ableitung von x_i von S_a auch eine in G' enthaltene Ableitung. Die Parsebäume dieser Ableitungen werden durch die unteren Dreiecke in Abbildung 7.5 dargestellt. Da dieser Parsebaum von G' das Ergebnis $x_1 x_2 \ldots x_n = w$ hat, schließen wir daraus, dass w in $L(G')$ enthalten ist.

(Nur-wenn-Teil) Nehmen wir nun an, w sei in $L(G')$ enthalten. Wir behaupten, dass der Parsebaum für w wie der in Abbildung 7.5 dargestellte aussehen muss, weil die Variablen der Grammatiken G und G_a für a aus Σ disjunkt sind. Folglich dürfen an der Spitze des Baums, der mit der Variablen S beginnt, nur Produktionen von G verwendet werden, bis ein Symbol S_a abgeleitet worden ist, und unterhalb des Symbols S_a dürfen nur Produktionen der Grammatik G_a verwendet werden. Infolgedessen können wir in jedem Parsebaum T von w eine Zeichenreihe $a_1 a_2 \ldots a_n$ aus $L(G)$ und Zeichenreihen x_i der Sprache $s(a_i)$ identifizieren, derart dass

1. $w = x_1 x_2 \ldots x_n$ und
2. die Zeichenreihe $S_{a_1} S_{a_2} \ldots S_{a_n}$ das Ergebnis eines Baums ist, der durch Löschen einiger Teilbäume aus T gebildet wird (siehe Abbildung 7.5).

Die Zeichenreihe $x_1 x_2 \ldots x_n$ ist schließlich in $s(L)$ enthalten, da sie gebildet wird, indem die einzelnen S_i durch die Zeichenreihe x_i ersetzt werden. Daraus können wir schließen, dass w in $s(L)$ enthalten ist. ■

7.3.2 Anwendungen des Satzes über die Substititution

Wir haben bei den regulären Sprachen einige Abgeschlossenheitseigenschaften studiert, die wir mithilfe von Satz 7.23 für kontextfreie Sprachen beweisen können. Wir werden diese Eigenschaften in einem Satz zusammenfassen.

Satz 7.24 Die kontextfreien Sprachen sind abgeschlossen bezüglich der folgenden Operationen:

1. Vereinigung
2. Verkettung
3. Hüllenbildung (*) und positive Hüllenbildung (+)
4. Homomorphismen.

Beweis: Jeder Beweis erfordert nur, dass wir die richtige Substitution wählen. In den unten angegebenen Beweisen werden jeweils Substitutionen einer kontextfreien Sprache durch eine andere kontextfreie Sprache verwendet, sodass sie nach Satz 7.23 kontextfreie Sprachen erzeugen.

1. *Vereinigung*: Seien L_1 und L_2 kontextfreie Sprachen. Dann ist $L_1 \cup L_2$ die Sprache $s(L)$, wobei L für die Sprache $\{1, 2\}$ und s für die durch $s(1) = L_1$ und $s(2) = L_2$ definierte Substitution steht.
2. *Verkettung*: Seien L_1 und L_2 wieder kontextfreie Sprachen. Dann ist L_1L_2 die Sprache $s(L)$, wobei für L für die Sprache $\{12\}$ und s für dieselbe Substitution wie im Fall (1) steht.
3. *Hüllenbildung und positive Hüllenbildung* : Wenn L_1 eine kontextfreie Sprache, L die Sprache $\{1\}^*$ und s die Substitution $s(1) = L_1$ ist, dann ist $L_1^* = s(L)$. Wenn L stattdessen die Sprache $\{1\}^+$ ist, dann gilt analog $L_1^+ = s(L)$.
4. Angenommen, L ist eine kfS über dem Alphabet Σ und h ist ein Homomorphismus über Σ. Sei s die Substitution, die jedes Symbol a aus Σ durch die Sprache ersetzt, die aus der einen Zeichenreihe $h(a)$ besteht. Das heißt, $s(a) = \{h(a)\}$ für alle a aus Σ. Dann gilt $h(L) = s(L)$. ■

7.3.3 Spiegelung

Die kontextfreien Sprachen sind auch bezüglich der Spiegelung abgeschlossen. Wir können den Substitutionssatz hier nicht verwenden. Es gibt jedoch eine einfache Konstruktion mit Grammatiken.

Satz 7.25 Wenn L eine kontextfreie Sprache ist, dann ist auch L^R eine kontextfreie Sprache.

Beweis: Sei $L = L(G)$ für eine kontextfreie Grammatik $G = (V, T, P, S)$. Wir konstruieren $G^R = (V, T, P^R, S)$, wobei P^R für die »Spiegelung« der einzelnen in P enthaltenen Produktionen steht. Das heißt, wenn $A \to \alpha$ eine Produktion von G ist, dann ist $A \to \alpha^R$ eine Produktion von G^R. Mithilfe einer einfachen Induktion über die Länge der Ableitungen in G und G^R lässt sich zeigen, dass $L(G^R) = L^R$. Im Wesentlichen sind sämtliche Satzformen von G^R Spiegelungen der Satzformen von G und umgekehrt. Wir überlassen den formalen Beweis dem Leser als Übung. ■

7.3.4 Durchschnitte mit einer regulären Sprache

Die kontextfreien Sprachen sind nicht abgeschlossen bezüglich des Durchschnitts. Es folgt ein einfaches Beispiel, das dies beweist.

Beispiel 7.26 Wir haben in Beispiel 7.19 herausgefunden, dass die Sprache

$$L = \{0^n1^n2^n \mid n \geq 1\}$$

keine kontextfreie Sprache ist. Die beiden folgenden Sprachen *sind* jedoch kontextfrei:

$$L_1 = \{0^n1^n2^i \mid n \geq 1, \mathrm{i} \geq 1\}$$
$$L_2 = \{0^i1^n2^n \mid n \geq 1, \mathrm{i} \geq 1\}\}$$

Eine Grammatik für L_1 lautet:

$$S \rightarrow AB$$
$$A \rightarrow 0A1 \mid 01$$
$$B \rightarrow 2B \mid 2$$

Nach dieser Grammatik erzeugt A alle Zeichenreihen der Form 0^n1^n, und B erzeugt alle Zeichenreihen aus Zweien. Eine Grammatik für L_2 lautet:

$$S \rightarrow AB$$
$$A \rightarrow 0A \mid 0$$
$$B \rightarrow 1B2 \mid 12$$

Diese Grammatik ist ähnlich; hier erzeugt A jedoch alle Zeichenreihen aus Nullen und B erzeugt alle Zeichenreihen der Form 1^n2^n.

Nun gilt, $L = L_1 \cap L_2$. Zur Begründung dieser Aussage stellen wir fest, dass L_1 Zeichenreihen mit der gleichen Anzahl von Nullen und Einsen erfordert, während L_2 Zeichenreihen mit der gleichen Anzahl von Einsen und Zweien erfordert. Eine Zeichenreihe, die beiden Sprachen angehören soll, muss daher die gleiche Anzahl von Nullen, Einsen und Zweien aufweisen und folglich in L enthalten sein. Dass $L = L_1 \cap L_2$ ist unmittelbar klar, weil $0^i1^i2^i$ für jedes $i \geq 1$ sowohl in L_1 als auch in L_2 enthalten ist.

Wären die kontextfreien Sprachen bezüglich des Durchschnitts abgeschlossen, könnten wir die falsche Aussage, dass L kontextfrei ist, beweisen. Wir schließen durch Widerspruch, dass die kontextfreien Sprachen bezüglich des Durchschnitts nicht abgeschlossen sind. ■

Andererseits können wir eine schwächere Behauptung bezüglich des Durchschnitts formulieren. Die kontextfreien Sprachen sind bezüglich des » Durchschnitts mit einer regulären Sprache« abgeschlossen. Die formale Aussage und deren Beweis finden Sie im nächsten Satz 7.27.

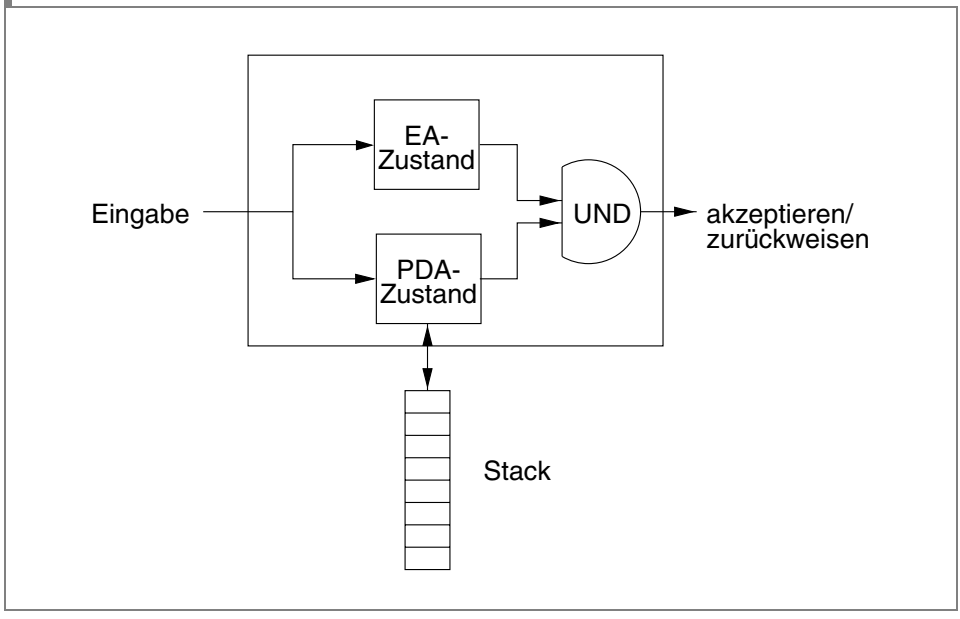
Satz 7.27 Wenn L eine kontextfreie Sprache und R eine reguläre Sprache ist, dann ist $L \cap R$ eine kontextfreie Sprache.

Beweis: Dieser Beweis erfordert, dass wir die kontextfreie Sprache als Pushdown-Automaten (PDA) und die reguläre Sprache als endlichen Automaten (EA) darstellen. Er stellt eine Verallgemeinerung des Beweises von Satz 4.8 dar, in dem wir zwei endliche Automaten »parallel« ausgeführt haben, um den Durchschnitt ihrer Sprachen zu erhalten. Hier führen wir einen endlichen Automaten und einen PDA »parallel« aus. Wir erhalten als Ergebnis einen anderen PDA, wie in Abbildung 7.6 dargestellt.

Formal ausgedrückt, sei

$$P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$$

Abbildung 7.6: Ein PDA und ein EA laufen parallel und erzeugen so einen neuen PDA



ein PDA, der L durch einen Endzustand akzeptiert, und sei

$$A = (Q_A, \Sigma, \delta_A, q_A, F_A)$$

ein DEA für R. Wir konstruieren den PDA

$$P' = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$$

wobei $\delta((q, p), a, X)$ gemäß Definition die Menge aller Paare $((r, s), \gamma)$ darstellt, derart dass

1. $s = \hat{\delta}_A(p, a)$ und
2. das Paar (r, γ) in $\delta_P(q, a, X)$ enthalten ist.

Das heißt, für jeden Schritt des PDA P ist ein Schritt des PDA P' festgelegt, und zudem wird in einer zweiten Zustandskomponente von P' der Zustand des DEA A mitgeführt. Beachten Sie, dass a ein Symbol aus Σ oder $a = \varepsilon$ sein kann. Im ersten Fall gilt $s = \hat{\delta}_A(p, a)$, und wenn $a = \varepsilon$, dann ist $s = p$, d. h. A ändert seinen Zustand nicht, während P einen ε-Schritt ausführt.

Durch eine einfache Induktion über die Anzahl der Schritte des PDA lässt sich zeigen, dass $(q_P, w, Z_0) \overset{*}{\vdash}_P (q, \varepsilon, \gamma)$ genau dann, wenn $((q_P, q_A), w, Z_0) \overset{*}{\vdash}_P (q, p)\ \varepsilon, \gamma)$, wobei p $= \hat{\delta}(q_A, w)$. Wir überlassen diesen Induktionsbeweis dem Leser als Übung. Da (q, p) genau dann ein akzeptierender Zustand von P' ist, wenn q ein akzeptierender Zustand von P und p ein akzeptierender Zustand von A ist, schließen wir, dass P' w genau dann akzeptiert, wenn sowohl P als auch A w akzeptieren, d. h. w in $L \cap R$ enthalten ist. ■

Beispiel 7.28 In Abbildung 6.6 entwarfen wir einen PDA namens F, der durch seinen Endzustand die Menge der Zeichenreihen aus den Symbolen i und e akzeptiert, die minimale Verletzungen der Regel hinsichtlich der Verwendung von `if` und `else` in

einem C-Programm darstellen. Wir nennen diese Sprache L. Der PDA F wurde definiert durch:

$$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, \{r\})$$

wobei δ_F aus folgenden Regeln besteht:

1. $\delta_F(p, \varepsilon, X_0) = \{(q, ZX_0)\}$
2. $\delta_F(q, i, Z) = \{(q, ZZ)\}$
3. $\delta_F(q, e, Z) = \{(q, \varepsilon)\}$
4. $\delta_F(q, e, X_0) = \{(r, \varepsilon, \varepsilon)\}$

Wir wollen nun einen endlichen Automaten einführen:

$$A = (\{s, t\}, \{i, e\}, \delta_A, s, \{s, t\})$$

der die Zeichenreihen der Sprache i^*e^* akzeptiert, d. h. alle Zeichenreihen, die aus beliebig vielen Vorkommen des Buchstabens i gefolgt von beliebig vielen Vorkommen des Buchstabens e bestehen. Wir nennen diese Sprache R. Die Übergangsfunktion δ_A wird durch folgende Regeln definiert:

a) $\delta_A(s, i) = s$

b) $\delta_A(s, e) = t$

c) $\delta_A(t, e) = t$

Streng genommen ist A kein DEA, wie in Satz 7.27 angenommen wird, weil er keinen Fangzustand für den Fall besitzt, dass er sich im Zustand t befindet und die Eingabe i liest. Die gleiche Konstruktion funktioniert aber sogar mit einem NEA, da der von uns konstruierte PDA nichtdeterministisch sein darf. In diesem Fall ist der konstruierte PDA tatsächlich deterministisch, obwohl er nach bestimmten Eingabefolgen in einem »Fangzustand endet«.

Wir werden folgenden PDA konstruieren:

$$P = (\{p, q, r\} \times \{s, t\}, \{i, e\}, \{Z, X_0\}, \delta, (p, s), X_0, \{r\} \times \{s, t\}$$

Die Übergänge von δ sind unten aufgeführt und nach der Regel des PDA F (eine Zahl zwischen 1 und 4) sowie der Regel des DEA A (Buchstabe a, b oder c) indiziert, aus denen diese Regel hervorgeht. In dem Fall, in dem der PDA F einen ε-Schritt ausführt, wird keine Regel von A verwendet. Beachten Sie, dass wir diese Regeln »sparsam« konstruieren, d. h. wir beginnen mit dem Zustand von P, der die Startzustände von F und A darstellt, und konstruieren nur dann Regeln für andere Zustände, wenn wir erkennen, dass P dieses Paar von Zuständen annehmen kann.

1: $\delta((p, s), \varepsilon, X_0) = \{((q, s), ZX_0)\}$

2a: $\delta((q, s), i, Z) = \{((q, s), ZZ)\}$

3b: $\delta((q, s), e, Z) = \{((q, t), \varepsilon)\}$

4: $\delta((q, s), \varepsilon, X_0) = \{((r, s), \varepsilon)\}$. Beachten Sie, dass man beweisen kann, dass diese Regel nie angewendet wird. Dies ist so, weil der Stack nur dann geleert werden kann, wenn e gelesen und die zweite Komponente des Zustands von P zu t wird, sobald P e liest.

3c: $\delta((q, t), e, Z) = \{((q, t), \varepsilon)\}$

4: $\delta((q, t), \varepsilon, X_0) = \{((r, t), \varepsilon)\}$

Die Sprache $L \cap R$ besteht aus der Menge $\{i^n e^{n+1} \mid n \geq 0\}$. Diese Menge enthält genau die minimalen `if-else`-Verstöße, die aus einem Block von `if`-Anweisungen gefolgt von einem Block von `else`-Anweisungen mit genau einer `else` mehr als `if` bestehen. Die Sprache ist offensichtlich eine kontextfreie Sprache, die von der Grammatik mit den Produktionen $S \to iSe \mid e$ erzeugt wird.

Beachten Sie, dass der PDA P genau die Sprache $L \cap R$ akzeptiert. Nachdem Z auf dem Stack abgelegt worden ist, legt er in Reaktion auf die Eingabe i weitere Z auf dem Stack ab und verbleibt im Zustand (q, s). Sobald er ein e liest, wechselt er in den Zustand (q, t) und beginnt, für jedes e ein Z vom Stack zu entfernen. Er kommt schließlich genau dann in den einzig erreichbaren akzeptierenden Zustand (r, t), wenn er eine Zeichenreihe $(i^n e^{n+1})$ mit $n = 0$ eingelesen hat. ■

Da wir jetzt wissen, dass die kontextfreien Sprachen bezüglich Durchschnitt nicht abgeschlossen sind, es jedoch bezüglich Durchschnitt mit einer regulären Sprache sind, wissen wir auch, wie sich die Mengendifferenz und Komplementbildung bei kontextfreien Sprachen verhalten. Wir fassen diese Eigenschaften in einem Satz zusammen:

Satz 7.29 Das Folgende gilt in Bezug auf die kontextfreien Sprachen L, L_1 und L_2 und eine reguläre Sprache R:

1. $L - R$ ist eine kontextfreie Sprache.
2. $\overline{L}$ ist nicht notwendigerweise eine kontextfreie Sprache.
3. $L_1 - L_2$ ist nicht notwendigerweise kontextfrei.

Beweis: Bei der Aussage (1) ist zu beachten, dass $L - R = L \cap \overline{R}$. Wenn R regulär ist, dann ist nach Satz 4.5 auch $\overline{R}$ regulär. Dann ist $L - R$ nach Satz 7.27 eine kontextfreie Sprache.

Für (2) nehmen wir an, dass $\overline{L}$ immer dann kontextfrei ist, wenn L es ist. Weil aber

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

und weil die kontextfreien Sprachen bezüglich der Vereinigung abgeschlossen sind, würde demnach gelten, dass die kontextfreien Sprachen bezüglich Durchschnitt abgeschlossen sind. Wir wissen jedoch aus dem Beispiel 7.26, dass sie es nicht sind.

Zuletzt wollen wir Aussage (3) beweisen. Wir wissen, dass Σ^* eine kontextfreie Sprache für jedes Alphabet Σ ist. Es ist leicht, eine Grammatik oder einen PDA für diese kontextfreie Sprache zu entwerfen. Für Σ_2 gilt nun: $\Sigma_2^* - L_2 = \overline{L}_2$. Wäre also $\Sigma_2^* - L_2$ für jede CFL L_2 kontextfrei, so wäre das Komplement einer beliebigen kontextfreien Sprache kontextfrei. Wir würden damit (2) widersprechen, und damit haben wir durch Widerspruch bewiesen, dass $L_1 - L_2$ nicht notwendigerweise eine kontextfreie Sprache ist. ■

7.3.5 Inverse Homomorphismen

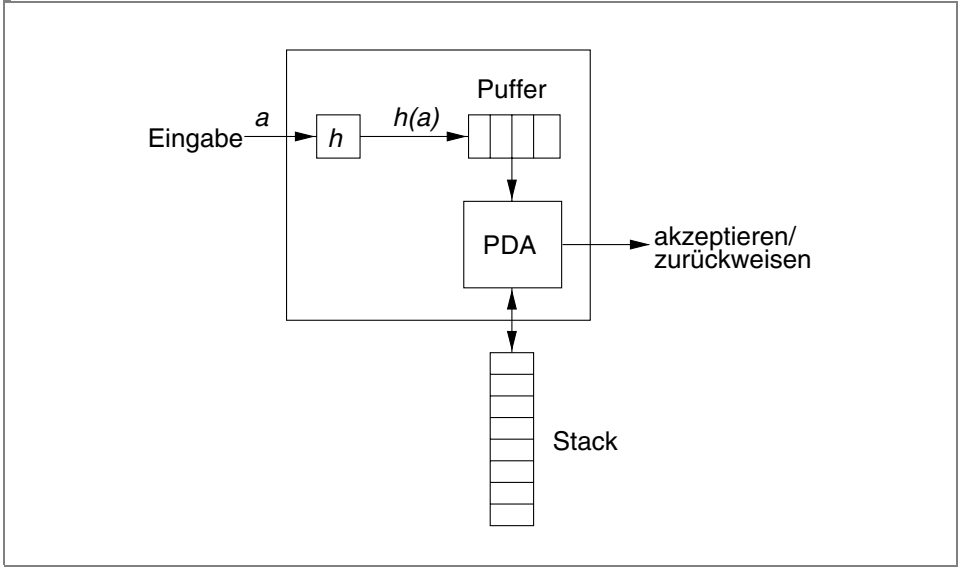
Wir wollen die in Abschnitt 4.2.4 beschriebene Operation namens »inverse Homomorphismen« erneut betrachten. Wenn h ein Homomorphismus und L eine beliebige

Sprache ist, dann bildet $h^{-1}(L)$ die Menge aller Zeichenreihen w, derart dass $h(w)$ in L enthalten ist. Der Beweis, dass reguläre Sprachen bezüglich inverser Homomorphismen abgeschlossen sind, wurde in Abbildung 4.6 dargestellt. Wir haben dort gezeigt, wie man einen endlichen Automaten entwirft, der seine Eingabesymbole a verarbeitet, indem er einen Homomorphismus h darauf anwendet und einen anderen endlichen Automaten für die Eingabesequenz $h(a)$ simuliert.

Wir können diese Eigenschaft der Abgeschlossenheit auch für kontextfreie Sprachen auf ähnliche Art beweisen, indem wir einen PDA statt eines endlichen Automaten verwenden. Beim Einsatz eines PDA stellt sich jedoch ein Problem, das bei Verwendung endlicher Automaten nicht gegeben ist. Endliche Automaten reagieren auf eine Folge von Eingabesymbolen mit einer Zustandsänderung und, was den konstruierten Automaten betrifft, sieht diese Aktion daher genauso wie ein Schritt aus, den ein endlicher Automat in Reaktion auf ein Eingabesymbol ausführt.

Wenn es sich bei dem Automaten dagegen um einen PDA handelt, dann ist die Reaktion auf ein einzelnes Symbol im Allgemeinen nicht ein einziger Schritt, sondern eine Folge von Schritten. Daher ist die Konstruktion eines PDA, die der Konstruktion aus Abbildung 4.6 entspricht, etwas komplizierter. Sie ist in Abbildung 7.7 dargestellt. Das wichtigste der hier zusätzlich eingeführten Konzepte besteht darin, dass nach dem Einlesen des Eingabesymbols a der Homomorphismus $h(a)$ in einem »Puffer« abgelegt wird. Die Symbole von $h(a)$ werden einzeln nacheinander verarbeitet und dem gegebenen PDA eingegeben. Nur wenn der Puffer leer ist, liest der konstruierte PDA ein weiteres Eingabesymbol und wendet den Homomorphismus darauf an. Wir werden diese Konstruktion im nächsten Satz formalisieren.

Abbildung 7.7: Konstruktion eines PDA, der das invers homomorphe Bild einer von einem gegebenen PDA akzeptierten Sprache akzeptiert



Satz 7.30 Sei L eine kontextfreie Sprache und h ein Homomorphismus; dann ist $h^{-1}(L)$ eine kontextfreie Sprache.

Beweis: Angenommen, h wird auf Symbole des Alphabets Σ angewandt und erzeugt in T^* enthaltene Zeichenreihen. Wie oben beschrieben, beginnen wir mit einem PDA $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$, der L durch einen Endzustand akzeptiert. Wir konstruieren einen neuen PDA

$$P' = (Q', \Sigma, \delta', (q_0, \varepsilon), Z_0, F \times \{\varepsilon\}) \quad (7.1)$$

wobei

1. Q' die Menge der Paare (q, x) ist, derart dass
   a) q ein in Q enthaltener Zustand ist.
   b) x ein Suffix einer Zeichenreihe $h(a)$ für ein Eingabesymbol a aus Σ ist.

   Das heißt, die erste Komponente des Zustands von P' besteht aus dem Zustand von P und die zweite Komponente aus dem Puffer. Wir nehmen an, der Puffer wird regelmäßig mit einer Zeichenreihe $h(a)$ gefüllt und dann FIFO geleert, da wir die im Puffer enthaltenen Symbole in der Reihenfolge ihrer Ankunft als Eingabe für den simulierten PDA P verwenden. Da Σ endlich ist und $h(a)$ für alle a endlich ist, ist nur eine endliche Anzahl von Zuständen für P' verfügbar.
2. δ' wird durch die folgenden Regeln definiert:
   a) $\delta'((q, \varepsilon), a, X) = \{((q, h(a)), X)\}$ für alle Symbole a aus Σ, alle Zustände q aus Q und alle Stacksymbole X aus Γ. Beachten Sie, dass a hier nicht gleich ε sein kann. Wenn der Puffer leer ist, kann P' das nächste Eingabesymbol a verarbeiten und $h(a)$ in den Puffer einfügen.
   b) Wenn $\delta(q, b, X)$ das Paar (p, γ) enthält, wobei b in T enthalten oder $b = \varepsilon$ ist, dann enthält

   $$\delta'((q, bx), \varepsilon, X)$$

   $((p, x), \gamma)$. Das heißt, P' hat stets die Möglichkeit, unter Verwendung des ersten Symbols im Puffer einen Schritt von P zu simulieren. Wenn b ein Symbol aus T ist, dann darf der Puffer nicht leer sein. Ist jedoch $b = \varepsilon$, dann kann der Puffer leer sein.
3. Beachten Sie, dass P' nach der Definition in (7.1) den Startzustand (q_0, ε) hat, d. h. P' startet im Startzustand von P und mit einem leeren Puffer.
4. Analog sind nach der Definition in (7.1) die akzeptierenden Zustände von P' die Zustände (q, ε), derart dass q ein akzeptierender Zustand von P ist.

Die folgende Aussage charakterisiert die Beziehung zwischen P' und P:

- $(q_0, h(w), Z_0) \overset{*}{\underset{P}{\vdash}} (p, \varepsilon, \gamma)$ genau dann, wenn $((q_0, \varepsilon), w, Z_0) \overset{*}{\underset{P'}{\vdash}} ((p, \varepsilon), \varepsilon, \gamma)$.

Beide Richtungen können durch Induktionsbeweise über die Anzahl der von den beiden Automaten ausgeführten Schritte bewiesen werden. Im »Wenn-Teil« ist festzuhalten, dass P', sobald der Puffer ein Zeichen enthält, kein weiteres Eingabesymbol lesen kann und P so lange simulieren muss, bis der Puffer leer ist (aber auch wenn der Puffer leer ist, kann P' noch weiter P simulieren). Wir überlassen dem Leser die Ergänzung der weiteren Details als Übung.

Wenn wir diese Beziehung zwischen P' und P bewiesen haben, dann stellen wir fest, dass auf Grund der Definition P $h(w)$ genau dann akzeptiert, wenn P' w akzeptiert. Folglich gilt $L(P') = h^{-1}(L(P))$. ■

7.3.6 Übungen zum Abschnitt 7.3

Übung 7.3.1 Zeigen Sie, dass die kontextfreien Sprachen abgeschlossen sind bezüglich der folgenden Operationen:

* a) *init*, wie in Übung 4.2.6 (c) definiert. *Hinweis*: Beginnen Sie mit einer CNF-Grammatik für die Sprache L.

*! b) Die Operation L/a, wie in Übung 4.2.2 definiert. *Hinweis*: Beginnen Sie wieder mit einer CNF-Grammatik für L.

!! c) *cycle*, wie in Übung 4.2.11 definiert. *Hinweis*: Arbeiten Sie mit einer auf einem PDA basierenden Konstruktion.

Übung 7.3.2 Betrachten Sie die folgenden beiden Sprachen:

$$L_1 = \{a^n b^{2n} c^m \mid n, m \geq 0\}$$
$$L_2 = \{a^n b^m c^{2m} \mid n, m \geq 0\}$$

a) Zeigen Sie, dass jede dieser Sprachen kontextfrei ist, indem Sie Grammatiken für diese Sprachen formulieren.

! b) Ist $L_1 \cap L_2$ eine kontextfreie Sprache? Begründen Sie Ihre Antwort.

!! **Übung 7.3.3** Zeigen Sie, dass die kontextfreien Sprachen bezüglich der folgenden Operationen nicht abgeschlossen sind:

* a) *min*, wie in Übung 4.2.6 (a) definiert

b) *max*, wie in Übung 4.2.6 (b) definiert

c) *half*, wie in Übung 4.2.8 definiert

d) *alt*, wie in Übung 4.2.7 definiert

Übung 7.3.4 Wir definieren das Ergebnis einer Funktion *shuffle(w, x)* für zwei Zeichenreihen w und x als Menge aller Zeichenreihen z, sodass gilt:

1. Jede Position von z kann w oder x, aber nicht beiden Zeichenreihen zugeordnet werden.
2. Die Positionen von z, die w zugeordnet sind, ergeben von links nach rechts gelesen w.
3. Die Positionen von z, die x zugeordnet sind, ergeben von links nach rechts gelesen x.

Wenn z. B. $w = 01$ und $x = 110$, dann ist *shuffle*(01, 110) die Menge der Zeichenreihen {01110, 01101, 10110, 10101, 11010, 11001}. Die dritte Zeichenreihe 10110 ergibt sich beispielsweise u.a. daraus, dass die zweite und die vierte Position der Zeichenreihe 01 und die Positionen eins, drei und fünf der Zeichenreihe 110 zugeordnet werden. Die

erste Zeichenreihe 01110 lässt sich auf dreierlei Weise bilden: Die erste Position und eine der Positionen zwei, drei oder vier werden 01 und die anderen drei Positionen werden 110 zugewiesen. Wir können zudem die Funktion *shuffle*(L_1, L_2) für Sprachen als Vereinigung aller *shuffle*(w, x) für alle Paare von Zeichenreihen w aus L_1 und x aus L_2 definieren.

a) Wie lautet das Ergebnis von *shuffle*(00, 111)?

* b) Wie lautet das Ergebnis von *shuffle*(L_1, L_2), wenn $L_1 = L(\mathbf{0}^*)$ und $L_2 = \{0^n1^n \mid n \geq 0\}$.

*! c) Zeigen Sie Folgendes: Wenn L_1 und L_2 reguläre Sprachen sind, dann ist auch

$$shuffle(L_1, L_2)$$

eine reguläre Sprache. *Hinweis*: Beginnen Sie mit DEAs für L_1 und L_2.

! d) Zeigen Sie, dass *shuffle*(L, R) eine kontextfreie Sprache ist, wenn L eine kontextfreie Sprache und R eine reguläre Sprache ist. *Hinweis*: Beginnen Sie mit einem PDA für L und einem DEA für R.

!! e) Führen Sie ein Gegenbeispiel an, um zu zeigen, dass *shuffle*(L_1, L_2) nicht notwendigerweise eine kontextfreie Sprache ist, wenn L_1 und L_2 kontextfreie Sprachen sind.

*!! **Übung 7.3.5** Eine Zeichenreihe y wird als Permutation der Zeichenreihe x bezeichnet, wenn die Symbole von y so umgestellt werden können, dass sie x ergeben. Beispielsweise besitzt die Zeichenreihe $x = 011$ die Permutationen 110, 101 und 011. Wenn L eine Sprache ist, dann ist *perm*(L) die Menge aller Zeichenreihen, die Permutationen der in L enthaltenen Zeichenreihen sind. Wenn z. B. $L = \{0^n1^n \mid n \geq 0\}$, dann ist *perm*($L$) die Menge aller Zeichenreihen mit der gleichen Anzahl von Nullen und Einsen.

a) Geben Sie ein Beispiel für eine reguläre Sprache L über dem Alphabet {0, 1} an, derart dass *perm*(L) nicht regulär ist, und begründen Sie Ihre Antwort. *Hinweis*: Versuchen Sie eine reguläre Sprache zu finden, deren Permution die Menge aller Zeichenreihen mit der gleichen Anzahl von Nullen und Einsen ist.

b) Geben Sie ein Beispiel für eine reguläre Sprache L über dem Alphabet {0, 1, 2} an, derart dass *perm*(L) nicht kontextfrei ist.

c) Beweisen Sie, dass für jede reguläre Sprache L über einem aus zwei Symbolen bestehenden Alphabet *perm*(L) kontextfrei ist.

Übung 7.3.6 Geben Sie einen formalen Beweis für Satz 7.25 an, dass die kontextfreien Sprachen bezüglich Spiegelung abgeschlossen sind.

Übung 7.3.7 Vervollständigen Sie den Beweis von Satz 7.27, indem Sie zeigen, dass

$$(q_P, w, \varepsilon, Z_0) \overset{*}{\vdash}_P (q, \varepsilon, \gamma)$$

genau dann, wenn $((q_P, q_A), w, Z_0) \overset{*}{\vdash}_{P'} ((q, p), \varepsilon, \gamma)$ und $p = \hat{\delta}(p_A, w)$.

7.4 Entscheidbarkeit kontextfreier Sprachen

Lassen Sie uns nun untersuchen, welche Fragen wir zu kontextfreien Sprachen beantworten können. Analog zu Abschnitt 4.3, in dem die Eigenschaften der Entscheidbarkeit regulärer Sprachen behandelt wurden, bildet eine Repräsentation einer kontextfreien Sprache – eine Grammatik oder ein PDA – stets den Ausgangspunkt in der Betrachtung einer Frage. Da wir aus Abschnitt 6.3 wissen, dass wir Grammatiken in PDAs umwandeln können und umgekehrt, können wir annehmen, dass eine dieser beiden Repräsentationen einer kfS gegeben ist, und zwar die am einfachsten zu handhabende.

Wir werden herausfinden, dass nur wenige Fragen in Bezug auf kontextfreie Sprachen entscheidbar sind. Die wichtigsten Tests, die wir durchführen können, erlauben die Beantwortung der Frage, ob eine Sprache leer ist, und der Frage, ob eine gegebene Zeichenreihe in der Sprache enthalten ist. Wir beschließen diesen Abschnitt mit einer kurzen Diskussion von Problemen, von denen wir an späterer Stelle (in Kapitel 9) zeigen werden, dass sie »unentscheidbar« sind, d. h. dass es keine Algorithmen zu ihrer Lösung gibt. Wir beginnen diesen Abschnitt mit einigen Beobachtungen hinsichtlich der Komplexität der Verfahren zur Umwandlung der Grammatik einer Sprache in einen PDA und der Umwandlung in umgekehrter Richtung. Diese Berechnungen beeinflussen alle Fragen hinsichtlich der Effizienz, mit der wir eine Frage zu einer Eigenschaft einer kontextfreien Sprache in einer gegebenen Repräsentation entscheiden können.

7.4.1 Komplexität der Umwandlung von kfGs in PDAs und umgekehrt

Bevor wir uns mit den Algorithmen zur Entscheidung von Fragen bezüglich kfLs befassen, wollen wir die Komplexität der Umwandlung einer Repräsentation in eine andere betrachten. Die Ausführungszeit dieser Umwandlung bildet einen Bestandteil der Kosten des Entscheidungsalgorithmus, wenn die Sprache in einer Form gegeben ist, die nicht der Form entspricht, für die der Algorithmus vorgesehen ist.

Im Folgenden soll n für die Länge der gesamten Repräsentation eines PDA oder einer kfG stehen. Mit diesem Parameter wird die Größe der Grammatik bzw. des Automaten in dem Sinn »grob« beschrieben, als die Ausführungszeit einiger Algorithmen durch spezifischere Parameter repräsentiert werden könnte, wie z. B. die Anzahl der Variablen einer Grammatik oder die Summe der Länge der Stackzeichenreihen, die in der Übergangsfunktion eines PDA vorkommen.

Die Angabe der Gesamtlänge reicht jedoch zum Aufzeigen der wichtigsten Probleme aus: Ist ein Algorithmus linear bezüglich der Länge (d. h. erfordert die Ausführung nur wenig mehr Zeit als das Lesen der Eingabe), ist er exponentiell bezüglich der Länge (d. h. die Umwandlung kann nur mit relativ kleinen Beispielen durchgeführt werden) oder ist er nichtlinear polynomial (d. h. der Algorithmus kann auch mit großen Beispielen ausgeführt werden, erfordert dann oft aber einen beträchtlichen Zeitaufwand)?

Wir haben bislang verschiedene Umwandlungen behandelt, die bezüglich der Größe der Eingabe linear sind. Da sie einen linearen Zeitaufwand erfordern, wird die von ihnen erzeugte Repräsentation nicht nur schnell erzeugt, sondern hat häufig auch eine der Größe der Eingabe vergleichbare Größe. Es handelt sich hierbei um folgende Umwandlungen:

1. Umwandlung einer kfG in einen PDA nach dem in Satz 6.13 beschriebenen Algorithmus.
2. Umwandlung eines PDA, der durch einen Endzustand akzeptiert, in einen PDA, der durch Leeren seines Stacks akzeptiert, nach der in Satz 6.11 beschriebenen Konstruktion.
3. Umwandlung eines PDA, der durch Leeren seines Stacks akzeptiert, in einen PDA, der durch einen Endzustand akzeptiert, nach der in Satz 6.9 beschriebenen Konstruktion.

Andererseits ist die Ausführungszeit für die Umwandlung von einem PDA in eine Grammatik (Satz 6.14) sehr viel komplexer. Erstens ist zu beachten, dass n, die Gesamtlänge der Eingabe, mit Sicherheit eine Obergrenze für die Anzahl der Zustände und Stacksymbole bildet, sodass nicht mehr als n^3 Variablen der Form $[pXq]$ für die Grammatik konstruiert werden können. Die Umwandlung kann jedoch eine exponentielle Ausführungszeit erfordern, wenn es einen Übergang des PDA gibt, mit dem viele Symbole auf dem Stack abgelegt werden. Beachten Sie, dass eine Regel nahezu n Symbole in den Stack einfügen könnte.

Wenn wir die Konstruktion von Grammatikproduktionen nach einer Regel wie »$\delta(q, a, X)$ enthält $(r_0, Y_1Y_2 \ldots Y_k)$« betrachten, stellen wir fest, dass damit eine Kollektion von Produktionen der Form $[qXr_k] \rightarrow [r_0Y_1r_1][r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]$ für alle Listen von Zuständen $r_1, r_2, \ldots, r_k$ erzeugt wird. Da k fast so groß wie n sein und es nahezu n Zustände geben könnte, könnte die Gesamtzahl der Produktionen auf n^n anwachsen. Wir können eine solche Konstruktion für einen vernünftig bemessenen PDA nicht ausführen, wenn der PDA auch nur eine lange Stackzeichenreihe schreiben muss.

Glücklicherweise muss dieser ungünstigste Fall nicht auftreten. Wie in Übung 6.2.8 nahegelegt wurde, können wir eine lange auf dem Stack abzulegende Zeichenreihe von Stacksymbolen in eine Folge von höchstens n Schritten aufteilen, die jeweils ein Symbol auf dem Stack ablegen. Das heißt, wenn $\delta(q, a, X)$ $(r_0, Y_1Y_2 \ldots Y_k)$ enthält, dann können wir die neuen Zustände $p_2, p_3, \ldots, p_{k-1}$ einführen. Anschließend ersetzen wir $(r_0, Y_1Y_2 \ldots Y_k)$ in $\delta(q, a, X)$ durch $(p_{k-1}, Y_{k-1}Y_k)$ und führen die neuen Übergänge ein:

$$\delta(p_{k-1}, Y_{k-1}) = \{(p_{k-2}, Y_{k-2}Y_{k-1})\}, \delta(p_{k-2}, Y_{k-2}) = \{(p_{k-3}, Y_{k-3}Y_{k-2})\}$$

und so weiter bis zu $\delta(p_2, Y_2) = \{(r_0, Y_1Y_2)\}$.

Nun besitzt kein Übergang mehr als zwei Stacksymbole. Wir haben höchstens n Zustände hinzugefügt, und die Gesamtlänge aller Übergangsregeln von δ ist höchstens um einen konstanten Faktor gewachsen, d. h. sie beträgt immer noch $O(n)$. Es gibt $O(n)$ Übergangsregeln und jede generiert $O(n^2)$ Produktionen, da jeweils nur zwei Zustände aus den Produktionen, die sich aus jeder Regel ergeben, gewählt werden müssen. Folglich hat die konstruierte Grammatik die Länge $O(n^3)$ und kann in kubischer Zeit konstruiert werden. Wir fassen diese informelle Analyse im nachfolgenden Satz zusammen:

Satz 7.31 Es gibt einen $O(n^3)$-Algorithmus, mit dem ein PDA P, dessen Repräsentation die Länge n hat, in eine kfG mit einer Länge von höchstens $O(n^3)$ transformiert wird. Diese kfG erzeugt dieselbe Sprache wie P durch Leeren des Stacks. Optional können wir auch eine kfG konstruieren, die die Sprache generiert, die P durch einen Endzustand akzeptiert. ■

7.4.2 Ausführungszeit der Umwandlung in Chomsky-Normalform

Da Entscheidungsalgorithmen gelegentlich davon abhängen, dass eine kfG zuerst in Chomsky-Normalform gebracht wird, sollten wir die Ausführungszeit verschiedener Algorithmen betrachten, die wir zur Umwandlung einer beliebigen Grammatik in Chomsky-Normalform verwendet haben. Die meisten Schritte verändern die Länge der Beschreibung der Grammatik nicht, von einem konstanten Faktor abgesehen. Das heißt, wenn wir mit einer Grammatik der Länge n beginnen, dann produzieren sie eine andere Grammatik der Länge $O(n)$. Die betreffenden Beobachtungen sind in der folgenden Liste zusammengefasst:

1. Mithilfe des richtigen Algorithmus (siehe Abschnitt 7.4.3) lassen sich die erreichbaren und die erzeugenden Symbole einer Grammatik in $O(n)$ Zeit ermitteln. Die Elimination der daraus resultierenden unnützen Symbole erfordert $O(n)$ Zeit und vergrößert den Umfang der Grammatik nicht.
2. Die Konstruktion von Einheitspaaren und das Entfernen von Einheitsproduktionen, wie in Abschnitt 7.1.4 beschrieben, erfordert $O(n^2)$ Zeit, und die resultierende Grammatik hat die Länge $O(n^2)$.
3. Terminale Symbole in den Produktionsrümpfen, wie in Abschnitt 7.1.5 (Chomsky-Normalform) beschrieben, durch Variable zu ersetzen, erfordert $O(n)$ Zeit und ergibt eine Grammatik mit einer Länge von $O(n)$.
4. Das Aufteilen der Produktionsrümpfe mit einer Länge von 3 oder mehr in Rümpfe mit der Länge 2, wie in Abschnitt 7.1.5 beschrieben, erfordert auch $O(n)$ Zeit und ergibt eine Grammatik mit einer Länge von $O(n)$.

Problematisch ist die Konstruktion aus Abschnitt 7.1.3 zur Elimination von ε-Produktionen. Wenn ein Produktionsrumpf der Länge k vorliegt, könnten wir aus dieser einen Produktion $2^k - 1$ Produktionen für die neue Grammatik erzeugen. Da k zu n proportional sein könnte, könnte dieser Teil der Konstruktion $O(2^n)$ Zeit erfordern und eine Grammatik mit der Länge $O(2^n)$ ergeben.

Um diesen exponentiellen Aufwand zu vermeiden, müssen wir nur die Länge der Produktionsrümpfe begrenzen. Der Trick aus Abschnitt 7.1.5 kann auf jeden Produktionsrumpf angewendet werden, nicht nur auf Produktionsrümpfe ohne terminale Symbole. Wir empfehlen daher als Zwischenschritt vor der Elimination von ε-Produktionen, alle langen Produktionsrümpfe in eine Folge von Produktionen mit Rümpfen der Länge 2 aufzuteilen. Dieser Schritt erfordert O(n) Zeit und vergrößert die Grammatik nur linear. Die Anwendung der Konstruktion aus Abschnitt 7.1.3 zur Elimination von ε-Produktionen auf Produktionsrümpfe mit einer maximalen Länge von 2 erfordert eine Ausführungszeit von $O(n)$, und die resultierende Grammatik hat die Länge $O(n)$.

Nach dieser Modifikation der allgemeinen CNF-Konstruktion besteht der einzige nichtlineare Schritt in der Elimination von Einheitsproduktionen. Da dieser Schritt $O(n^2)$ Zeit erfordert, schließen wir auf Folgendes:

Satz 7.32 Wenn eine Grammatik der Länge n gegeben ist, können wir eine äquivalente Grammatik in Chomsky-Normalform für G in $O(n^2)$ Zeit finden. Die resultierende Grammatik hat die Länge $O(n^2)$. ■

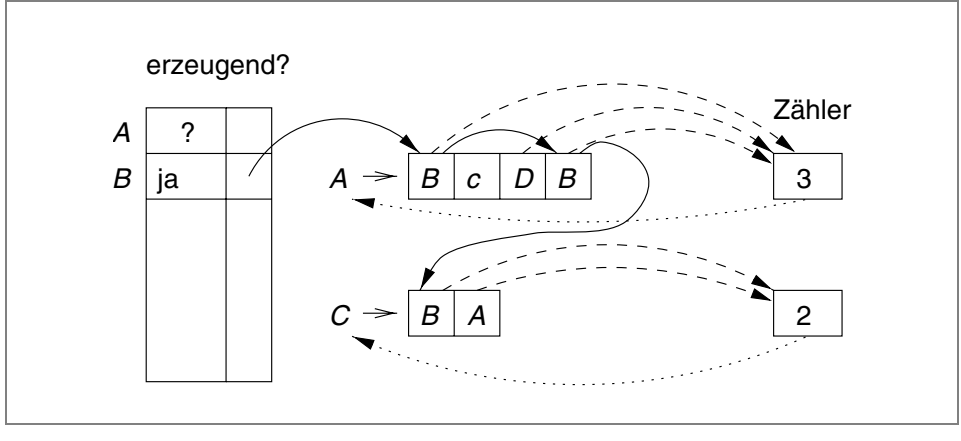
7.4.3 Prüfen, ob eine kontextfreie Sprache leer ist

Wir haben bereits den Algorithmus vorgestellt, mit dem festgestellt werden kann, ob eine kontextfreie Sprache L leer ist. Wenn eine Grammatik G für die Sprache L gegeben ist, dann können wir mithilfe des Algorithmus aus Abschnitt 7.1.2 bestimmen, ob das Startsymbol von G erzeugend ist, d. h. ob S mindestens eine Zeichenreihe ableitet. L ist genau dann leer, wenn S nichts erzeugt.

Auf Grund der Bedeutung dieses Tests werden wir nun genauer betrachten, wie viel Zeit zum Auffinden aller erzeugenden Symbole einer Grammatik G benötigt wird. Angenommen, G hat die Länge n, dann könnte sie $O(n)$ Variable enthalten, und jeder Durchlauf der induktiven Erkennung erzeugender Variablen könnte $O(n)$ Zeit für die Überprüfung aller Produktionen von G erfordern. Wenn in jedem Durchlauf nur eine Variable erkannt wird, dann könnten $O(n)$ Durchläufe notwendig sein. Folglich erfordert eine naive Implementierung des Tests zur Erkennung erzeugender Symbole $O(n^2)$ Zeit.

Es gibt allerdings einen sorgfältiger entworfenen Algorithmus, der vorab eine Datenstruktur einrichtet, sodass die Erkennung erzeugender Symbole nur $O(n)$ Zeit beansprucht. Die Datenstruktur, die in Abbildung 7.8 dargestellt ist, basiert auf einer Reihung (links in der Abbildung), die durch Variable indiziert ist und angibt, ob eine Variable bereits als erzeugend erkannt wurde. Die in Abbildung 7.8 dargestellte Reihung zeigt an, dass die Variable B als erzeugende Variable erkannt wurde und von der Variablen A noch nicht bekannt ist, ob sie erzeugend ist oder nicht. Am Ende des Algorithmus wird jedes Fragezeichen zu einem »Nein«, da jede Variable, die vom Algorithmus nicht als erzeugende Variable erkannt wurde, in der Tat nichts erzeugt.

Abbildung 7.8: Datenstruktur für den in linearer Zeit auszuführenden Test, der ermittelt, ob eine Sprache leer ist



Die Produktionen werden verarbeitet, indem verschiedene hilfreiche Verknüpfungen eingerichtet werden. Erstens existiert für jede Variable eine Kette all jener Positionen, an denen diese Variable vorkommt. Beispielsweise wird die Kette für die Variable B durch durchgezogene Linien dargestellt. Für jede Produktion gibt es einen Zähler für die Anzahl von Positionen, die Variable enthalten, deren Fähigkeit zur Erzeugung einer terminalen Zeichenreihe noch nicht berücksichtigt worden ist. Die gestrichelten Linien stellen die Verbindungen zwischen den Produktionen und ihren Zählern dar.

Die in Abbildung 7.8 dargestellten Zähler zeigen an, dass bislang noch gar keine Variablen berücksichtigt wurden, obwohl wir festgestellt haben, dass B erzeugend ist.

Angenommen, wir haben erkannt, dass B erzeugend ist. Wir gehen die Liste der Positionen in den Produktionsrümpfen durch, die B enthalten. Für jede dieser Positionen verringern wir den Zähler für diese Produktion um 1. Damit müssen wir von einer Position weniger zeigen, dass sie ein erzeugendes Symbole enthält, um schließen zu können, dass die Variable im Kopf auch erzeugend ist.

Wird der Zähler 0 erreicht, wissen wir, dass die Variable im Produktionskopf erzeugend ist. Eine durch gepunktete Linien dargestellte Verknüpfung führt uns zu der Variablen, und wir können diese Variable in eine Warteschlange für erzeugende Symbole einfügen, deren Auswirkungen noch untersucht werden müssen (wie wir es eben mit der Variablen B getan haben). Diese Warteschlange ist hier nicht dargestellt.

Wir müssen zeigen, dass dieser Algorithmus $O(n)$ Zeit erfordert. Folgende Punkte sind hierbei wichtig:

- Da es in einer Grammatik der Größe n höchstens n Variable gibt, erfordert die Erstellung und Initialisierung der Reihung $O(n)$ Zeit.
- Es gibt höchstens n Produktionen mit einer Gesamtlänge von höchstens n. Daher kann die Initialisierung der Verknüpfungen und Zähler, die in Abbildung 7.8 dargestellt sind, in $O(n)$ Zeit erfolgen.
- Wenn wir erkennen, dass eine Produktion den Zähler 0 hat (d. h. alle Positionen in ihrem Rumpf sind erzeugend), dann kann der damit verbundene Aufwand in zwei Kategorien eingeteilt werden:

    1. Arbeitsaufwand im Zusammenhang mit der Produktion selbst: erkennen, dass der Zähler 0 ist; ermitteln der Variablen im Produktionskopf, z. B. A; prüfen, ob diese Variable bereits als erzeugende Variable erkannt wurde, und einfügen dieser Variablen in die Warteschlange, falls sie noch nicht als erzeugend bekannt ist. Der Aufwand für jeden dieser Schritte ist 0(1), der Gesamtaufwand also höchstens 0(n).
    2. Arbeitsaufwand im Zusammenhang mit der Überprüfung der Positionen in den Produktionsrümpfen, deren Produktionskopf die Variable A ist. Der zur Verarbeitung aller generierenden Symbole erforderliche gesamte Arbeitsaufwand ist daher proportional zur Summe der Länge der Produktionsrümpfe, und diese beträgt $O(n)$.

Wir schließen daraus, dass der von diesem Algorithmus geleistete Gesamtarbeitsaufwand gleich $O(n)$ ist.

Andere Anwendungen für den linearen Test zur Erkennung leerer Sprachen

Der Trick, den wir in Abschnitt 7.4.3 verwendet haben und der in der Verwendung einer Datenstruktur und dem Buchführen über den Arbeitsfortschritt bestand, um festzustellen, ob eine Variable erzeugend ist, kann auch auf einige andere der in Abschnitt 7.1 genannten Tests angewendet werden. Zwei wichtige Beispiele sind:

1. Welche Symbole sind erreichbar?
2. Welche Symbole sind eliminierbar?

7.4.4 Die Zugehörigkeit zu einer kontextfreien Sprache prüfen

Wir können zudem die Frage entscheiden, ob eine Zeichenreihe w in einer kontextfreien Sprache L enthalten ist. Es gibt verschiedene ineffiziente Methoden zur Durchführung dieses Tests. Sie erfordern einen Zeitaufwand, der in Bezug auf $|w|$ exponentiell ist, wenn davon ausgegangen wird, dass eine Grammatik oder ein PDA für die Sprache L gegeben ist und deren Größe als Konstante unabhängig von w behandelt wird. Beispielsweise beginnen wir, indem wir die gegebene Repräsentation von L in eine CNF-Grammatik für L umwandeln. Da die Parsebäume für Grammatiken in Chomsky-Normalform Binärbäume sind, verfügt der Baum für eine Zeichenreihe w der Länge n über genau $2n - 1$ Knoten, die mit Variablen beschriftet sind (dieses Ergebnis lässt sich durch eine einfache Induktion beweisen, die wir Ihnen überlassen). Die Anzahl der möglichen Bäume und Knotenbeschriftungen ist daher exponentiell in n, sodass wir sie theoretisch alle auflisten und überprüfen können, ob einer bzw. eine von ihnen w ergibt.

Es gibt eine viel effizientere Technik, die auf dem Konzept der »dynamischen Programmierung« beruht, das Ihnen möglicherweise auch unter der Bezeichnung »Table-filling-Algorithmus« oder »Tabulation« bekannt sein mag. Dieser Algorithmus, der auch CYK-Algorithmus[5] genannt wird, beginnt mit einer CNF-Grammatik $G = (V, T, P, S)$ für die Sprache L. Der Algorithmus erhält als Eingabe eine Zeichenreihe $w = a_1a_2 \ldots a_n$ aus T^*. In $O(n^3)$ Zeit konstruiert der Algorithmus eine Tabelle, die darüber Aufschluss gibt, ob w in L enthalten ist. Beachten Sie, dass die Grammatik bei der Berechnung dieser Ausführungszeit als fest betrachtet wird und dass ihre Größe nur einen konstanten Faktor zur Ausführungszeit beiträgt, die bezogen auf die Länge der Zeichenreihe w gemessen wird, deren Zugehörigkeit zu L geprüft werden soll.

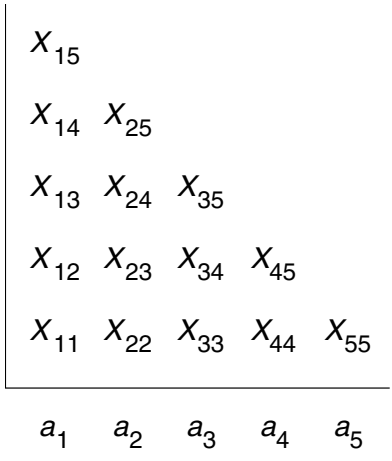
Im CYK-Algorithmus konstruieren wir eine dreieckige Tabelle, wie in Abbildung 7.9 dargestellt. Auf der horizontalen Achse werden die Positionen der Zeichenreihe $w = a_1a_2 \ldots a_n$ dargestellt, die in der Abbildung fünf Positionen umfasst. Bei dem Tabelleneintrag X_{ij} handelt es sich um die Menge der Variablen A, derart dass $A \stackrel{*}{\Rightarrow} a_ia_{i+1} \ldots a_j$. Beachten Sie besonders, dass uns interessiert, ob S in der Menge X_{1n} enthalten ist, weil dies der Aussage $S \stackrel{*}{\Rightarrow} w$, das heißt w ist in L enthalten, entspricht.

Um die Tabelle zu füllen, gehen wir zeilenweise von unten nach oben vor. Beachten Sie, dass jede Zeile Teilzeichenreihen gleicher Länge entspricht. Die unterste Zeile repräsentiert Zeichenreihen der Länge 1, die zweite Zeichenreihen der Länge 2 usw. bis zur obersten Zeile, die genau einer Teilzeichenreihe der Länge n entspricht, bei der es sich um w selbst handelt. Die Berechnung eines Tabelleneintrags erfordert nach der Methode, die wir als Nächstes erörtern werden, $O(n)$ Zeit. Da es $n(n + 1)/2$ Tabelleneinträge gibt, erfordert die gesamte Tabellenkonstruktion $O(n^3)$ Zeit. Der Algorithmus zur Berechnung der X_{ij} sieht so aus:

INDUKTIONSBEGINN: Wir berechnen die erste Zeile wie folgt. Da es sich bei der Zeichenreihe, die an der Position i beginnt und endet, um das terminale Symbol a_i handelt und die Grammatik in CNF vorliegt, kann die Zeichenreihe nur mithilfe einer Produktion der Form $A \to a$ abgeleitet werden. Folglich ist X_{ii} die Menge der Variablen A, derart dass $A \to a_i$ eine Produktion von G ist.

5. Er wurde nach den Personen benannt, die unabhängig voneinander dasselbe Konzept entdeckt haben: J. Cocke, D. Younger und T. Kasami.

Abbildung 7.9: Die vom CYK-Algorithmus konstruierte Tabelle



Induktionsschritt: Angenommen, wir möchten X_{ij} berechnen, das sich in Zeile $j - i + 1$ befindet, und wir haben alle X in den darunter liegenden Zeilen berechnet. Das heißt, wir wissen von allen Zeichenreihen, die kürzer als $a_i a_{i+1} \ldots a_j$ sind, und damit insbesondere von allen echten Präfixen und echten Suffixen dieser Zeichenreihe, ob sie in L enthalten sind. Da davon ausgegangen werden kann, dass $j - i > 0$ (da der Fall $i = j$ den Induktionsbeginn darstellt), wissen wir, dass jede Ableitung $A \stackrel{*}{\Rightarrow} a_i a_{i+1} \ldots a_j$ mit einem Schritt der Form $A \Rightarrow BC$ beginnen muss. B leitet dann ein Präfix von $a_i a_{i+1} \ldots a_j$, z. B. $B \stackrel{*}{\Rightarrow} a_i a_{i+1} \ldots a_k$ für $k < j$, ab und C muss den restlichen Teil von $a_i a_{i+1} \ldots a_j$ ableiten, d. h. $C \stackrel{*}{\Rightarrow} a_{k+1} a_{k+2} \ldots a_j$.

Wir schließen daraus, dass A in X_{ij} ist, wenn es die Variablen B und C und eine ganze Zahl k gibt, derart dass

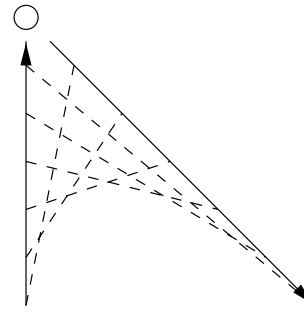
1. $i \leq k < j$.
2. B in X_{ik} enthalten ist.
3. C in X_{k+1j} enthalten ist.
4. $A \rightarrow BC$ eine Produktion von G ist.

Um solche Variablen A zu finden, müssen wir höchstens n Paare von zuvor berechneten Mengen untersuchen: $(X_{ii}, X_{i+1,j})$, $(X_{i,i+1}, X_{i+2,j})$ usw. bis $(X_{i,j-1}, X_{jj})$. Das Muster, nach dem wir die Spalte unter X_{ij} nach oben gehen, während wir gleichzeitig die Diagonale nach rechts unten gehen, ist in Abbildung 7.10 dargestellt.

Satz 7.33 Der oben beschriebene Algorithmus berechnet X_{ij} für alle i und j korrekt; folglich ist w genau dann in $L(G)$ enthalten, wenn S in X_{1n} enthalten ist. Zudem hat der Algorithmus eine Ausführungszeit von $O(n^3)$.

Beweis: Der Grund dafür, dass dieser Algorithmus die korrekten Mengen von Variablen findet, wurde dargelegt, als wir den induktiven Algorithmus eingeführt haben. Was die Ausführungszeit betrifft, berücksichtigen Sie, dass $O(n^2)$ Einträge berechnet

Abbildung 7.10: Schematische Darstellung, welche Paare für die Berechnung eines X_{ij} untersucht werden müssen



werden müssen, wobei für jeden Eintrag die Untersuchung von n Paaren erforderlich ist. Sie dürfen nicht vergessen, dass jede Menge X_{ij} zwar viele Variable enthalten kann, die Anzahl der Variablen der Grammatik G jedoch nicht von der Länge der Zeichenreihe w abhängt. Die Untersuchung eines Paars $(X_{ik}, X_{k+1,j})$ erfordert also einen Aufwand von $O(1)$. Da für X_{ij} höchstens n solcher Paare zu untersuchen sind, ist der Aufwand für seine Berechnung $O(n)$. Mithin ist der Gesamtaufwand für den CYK-Algorithmus $O(n^3)$. ■

Beispiel 7.34 Nachfolgend sind die Produktionen einer CNF-Grammatik G aufgeführt:

$$S \to AB \mid BC$$
$$A \to BA \mid a$$
$$B \to CC \mid b$$
$$C \to AB \mid a$$

Wir werden prüfen, ob die Zeichenreihe *baaba* in $L(G)$ enthalten ist. Abbildung 7.11 zeigt die ausgefüllte Tabelle für diese Zeichenreihe.

Zur Erstellung der ersten (untersten) Zeile verwenden wir die Induktionsbeginnregel. Wir müssen nur betrachten, welche Variablen den Produktionsrumpf *a* besitzen (die Variablen A und C) und welche Variablen den Produktionsrumpf *b* besitzen (nur die Variable B). Über den Positionen, die *a* enthalten, steht daher der Eintrag $\{A, C\}$ und über den Positionen, die *b* enthalten, steht $\{B\}$. Das heißt, $X_{11} = X_{44} = \{B\}$ und $X_{22} = X_{33} = X_{55} = \{A, C\}$.

In der zweiten Zeile stehen die Werte für X_{12}, X_{23}, X_{34} und X_{45}. Sehen wir uns an, wie z. B. X_{12} berechnet wird. Es gibt nur eine Möglichkeit, die Zeichenreihe aus den Positionen 1 und 2, die *ba* lautet, in zwei nichtleere Teilzeichenreihen aufzuteilen. Die erste Teilzeichenreihe muss Position 1 und die zweite Position 2 entsprechen. Damit eine Variable *ba* erzeugen kann, muss sie einen Rumpf besitzen, dessen erste Variable in $X_{11} = \{B\}$ (d. h. das *b* erzeugt) und dessen zweite Variable in $X_{22} = \{A, C\}$ (d. h. das *a* erzeugt) enthalten ist. Dieser Rumpf kann lediglich BA oder BC lauten. Wenn wir die Grammatik betrachten, finden wir heraus, dass nur die Produktionen $A \to BA$ und $S \to BC$ diese Rümpfe aufweisen. Folglich bilden die beiden Köpfe A und S X_{12}.

Abbildung 7.11: Die vom CYK-Algorithmus erstellte Tabelle für die Zeichenreihe *baaba*

Ein komplexeres Beispiel stellt die Berechnung von X_{24} dar. Wir können die Zeichenreihe *aab*, die die Positionen 2 bis 4 belegt, aufteilen, indem wir die erste Zeichenreihe nach Position 2 oder 3 enden lassen. Das heißt, wir können in der Definition von X_{24} $k = 2$ oder $k = 3$ wählen. Folglich müssen wir alle Produktionsrümpfe der Menge $X_{22}X_{34} \cup X_{23}X_{44}$ betrachten. Diese Menge von Zeichenreihen lautet $\{A, C\}\{S, C\} \cup \{B\}\{B\} = \{AS, AC, CS, CC, BB\}$. Von den fünf Zeichenreihen dieser Menge stellt nur $\{CC\}$ einen Produktionsrumpf dar und dessen Kopf lautet B. Somit ist $X_{24} = \{B\}$. ■

7.4.5 Vorschau auf unentscheidbare kfL-Probleme

In den nächsten Kapiteln werden wir eine bemerkenswerte Theorie entwickeln, mit der wir formal beweisen können, dass es Probleme gibt, die wir mit keinem Algorithmus, der auf einem Rechner ausgeführt werden kann, lösen können. Wir werden damit zeigen, dass für eine Reihe von einfach zu formulierenden Fragen zu Grammatiken und kontextfreien Sprachen, die als »unentscheidbare Probleme« bezeichnet werden, keine Algorithmen existieren. Vorerst müssen wir uns mit einer Liste der bedeutendsten unentscheidbaren Fragen zu kontextfreien Grammatiken und Sprachen begnügen. Die folgenden Fragen sind unentscheidbar:

1. Ist eine gegebene kontextfreie Grammatik G mehrdeutig?
2. Ist eine gegebene kontextfreie Sprache inhärent mehrdeutig?
3. Ist der Durchschnitt zweier kfLs leer?
4. Sind zwei kontextfreie Sprachen identisch?
5. Ist eine gegebene kfG identisch mit Σ^*, wobei Σ für das Alphabet der Sprache steht?

Beachten Sie, dass sich Frage (1) zur Mehrdeutigkeit insofern von den übrigen Fragen unterscheidet, als hier von einer Grammatik und keiner Sprache die Rede ist. Alle

anderen Fragen setzen voraus, dass die Sprache durch eine Grammatik oder einen PDA repräsentiert wird, die Frage betrifft jedoch die durch die Grammatik bzw. den PDA definierte(n) Sprache(n). Im Gegensatz zu Frage (1) läuft die zweite Frage im Wesentlichen darauf hinaus, ob es für eine gegebene Grammatik G (oder einen entsprechenden PDA) eine äquivalente Grammatik G' gibt, die eindeutig ist. Wenn G selbst eindeutig ist, dann lautet die Antwort mit Sicherheit »Ja«. Ist G jedoch mehrdeutig, dann könnte es eine andere Grammatik G' für dieselbe Sprache geben, die eindeutig ist, wie wir an der Grammatik für einfache Ausdrücke aus Beispiel 5.27 gesehen haben.

7.4.6 Übungen zum Abschnitt 7.4

Übung 7.4.1 Geben Sie Algorithmen an, mit denen die folgenden Fragen entschieden werden können:

* a) Ist $L(G)$ für eine gegebene kfG G endlich? *Hinweis*: Verwenden Sie das Pumping-Lemma.

! b) Enthält $L(G)$ für eine gegebene kfG G mindestens 100 Zeichenreihen?

!! c) Wenn eine kfG G und eine ihrer Variablen A gegeben sind, gibt es dann eine Satzform, in der A das erste Symbol bildet? *Hinweis*: Denken Sie daran, dass A zuerst in der Mitte einer Satzform vorkommen und dann alle links davon stehenden Symbole ε ableiten kann.

Übung 7.4.2 Entwickeln Sie mithilfe der in Abschnitt 7.4.3 beschriebenen Technik in linearer Zeit ausführbare Algorithmen für die folgenden Fragen zu kontextfreien Grammatiken:

a) Welche Symbole kommen in den Satzformen vor?

b) Welche Symbole sind eliminierbar (leiten ε ab)?

Übung 7.4.3 Entscheiden Sie mithilfe des CYK-Algorithmus und unter Verwendung der Grammatik G aus dem Beispiel 7.34, ob jede der folgenden Zeichenreihen in $L(G)$ enthalten ist:

* *a) ababa*

b) *baaab*

c) aabab

* **Übung 7.4.4** Zeigen Sie, dass in jeder CNF-Grammatik sämtliche Parsebäume für Zeichenreihen der Länge n $2n - 1$ innere Knoten haben (d. h. $2n - 1$ Knoten, deren Beschriftung aus einer Variablen besteht).

! **Übung 7.4.5** Modifizieren Sie den CYK-Algorithmus, sodass er die Anzahl verschiedener Parsebäume für die gegebene Eingabe ermittelt und nicht einfach nur feststellt, ob eine Eingabe in der Sprache enthalten ist.

7.5 Zusammenfassung von Kapitel 7

- *Elimination unnützer Symbole* : Eine Variable kann aus einer kfG entfernt werden, wenn sie keine Zeichenreihe aus terminalen Symbolen erzeugt und nicht in mindestens einer Zeichenreihe vorkommt, die vom Startsymbol abgeleitet ist. Um solche unnützen Symbole korrekt zu eliminieren, müssen wir zuerst ermitteln, ob eine Variable eine terminale Zeichenreihe ableitet, und dann alle Variablen, die dies nicht tun, zusammen mit all ihren Produktionen entfernen. Nur dann können wir Variable eliminieren, die nicht vom Startsymbol ableitbar sind.
- *Elimination von ε- und Einheitsproduktionen:* Wenn eine kfG gegeben ist, die mindestens eine nichtleere Zeichenreihe erzeugt, dann können wir eine kfG finden, die dieselbe Sprache mit Ausnahme der Zeichenreihe ε erzeugt, jedoch weder ε-Produktionen (Produktionen mit ε als Rumpf) noch Einheitsproduktionen (Produktionen mit einer einzelnen Variablen als Rumpf) besitzt.
- *Chomsky-Normalform*: Wenn eine kfG gegeben ist, die mindestens eine nichtleere Zeichenreihe erzeugt, dann können wir eine kfG finden, die dieselbe Sprache mit Ausnahme der Zeichenreihe ε erzeugt und Chomsky-Normalform hat. Das heißt, sie enthält keine unnützen Symbole, und jeder Produktionsrumpf besteht entweder aus zwei Variablen oder einem terminalen Symbol.
- *Pumping-Lemma*: In jeder kontextfreien Sprache ist es möglich, in jeder ausreichend langen Zeichenreihe der Sprache eine kurze Teilzeichenreihe zu finden, derart dass ein Präfix v und ein Suffix x dieser Zeichenreihe simultan beliebig oft wiederholt werden können. Die zu wiederholenden Zeichenreihen dürfen nicht beide gleich ε sein. Dieses Lemma sowie eine mächtigere Variante namens Odgens Lemma, die in Übung 7.2.3 erwähnt wird, ermöglichen uns bei vielen Sprachen den Beweis, dass sie nicht kontextfrei sind.
- *Operationen, bezüglich derer kontext freie Sprachen abgeschlossen sind*: Die kontextfreien Sprachen sind abgeschlossen bezüglich Substitution, Vereinigung, Verkettung, Hüllenbildung (Sternoperator), Spiegelung und inverser Homomorphismen. Kontextfreie Sprachen sind bezüglich Durchschnitt und Komplementbildung nicht abgeschlossen. Allerdings ist der Durchschnitt aus einer kontextfreien Sprache und einer regulären Sprache stets eine kontextfreie Sprache.
- *Prüfen, ob eine kontextfreie Sprache leer ist* : Es gibt einen Algorithmus, mit dem ermittelt werden kann, ob eine gegebene kontextfreie Grammatik Zeichenreihen generiert. Eine sorgfältige Implementierung ermöglicht es, dass dieser Test mit einem zur Größe der Grammatik proportionalen Aufwand durchgeführt wird.
- *Zugehörigkeit zu einer kfL prüfen* : Der Cocke-Younger-Kasami-Algorithmus gibt darüber Aufschluss, ob eine gegebene Zeichenreihe in einer gegebenen kontextfreien Sprache enthalten ist. Für eine feste kfL erfordert dieser Test $O(n^3)$ Zeit, wenn n die Länge der zu prüfenden Zeichenreihe ist.

LITERATURANGABEN ZU KAPITEL 7

Die Chomsky-Normalform stammt aus [2]. Die Greibach-Normalform ist [4] entnommen, aber die in Übung 7.1.11 skizzierte Konstruktion stammt von M. C. Paull.

Viele der grundlegenden Eigenschaften kontextfreier Sprachen wurden aus [1] übernommen. Zu diesen Konzepten gehören das Pumping-Lemma, grundlegende Eigenschaften der Abgeschlossenheit und Tests für einfache Fragen, wie z. B. ob eine kontextfreie Sprache leer oder endlich ist. [6] ist die Quelle für die Feststellung, dass kontextfreie Sprachen unter Durchschnitt und Komplementbildung nicht abgeschlossen sind. Zudem werden in [3] weitere Ergebnisse hinsichtlich der Abgeschlossenheit dargestellt, einschließlich der Abgeschlossenheit von kontextfreien Sprachen bezüglich inverser Homomorphismen. Ogdens Lemma stammt aus [5].

Der CYK-Algorithmus hat drei voneinander unabhängige Quellen. J. Cockes Arbeit wurde privat weitergegeben und nie publiziert. T. Kasamis Beschreibung des praktisch identischen Algorithmus erschien lediglich in einem internen Memorandum der U.S. Air Force. Allerdings wurde die Arbeit von D. Younger in konventioneller Weise veröffentlicht [7].

1. Y. Bar-Hillel, M. Persel und E. Shamir [1961]. »On formal properites of simple phase-structure grammars«, *Z. Phonetik. Sprachwiss. Kommnikationsforsch.* **14**, S. 143–172.
2. N. Chomsky [1959]. »On certain formal properties of grammars«, *Information and Control* **2**:2, S. 137–167.
3. S. Ginsburg und G. Rose [1963]. »Operations which preserve definability in languages«, *J. ACM* **10**:2, S. 175–195.
4. S. A. Greibach [1965]. »A new normal-form theorem for context-free phrase structure grammars«, *J. ACM* **12**:1, S, 42–52.
5. W. Ogden [1969]. »A helpful result for proving inherent ambiguity«, *Mathematical Systems Theory* **2**:3, S. 31–42.
6. S. Scheinberg [1960]. »Note on the boolean properties of context-free languages«, *Information and Control* **3**:4, S. 372–375.
7. D. H. Younger [1967]. »Recognition and parsing of context-free languages in time n^3«, *Information and Control* **10**:2, S. 372–375.

KAPITEL 8

Einführung in Turing-Maschinen

In diesem Kapitel schlagen wir eine andere Richtung ein. Bisher waren wir primär an einfachen Sprachklassen und den Möglichkeiten interessiert, wie diese bei verhältnismäßig eingeschränkten Problemen, wie etwa der Analyse von Protokollen, dem Durchsuchen von Texten oder der Konstruktion von Parsebäumen zu Programmen, einsetzbar sind. Nun wenden wir uns zunächst der Frage zu, welche Sprachen von einem beliebigen Rechengerät definiert werden können. Diese Frage ist gleichbedeutend mit der Frage, welche Aufgaben Computer ausführen können, denn das Erkennen der in einer Sprache enthaltenen Zeichenreihen ist eine formale Möglichkeit, beliebige Probleme auszudrücken, und das Lösen eines Problems ist ein vollkommenes Substitut für das, was Computer leisten.

Wir beginnen mit einem informellen Argument und setzen Grundkenntnisse in der C-Programmierung voraus. Wir zeigen, dass es bestimmte Probleme gibt, die wir nicht mithilfe eines Computers lösen können. Diese Probleme werden »unentscheidbar« genannt. Danach stellen wir einen berühmten Formalismus für Computer vor, die so genannte Turing-Maschine. Eine solche Turing-Maschine besitzt zwar keinerlei Ähnlichkeit mit einem PC und wäre äußerst ineffizient, wenn sich ein Startup-Unternehmen deren Bau und Verkauf zum Ziel setzen würde, doch die Turing-Maschine dient schon seit langem als akkurates Modell dafür, wozu ein physikalisches Rechengerät in der Lage ist.

In Kapitel 9 entwickeln wir anhand der Turing-Maschine eine Theorie »unentscheidbarer« Probleme. Solche Probleme können von keinem Computer gelöst werden. Wir zeigen, dass einige Probleme zwar einfach auszudrücken, jedoch tatsächlich nicht lösbar sind. Dazu gehört beispielsweise die Beantwortung der Frage, ob eine gegebene Grammatik mehrdeutig ist. Sie werden noch weitere Probleme dieser Art kennen lernen.

8.1 Probleme, die nicht von Computern gelöst werden können

Dieser Abschnitt soll eine informelle, auf der C-Programmierung basierende Einführung des Beweises bieten, dass ein spezielles Problem nicht von Computern gelöst werden kann. Wir erörtern das Problem, ob ein C-Programm als Erstes `hello, world` ausgibt. Wir können uns zwar vorstellen, dass uns die Simulation des Programms ermöglicht, dessen Funktion zu beschreiben, doch in der Realität müssen wir mit Programmen umgehen, die unvorstellbar lange dazu brauchen, überhaupt etwas auszu-

geben. Dieses Problem – nicht zu wissen, wann etwas, falls überhaupt, geschieht – ist letztendlich die Ursache für unsere Unfähigkeit, die Arbeitsweise eines Programms zu beschreiben. Allerdings ist der formale Beweis, dass es kein Programm gibt, das eine gegebene Aufgabe erledigt, relativ kompliziert, und wir müssen daher einige formale Mechanismen entwickeln. Dieser Abschnitt erläutert die Absicht, die hinter den formalen Beweisen steht.

8.1.1 Programme, die »Hello, World« ausgeben

Listing 8.1 zeigt das erste C-Programm, auf das die Leser des Klassikers[1] von Kernighan und Ritchie treffen. Es ist leicht erkennbar, dass dieses Programm `hello, world` ausgibt und dann endet. Dieses Programm ist so transparent, dass es inzwischen allgemein üblich ist, Sprachen anhand eines Programms vorzustellen, das `hello, world` ausgibt.

Listing 8.1: Das »Hello, World«-Programm von Kernighan und Ritchie

```
main()
{
    printf("hello, world\n");
}
```

Es gibt aber auch andere Programme, die ebenfalls `hello, world` ausgeben, obwohl dies weit weniger offensichtlich ist. Listing 8.2 zeigt ein weiteres Programm, das `hello, world` ausgeben könnte. Dieses Programm verarbeitet die Eingabe n und sucht nach positiven ganzzahligen Lösungen der Gleichung $x^n + y^n = z^n$. Wird eine Lösung gefunden, gibt das Programm `hello, world` aus. Findet das Programm allerdings keine Ganzzahlen x, y und z, die die Gleichung erfüllen, setzt es die Suche unendlich lange fort und gibt niemals `hello, world` aus.

Zum Verständnis der Funktionsweise dieses Programms beachten Sie zunächst die Hilfsfunktion `exp`, die einen Exponentialausdruck berechnet. Das Hauptprogramm muss die Tripel (x, y, z) in einer Reihenfolge durchsuchen, die garantiert, dass jedes Tripel positiver Ganzzahlen berücksichtigt wird. Um die Suche entsprechend zu organisieren, verwenden wir eine vierte Variable, `total`, die mit `3` beginnt, in der `while`-Schleife jeweils um `1` erhöht wird und somit jeden endlichen Integerwert annimmt. Innerhalb der `while`-Schleife teilen wir `total` in drei positive Ganzzahlen x, y und z auf, wobei x zunächst Werte von `1` bis zu `total-2` und y innerhalb dieser for-Schleife Werte von `1` bis zu `total-x-1` annehmen. Was übrig bleibt, also Werte zwischen `1` und `total-2`, wird z zugewiesen.

In der innersten Schleife wird das Tripel (x, y, z) daraufhin geprüft, ob $x^n + y^n = z^n$. Ist dem so, gibt das Programm `hello, world` aus, im anderen Fall dagegen nichts.

Liest das Programm für n den Wert 2, dann findet es Kombinationen wie etwa `total` = 12, $x = 3$, $y = 4$ und $z = 5$, für die $x^n + y^n = z^n$ gilt. Beim Eingabewert 2 gibt das Programm daher `hello, world` aus.

1. B. W. Kernighan und D. M. Ritchie [1978]. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.

Listing 8.2: Fermats letzter Satz als »Hello, World«-Programm

```
int exp(int i, n)
/* berechnet i hoch n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return (ans);
}

main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp (z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

Für Ganzzahlen > 2 wird das Programm allerdings kein Tripel positiver Ganzzahlen finden, die die Gleichung $x^n + y^n = z^n$ erfüllen, und daher niemals `hello, world` ausgeben. Interessanterweise hat man erst vor einigen Jahren herausgefunden, dass dieses Programm für einen größeren ganzzahligen Wert n `hello, world` ausgibt. Die Behauptung, dass keine Ausgabe erfolgt, also dass die Gleichung $x^n + y^n = z^n$ für $n > 2$ keine ganzzahlige Lösung besitzt, stellte Fermat vor 300 Jahren auf, doch erst vor kurzem wurde ein Beweis gefunden. Diese Behauptung wird häufig »Fermats letzter Satz« genannt.

Wir definieren das »Hello, World«-Problem wie folgt: Es soll festgestellt werden, ob ein gegebenes C-Programm bei gegebener Eingabe als erste zwölf Ausgabezeichen `hello, world` ausgibt. Im Folgenden verwenden wir für die Behauptung über ein Programm, dass es `hello, world` als die ersten zwölf Zeichen ausgibt, häufig die Kurzform, dass es `hello, world` ausgibt.

Die Tatsache, dass Mathematiker 300 Jahre brauchten, um eine Frage hinsichtlich eines einzelnen Programms mit 22 Zeilen zu beantworten, scheint darauf hinzudeuten, dass das allgemeine Problem der Bestimmung, ob ein gegebenes Programm mit einer gegebenen Eingabe `hello, world` ausgibt, sehr schwierig sein muss. Tatsächlich kann jedes Problem, das von Mathematikern bisher nicht gelöst werden konnte, in eine Frage der Form »Gibt dieses Programm mit dieser Eingabe `hello, world` aus?« umgewandelt werden. Es wäre also wirklich bemerkenswert, wenn wir ein Pro-

gramm schreiben könnten, das jedes Programm *P* mit der Eingabe *I* für *P* untersuchen und bestimmen könnte, ob *P* mit *I* als Eingabe `hello, world` ausgibt. Wir werden beweisen, dass es ein solches Programm nicht gibt.

Warum es unentscheidbare Probleme geben muss

Obwohl der Beweis schwierig ist, dass ein bestimmtes Problem, wie etwa das hier vorgestellte »Hello, World«-Problem, unentscheidbar sein muss, ist leicht feststellbar, warum fast alle Probleme für ein System, das Programmierung erfordert, unentscheidbar sind. Sie erinnern sich, dass ein »Problem« eigentlich gleichbedeutend ist mit der Zugehörigkeit einer Zeichenreihe zu einer Sprache. Die Anzahl unterschiedlicher Sprachen über einem beliebigen, mehr als ein Symbol umfassenden Alphabet ist nicht abzählbar. Daher besteht keine Möglichkeit, den Sprachen ganzzahlige Werte zuzuweisen, sodass jeder Sprache eine ganze Zahl und jeder ganzen Zahl eine Sprache zugeordnet ist.

Andererseits *sind* Programme als endliche Zeichenreihen über einem endlichen Alphabet (normalerweise einer Teilmenge des ASCII-Alphabets) abzählbar. Wir können sie also nach der Länge und bei Programmen mit gleicher Länge lexikografisch ordnen. Daher können wir vom ersten Programm, vom zweiten Programm und allgemein vom *i*-ten Programm für jeden ganzzahligen Wert *i* reden.

Daraus folgt die Erkenntnis, dass es unendlich viel weniger Programme als Probleme gibt. Wählten wir zufällig eine Sprache, dann wäre sie mit sehr hoher Wahrscheinlichkeit ein unentscheidbares Problem. Der einzige Grund dafür, dass die meisten Probleme entscheidbar *erscheinen*, liegt darin, dass wir nur selten an zufälligen Problemen interessiert sind. Stattdessen neigen wir zu eher einfachen, gut strukturierten Problemen, die tatsächlich häufig entscheidbar sind. Jedoch gibt es auch unter den Problemen, die uns interessieren und klar und prägnant darstellbar sind, viele unentscheidbare Probleme; das »Hello, World«-Problem ist ein typisches Beispiel.

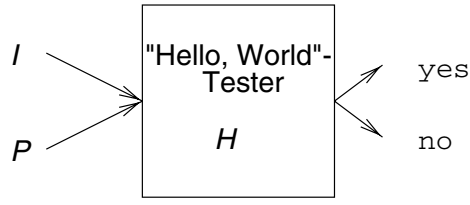
8.1.2 Der hypothetische »Hello, World«-Tester

Die Unmöglichkeit des »Hello, World«-Tests wird mit einem Beweis durch Widerspruch belegt. Angenommen, es gibt ein Programm *H*, das als Eingabe ein Programm *P* sowie die Eingabe *I* erhält und ausgibt, ob *P* bei der Eingabe *I* die Ausgabe `hello, world` liefert. Abbildung 8.3 zeigt die Arbeitsweise von *H*. *H* gibt entweder die drei Zeichen `yes` oder die beiden Zeichen `no` aus. Es produziert stets eine dieser beiden Ausgaben.

Gibt es für ein Problem einen Algorithmus wie *H*, der immer korrekt feststellt, ob eine Instanz des Problems die Antwort »yes« oder »no« besitzt, dann wird das Problem »entscheidbar« genannt. Andernfalls ist das Problem »unentscheidbar«. Wir wollen beweisen, dass *H* nicht existiert, dass das »Hello, World«-Problem also unentscheidbar ist.

Um diese Aussage durch einen Beweis durch Widerspruch zu belegen, werden wir einige Änderungen an *H* vornehmen und damit ein weiteres Programm H_2 erstellen, von dem wir zeigen, dass es nicht existiert. Da es sich bei den Änderungen an *H* um einfache Umformungen handelt, die an jedem C-Programm vorgenommen werden können, ist die Existenz von *H* die einzig fragliche Aussage, und damit haben wir diese Aussage widerlegt.

Abbildung 8.3: Das hypothetische Programm *H* als »Hello, World«-Tester

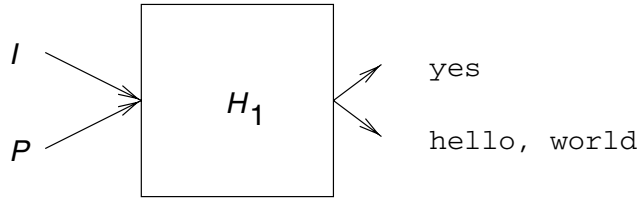


Zur Vereinfachung gehen wir von einigen Annahmen über C-Programme aus. Diese Annahmen erleichtern die Arbeitsweise von *H*, ohne die Klasse der behandelbaren Probleme einzuschränken. Wir nehmen Folgendes an:

1. Die gesamte Ausgabe ist textorientiert. Wir verwenden kein Grafikpaket oder andere Möglichkeiten, um etwas anderes als Zeichen auszugeben.
2. Die gesamte textorientierte Ausgabe wird mit `printf` statt mit `putchar()` oder einer anderen textorientierten Ausgabefunktion ausgeführt.

Nun nehmen wir an, dass das Programm *H* existiert. Zuerst ändern wir die Ausgabe `no`, also die Antwort von *H*, wenn dessen Eingabeprogramm *P* auf die Eingabe *I* hin nicht als erste Ausgabe `hello, world` liefert. Sobald *H* »n« ausgibt, wissen wir, dass danach das »o« folgt[2]. Wir können daher in *H* jede `printf`-Anweisung , die »n« ausgibt, ändern, sodass sie stattdessen `hello, world` druckt. Eine zweite `printf`-Anweisung, die zwar »o«, nicht aber »n« ausgibt, wird weggelassen. Daraus folgt, dass sich das neue Programm, das wir H_1 nennen, wie *H* verhält, jedoch genau dann `hello, world` ausgibt, wenn *H* `no` ausgeben würde. Abbildung 8.4 zeigt H_1.

Abbildung 8.4: H_1 verhält sich wie *H*, gibt jedoch `hello, world` statt `no` aus



Unsere nächste Änderung am Programm *H* ist etwas raffinierter. Es handelt sich im Wesentlichen um die Einsicht, die Alan Turing den Beweis der Unentscheidbarkeit bestimmter Probleme auf Turing-Maschinen ermöglichte. Da wir an Programmen

2. Wahrscheinlich wird das Programm `no` mit einem einzigen `printf` ausgeben, doch es könnte auch »n« mit einer und »o« mit einer anderen `printf`-Anweisung ausgeben.

interessiert sind, die andere Programme als Eingabe erhalten und Aussagen zu diesen Programmen machen, schränken wir H_1 folgendermaßen ein:

a) H_1 akzeptiert nur die Eingabe *P*, nicht jedoch *P* und *I*.

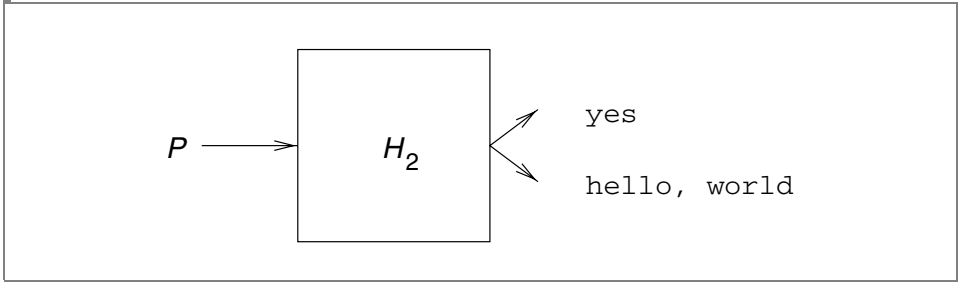
b) H_1 fragt, wie sich *P* verhalten würde, wenn es den eigenen Code als Eingabe erhielte. Das heißt, wie verhält sich H_1 bei einer Eingabe von *P* als Programm und von *P* als Eingabe *I*?

Wir müssen die folgenden Änderungen an H_1 vornehmen, um das in Abbildung 8.5 dargestellte Programm H_2 zu erhalten:

1. H_2 liest zuerst die gesamte Eingabe *P* und speichert sie in einer Reihung *A*, die zu diesem Zweck (mit `malloc`) reserviert wird.[3]

2. H_2 simuliert H_1, doch immer dann, wenn H_1 Eingaben aus *P* oder *I* lesen würde, liest H_2 aus der in *A* gespeicherten Kopie. Zur Kontrolle, wie viel H_1 von *P* und *I* gelesen hat, kann H_2 zwei Zeiger verwenden, die Positionen in *A* markieren.

Abbildung 8.5: H_2 verhält sich wie H_1, verwendet jedoch die Eingabe *P* sowohl für *P* als auch für *I*



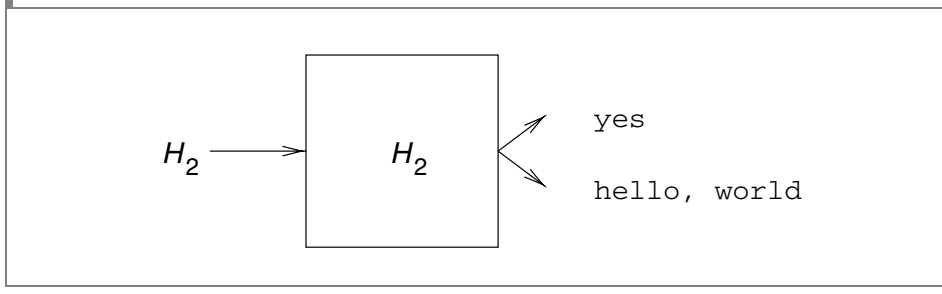
Wir beweisen nun, dass es kein H_2 geben kann. Folglich kann es auch H_1 und entsprechend *H* nicht geben. Der Kern der Argumentation beruht auf der Vorstellung, wie H_2 arbeitet, wenn es sich selbst als Eingabe erhält. Diese Situation stellt Abbildung 8.6 dar. Sie wissen, dass H_2 `yes` ausgibt, wenn es ein beliebiges Programm *P* als Eingabe erhält, das `hello, world` ausgibt, wenn es sich selbst als Eingabe erhält. Ebenso gibt H_2 `hello, world` aus, falls *P* mit sich selbst als Eingabe als erste Ausgabe nicht `hello, world` liefert.

Angenommen, das im Quadrat in Abbildung 8.6 dargestellte Programm H_2 gibt `yes` aus. Dann sagt H_2 im Quadrat über seine Eingabe H_2 aus, dass H_2 mit sich selbst als Eingabe `hello, world` als erste Ausgabe produziert. Jedoch haben wir gerade vorausgesetzt, dass H_2 in dieser Situation als Erstes die Ausgabe `yes` statt `hello, world` produziert.

Also nehmen wir an, das Quadrat in Abbildung 8.6 gibt `hello, world` aus, da eine der beiden Ausgaben erfolgen muss. Wenn H_2 jedoch mit sich selbst als Eingabe zuerst

3. Die Unix-Systemfunktion `malloc` reserviert einen Speicherblock der im Aufruf von `malloc` angegebenen Größe. Diese Funktion wird eingesetzt, wenn die notwendige Speichergröße erst zur Programmlaufzeit ersichtlich ist, beispielsweise beim Einlesen einer Eingabe beliebiger Länge. Normalerweise wird `malloc` mehrmals aufgerufen, falls immer mehr Eingabedaten gelesen werden und zunehmend mehr Speicher benötigt wird.

Abbildung 8.6: Wie verhält sich H_2, wenn die Eingabe aus H_2 selbst besteht?



`hello, world` ausgibt, dann muss die Ausgabe des Quadrats in Abbildung 8.6 aus `yes` bestehen. Gleichgültig, welche Ausgabe wir H_2 unterstellen, können wir also stets zeigen, dass die andere Ausgabe erfolgt.

Diese Situation ist paradox, und wir schließen daraus, dass es das Programm H_2 nicht geben kann. Somit haben wir die Annahme widerlegt, dass H existiert. Wir haben also bewiesen, dass es kein Programm H geben kann, das für jedes Programm P und jede Eingabe I entscheidet, ob P mit der Eingabe I als Erstes die Ausgabe `hello, world` produziert oder nicht.

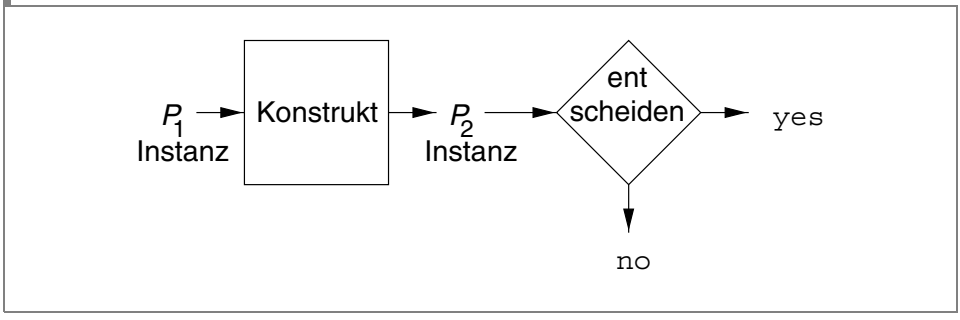
8.1.3 Ein Problem auf ein anderes Problem reduzieren

Wir haben nun ein Problem (gibt ein gegebenes Programm mit gegebener Eingabe als Erstes `hello, world` aus?) von dem wir wissen, dass es von keinem Computerprogramm in Allgemeinheit gelöst werden kann. Ein solches Problem, das nicht von Computern lösbar ist, wird *unentscheidbar* genannt. Die formale Definition der Unentscheidbarkeit finden Sie in Abschnitt 9.3, wir verwenden den Begriff zunächst jedoch informell. Angenommen, wir möchten bestimmen, ob ein Problem von einem Computer gelöst werden kann. Wir könnten natürlich versuchen, ein Programm zur Problemlösung zu schreiben. Doch wenn wir nicht wissen, wie wir dies angehen sollen, könnten wir zu beweisen versuchen, dass es ein solches Programm nicht gibt.

Vielleicht könnten wir mit einer ähnlichen Technik wie beim »Hello, World«-Problem beweisen, dass dieses neue Problem unentscheidbar ist: Wir nehmen an, es gibt ein Programm zur Lösung des Problems und entwickeln ein paradoxes Programm, das ähnlich dem Programm H_2 auf einen unauflösbaren Widerspruch führt. Wissen wir allerdings von einem Problem, dass es unentscheidbar ist, dann müssen wir für ein neues Problem nicht unbedingt die Existenz einer paradoxen Situation beweisen. Es ist ausreichend, Folgendes zu zeigen: Könnten wir das neue Problem lösen, dann wäre es uns möglich, diese Lösung einsetzen, um das Problem zu lösen, von dem wir schon wissen, dass es unentscheidbar ist. Abbildung 8.7 zeigt diese Strategie; die Technik wird *Reduktion* von P_1 auf P_2 genannt.

Angenommen, wir wissen, dass das Problem P_1 unentscheidbar ist, und wollen beweisen, dass das neue Problem P_2 ebenfalls unentscheidbar ist. Wir nehmen an, es gibt ein Programm, wie in Abbildung 8.7 durch die Raute »Entscheiden« dargestellt,

Abbildung 8.7: Können wir Problem P_2 lösen, dann könnten wir dessen Lösung zum Lösen von Problem P_1 verwenden



das `yes` oder `no` ausgibt, abhängig davon, ob die Eingabeinstanz des Problems P_2 in der Sprache dieses Problems enthalten ist.[4]

Zum Beweis, dass Problem P_2 unentscheidbar ist, müssen wir eine Konstruktion einführen, die in Abbildung 8.7 durch das quadratische Feld repräsentiert wird. Diese Konstruktion konvertiert Instanzen von P_1 in Instanzen von P_2, welche die gleiche Antwort besitzen. Das bedeutet, dass jede Zeichenreihe der Sprache P_1 in eine Zeichenreihe der Sprache P_2 umgewandelt wird, jede Zeichenreihe über dem Alphabet von P_1, die *nicht* in der Sprache von P_1 enthalten ist, dagegen in eine Zeichenreihe, die nicht in der Sprache P_2 enthalten ist. Sobald wir dieses Konstrukt besitzen, können wir das Entscheidbarkeitsproblem für P_2 wie folgt lösen:

1. Gegeben sei eine Instanz von P_1, also eine Zeichenreihe w, die in der Sprache P_1 enthalten sein kann oder auch nicht. Wende den Konstruktionsalgorithmus an, um eine Zeichenreihe x zu produzieren.
2. Teste, ob x in P_2 enthalten ist, und gib die gleiche Antwort für w und P_1 an.

Ist x in P_2 enthalten, dann ist w in P_1 enthalten, und der Algorithmus liefert die Antwort `yes`. Ist x nicht in P_2 enthalten, dann ist auch w nicht in P_1 enthalten, und der Algorithmus gibt `no` aus. In jedem Fall wird die Wahrheit über w ausgesagt. Da wir angenommen haben, dass kein Algorithmus existiert, der über die Zugehörigkeit aller Zeichenreihen zu P_1 entscheidet, haben wir mit einem Beweis durch Widerspruch gezeigt, dass der hypothetische Entscheidungsalgorithmus für P_2 nicht existiert. P_2 ist daher unentscheidbar.

Beispiel 8.1 Wir verwenden diese Methode, um zu zeigen, dass die folgende Frage unentscheidbar ist: »Ruft ein Programm Q mit gegebener Eingabe y jemals die Funktion `foo` auf?« Beachten Sie, dass in Q unter Umständen keine Funktion `foo` enthalten ist, wobei das Problem in diesem Fall einfach ist. Schwieriger wird es, wenn Q eine Funktion `foo` besitzt und fraglich ist, ob Q mit der Eingabe y einen Aufruf von `foo` erreicht. Da wir nur ein unentscheidbares Problem kennen, wird die Rolle von P_1 in

4. Sie wissen, dass es sich bei einem Problem eigentlich um eine Sprache handelt. Als wir vom Problem des Entscheidens sprachen, ob ein gegebenes Programm mit gegebener Eingabe als Erstes die Ausgabe `hello, world` produziert, sprachen wir eigentlich von Zeichenreihen, die aus einem C-Quellprogramm, gefolgt von einer Eingabedatei/Eingabedateien, die das Programm liest/lesen, bestehen. Diese Menge von Zeichenreihen ist eine Sprache aus dem Alphabet der ASCII-Zeichen.

Abbildung 8.7 vom »Hello, World«-Problem übernommen. P_2 entspricht dem oben beschriebenen *»foo-Aufruf«-Problem* . Wir nehmen an, es gibt ein Programm, das das »foo-Aufruf«-Problem löst. Wir müssen einen Algorithmus entwerfen, der das »Hello, World«-Problem in das »foo-Aufruf«-Problem konvertiert.
Für ein gegebenes Programm Q und dessen Eingabe y müssen wir also ein Programm R und eine Eingabe z konstruieren, sodass R mit der Eingabe z `foo` wirklich nur dann aufruft, wenn Q mit der Eingabe y `hello, world` ausgibt. Diese Konstruktion ist nicht schwierig:

1. Enthält Q eine Funktion namens `foo`, dann benennen wir diese Funktion sowie alle Aufrufe der Funktion um. Es ist offensichtlich, dass das neue Programm Q_1 die gleiche Funktionalität wie Q besitzt.
2. Wir fügen zu Q_1 eine Funktion `foo` hinzu. Diese Funktion führt nichts aus und wird nicht aufgerufen. Daraus resultiert das Programm Q_2.
3. Wir ändern Q_2, sodass die ersten zwölf Zeichen der Ausgabe in einer globalen Reihung A gespeichert werden. Daraus resultiert das Programm Q_3.
4. Wir ändern Q_3, sodass bei jeder Ausführung einer Ausgabeanweisung die Reihung A daraufhin geprüft wird, ob zwölf oder mehr Zeichen geschrieben wurden. Ist dem so, wird geprüft, ob die zwölf Zeichen `hello, world` sind. Nur in diesem einen Fall wird die in Schritt 2 hinzugefügte neue Funktion `foo` aufgerufen. Daraus resultiert das Programm R, und die Eingabe z ist identisch mit y.

Angenommen, Q gibt bei der Eingabe y als Erstes `hello, world` aus. Dann wird R, wie wir es konstruiert haben, `foo` aufrufen. Falls Q mit der Eingabe y jedoch nicht als Erstes `hello, world` ausgibt, ruft R niemals `foo` auf. Wenn wir entscheiden können, ob R mit der Eingabe y `foo` aufruft, dann wissen wir auch, ob Q mit der Eingabe y (denn $y = z$) `hello, world` ausgibt. Da wir wissen, dass es keinen Algorithmus zur Lösung des »Hello, World«-Problems gibt, und da alle vier Schritte der Konstruktion von R aus Q von einem Programm ausgeführt werden könnten, das den Quelltext von Programmen editiert, ist unsere Annahme falsch, dass es einen »foo-Aufruf«-Tester gibt. Ein solches Programm existiert nicht, und das »foo-Aufruf«-Problem ist unentscheidbar. ■

Kann ein Computer wirklich all das leisten?

Wenn wir ein Programm wie das in Listing 8.2 dargestellte untersuchen, könnten wir infrage stellen, ob es wirklich nach Gegenbeispielen zu Fermats letztem Satz sucht. Im Grunde sind ganzzahlige Werte bei einem typischen Computer lediglich 32 Bit lang, und wenn das kleinste Gegenbeispiel ganze Zahlen im Bereich der Milliarden verlangen würde, träte ein Überlauffehler auf, bevor die Lösung gefunden würde. Tatsächlich könnte man argumentieren, dass ein Computer mit 128 Mbyte Hauptspeicher und einer 30 Gbyte großen Festplatte »nur« $256^{30128000000}$ Zustände besitzt und daher ein endlicher Automat ist.

Es ist jedoch unproduktiv, Computer als endliche Automaten anzusehen (oder Gehirne als endliche Automaten zu betrachten, worauf das Konzept der endlichen Automaten zurückgeht). Die Anzahl der Zustände ist so hoch und die Beschränkungen sind so unklar, dass man daraus keine nützlichen Schlussfolgerungen ziehen kann. Stattdessen spricht alles dafür, dass wir die Menge der Zustände eines Computers bei Bedarf beliebig erweitern könnten.

→

→ Beispielsweise könnten wir ganze Zahlen als verknüpfte Listen von Zahlen beliebiger Länge darstellen. Steht nicht mehr genug Speicher zur Verfügung, kann das Programm eine Anweisung an den menschlichen Benutzer ausgeben, die Festplatte gegen eine leere Festplatte auszutauschen und zu archivieren. Im Lauf der Zeit könnte der Computer die benötigte Anzahl von Festplatten anfordern. Dieses Programm wäre weit komplexer als das in Listing 8.2 dargestellte, läge jedoch im Rahmen unserer Möglichkeiten. Ähnliche Tricks erlaubten jedem beliebigen Programm, endliche Beschränkungen der Speichergröße oder der Größe der ganzzahligen Werte oder anderer Datenelemente zu umgehen.

Die Richtung der Reduktion ist wichtig

Häufig wird der Fehler begangen, den Beweis für die Unentscheidbarkeit eines Problems P_2 über die Reduktion von P_2 auf ein als unentscheidbar bekanntes Problem P_1 zu führen. Es soll also die Aussage »Wenn P_1 entscheidbar ist, dann ist auch P_2 entscheidbar« gezeigt werden. Diese Aussage ist zwar mit Sicherheit richtig, jedoch nutzlos, da deren Hypothese »P_1 ist entscheidbar« falsch ist.

Es gibt nur eine Möglichkeit, die Unentscheidbarkeit eines Problems P_2 durch Reduktion zu beweisen: Ein bekanntes unentscheidbares Problem P_1 muss auf P_2 reduziert werden. Auf diese Art beweisen wir die Aussage »Wenn P_2 entscheidbar ist, dann ist auch P_1 entscheidbar«. Die Umkehrung dieser Aussage lautet: »Wenn P_1 unentscheidbar ist, dann ist auch P_2 unentscheidbar.« Da wir wissen, dass P_1 unentscheidbar ist, können wir schließen, dass P_2 unentscheidbar ist.

8.1.4 Übungen zum Abschnitt 8.1

Übung 8.1.1 Reduzieren Sie das »Hello, World«-Problem auf jedes der unten aufgeführten Probleme. Verwenden Sie den informellen Stil dieses Abschnitts zur Beschreibung plausibler Programmtransformationen, und lassen Sie reale Beschränkungen wie maximale Datei- oder Speichergröße, die durch reale Computer auferlegt werden, außer Acht.

***!** a) Gegeben seien ein Programm und eine Eingabe. Hält das Programm an, wird es also bei dieser Eingabe nicht in eine Endlosschleife münden?

b) Gegeben seien ein Programm und eine Eingabe. Wird das Programm *überhaupt* jemals eine Ausgabe produzieren?

! c) Gegeben seien zwei Programme und eine Eingabe. Werden die Programme für die gegebene Eingabe die gleiche Ausgabe produzieren?

8.2 Die Turing-Maschine

Die Theorie der unentscheidbaren Probleme dient nicht nur dazu, die Existenz solcher Probleme zu bestätigen (eine an sich intellektuell reizvolle Vorstellung), sondern sie soll Programmierern auch eine Leitlinie bieten, was durch Programmierung möglich ist und was nicht. Die Theorie hat außerdem einen großen pragmatischen Einfluss auf die Diskussion von Problemen, die zwar entscheidbar sind, deren Lösung jedoch einen hohen Zeitaufwand erfordert (siehe Kapitel 10). Diese so genannten nicht handhabbaren (engl. *intractable*) Probleme stellen für Programmierer und Systementwick-

ler tendenziell eine größere Schwierigkeit dar als die unentscheidbaren Probleme. Der Grund dafür ist, dass unentscheidbare Probleme für gewöhnlich als solche erkennbar sind und sich nur selten jemand an deren Lösung versucht, während nicht handhabbare Probleme tagtäglich auftreten. Außerdem werden sie oft nach kleinen Änderungen an den Voraussetzungen oder durch heuristische Lösungen handhabbar. Daher muss häufig entschieden werden, ob ein Problem zu dieser Problemklasse gehört und wie damit umzugehen ist.

Wir benötigen Werkzeuge, die uns den Beweis ermöglichen, dass alltägliche Fragen nicht entscheidbar oder nicht handhabbar sind. Die in Abschnitt 8.1 vorgestellten Techniken sind bei Fragen im Umfeld von Programmen nützlich, lassen sich jedoch nicht einfach auf Probleme aus nicht miteinander in Beziehung stehenden Domänen übertragen. Beispielsweise würde es uns schwer fallen, das »Hello, World«-Problem auf die Frage zu reduzieren, ob eine Grammatik mehrdeutig ist.

Daher müssen wir unsere Theorie der Unentscheidbarkeit anders definieren, sodass sie nicht auf Programmen in C oder einer anderen Sprache, sondern auf einem sehr einfachen Modell eines Computers, nämlich der Turing-Maschine, basiert. Bei diesem Gerät handelt es sich im Wesentlichen um einen endlichen Automaten, der ein einzelnes unendlich langes Band besitzt, von dem er Daten liest und auf das er Daten schreibt. Ein Vorteil der Turing-Maschine gegenüber Programmen als Repräsentation dessen, was berechnet werden kann, liegt in ihrer Einfachheit, sodass wir ihre Konfiguration mit einer einfachen Notation, ähnlich den Konfigurationen eines PDA, präzise darstellen können. Zwar besitzen C-Programme einen Zustand, in den alle Variablen durch die Folgen von Funktionsaufrufen eingehen, doch die Beschreibung dieser Zustände ist viel zu komplex, um uns verständliche formale Beweise zu ermöglichen.

Wir verwenden die Notation der Turing-Maschine, um die Unentscheidbarkeit verschiedener Probleme zu beweisen, die in keinem Zusammenhang mit der Programmierung zu stehen scheinen. Beispielsweise werden wir in Abschnitt 9.4 zeigen, dass das »Post'sche Korrespondenzproblem«, eine einfache Frage hinsichtlich zweier Listen von Zeichenreihen, unentscheidbar ist, und dieses Problem erleichtert es, die Unentscheidbarkeit von Fragen in Bezug auf Grammatiken, z. B. die Unentscheidbarkeit der Mehrdeutigkeit, zu beweisen. Entsprechend werden wir bei der Einführung nicht handhabbarer Probleme sehen, dass bestimmte Fragen, die scheinbar kaum in Zusammenhang mit Computern stehen (etwa die Erfüllbarkeit boolescher Formeln), nicht handhabbar sind.

8.2.1 Das Streben danach, alle mathematischen Fragen zu entscheiden

Zu Anfang des 20. Jahrhunderts untersuchte der Mathematiker David Hilbert die Frage, ob es möglich sei, einen Algorithmus zu finden, mit dem sich die Wahrheit bzw. Falschheit jeder mathematischen Aussage bestimmen ließe. Er ging insbesondere der Frage nach, ob sich auf irgendeine Weise feststellen lässt, ob eine auf ganze Zahlen angewandte Formel des Prädikatenkalküls erster Stufe wahr ist. Da der Prädikatenkalkül erster Stufe für ganze Zahlen ausreichend mächtig ist, um Aussagen wie »Diese Grammatik ist mehrdeutig« oder »Dieses Programm gibt `hello, world` aus« auszudrücken, gäbe es für diese Probleme Algorithmen, wäre Hilbert erfolgreich gewesen. Wir wissen heute, dass solche Algorithmen nicht existieren.

1931 veröffentlichte Kurt Gödel seinen berühmten Unvollständigkeitssatz. Er konstruierte eine Formel des Prädikatenkalküls erster Stufe angewandt auf ganze Zahlen,

die aussagt, dass die Formel selbst innerhalb des Prädikatenkalküls weder bewiesen noch widerlegt werden kann. Gödels Technik gleicht der Konstruktion des sich selbst widersprechenden Programms H_2 in Abschnitt 8.1.2, hat jedoch Funktionen für ganze Zahlen statt C-Programme zum Inhalt.

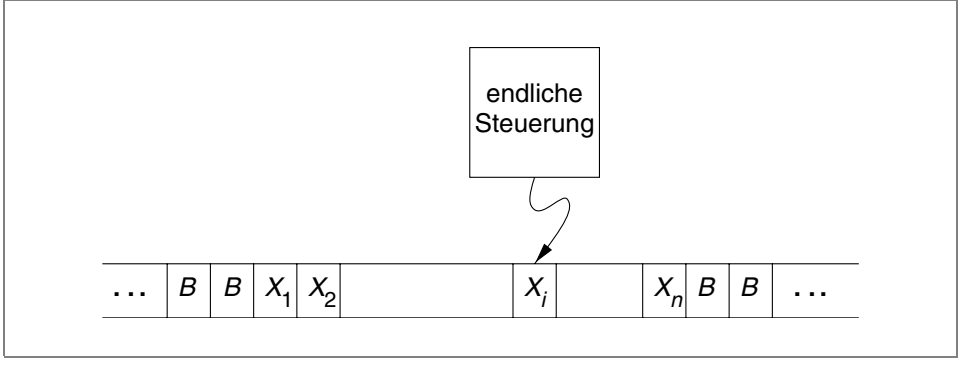
Den Mathematikern stand für »jede mögliche Berechnung« nicht nur der Prädikatenkalkül zur Verfügung. Der Prädikatenkalkül, der deklarativer statt berechnender Natur ist, konkurrierte mit einer Vielzahl von Notationen, darunter den »partiell-rekursiven Funktionen«, einer programmiersprachenähnlichen Notation, sowie anderen ähnlichen Notationen. 1936 stellte Alan Mathison Turing die Turing-Maschine als ein Modell für »jede mögliche Berechung« vor. Dieses Modell ähnelt mehr einem Computer als einem Programm, auch wenn die Entwicklung wirklich elektronischer oder gar elektromechanischer Computer in der Zukunft lag. (Turing selbst arbeitete während des Zweiten Weltkriegs an der Konstruktion einer solchen Maschine mit.).

Es ist bemerkenswert, dass alle ernsthaften Vorschläge für ein Berechnungsmodell die gleiche Mächtigkeit besitzen: Alle berechnen die gleichen Funktionen oder erkennen die gleichen Sprachen. Die nicht beweisbare Aussage, dass jede allgemeine Berechnung lediglich erlaubt, die partiell-rekursiven Funktionen zu berechnen (oder entsprechend das, was Turing-Maschinen oder moderne Computer berechnen können), wird als *Church´sche Hypothese* oder *Church-Turing-These* bezeichnet (nach dem Logiker Alonzo Church).

8.2.2 Die Notation der Turing-Maschine

Abbildung 8.8 zeigt, wie wir uns eine Turing-Maschine bildlich vorstellen können. Die Maschine besteht aus einer *endlichen Steuerung*, die jeden Zustand aus einer endlichen Menge von Zuständen annehmen kann. Ein *Band* ist in einzelne Abschnitte oder *Zellen* aufgeteilt; jede Zelle kann ein beliebiges Symbol aus einer endlichen Menge von Symbolen enthalten.

Abbildung 8.8: Eine Turing-Maschine



Zur Initialisierung enthält das Band die *Eingabe*, d. h. eine endlich lange Zeichenreihe von Symbolen, die aus dem *Eingabealphabet* gewählt wurden. Alle weiteren Zellen des Bandes, die sich unendlich nach rechts und links erstrecken, enthalten das besondere Symbol *Leerzeichen* (Blank). Das Leerzeichen ist ein *Bandsymbol*, aber kein Eingabesymbol. Zusätzlich zu den Eingabesymbolen und dem Leerzeichen kann es noch weitere Bandsymbole geben.

Ein *Schreib-/Lesekopf* ist immer über einer der Bandzellen positioniert. Die Turing-Maschine *liest* diese Zelle. Zu Anfang befindet sich der Schreib-/Lesekopf über der am weitesten links stehenden Eingabezelle.

Eine *Bewegung* der Turing-Maschine ist eine Funktion des Zustands der endlichen Steuerung und des gelesenen Bandsymbols. Mit einer Bewegung führt die Turing-Maschine Folgendes aus:

1. Sie ändert den Zustand. Der nächste Zustand kann mit dem aktuellen Zustand übereinstimmen.
2. Sie schreibt ein Bandsymbol in die gelesene Zelle. Dieses Bandsymbol ersetzt das in dieser Zelle zuvor enthaltene Symbol. Es kann optional das gleiche Symbol wie das bereits vorhandene geschrieben werden.
3. Sie bewegt den Schreib-/Lesekopf nach links oder rechts. In unserem Formalismus ist eine Bewegung erforderlich; der Schreib-/Lesekopf darf nicht an der gleichen Stelle verbleiben. Diese Einschränkung wirkt sich nicht darauf aus, was eine Turing-Maschine berechnen kann, da sich jede Folge von Bewegungen mit stationärem Schreib-/Lesekopf im Wesentlichen zusammen mit dessen nächster Bewegung in eine einzige Zustandsänderung, ein neues Bandsymbol und eine Bewegung nach links oder rechts zusammenfassen lässt.

Wir verwenden für eine *Turing-Maschine* (TM) eine formale Notation, die derjenigen für endliche Automaten oder PDAs gleicht. Eine Turing-Maschine wird beschrieben durch

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

wobei die Komponenten folgende Bedeutungen besitzen:

Q: Die endliche Menge der *Zustände* der endlichen Steuerung.

Σ: Die endliche Menge der *Eingabesymbole*.

Γ: Die vollständige Menge der *Bandsymbole*; Σ ist immer eine Teilmenge von Γ.

δ: Die *Übergangsfunktion*. Die Übergangsfunktion $\delta(q, X)$ besitzt als Argumente die Zustände q und ein Bandsymbol X. Falls definiert, ist der Wert von $\delta(q, X)$ ein Tripel (p, Y, D), wobei gilt:

1. p ist der nächste Zustand in Q.
2. Y ist das Symbol aus Γ, das in die gelesene Zelle geschrieben wird und das vorherige Symbol ersetzt.
3. D ist die *Richtung* L (links) oder R (rechts), in die sich der Schreib-/Lesekopf bewegt.

q_0: Der *Startzustand*, in dem sich die endliche Steuerung anfänglich befindet, ist ein Element von Q.

B: Das *Leerzeichen*. Dieses Symbol ist in Γ enthalten, nicht jedoch in Σ, ist also kein Eingabesymbol. Das Leerzeichen steht zu Anfang in allen Zellen bis auf die endliche Menge der Anfangszellen mit den Eingabesymbolen.

F: Die Menge der *Endzustände* oder *akzeptierenden Zustände* ist eine Teilmenge von Q.

8.2.3 Beschreibung von Turing-Maschinen

Um die Arbeitsweise einer Turing-Maschine formal zu beschreiben, müssen wir eine Notation für Konfigurationen oder IDs (Instantaneous Descriptions) entwickeln, ähnlich der Notation für PDAs. Da eine TM im Allgemeinen ein Endlosband besitzt, ist es vorstellbar, dass sich die Konfigurationen oder Instanzdeskriptoren (IDs) einer TM nicht präzise beschreiben lassen. Allerdings kann eine TM nach jeder endlichen Anzahl von Bewegungen nur eine endliche Anzahl von Zellen besucht haben, auch wenn die Anzahl besuchter Zellen irgendwann jede endliche Beschränkung übersteigt. Eine Konfiguration umfasst daher ein unendliches Präfix sowie ein unendliches Suffix nicht besuchter Zellen. Jede dieser Zellen enthält entweder ein Leerzeichen oder eines der endlich vielen Eingabesymbole. Wir zeigen in einer Konfiguration im Allgemeinen nur die Zellen zwischen der am weitesten links und der am weitesten rechts stehenden Zelle; letztere enthalten beide kein Leerzeichen. Unter speziellen Bedingungen (wenn der Schreib-/Lesekopf eines der führenden oder folgenden Leerzeichen liest) muss auch eine endliche Anzahl von Leerzeichen links oder rechts vom nichtleeren Abschnitt des Bandes in die Konfiguration aufgenommen werden.

Zusätzlich zum Band müssen wir auch die endliche Steuerung sowie die Position des Schreib-/Lesekopfes beschreiben. Dazu betten wir den Zustand in das Band ein und legen ihn links von der gelesenen Zelle ab. Um die Band-plus-Zustand-Zeichenreihe eindeutig zu machen, müssen wir sicherstellen, dass wir kein Bandsymbol als Zustand verwenden. Es ist jedoch einfach, die Namen der Zustände zu ändern, sodass keine Gemeinsamkeit mit den Bandsymbolen vorliegt, da die Arbeitsweise der TM nicht von den Namen der aufgerufenen Zustände abhängt. Daher verwenden wir die Zeichenreihe $X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n$ zur Darstellung einer Konfiguration, wobei die Komponenten folgende Bedeutungen haben:

1. q ist der Zustand der Turing-Maschine.
2. Der Schreib-/Lesekopf liest das i-te Symbol von links.
3. $X_1X_2\ldots X_n$ bezeichnet den Abschnitt des Bandes zwischen dem am weitesten links und dem am weitesten rechts stehenden Zeichen, die keine Leerzeichen sind. Als Ausnahme gilt, wenn sich der Lese-/Schreibkopf links bzw. rechts von dem am weitesten links bzw. rechts stehenden Zeichen befindet, das kein Leerzeichen ist. In diesem Ausnahmefall ist ein Präfix oder Suffix von $X_1X_2\ldots X_n$ leer, und i ist 1 bzw. n.

Wir beschreiben die Bewegungen $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ der Turing-Maschine mit der für PDAs verwendeten $\vdash_M$-Notation. Wenn klar ist, um welche TM M es sich handelt, verwenden wir lediglich $\vdash$ zur Darstellung von Bewegungen. Wie üblich, zeigt $\vdash_M^*$ oder einfach nur $\vdash^*$ keine, eine oder mehrere Bewegungen der TM M an.
Angenommen, $\delta(q, X_i) = (p, Y, L)$; d. h. die nächste Bewegung erfolgt nach links. Dann gilt:

$$X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n \vdash_M X_1X_2\ldots X_{i-2}pX_{i-1}YX_{i+1}\ldots X_n$$

Beachten Sie, wie diese Bewegung die Änderung in Zustand p und die Tatsache reflektiert, dass der Schreib-/Lesekopf nun über der Zelle $i - 1$ positioniert ist. Es gibt zwei wichtige Ausnahmen:

1. Ist $i = 1$, dann führt M eine Bewegung zum Leerzeichen links von X_1 aus. In diesem Fall gilt:

$$X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n \vdash_M pBY\ X_2\ldots X_n$$

2. Ist $i = n$ und $Y = B$, dann wird das über X_n geschriebene Symbol B mit der Folge nachfolgender Leerzeichen vereinigt und erscheint nicht in der nächsten Konfiguration. Daher gilt:

$$X_1X_2\ldots X_{n-1}qX_n \vdash_M X_1X_2\ldots X_{n-2}pX_{n-1}$$

Angenommen, $\delta(q, X_i) = (p, Y, R)$; die nächste Bewegung sei also nach rechts gerichtet. Dann gilt:

$$X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n \vdash_M X_1X_2\ldots X_{i-1}YpX_{i+1}\ldots X_n$$

Diese Bewegung spiegelt hier das Verschieben des Schreib-/Lesekopfes zur Zelle $i + 1$ wider. Auch hier gibt es zwei wichtige Ausnahmen:

1. Ist $i = n$, dann enthält Zelle $i + 1$ ein Leerzeichen, und diese Zelle gehörte nicht zur vorherigen Konfiguration. Daher gilt stattdessen:

$$X_1X_2\ldots X_{n-1}qX_n \vdash_M X_1X_2\ldots X_{n-1}YpB$$

2. Ist $i = 1$ und $Y = B$, dann wird das über X_1 geschriebene Symbol B mit der unendlichen Folge führender Leerzeichen vereinigt und erscheint nicht in der nächsten Konfiguration. Daher gilt:

$$qX_1X_2\ldots X_n \vdash_M pX_2\ldots X_n$$

Beispiel 8.2 Wir wollen eine Turing-Maschine entwerfen und deren Verhalten bei einer typischen Eingabe beobachten. Diese TM akzeptiert die Sprache $\{0^n1^n \mid n \geq 1\}$. Anfangs ist das Band mit einer endlichen Folge von Nullen und Einsen beschrieben, der unendlich viele Leerzeichen voranstehen und nachfolgen. Alternierend ändert die TM eine Null in ein X und dann eine Eins in ein Y, bis alle Nullen und Einsen überschrieben sind.

Genauer gesagt, beginnt die TM am linken Ende der Eingabe, ändert eine 0 in ein X und bewegt sich dann nach rechts über Nullen und Y-Zeichen hinweg, bis eine 1 erreicht ist. Die TM ändert die 1 in ein Y und bewegt sich dann nach links, über jede 0 und jedes Y hinweg, bis ein X gefunden wird. Sie sucht in der rechten Nachbarzelle des X nach einer 0 und ändert sie, falls vorhanden, in ein X, wiederholt dann den Vorgang und ändert eine 1 in ein Y.

Ist die nichtleere Eingabe nicht in **0*1*** enthalten, dann kann die TM irgendwann keine Bewegung ausführen und gerät in einen undefinierten Zustand, ohne zu akzeptieren. Wurden jedoch in der gleichen Runde, in der die letzte 1 in Y geändert wurde, auch alle Nullen in X geändert, dann hat die TM die Eingabe als Eingabe der Form 0^n1^n erkannt und akzeptiert. Die formale Spezifikation der TM M lautet wie folgt:

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\}$$

δ ist in der Tabelle 8.1 festgelegt.

Tabelle 8.1: Eine Turing-Maschine, die $\{0^n1^n \mid n \geq 1\}$ akzeptiert

	Symbol				
Zustand	**0**	**1**	***X***	***Y***	***B***
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Während *M* die Berechnung ausführt, besteht der Abschnitt des Bandes, der von *M*s Schreib-/Lesekopf besucht wurde, immer aus einer Folge, die der reguläre Ausdruck **X*0*Y*1*** beschreibt. Das heißt es gibt einige Nullen, die in *X* geändert wurden, gefolgt von einigen unveränderten Nullen. Außerdem gibt es einige Einsen, die in *Y* geändert wurden, sowie einige unveränderte Einsen. Überdies kann es nachfolgende Nullen und Einsen geben.

Zustand q_0 ist der Startzustand, den die TM *M* jedes Mal wieder erreicht, wenn sie zur Null zurückkehrt, die am weitesten links steht. Wenn sich *M* im Zustand q_0 befindet und eine Null liest, dann weist die Regel, die in der linken oberen Ecke von Tabelle 8.1 dargestellt ist, die TM an, in den Zustand q_1 zu wechseln, die Null in ein *X* zu ändern und sich nach rechts zu bewegen. Im Zustand q_1 bewegt sich *M* über alle Nullen und *Y* auf dem Band nach rechts und verbleibt in diesem Zustand. Trifft *M* auf ein *X* oder ein *B*, dann gerät sie in einen undefinierten Zustand. Liest *M* jedoch im Zustand q_1 eine Eins, dann ändert *M* die Eins in *Y*, wechselt in den Zustand q_2 und beginnt, sich nach links zu bewegen.

Im Zustand q_2 bewegt sich *M* über alle Nullen und *Y* nach links und verbleibt in diesem Zustand. Erreicht *M* das am weitesten links stehende *X*, welches das rechte Ende des Blocks aus Nullen markiert, die schon in *X* geändert wurden, kehrt *M* in den Zustand q_0 zurück und führt eine Bewegung nach rechts aus. Zwei Fälle sind möglich:

1. Liest *M* nun eine Null, wird der oben beschriebene Zyklus wiederholt.
2. Liest *M* ein *Y*, dann wurden alle Nullen in *X* geändert. Wurden alle Einsen in *Y* geändert, dann lag eine Eingabe der Form 0^n1^n vor, und *M* sollte akzeptieren. *M* wechselt daher in den Zustand q_3 und bewegt sich über die Zeichen *Y* hinweg nach rechts. Handelt es sich bei dem ersten Zeichen ungleich *Y*, auf das *M* trifft, um ein Leerzeichen, dann lag tatsächlich eine identische Anzahl von Nullen und Einsen vor, und *M* wechselt in den Zustand q_4 und akzeptiert. Trifft *M* jedoch auf eine weitere Eins, dann sind zu viele Einsen vorhanden, und *M* gerät in einen undefinierten Zustand, ohne zu akzeptieren. Trifft *M* auf eine Null, dann hatte die Eingabe das falsche Format, und *M* gerät ebenfalls in einen undefinierten Zustand.

Es folgt ein Beispiel für eine akzeptierende Berechnung von M. Die Eingabe lautet 0011. Anfangs befindet sich M im Zustand q_0 und liest die erste 0; Ms Anfangs-Konfiguration ist also $q_0 0011$. Die gesamte Zugfolge lautet:

$$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1 \vdash q_2 X0Y1 \vdash$$
$$Xq_0 0Y1 \vdash XXq_1 Y1 \vdash XXYq_1 1 \vdash XXq_2 YY \vdash Xq_2 XYY \vdash$$
$$XXq_0 YY \vdash XXYq_3 Y \vdash XXYYq_3 B \vdash XXYYBq_4 B$$

Sehen Sie sich in einem weiteren Beispiel an, wie M die Eingabe 0010 verarbeitet, die nicht aus der akzeptierten Sprache stammt:

$$q_0 0010 \vdash Xq_1 010 \vdash X0q_1 10 \vdash Xq_2 0Y0 \vdash q_2 X0Y0 \vdash$$
$$Xq_0 0Y0 \vdash XXq_1 Y0 \vdash XXYq_1 0 \vdash XXY0q_1 B$$

Bei 0010 zeigt M das gleiche Verhalten wie bei 0011, bis M in Konfiguration $XXYq_1 0$ zum ersten Mal die letzte 0 liest. M verbleibt im Zustand q_1, muss eine Bewegung nach rechts ausführen und kommt zu Konfiguration $XXY0q_1 B$. Im Zustand q_1 gibt es jedoch keine Bewegung für das Bandsymbol B, und daher gerät M in einen undefinierten Zustand und akzeptiert die Eingabe nicht. ■

8.2.4 Übergangsdiagramme für Turing-Maschinen

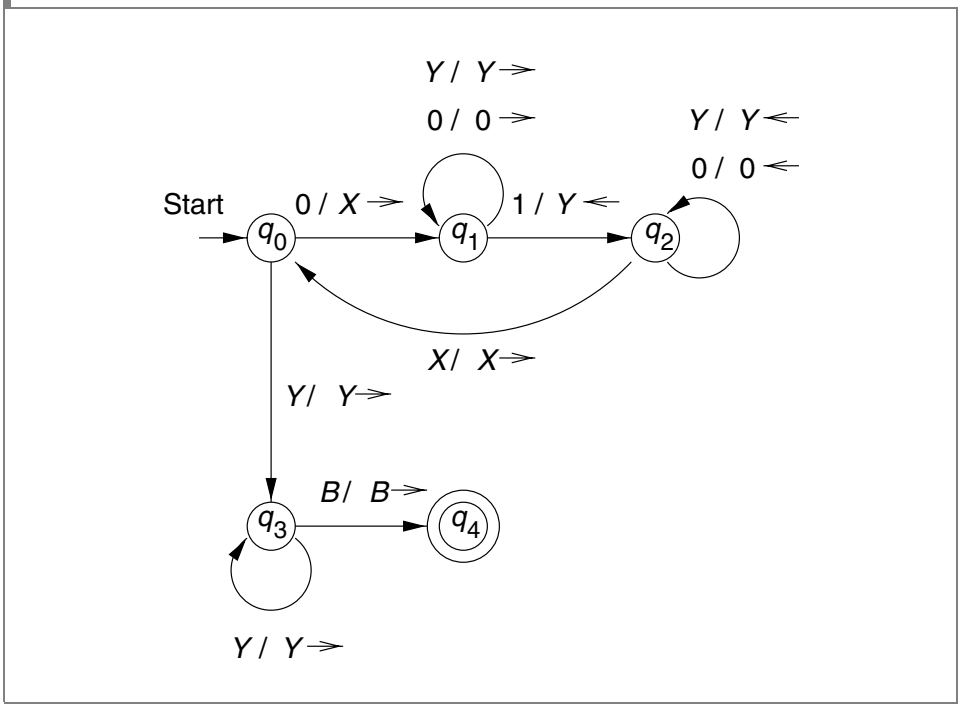
Wir können wie bei PDAs die Zustandsübergänge einer Turing-Maschine bildlich darstellen. Ein *Übergangsdiagramm* besteht aus einer Folge von Knoten, die den Zuständen der TM entsprechen. Eine Kante vom Zustand q zum Zustand p wird durch ein oder mehrere Elemente $X / Y\ D$ beschriftet, wobei X und Y für Bandsymbole und D für eine der Richtungen L oder R stehen. Wenn $\delta(q, X) = (p, Y, D)$, dann wird die Kante zwischen p und q mit $X / Y\ D$ beschriftet. In unseren Diagrammen wird die Richtung allerdings bildlich durch $\leftarrow$ für »links« und $\rightarrow$ für »rechts« dargestellt.

Wie bei anderen Typen von Übergangsdiagrammen repräsentieren wir den Startzustand durch das Wort »Start« sowie einen Pfeil, der auf diesen Zustand zeigt. Akzeptierende Zustände werden durch einen doppelten Kreis angezeigt. Die einzige Information über die TM, die nicht direkt aus dem Diagramm ersichtlich ist, ist das Symbol für das Leerzeichen. Wir nehmen an, dieses Symbol sei B, sofern nichts anderes angegeben ist.

Beispiel 8.3 Abbildung 8.9 zeigt das Übergangsdiagramm für die Turing-Maschine aus Beispiel 8.2, deren Übergangsfunktion in Tabelle 8.1 dargestellt wurde. ■

Beispiel 8.4 Heutzutage sehen wir Turing-Maschinen als Sprachenerkenner an bzw. als Problemlöser, was damit gleichbedeutend ist. Turing betrachtete seine Maschine ursprünglich jedoch als einen Computer für ganzzahlige Funktionen. In seinem Schema wurden ganze Zahlen unär als aus einem Zeichen bestehende Blöcke dargestellt, und die Maschine führte Berechnungen durch, indem sie die Blocklängen änderte oder neue Blöcke in einem anderen Bandabschnitt erstellte. In diesem einfachen Beispiel zeigen wir, wie eine Turing-Maschine die Funktion $\dot{-}$ berechnen könnte, die *Monus-Operation* oder *eingeschränkte Subtraktion* genannt wird und durch $m \dot{-} n = max\ (m - n, 0)$ definiert ist. $m \dot{-} n$ ist also $m - n$, falls $m \geq n$ und 0, falls $m < n$.

Abbildung 8.9: Übergangsdiagramm für eine TM, die Zeichenreihen der Form 0^n1^n akzeptiert



Eine Turing-Maschine, die diese Operation ausführt, ist wie folgt definiert:

$$M = (\{q_0, q_1, \ldots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B)$$

Beachten Sie, dass auf die siebte Komponente, die Menge der akzeptierenden Zustände, verzichtet wurde, da diese TM nicht verwendet wird, um Eingaben zu akzeptieren. M startet mit einem Band, das 0^m10^n umgeben von Leerzeichen enthält. M hält an, wenn das Band $0^{m \dot{-} n}$ umgeben von Leerzeichen enthält.

M sucht wiederholt nach der am weitesten links stehenden 0 und ersetzt sie durch ein Leerzeichen. Dann wird nach rechts nach einer 1 gesucht. Findet M eine 1, wird die Suche nach rechts fortgesetzt, bis eine 0 erreicht und durch eine 1 ersetzt wird. M sucht dann nach links nach der am weitesten links stehenden 0 und identifiziert sie, wenn ein Leerzeichen auftritt. Die 0 steht in der Zelle unmittelbar rechts davon. Die Wiederholung endet in einem der beiden folgenden Fälle:

1. Die nach rechts gerichtete Suche nach einer Null führt zu einem Leerzeichen. In diesem Fall wurden n Nullen in 0^m10^n durch Einsen und $n + 1$ der m Nullen durch B ersetzt. M ersetzt die $n + 1$ Einsen durch eine Null und n B, wobei $m - n$ Nullen auf dem Band belassen werden. Da in diesem Fall $m > n$ gilt, ist $m - n = m \dot{-} n$.

2. Zu Beginn des Zyklus findet M keine Null, die durch ein Leerzeichen zu ersetzen ist, da die ersten m Nullen schon durch B ersetzt worden sind. Dann ist $n \geq m$ und somit $m \dot{-} n = 0$. M ersetzt die restlichen Einsen und Nullen durch B und endet mit einem vollständig leeren Band.

Tabelle 8.2: Eine Turing-Maschine, welche die Monus-Funktion berechnet

	Symbol		
Zustand	**0**	**1**	***B***
q_0	(q_1, B, R)	(q_5, B, R)	–
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	–
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	–	–	–

Tabelle 8.2 zeigt die Regeln der Übergangsfunktion δ. δ wird außerdem in Abbildung 8.10 als Übergangsdiagramm dargestellt. Es folgt eine Zusammenfassung der Rollen der sieben Zustände:

q_0: Dieser Zustand startet den Zyklus und bricht ihn gegebenenfalls auch ab. Liest M eine 0, dann muss der Zyklus wiederholt werden. Die 0 wird durch B ersetzt, der Schreib-/Lesekopf wandert nach rechts, und Zustand q_1 wird erreicht.

q_1: In diesem Zustand durchsucht M den ursprünglichen Block von Nullen nach rechts nach der am weitesten links stehenden 1. Im Erfolgsfall wechselt M in den Zustand q_2.

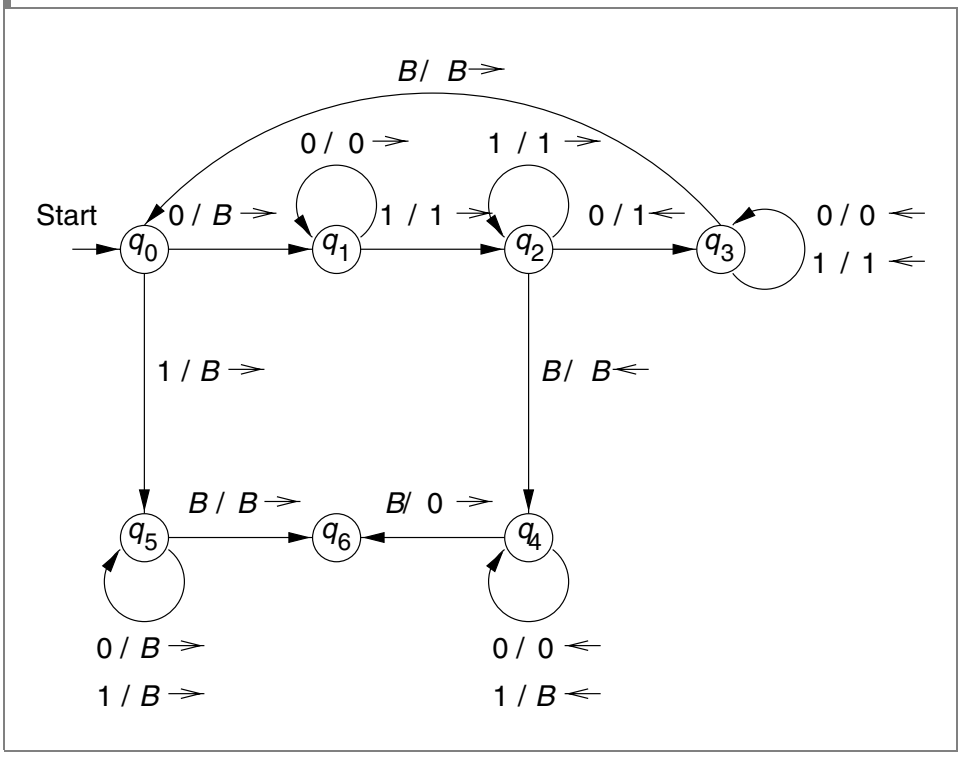
q_2: M bewegt sich nach rechts und überspringt die Einsen, bis eine Null gefunden wird. M ändert die 0 in eine 1, ändert die Suchrichtung und wechselt in Zustand q_3. Es ist jedoch möglich, dass nach dem Block mit Einsen keine weitere Null folgt. In diesem Fall trifft M in Zustand q_2 auf ein Leerzeichen. Dieser Fall (1) ist oben beschrieben: n Nullen aus dem zweiten Block von Nullen wurden verwendet, um $n + 1$ der m Nullen im ersten Block zu eliminieren. Die Subtraktion ist abgeschlossen. M wechselt in den Zustand q_4, der dazu dient, die Einsen auf dem Band in Leerzeichen umzuwandeln.

q_3: M sucht nach links nach einem Leerzeichen und überspringt dabei alle Nullen und Einsen. Wird B gefunden, wechselt M die Richtung und tritt in den Zustand q_0 ein. Der Zyklus wird erneut gestartet.

q_4: Nun ist die Subtraktion abgeschlossen, doch eine Null im ersten Block wurde zu viel in ein B geändert. M ändert nach links die Einsen in B, bis ein B auf dem Band gefunden wird. M ändert dieses B dann wieder in eine Null und wechselt in den Zustand q_6, in dem M anhält.

q_5: Aus dem Zustand q_0 wird in den Zustand q_5 gewechselt, wenn alle Nullen im ersten Block in B geändert worden sind. In diesem in (2) beschriebenen Fall lautet das Ergebnis der eingeschränkten Subtraktion 0. M ändert alle verbliebenen Nullen und Einsen in B und wechselt in den Zustand q_6.

Abbildung 8.10: Das Übergangsdiagramm für die TM aus Beispiel 8.4



q_6: Der einzige Zweck dieses Zustands besteht darin, es der TM M zu ermöglichen, nach erledigter Arbeit anzuhalten. Wäre die Subtraktion eine Subroutine einer komplexeren Funktion, dann würde q_6 den nächsten Schritt dieser umfangreicheren Berechnung initiieren. ■

8.2.5 Die Sprache einer Turing-Maschine

Wir haben intuitiv ein Verfahren nahe gelegt, wie eine Turing-Maschine eine Sprache akzeptiert. Die Eingabezeichenreihe wird auf dem Band abgelegt, und der Schreib-/Lesekopf beginnt mit dem am weitesten links stehenden Symbol. Erreicht die TM einen akzeptierenden Zustand, dann wird die Eingabe akzeptiert, sonst jedoch nicht.

Formaler ausgedrückt sei $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ eine Turing-Maschine. Dann ist $L(M)$ die Menge der Zeichenreihen w in Σ^*, sodass $q_0 w \overset{*}{\vdash} \alpha p \beta$ für einige Zustände p in F und alle Bandzeichenreihen α und β. Diese Definition bildete die Grundlage der Diskussion der Turing-Maschine aus Beispiel 8.2, die Zeichenreihen der Form 0^n1^n akzeptiert.

Die Menge der Sprachen, die wir unter Verwendung einer Turing-Maschine akzeptieren können, wird häufig die Menge der *rekursiv aufzählbaren Sprachen* genannt. Der Begriff »rekursiv aufzählbar« stammt aus einem Berechnungsformalismus, der der Turing-Maschine vorausging, aber die gleiche Klasse von Sprachen oder arithmetischen Funktionen definiert. Den Ursprung des Begriffs erläutert ein Exkurs in Abschnitt 9.2.1.

Notationskonventionen für Turing-Maschinen

Die Symbole, die wir normalerweise für Turing-Maschinen verwenden, entsprechen denjenigen, die wir bei anderen Typen von Automaten einsetzen.

1. Kleinbuchstaben vom Alphabetanfang repräsentieren Eingabesymbole.
2. Großbuchstaben vom Alphabetende stehen für Bandsymbole, bei denen es sich auch um Eingabesymbole handeln kann. Im Allgemeinen wird *B* für das Leerzeichen verwendet.
3. Kleinbuchstaben vom Alphabetende repräsentieren Zeichenreihen aus Eingabesymbolen.
4. Griechische Buchstaben stehen für Zeichenreihen aus Bandsymbolen.
5. Buchstaben aus der Alphabetmitte wie *p* und *q* bezeichnen Zustände.

8.2.6 Turing-Maschinen und das Halteproblem

Für Turing-Maschinen wird für gewöhnlich ein weiterer Begriff für »Akzeptieren« verwendet: das Akzeptieren durch Anhalten. Eine Turing-Maschine *hält an*, wenn sie in einen Zustand q eintritt, ein Bandsymbol X liest und in dieser Situation keine weitere Bewegung erfolgen kann. $\delta(q, X)$ ist also undefiniert.

Beispiel 8.5 Die Turing-Maschine M aus Beispiel 8.4 sollte keine Sprache verarbeiten; sie diente zur Berechnung einer arithmetischen Funktion. Beachten Sie jedoch, dass M bei allen Zeichenreihen aus Nullen und Einsen anhält, denn unabhängig davon, welche Zeichenreihe M auf dem Band findet, wird M die zweite Gruppe von Nullen, falls vorhanden, gegen die erste Gruppe von Nullen eliminieren und muss daher Zustand q_6 erreichen und anhalten. ■

Wir können stets annehmen, dass eine TM anhält, wenn sie akzeptiert. Ohne die akzeptierte Sprache zu ändern, können wir $\delta(q, Y)$ als undefiniert bezeichnen, wann immer q ein akzeptierender Zustand ist. Im Allgemeinen gilt Folgendes, sofern nichts anderes angegeben wird:

- Wir nehmen an, dass eine TM immer anhält, wenn sie sich in einem akzeptierenden Zustand befindet.

Leider kann nicht immer gefordert werden, dass eine TM anhält, auch wenn sie nicht akzeptiert. Die Sprachen, bei denen Turing-Maschinen unabhängig vom Akzeptieren irgendwann anhalten, werden *rekursiv* genannt. In Abschnitt 9.2.1 ff. werden deren wichtige Eigenschaften vorgestellt. Turing-Maschinen, die unabhängig vom Akzeptieren immer anhalten, sind ein gutes Modell für einen »Algorithmus«. Existiert ein Algorithmus zur Lösung eines gegebenen Problems, dann bezeichnen wir dieses Problem als *entscheidbar*. TMs, die stets anhalten, sind daher in der Entscheidbarkeitstheorie (Kapitel 9) von Bedeutung.

8.2.7 Übungen zum Abschnitt 8.2

Übung 8.2.1 Zeigen Sie die Konfigurationen der Turing-Maschine aus Abbildung 8.9, falls das Eingabeband Folgendes enthält:

* a) 00

b) 000111

c) 00111

! **Übung 8.2.2** Entwerfen Sie für die folgenden Sprachen Turing-Maschinen:

* a) Die Menge der Zeichenreihen mit gleicher Anzahl von Nullen und Einsen

b) $\{a^n b^n c^n \mid n \geq 1\}$

c) $\{ww^R \mid w$ ist eine beliebige Zeichenreihe aus Nullen und Einsen$\}$.

Übung 8.2.3 Entwerfen Sie eine Turing-Maschine, die eine Zahl N als Eingabe verarbeitet und binär 1 dazu addiert. Präziser ausgedrückt, enthält das Band zu Anfang ein \$, gefolgt von N in Binärdarstellung. Der Schreib-/Lesekopf liest zu Anfang das \$ im Zustand q_0. Ihre TM sollte anhalten, wenn $N + 1$ in Binärdarstellung auf dem Band steht und die TM im Zustand q_f das am weitesten links stehende Symbol von $N + 1$ liest. Sie können das Zeichen \$ während der Erstellung von $N + 1$ löschen, wenn dies notwendig sein sollte. Beispielsweise gilt $q_0\$10011 \vdash^* \$q_f10100$ und $q_0\$11111 \vdash^* q_f100000$.

a) Zeigen Sie die Übergänge Ihrer Turing-Maschine, und erläutern Sie den Zweck jedes Zustands.

b) Zeigen Sie die Folge der Konfigurationen Ihrer Turing-Maschine bei Eingabe \$111.

*! **Übung 8.2.4** In dieser Übung untersuchen wir die Äquivalenz zwischen Funktionsberechnung und Spracherkennung bei Turing-Maschinen. Der Einfachheit halber betrachten wir nur Funktionen von nicht negativen ganzen Zahlen auf nicht negative Ganzzahlen, doch die Ideen dieses Problems sind auf alle berechenbaren Funktionen anwendbar. Die beiden zentralen Definitionen lauten:

- Der *Graph* einer Funktion f sei definiert als Menge aller Zeichenreihen der Form $[x, f(x)]$, wobei x eine nicht negative ganze Zahl in Binärform und $f(x)$ der Wert der Funktion f mit dem Argument x, ebenfalls in Binärform, sei.
- Eine Turing-Maschine *berechnet* Funktion f, wenn sie, mit einer beliebigen nicht negativen ganzen Zahl x in Binärform auf dem Band beginnend, mit $f(x)$ (in Binärform) auf dem Band anhält.

Beantworten Sie die folgenden Fragen mit informellen, aber klaren Konstruktionen:

a) Gegeben sei eine TM, die f berechnet. Zeigen Sie, wie Sie eine TM konstruieren, die den Graph von f als Sprache akzeptiert.

b) Gegeben sei eine TM, die den Graph von f akzeptiert. Zeigen Sie, wie Sie eine TM konstruieren, die f berechnet.

c) Eine Funktion ist partiell, wenn sie für einige Argumente undefiniert ist. Wenn wir die Ideen dieser Übung auf partielle Funktionen ausdehnen, dann fordern wir nicht, dass die TM, die f berechnet, anhält, wenn ihre Eingabe x eine der ganzen Zahl ist, für die $f(x)$ nicht definiert ist. Können Sie Ihre Konstruktionen aus a) und b) auf partielle Funktionen anwenden? Wenn nicht, erläutern Sie, wie die Konstruktion zu modifizieren ist, damit sie auch für partielle Funktionen arbeitet.

Übung 8.2.5 Betrachten Sie die folgende Turing-Maschine:

$$M = (\{q_0, q_1, q_2, q_f\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_f\})$$

Beschreiben Sie die Sprache $L(M)$ informell, jedoch klar, wenn δ aus den folgenden Regelmengen besteht:

* a) $\delta(q_0, 0) = (q_1, 1, R)$; $\delta(q_1, 1) = (q_0, 0, R)$; $\delta(q_1, B) = (q_f, B, R)$

b) $\delta(q_0, 0) = (q_0, B, R)$; $\delta(q_0, 1) = (q_1, B, R)$; $\delta(q_1, 1) = (q_1, B, R)$; $\delta(q_1, B) = (q_f, B, R)$

! c) $\delta(q_0, 0) = (q_1, 1, R)$; $\delta(q_1, 1) = (q_2, 0, L)$; $\delta(q_2, 1) = (q_0, 1, R)$; $\delta(q_1, B) = (q_f, B, R)$

8.3 Programmiertechniken für Turing-Maschinen

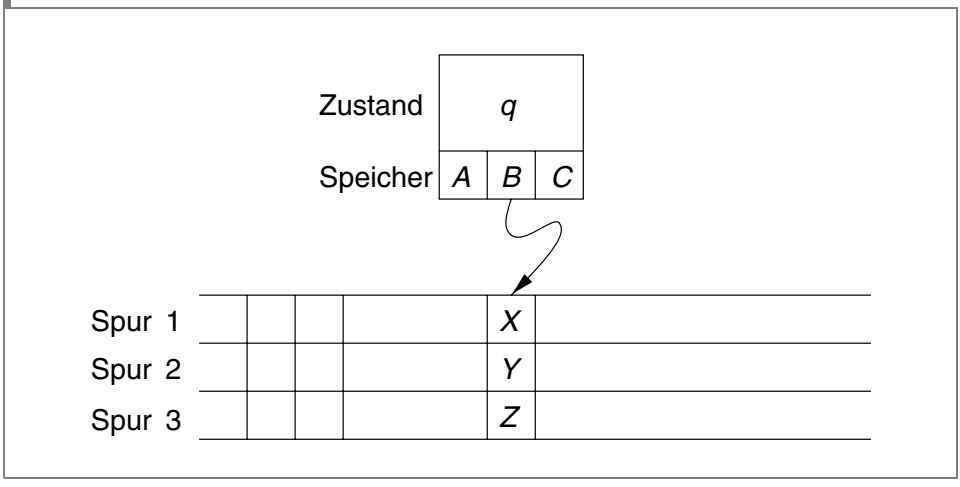
Wir möchten Ihnen einen Eindruck davon vermitteln, wie eine Turing-Maschine eingesetzt werden kann, um ähnlich wie ein konventioneller Computer Berechnungen auszuführen. Wir möchten Sie davon überzeugen, dass eine TM genauso leistungsfähig wie ein konventioneller Computer ist. Insbesondere werden Sie erfahren, dass die Turing-Maschine die Art von Berechnungen anderer Turing-Maschinen ausführen kann, die in Abschnitt 8.1.2 von einem Programm ausgeführt wurde, das andere Programme untersucht hat. Diese »introspektive« Fähigkeit von Turing-Maschinen und Computerprogrammen ermöglicht uns den Beweis, dass Probleme unentscheidbar sind.

Um die Fähigkeiten einer TM klarer darzustellen, erläutern wir anhand einiger Beispiele, wie wir uns das Band und die endliche Steuerung der Turing-Maschine vorstellen können. Keines dieser Beispiele erweitert das Basismodell der TM; es handelt sich lediglich um Abwandlungen der Notation. Später werden wir darauf zurückgreifen, um erweiterte Modelle von Turing-Maschinen zu simulieren, die gegenüber dem grundlegenden TM-Modell zusätzliche Eigenschaften besitzen, beispielsweise mehrere Bänder.

8.3.1 Im Zustand Daten speichern

Wir können die endliche Steuerung nicht nur zur Repräsentation einer Position im »Programm« der Turing-Maschine, sondern auch zum Speichern einer endlichen Datenmenge einsetzen. Abbildung 8.11 stellt diese Technik dar (und eine weitere Idee: mehrere Spuren). Die endliche Steuerung besteht nicht nur aus einem »Steuerzustand« q, sondern auch aus drei Datenelementen A, B und C. Diese Technik erfordert keine Erweiterung des TM-Modells; wir stellen uns den Zustand als ein Tupel vor. In Abbildung 8.11 entspricht $[q, A, B, C]$ diesem Tupel. Betrachten wir Zustände auf diese Weise, dann können wir Zustandsübergänge systematischer beschreiben, wodurch die Strategie hinter dem TM-Programm oft transparenter wird.

Abbildung 8.11: Eine Turing-Maschine mit Speicherung in der endlichen Steuerung und mehreren Spuren



Beispiel 8.6 Wir entwerfen eine TM

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], \{[q_1, B]\})$$

die in ihrer endlichen Steuerung das erste gelesene Symbol (0 oder 1) speichert und prüft, dass es an keiner anderen Stelle in der Eingabe auftritt. M akzeptiert daher die Sprache **01* + 10***. Das Akzeptieren regulärer Sprachen wie dieser übersteigt die Fähigkeiten von Turing-Maschinen nicht, sondern dient als einfaches Beispiel.

Die Menge der Zustände Q lautet $\{q_0, q_1\} \times \{0, 1, B\}$. Die Zustände können also als Paare angesehen werden, die aus zwei Komponenten bestehen:

a) einem Steuerabschnitt q_0 oder q_1, der speichert, was die TM ausführt. Steuerzustand q_0 zeigt an, dass M das erste Symbol noch nicht gelesen hat, während q_1 anzeigt, dass M das Symbol gelesen *hat* und prüft, ob es nicht noch an anderer Stelle vorkommt, indem sich M nach rechts bewegt und nach einer Zelle mit einem Leerzeichen sucht.

b) einem Datenabschnitt, der das erste gelesene Symbol, eine 0 oder eine 1, speichert. Das Symbol B in dieser Komponente bedeutet, dass noch kein Symbol gelesen wurde.

Die Übergangsfunktion δ von M lautet wie folgt:

1. $\delta([q_0, B], a) = ([q_1, a], a, R)$ für $a = 0$ oder $a = 1$. Zu Anfang ist q_0 der Steuerzustand, und B ist der Datenabschnitt des Zustands. Das gelesene Symbol wird in die zweite Komponente des Zustands kopiert; M bewegt sich nach rechts und tritt dabei in den Steuerzustand q_1 ein.

2. $\delta([q_1, a], \overline{a}) = ([q_1, a], \overline{a}, R)$, wobei $\overline{a}$ das »Komplement« von a ist, also 0, falls $a = 1$, oder 1, falls $a = 0$. In Zustand q_1 überspringt M jedes Symbol 0 oder 1, das sich von dem im Zustand gespeicherten Zeichen unterscheidet und bewegt sich dabei nach rechts.

3. $\delta([q_1, a], B) = ([q_1, B], B, R)$ für $a = 0$ oder $a = 1$. Findet M das erste Leerzeichen, wechselt M in den akzeptierenden Zustand $[q_1, B]$.

Beachten Sie, dass M keine Definition für $\delta([q_1, a], a)$ für $a = 0$ oder $a = 1$ besitzt. Findet M also ein zweites Auftreten des Zeichens, das zu Anfang in der endlichen Steuerung gespeichert wurde, hält M an, ohne den akzeptierenden Zustand zu erreichen. ■

8.3.2 Mehrere Spuren

Ein weiterer nützlicher »Trick« besteht darin, sich das Band einer Turing-Maschine so vorzustellen, als würde es sich aus mehreren Spuren zusammensetzen. Jede Spur kann ein Symbol speichern, und das Bandalphabet der TM besteht aus Tupeln mit einer Komponente für jede »Spur«. Beispielsweise enthält die Zelle, die vom Schreib-/Lesekopf in Abbildung 8.11 gelesen wird, das Symbol $[X, Y, Z]$. Wie die Technik des Speicherns in der endlichen Steuerung erweitern auch mehrere Spuren die Funktionalität der Turing-Maschine nicht. Es ist eine einfache Möglichkeit, Bandsymbole zu lesen und sich vorzustellen, dass sie eine nützliche Struktur besitzen.

Beispiel 8.7 Für gewöhnlich werden mehrere Spuren eingesetzt, um auf einer Spur Daten und auf einer anderen Markierungen zu speichern. Wir können jedes Symbol abhaken, wenn wir es »verwenden«, oder wir können einige Positionen innerhalb der Daten verfolgen, indem wir nur diese Positionen markieren. Die Beispiele 8.2 und 8.4 zeigen diese Technik, doch in keinem dieser Beispiele haben wir uns das Band explizit als aus mehreren Spuren bestehend vorgestellt. Im aktuellen Beispiel werden wir explizit eine zweite Spur einsetzen, um die nicht kontextfreie Sprache

$$L_{wcw} = \{wcw \mid w \text{ is in } (\mathbf{0} + \mathbf{1})^+ \text{ enthalten}\}$$

zu erkennen.

Wir werden die folgende Turing-Maschine entwerfen:

$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_9, B]\})$$

wobei gilt:

Q: Die Menge der Zustände besteht aus $\{q_1, q_2, \ldots, q_9\} \times \{0, 1\}$, also aus Paaren, die aus einem Steuerzustand q_i und einer Datenkomponente 0 oder 1 bestehen. Wir setzen die Technik des Speicherns in der endlichen Steuerung wiederum ein, indem der Zustand ein Eingabesymbol 0 oder 1 speichern kann.

Γ: $\{B, *\} \times \{0, 1, c, B\}$ ist die Menge der Bandsymbole. Die erste Komponente oder Spur kann leer oder »markiert« sein, was durch die Symbole B bzw. $*$ repräsentiert wird. Wir verwenden $*$, um Symbole der ersten und zweiten Gruppe von Nullen und Einsen zu markieren und damit zu bestätigen, ob die Zeichenreihe links von der zentralen Markierung c mit der Zeichenreihe rechts davon übereinstimmt. Die zweite Komponente des Bandsymbols beschreibt, wie wir das Bandsymbol selbst sehen. Wir stellen uns das Symbol $[B, X]$ als Bandsymbol X vor, wobei gilt: $X = 0, 1, c, B$.

Σ: $[B, 0]$ und $[B, 1]$ sind die Eingabesymbole, die wir, wie schon angemerkt, mit 0 bzw. 1 identifizieren.

δ: Die Übergangsfunktion δ wird durch die folgenden Regeln definiert, in denen a und b für 0 oder 1 stehen.

1. $\delta([q_1, B],[B, a]) = ([q_2, a], [*, a], R)$. Im Startzustand liest M das Symbol a (das entweder 0 oder 1 lautet), speichert es in der endlichen Steuerung, wechselt in Steuerzustand q_2, »hakt« das gerade gelesene Symbol ab und führt eine Bewegung nach rechts aus. Beachten Sie, dass das Abhaken durch die Änderung der ersten Komponente des Bandsymbols von B in $*$ erfolgt.
2. $\delta([q_2, a],[B, b]) = ([q_2, a], [B, b], R)$. M sucht rechtsgerichtet nach dem Symbol c. Sie wissen, dass a und b voneinander unabhängig 0 oder 1 sind, jedoch nicht c sein können.
3. $\delta([q_2, a],[B, c]) = ([q_3, a], [B, c], R)$. Trifft M auf das c, wird die Bewegung nach rechts nicht unterbrochen. M wechselt jedoch in den Zustand q_3.
4. $\delta([q_3, a],[*, b]) = ([q_3, a], [*, b], R)$. Im Zustand q_3 bewegt sich M über alle markierten Symbole hinweg.
5. $\delta([q_3, a],[B, a]) = ([q_4, B], [*, a], L)$. Findet M als erstes unmarkiertes Symbol das Gleiche wie das in der endlichen Steuerung, dann wird dieses Symbol markiert, da es dem entsprechenden Symbol aus dem ersten Block mit Nullen und Einsen entspricht. M wechselt in den Steuerzustand q_4, entfernt das Symbol aus der endlichen Steuerung und beginnt mit einer Bewegung nach links.
6. $\delta([q_4, B],[*, a]) = ([q_4, B], [*, a], L)$. M bewegt sich über alle markierten Symbole hinweg nach links.
7. $\delta([q_4, B],[B, c]) = ([q_5, B], [B, c], L)$. Trifft M auf das Symbol c, dann wechselt M in den Zustand q_5 und bewegt sich weiter nach links. Im Zustand q_5 muss M eine Entscheidung treffen, abhängig davon, ob das Symbol unmittelbar links vom c markiert oder unmarkiert ist. Ist es markiert, dann haben wir schon den gesamten ersten Block mit Nullen und Einsen geprüft – alle Zeichen links vom c. Wir müssen sicherstellen, dass alle Nullen und Einsen rechts vom c ebenfalls markiert werden, und akzeptieren, falls rechts vom c keine unmarkierten Zeichen verbleiben. Ist das Symbol unmittelbar links vom c nicht markiert, suchen wir nach dem am weitesten links stehenden unmarkierten Symbol, speichern es und starten den Zyklus, der mit Zustand q_1 beginnt, neu.
8. $\delta([q_5, B],[B, a]) = ([q_6, B], [B, a], L)$. Dieser Zweig behandelt den Fall, dass das Symbol links vom c nicht markiert ist. M wechselt in den Zustand q_6, fährt mit der Bewegung nach links fort und sucht nach einem markierten Symbol.
9. $\delta([q_6, B],[B, a]) = ([q_6, B], [B, a], L)$. M verbleibt im Zustand q_6 und bewegt sich nach links, solange die Symbole nicht markiert sind.
10. $\delta([q_6, B],[*, a]) = ([q_1, B], [*, a], R)$. Ist das markierte Symbol gefunden, wechselt M in den Zustand q_1 und bewegt sich nach rechts, um das erste nicht markierte Symbol zu lesen.

11. $\delta([q_5, B],[*, a]) = ([q_7, B], [*, a], R)$. Wir behandeln nun den Zweig im Zustand q_5, wenn unmittelbar links von c ein markiertes Symbol steht. M bewegt sich nach rechts und wechselt in den Zustand q_7.

12. $\delta([q_7, B],[B, c]) = ([q_8, B], [B, c], R)$. In Zustand q_7 sehen wir sicher das c. Dabei wechseln wir in Zustand q_8 und fahren rechtsgerichtet fort.

13. $\delta([q_8, B],[*, a]) = ([q_8, B], [*, a], R)$. M bewegt sich im Zustand q_8 nach rechts und überspringt jede markierte Null oder Eins.

14. $\delta([q_8, B],[B, B]) = ([q_9, B], [B, B], R)$. Erreicht M im Zustand q_8 eine leere Zelle, ohne auf eine nicht markierte Null oder Eins zu treffen, dann akzeptiert M. Findet M allerdings eine nicht markierte Null oder Eins, dann stimmen die Blöcke vor und hinter dem c nicht überein, und M hält an, ohne zu akzeptieren. ■

8.3.3 Unterprogramme

Wie bei Programmen allgemein, ist es hilfreich, Turing-Maschinen als eine Konstruktion miteinander zusammenarbeitender Komponenten oder »Unterprogramme« anzusehen. Ein Unterprogramm einer Turing-Maschine besteht aus einer Menge von Zuständen, die einen nützlichen Prozess ausführen. Diese Menge von Zuständen umfasst einen Startzustand sowie einen weiteren Zustand, der temporär keine Bewegung ausführt und als »Rückgabe«-Zustand dient, um die Steuerung an die andere Menge von Zuständen zu übergeben, von denen aus das Unterprogramm aufgerufen wurde. Der »Aufruf« eines Unterprogramms findet statt, wenn ein Zustandsübergang zu seinem Startzustand erfolgt. Da die TM keinen Mechanismus besitzt, um eine »Rücksprungadresse« zu speichern – also einen Zustand, in den nach Beendigung gewechselt wird, falls unser Entwurf einer TM vorsieht, dass ein Unterprogramm aus verschiedenen Zuständen aufgerufen wird –, können wir das Unterprogramm kopieren und für jede Kopie eine neue Menge von Zuständen verwenden. Die »Aufrufe« werden in den Startzuständen verschiedener Kopien des Unterprogramms ausgeführt, und jede Kopie führt einen »Rücksprung« in einen anderen Zustand aus.

Beispiel 8.8 Wir entwerfen eine TM zur Implementierung der Funktion »Multiplikation«. Unsere TM wird mit 0^m10^n1 auf dem Band starten und mit 0^{mn} auf dem Band enden. Ein Überblick über die Strategie:

1. Das Band enthält im Allgemeinen eine Zeichenreihe der Form $0^i10^n10^{kn}$ für ein i und ein k.

2. In einem Schritt ändern wir eine Null aus der ersten Gruppe in B und addieren dann n Nullen zur letzten Gruppe. Wir erhalten eine Zeichenreihe der Form $0^{i-1}10^n10^{(k+1)n}$.

3. Wir kopieren so die Gruppe mit n Nullen m-mal an das Ende, und zwar jeweils einmal, wenn wir in der ersten Gruppe eine 0 in B ändern. Nachdem die erste Gruppe mit Nullen vollständig in Leerzeichen geändert wurde, enthält die letzte Gruppe mn Nullen.

4. Im letzten Schritt werden die führenden 10^n1 in Leerzeichen geändert, und die Aufgabe ist abgeschlossen.

Den Kern dieses Algorithmus bildet das Unterprogramm `Copy`. Es implementiert Schritt (2) und kopiert den Block mit n Nullen an das Ende. Präziser ausgedrückt konvertiert `Copy` eine Konfiguration der Form $0^{m-k}1q_1 0^n 10^{(k-1)n}$ in die Konfiguration $0^{m-k}1q_5 0^n 10^{kn}$. Abbildung 8.12 zeigt die Übergänge des Unterprogramms `Copy`. Dieses Unterprogramm markiert die erste Null mit einem X, bewegt sich in Zustand q_2 nach rechts bis zu einem Leerzeichen, kopiert die 0 an diese Position und bewegt sich in Zustand q_3 nach links bis zur Markierung X. Dieser Zyklus wird wiederholt, bis in Zustand q_1 eine 1 statt einer 0 gefunden wird. Nun verwendet `Copy` den Zustand q_4, um die Zeichen X wieder in 0 zu ändern, und endet dann in Zustand q_5.

Die vollständige Turing-Maschine für Multiplikation startet in Zustand q_0. Sie wechselt zuerst in mehreren Schritten von Konfiguration $q_0 0^m 10^{n1}$ zur Konfiguration $0^{m-1}1q_1 0^n 1$. Die notwendigen Übergänge zeigt der Abschnitt links vom Unterprogrammaufruf in Abbildung 8.13. In diese Übergänge sind lediglich die Zustände q_0 und q_6 involviert.

Rechts vom Unterprogrammaufruf in Abbildung 8.13 sehen Sie die Zustände q_7 bis q_{12}. Diese Zustände übernehmen die Steuerung, nachdem `Copy` einen Block mit n Nullen kopiert hat und sich in Konfiguration $0^{m-k}1q_5 0^n 10^{kn}$ befindet. Die Zustände q_7, q_8, q_9 bringen uns zum Zustand $q_0 0^{m-k}10^n 10^{kn}$. Hier startet der Zyklus erneut, und `Copy` wird aufgerufen, um den Block mit n Nullen erneut zu kopieren.

Eine Ausnahme tritt auf, wenn die TM im Zustand q_8 auf die Situation trifft, dass alle m Nullen in Leerzeichen ($k = m$) geändert wurden. In diesem Fall tritt ein Wechsel in den Zustand q_{10} auf. Dieser Zustand ändert mit Hilfe des Zustands q_{11} die führenden $10^n 1$ in Leerzeichen und wechselt in den Haltezustand q_{12}. Nun befindet sich die TM in Konfiguration q_{12}0mn und die Aufgabe ist erledigt.

Abbildung 8.12: Das Unterprogramm `Copy`

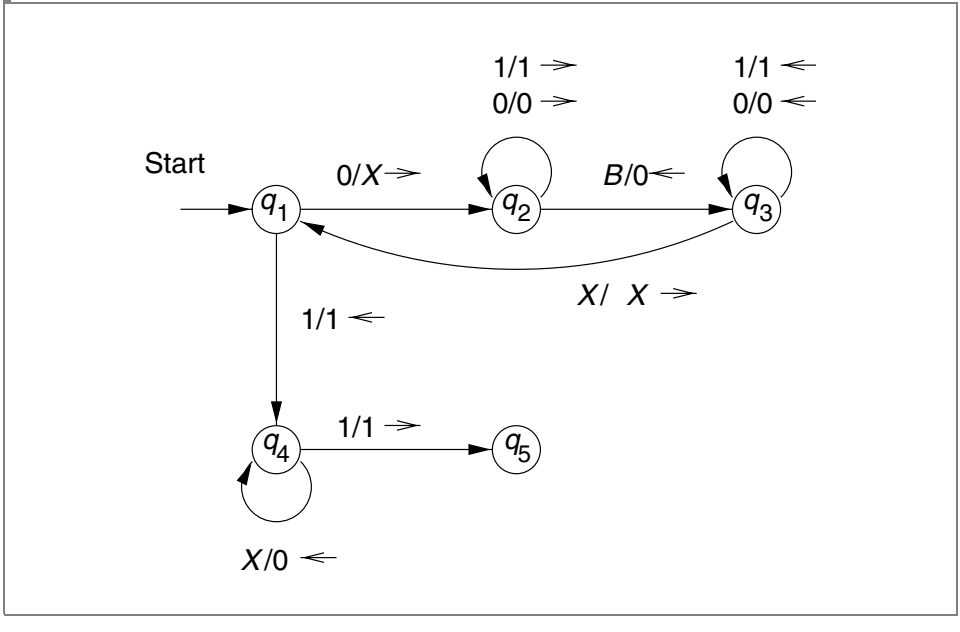
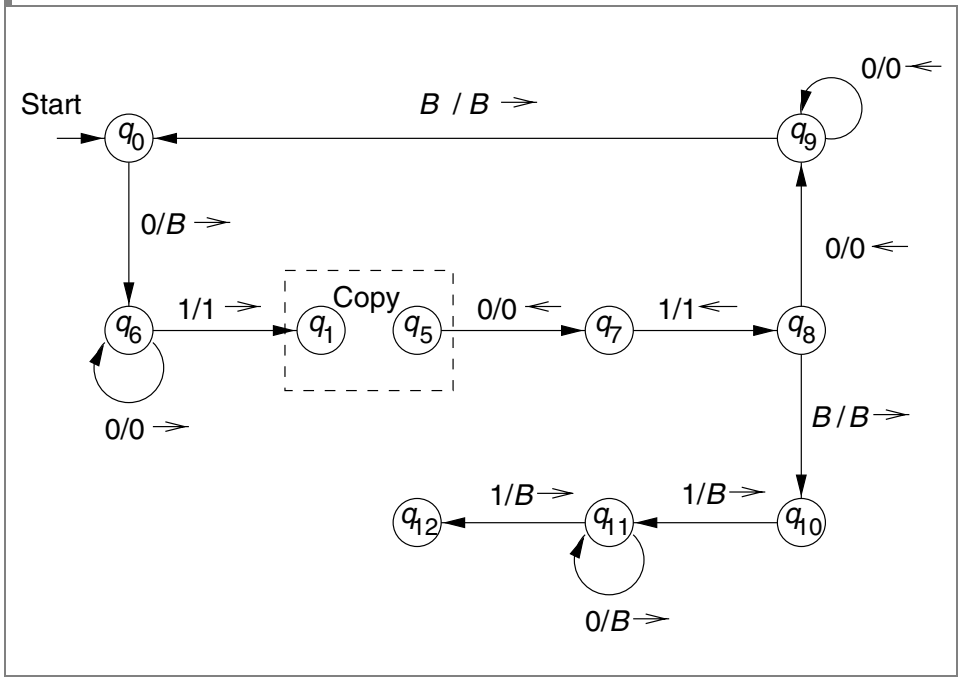


Abbildung 8.13: Das vollständige Multiplikationsprogramm verwendet das Unterprogramm Copy



8.3.4 Übungen zum Abschnitt 8.3

! **Übung 8.3.1** Ändern Sie Ihre Turing-Maschine aus Übung 8.2.2, sodass sie die in Abschnitt 8.3 vorgestellten Programmiertechniken nutzt.

! **Übung 8.3.2** Eine übliche Operation in Programmen von Turing-Maschinen ist das »Verschieben«. Im Idealfall würden wir an der aktuellen Position des Schreib-/Lesekopfes eine zusätzliche Zelle kreieren, in der wir ein Zeichen speichern könnten. Allerdings können wir das Band nicht auf diese Weise bearbeiten. Stattdessen müssen wir die Inhalte aller Zellen rechts von der aktuellen Position des Schreib-/Lesekopfes jeweils eine Zelle nach rechts verschieben und dann wieder zur aktuellen Position zurückkehren. Zeigen Sie, wie diese Operation ausgeführt wird.

Hinweis: Verwenden Sie ein besonderes Symbol, um die Position zu markieren, zu der der Schreib-/Lesekopf zurückkehren muss.

* **Übung 8.3.3** Entwerfen Sie ein Unterprogramm, um einen TM-Kopf von dessen aktueller Position nach rechts zu verschieben und dabei alle Nullen zu überspringen, bis eine Eins oder ein Leerzeichen erreicht ist. Ist an der aktuellen Position keine Null gespeichert, dann sollte die TM anhalten. Sie können annehmen, dass lediglich die Bandsymbole 0, 1 und B (Leerzeichen) vorhanden sind. Verwenden Sie dieses Unterprogramm dann, um eine TM zu entwerfen, die alle Zeichenreihen aus Nullen und Einsen akzeptiert, in denen niemals zwei Einsen unmittelbar nebeneinander vorkommen.

8.4 Erweiterte Turing-Maschinen

In diesem Abschnitt werden gewisse Computermodelle vorgestellt, die zu Turing-Maschinen in Beziehung stehen und die gleiche Leistungsfähigkeit bei der Spracherkennung wie das Basismodell einer TM besitzen, mit dem wir bisher gearbeitet haben. Eines davon, die mehrbändige Turing-Maschine, ist wichtig, da mit dieser Turing-Maschine die Simulation realer Computer (oder anderer Typen von Turing-Maschinen) wesentlich einfacher gezeigt werden kann, als es mit dem bisher vorgestellten einbändigen Modell möglich wäre. Allerdings erhöhen die zusätzlichen Bänder die Leistungsfähigkeit des Modells im Hinblick auf die Fähigkeit, Sprachen zu akzeptieren, nicht.

Danach betrachten wir die nichtdeterministische Turing-Maschine, eine Erweiterung des Basismodells, die in einer gegebenen Situation aus einer endlichen Menge von Bewegungen eine Auswahl treffen kann. Diese Erweiterung erleichtert in einigen Situationen ebenfalls die »Programmierung« von Turing-Maschinen, erhöht jedoch die Leistungsfähigkeit der Sprachdefinition des Basismodells nicht.

8.4.1 Turing-Maschinen mit mehreren Bändern

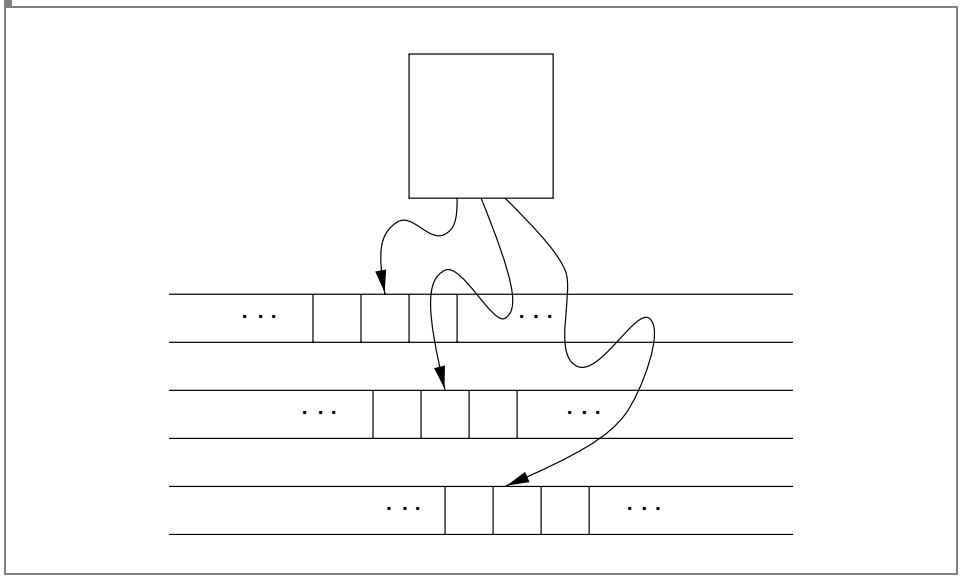
Abbildung 8.14 zeigt eine mehrbändige Turing-Maschine. Das Gerät besitzt eine endliche Steuerung (Zustand) und eine endliche Anzahl von Bändern. Jedes Band ist in Zellen aufgeteilt, und jede Zelle kann ein beliebiges Symbol aus dem endlichen Bandalphabet speichern. Wie bei der einbändigen TM enthält die Menge der Bandsymbole ein Leerzeichen sowie eine Teilmenge, die Eingabesymbole, in der das Leerzeichen nicht enthalten ist. Die Menge der Zustände enthält einen Startzustand sowie einige akzeptierende Zustände. Die TM wird wie folgt initialisiert:

1. Die Eingabe, eine endliche Folge von Eingabesymbolen, wird auf dem ersten Band gespeichert.
2. Alle weiteren Zellen auf allen Bändern enthalten Leerzeichen.
3. Die endliche Steuerung befindet sich im Startzustand.
4. Der Schreib-/Lesekopf des ersten Bandes befindet sich am linken Ende der Eingabe.
5. Alle weiteren Bandköpfe sind über einer beliebigen Zelle positioniert. Da abgesehen vom ersten Band alle Bänder leer sind, ist es nicht von Bedeutung, wo sich diese Köpfe anfänglich befinden; alle Zellen dieser Bänder »sehen« gleich aus.

Eine Bewegung der mehrbändigen TM hängt vom Zustand und dem Symbol ab, das jeder der Bandköpfe liest. Die mehrbändige TM führt in einer Bewegung Folgendes aus:

1. Die Steuerung wechselt in einen neuen Zustand, der mit dem vorherigen Zustand übereinstimmen kann.
2. Auf jedem Band wird ein neues Bandsymbol, das mit dem gelesenen übereinstimmen kann, in die gelesene Zelle geschrieben.

Abbildung 8.14: Eine Turing-Maschine mit mehreren Bändern



3. Jeder der Bandköpfe führt eine Bewegung nach links oder rechts aus oder bleibt stationär. Da sich die Köpfe unabhängig voneinander bewegen, können sich die Köpfe in verschiedene Richtungen bewegen oder an der gleichen Position verbleiben.

Wir werden hier nicht die formale Notation von Übergangsregeln angeben, deren Form einer einfachen Verallgemeinerung der Notation für die einbändige TM entspricht, mit Ausnahme der Richtung, die nun durch L, R oder S definiert wird. Bei einbändigen Maschinen haben wir keine stationären Köpfe erlaubt, und daher existierte die Option S nicht. Sie sollten sich eine passende Notation zur Beschreibung (Konfigurationen) der Konfiguration einer mehrbändigen TM vorstellen können. Wir werden diese formale Notation hier nicht darstellen. Mehrbändige Turing-Maschinen akzeptieren wie einbändige TMs, indem sie in einen akzeptierenden Zustand wechseln.

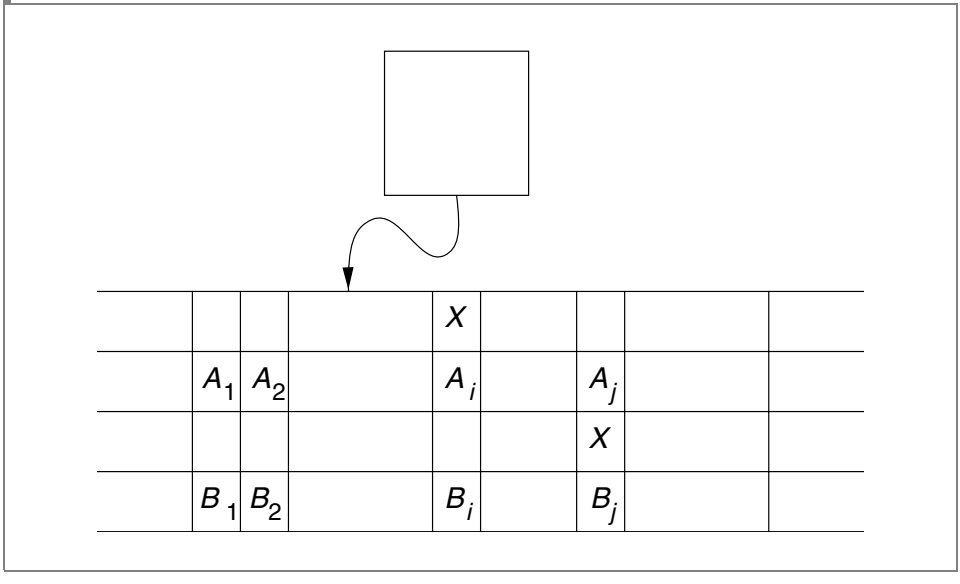
8.4.2 Äquivalenz zwischen ein- und mehrbändigen TMs

Sie wissen, dass die rekursiv aufzählbaren Sprachen als diejenigen Sprachen definiert sind, die eine einbändige TM akzeptiert. Natürlich akzeptieren mehrbändige TMs alle rekursiv aufzählbaren Sprachen, denn eine einbändige TM *ist* eine mehrbändige TM. Gibt es jedoch Sprachen, die nicht rekursiv aufzählbar sind, aber von mehrbändigen TMs akzeptiert werden? Die Antwort lautet »nein«, und wir werden diese Tatsache beweisen, indem wir zeigen, wie eine mehrbändige TM von einer einbändigen TM simuliert wird.

Satz 8.9 Jede Sprache, die von einer mehrbändigen TM akzeptiert wird, ist rekursiv aufzählbar.

Beweis: Abbildung 8.15 legt den Beweis nahe. Angenommen, die Sprache L wird von einer k-bändigen TM M akzeptiert. Wir simulieren M mit einer einbändigen TM N, deren Band wir uns aus $2k$ Spuren bestehend vorstellen. Die Hälfte dieser Spuren speichert die Bänder von M, die anderen Spuren speichern jeweils nur eine einzige Markierung, die anzeigt, wo sich der Kopf des zugehörigen Bandes von M gerade befindet. Abbildung 8.15 geht von der Annahme $k = 2$ aus. Die zweite und vierte Spur speichern die Inhalte der ersten und zweiten Spur von M, die erste Spur speichert die Position des Kopfes von Band 1, und Spur 3 speichert die Position des zweiten Bandkopfes.

Abbildung 8.15: Simulation einer zweibändigen Turing-Maschine durch eine einbändige Turing-Maschine



Um eine Bewegung von M zu simulieren, muss der Kopf von N die k Kopfmarkierungen besuchen. Damit N korrekt arbeitet, muss gespeichert werden, wie viele Kopfmarkierungen sich zu jedem Zeitpunkt links vom Kopf befinden. Dieser Zähler wird als eine Komponente der endlichen Steuerung von N gespeichert. Nachdem alle Markierungen besucht worden sind, weiß N, welche Bandsymbole von jedem der Köpfe von M gelesen werden. N kennt außerdem den Zustand von M, der in Ns eigener endlicher Steuerung gespeichert wird. N weiß daher, welche Bewegung M ausführen wird.

N besucht nun jede der Kopfmarkierungen auf dem eigenen Band erneut, ändert die Symbole auf dem Band in den Spuren, die den jeweiligen Bändern von M entsprechen, und verschiebt die Kopfmarkierungen, falls notwendig, nach links oder nach rechts. Zuletzt ändert N den Zustand von M, wie er in der eigenen endlichen Steuerung gespeichert ist. Zu diesem Zeitpunkt hat N eine Bewegung von M simuliert.

Als Ns akzeptierende Zustände wählen wir genau die Zustände, die Ms Zustand als einen der akzeptierenden Zustände von M aufzeichnen. N akzeptiert also genau dann, wenn die simulierte TM M akzeptiert. ■

Ein Exkurs über die Endlichkeit

Häufig wird der Fehler begangen, dass ein Wert, der zu jedem Zeitpunkt endlich ist, mit einer endlichen Menge von Werten verwechselt wird. Die Konstruktion Viele-Bänder-in-eins mag uns den Unterschied verdeutlichen. Wir haben in dieser Konstruktion Spuren auf dem Band verwendet, um die Positionen der Bandköpfe aufzuzeichnen. Warum konnten wir diese Positionen nicht als ganze Zahlen in der endlichen Steuerung speichern? Man könnte nachlässig argumentieren, dass die TM nach n Bewegungen Bandkopfpositionen festhalten muss, die sich innerhalb von n Positionen der ursprünglichen Bandpositionen befinden, und der Kopf somit lediglich ganze Zahlen = n speichern muss.

Das Problem besteht darin, dass die vollständige Menge von *möglichen* Positionen zu jedem Zeitpunkt unendlich (n ist zwar endlich, aber unbeschränkt) ist, die Positionen selbst zu jedem Zeitpunkt dagegen endlich sind. Soll der Zustand jede Kopfposition repräsentieren, dann muss es eine Datenkomponente der Zustände geben, die eine beliebige ganze Zahl als Wert besitzt. Diese Komponente erzwänge, dass die Menge der Zustände unendlich ist, auch wenn nur eine endliche Anzahl von Zuständen zu jedem endlichen Zeitpunkt verwendet werden kann. Die Definition einer Turing-Maschine fordert aber, dass die *Menge* der Zustände endlich ist. Daher ist es nicht zulässig, die Position eines Bandkopfes in der endlichen Steuerung zu speichern.

8.4.3 Ausführungszeit und die Viele-Bänder-in-eins-Konstruktion

Sie lernen nun ein Konzept kennen, das später sehr wichtig sein wird: die »Zeitkomplexität« oder »Ausführungszeit« einer Turing-Maschine. Die *Ausführungszeit* der TM M mit der Eingabe w ist die Anzahl der Schritte, die M vor dem Anhalten ausführt. Hält M mit w nicht an, dann ist die Ausführungszeit von M mit w unendlich. Die Zeitkomplexität von TM M ist die Funktion $T(n)$, das ist das Maximum der Ausführungszeiten von M mit w für alle Eingaben w der Länge n. Für Turing-Maschinen, die nicht mit allen Eingaben anhalten, kann $T(n)$ für einige oder sogar für alle n unendlich sein. Wir achten besonders auf TMs, die bei allen Eingaben anhalten, insbesondere auf diejenigen, die eine polynomiale Zeitkomplexität $T(n)$ besitzen; in dieses Thema wird in Abschnitt 10.1 eingeführt.

Die Konstruktion im Beweis von Satz 8.9 scheint schwerfällig. Tatsächlich könnte die konstruierte einbändige TM eine wesentlich längere Ausführungszeit als die mehrbändige TM benötigen. Allerdings sind die Zeiträume, die beide Turing-Maschinen benötigen, nicht immens verschieden: Der Zeitaufwand der einbändigen TM ist niemals größer als das Quadrat des Zeitaufwands der mehrbändigen TM. »Quadrieren« ist keine sehr gute Garantie, doch bewahrt dies polynomiale Ausführungszeit. In Kapitel 10 werden Sie Folgendes erfahren:

a) Der Unterschied zwischen polynomialem Zeitaufwand und höheren Wachstumsraten der Ausführungszeit bildet tatsächlich die Grenze zwischen dem, was wir mit Computern bearbeiten können, und dem, was praktisch nicht lösbar ist.

b) Trotz intensiver Forschung liegt die Ausführungszeit, die zum Lösen vieler Probleme notwendig ist, noch immer im polynomialen Bereich. Daher ist bei der Untersuchung des benötigten Zeitaufwands zur Lösung eines bestimmten Problems die Frage, ob wir eine ein- oder eine mehrbändige TM verwenden, nicht kritisch.

Wir beweisen nun, dass die Ausführungszeit der einbändigen TM innerhalb des Quadrats der Ausführungszeit der mehrbändigen TM liegt:

Satz 8.10 Der Zeitaufwand, den die einbändige TM N aus Satz 8.9 benötigt, um n Bewegungen der k-bändigen TM M zu simulieren, ist $O(n^2)$.

Beweis: Nach n Bewegungen von M können sich die Bandkopfmarkierungen in N nicht weiter als $2n$ Zellen voneinander entfernt haben. Wenn N an der am weitesten links stehenden Markierung beginnt, kann die Suche nach allen Kopfmarkierungen daher nicht weiter als $2n$ Zellen nach rechts führen. M kann danach linksgerichtet die Inhalte der simulierten Bänder von M ändern und die Kopfmarkierungen nach Bedarf nach links oder rechts verschieben. Dies erfordert nicht mehr als $2n$ Bewegungen nach links plus höchstens $2k$ Bewegungen, um die Richtung zu ändern und eine Markierung X in die Zelle unmittelbar rechts davon zu schreiben (falls ein Bandkopf von M eine rechtsgerichtete Bewegung ausführt).

N benötigt daher höchstens $4n + 2k$ Bewegungen, um eine der ersten n Bewegungen zu simulieren. Da k eine Konstante und somit von der Anzahl der simulierten Bewegungen unabhängig ist, ist $O(n)$ die Anzahl der Bewegungen. Die Simulation von n Bewegungen erfordert höchstens das n-fache dieser Anzahl, also $O(n^2)$. ■

8.4.4 Nichtdeterministische Turing-Maschinen

Eine *nichtdeterministische* Turing-Maschine (*NTM*) unterscheidet sich von der bisher vorgestellten deterministischen Variante durch ihre Übergangsfunktion δ, sodass für jeden Zustand q und jedes Bandsymbol X $\delta(q, X)$ eine Menge von Tripeln

$$\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

ist, wobei k eine beliebige endliche ganze Zahl ist. Die NTM kann bei jedem Schritt wählen, welches Tripel die nächste Bewegung sein wird. Jedoch kann nicht ein Zustand aus einem Tripel, ein Bandsymbol aus einem anderen und die Richtung aus einem weiteren gewählt werden.

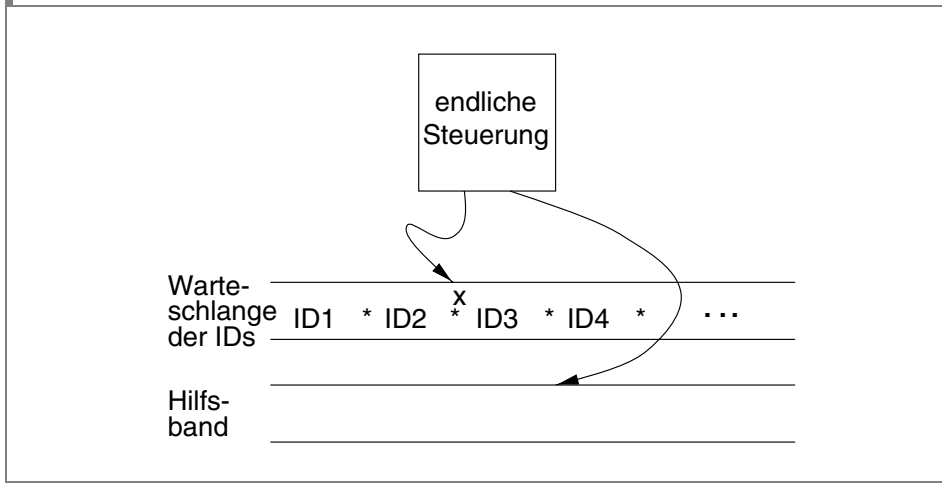
Die von einer NTM M akzeptierte Sprache wird erwartungsgemäß analog zu anderen nichtdeterministischen Geräten wie den vorgestellten NEAs und PDAs definiert. Das heißt, M akzeptiert eine Eingabe, wenn es eine Folge möglicher Bewegungen gibt, die von der anfänglichen Konfiguration mit w als Eingabe zu einer Konfiguration mit einem akzeptierenden Zustand führt. Die Existenz weiterer Auswahlmöglichkeiten, die nicht zu einem akzeptierenden Zustand führen, ist wie bei NEAs und PDAs irrelevant.

NTMs akzeptieren keine Sprache, die nicht auch von einer deterministischen TM (oder *DTM*, wenn betont werden soll, dass es sich um eine deterministische TM handelt) akzeptiert wird. Im Beweis wird gezeigt, dass wir für jede NTM M_N eine DTM M_D konstruieren können, die die Konfigurationen erforscht, die M_N über eine Folge aus den Wahlmöglichkeiten erreichen kann. Findet M_D eine Folge mit einem akzeptierenden Zustand, dann geht M_D selbst in einen akzeptierenden Zustand. M_D muss systematisch neue Konfigurationen an eine Warteschlange anfügen, statt sie auf einem Stack abzulegen, damit M_D nach endlicher Zeit alle Folgen von bis zu k Bewegungen von M_N für $k = 1, 2, \ldots$ simuliert.

Satz 8.11 Ist M_N eine nichtdeterministische Turing-Maschine, dann gibt es eine deterministische Turing-Maschine, sodass $L(M_N) = L(M_D)$.

Beweis: M_D wird als mehrbändige TM entworfen, wie in Abbildung 8.16 gezeigt. Das erste Band von M_D speichert eine Folge von Konfigurationen von M_N, einschließlich des Zustands von M_N. Eine dieser Konfigurationen ist als »aktuelle« Konfiguration markiert, wobei die nachfolgenden Konfigurationen während der Verarbeitung erkannt werden. In Abbildung 8.16 ist die dritte Konfiguration durch ein x sowie den Konfigurations-Separator * markiert. Alle Konfigurationen links von der aktuellen wurden schon untersucht und können nachfolgend ignoriert werden.

Abbildung 8.16: Simulation einer NTM durch eine DTM



Während der Verarbeitung der aktuellen Konfiguration führt M_D Folgendes aus:

1. M_D untersucht Zustand und gelesenes Symbol der aktuellen Konfiguration. In die endliche Steuerung von M_D ist das Wissen integriert, welche Auswahlmöglichkeiten für Bewegungen M_N in jedem Zustand und für jedes Symbol zur Verfügung stehen. Akzeptiert der Zustand in der aktuellen Konfiguration, dann akzeptiert M_D und bricht die Simulation von M_N ab.
2. Akzeptiert der Zustand jedoch nicht und besitzt die Kombination Zustand–Symbol k Wahlmöglichkeit für Bewegungen, dann verwendet M_D das zweite Band, um die Konfiguration zu kopieren und dann k Kopien dieser Konfiguration am Ende der Konfigurations-Warteschlange auf Band 1 zu speichern.
3. M_D ändert jede dieser k Konfigurationen entsprechend den k Wahlmöglichkeiten für Bewegungen, die M_N bei der aktuellen Konfiguration hat.
4. M_D kehrt zur markierten aktuellen Konfiguration zurück, löscht die Markierung und markiert stattdessen die nächste Konfiguration unmittelbar rechts davon. Der Zyklus wird nun mit Schritt (1) wiederholt.

Es sollte offensichtlich sein, dass die Simulation in dem Sinn akkurat ist, als M_D nur dann akzeptiert, wenn M_N zu einer akzeptierenden Konfiguration kommen kann. Wir müssen allerdings Folgendes bestätigen: Erreicht M_N nach einer Folge von n eigenen Bewegungen eine akzeptierende Konfiguration, dann wird M_D diese Konfiguration schließlich einmal als aktuelle Konfiguration erreichen und akzeptieren.

Angenommen, m ist die maximale Anzahl von Wahlmöglichkeiten, die M_N in ihren Konfigurationen (IDs) hat. Dann gibt es eine Start-Konfiguration, und M_N kann bis zur ersten Bewegung ausschließlich höchstens $1 + m$ Konfigurationen erreichen. Bis zur zweiten Bewegung einschließlich sind höchstens $1 + m + m^2$ Konfigurationen erreichbar und so weiter. Nach bis zu n Bewegungen kann M_N also höchstens $1 + m + m^2 + \ldots + m^n$ Konfigurationen erreichen. Diese Anzahl beträgt höchstens nm^n Konfigurationen für $m, n > 1$.

Die Reihenfolge, in der M_D Konfigurationen von M_N untersucht, entspricht einer »Breitensuche«. M_D prüft also alle Konfigurationen, die mit 0 Bewegungen (also die anfängliche Konfiguration) erreichbar sind, dann alle Konfigurationen, die mit einer Bewegung erreichbar sind, dann die mit zwei Bewegungen erreichbaren und so weiter. Insbesondere wird M_D alle Konfigurationen zur aktuellen Konfiguration machen und deren Nachfolger untersuchen, die mit bis zu n Bewegungen erreichbar sind, bevor andere Konfigurationen behandelt werden, die nur mit mehr als n Bewegungen zu erreichen sind.

Wenn also M_N in endlicher Zeit, d. h. für eine ganze Zahl n nach n Bewegungen einen akzeptierenden Zustand erreichen kann, dann erreicht M_D nach höchstens nm^n Bewegungen, d. h. in endlicher Zeit, einen akzeptierenden Zustand. Wenn M_N also akzeptiert, dann akzeptiert auch M_D. Da wir schon gesehen haben, dass M_D nur dann akzeptiert, wenn auch M_N akzeptiert, schließen wir, dass $L(M_N) = L(M_D)$. ■

Beachten Sie, dass die konstruierte deterministische TM exponentiell mehr Zeit als die nichtdeterministische TM benötigen kann. Es ist nicht bekannt, ob diese exponentielle Verlangsamung unvermeidlich ist. Kapitel 10 behandelt diese Frage und die Konsequenzen, falls jemand eine bessere Möglichkeit findet, NTMs deterministisch zu simulieren.

8.4.5 Übungen zum Abschnitt 8.4

Übung 8.4.1 Beschreiben Sie informell, aber klar mehrbändige Turing-Maschinen, die jede der Sprachen aus Übung 8.2.2 akzeptieren. Versuchen Sie zu erreichen, dass der Zeitaufwand jeder Ihrer Turing-Maschinen proportional zur Länge der Eingabe ist.

Übung 8.4.2 Es folgt die Übergangsfunktion einer nichtdeterministischen TM $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_2\})$:

δ	0	1	B
q_0	$\{(q_0, 1, R)\}$	$\{(q_1, 0, R)\}$	$\emptyset$
q_1	$\{(q_1, 0, R), (q_0, 0, L)\}$	$\{(q_1, 1, R), (q_0, 1, L)\}$	$\{(q_2, B, R)\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$

Zeigen Sie, welche Konfigurationen von der Start-Konfiguration erreichbar sind, wenn die Eingabe folgendermaßen lautet:

* a) 01

b) 001

! **Übung 8.4.3** Beschreiben Sie informell, aber klar nichtdeterministische Turing-Maschinen (wenn Sie möchten, mehrbändige), die die folgenden Sprachen akzeptieren. Versuchen Sie, den Nichtdeterminismus zu nutzen, um Iteration zu vermeiden und Zeit im nichtdeterministischen Sinn einzusparen. Gestalten Sie Ihre NTM also so, dass sie zwar stark verzweigt, aber jeder Zweig kurz ist.

* a) Die Sprache aller Zeichenreihen aus Nullen und Einsen, die eine Zeichenreihe der Länge 100 enthalten, die sich wiederholt, jedoch nicht notwendigerweise unmittelbar aufeinander folgend. Formal besteht diese Sprache aus der Menge der Zeichenreihen aus Nullen und Einsen der Form $wxyxz$, wobei $|x| = 100$ und w, y und z von beliebiger Länge sind.

b) Die Sprache aller Zeichenreihen der Form $w_1\#w_2\#\ldots\#w_n$ für ein beliebiges (festes) n, sodass jedes w_i eine Zeichenreihe aus Nullen und Einsen sowie für irgendein j w_j die ganze Zahl j in Binärdarstellung ist.

c) Die Sprache aller Zeichenreihen von der gleichen Form wie in b), doch für mindestens zwei Werte von j ist w_j die ganze Zahl j in Binärdarstellung.

! **Übung 8.4.4** Betrachten Sie die nichtdeterministische Turing-Maschine

$$M = (\{q_0, q_1, q_2, q_f\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_f\})$$

Beschreiben Sie die Sprache $L(M)$ informell, aber klar, wenn δ aus der folgenden Regelmenge besteht: $\delta(q_0, 0) = \{(q_0, 1, R), (q_1, 1, R)\}$; $\delta(q_1, 1) = \{(q_2, 0, L)\}$; $\delta(q_2, 1) = \{(q_0, 1, R)\}$; $\delta(q_1, B) = \{(q_f, B, R)\}$.

* **Übung 8.4.5** Betrachten Sie eine nichtdeterministische TM, deren Band in beiden Richtungen unendlich lang ist. Zu einem Zeitpunkt ist das Band vollkommen leer, abgesehen von einer Zelle, in der das Symbol \$ gespeichert ist. Der Kopf befindet sich zu diesem Zeitpunkt über einer leeren Zelle, und q ist der Zustand.

a) Schreiben Sie Übergänge, die der NTM erlauben, mit dem Kopf über dem Symbol \$ den Zustand p zu erreichen.

! b) Angenommen, die TM sei stattdessen deterministisch. Wie würden Sie der TM ermöglichen, das Symbol \$ zu finden und in Zustand p zu wechseln?

Übung 8.4.6 Entwerfen Sie die folgende zweibändige TM, um die Sprache aller Zeichenreihen aus Nullen und Einsen, die jeweils gleich oft vorkommen, zu akzeptieren. Das erste Band enthält die Eingabe und wird von links nach rechts gelesen. Das zweite Band dient zum Speichern des Überschusses von Nullen über Einsen (bzw. umgekehrt) aus dem bisher gelesenen Abschnitt der Eingabe. Spezifizieren Sie die Zustände, die Übergänge und den intuitiven Zweck jedes Zustands.

Übung 8.4.7 In dieser Übung werden wir unter Verwendung einer besonderen dreibändigen TM einen Stack implementieren.

1. Das erste Band dient nur zum Speichern und Lesen der Eingabe. Das Eingabealphabet besteht aus dem Symbol ↑, das wir als »Entfernen vom Stack« interpretieren, sowie den Symbolen a und b, die als »Hinzufügen von a (bzw. b) auf den Stack« zu interpretieren sind.

2. Das zweite Band dient zum Speichern des Stacks.
3. Das dritte Band ist das Ausgabeband. Jedes Mal, wenn ein Symbol vom Stack entfernt wird, muss es hinter die bisher geschriebenen Symbole auf das Ausgabeband geschrieben werden.

Die Turing-Maschine muss mit einem leeren Stack beginnen und die Folge der Push&Pop-Operationen (Hinzufügen und Entfernen) von links nach rechts implementieren, wie es die Eingabe vorgibt. Bedingt die Eingabe, dass die TM einmal versucht, ein Symbol vom leeren Stack zu entfernen, dann muss sie in einem besonderen Fehlerzustand q_e anhalten. Ist nach Einlesen der vollständigen Eingabe der Stack leer, dann wird die Eingabe durch Übergang in den Zustand q_f akzeptiert. Beschreiben Sie die Übergangsfunktion der TM informell, aber klar. Beschreiben Sie außerdem den Zweck jedes von Ihnen verwendeten Zustands.

Übung 8.4.8 Abbildung 8.15 zeigt ein Beispiel für die allgemeine Simulation einer k-bändigen TM durch eine einbändige TM.

* a) Angenommen, diese Technik wird verwendet, um eine fünfbändige TM zu simulieren, die ein Bandalphabet aus sieben Symbolen besitzt. Wie viele Bandsymbole würde die einbändige TM besitzen?

* b) Alternativ können k Bänder durch ein einziges Band simuliert werden, indem eine Spur $k + 1$ zum Speichern der Kopfpositionen aller k Bänder verwendet wird, während die ersten k Spuren die k Bänder in offensichtlicher Weise simulieren. Beachten Sie, dass auf der Spur $k + 1$ sorgfältig zwischen den Bandköpfen unterschieden und auch die Möglichkeit in Betracht gezogen werden muss, dass zwei oder mehr Bandköpfe über der gleichen Zelle positioniert sind. Reduziert diese Methode die Anzahl der Bandsymbole, die für die einbändige TM notwendig sind?

c) Bänder können auch durch ein einziges Band simuliert werden, indem das Speichern der Kopfpositionen vollständig vermieden wird. Stattdessen wird eine Spur $k + 1$ verwendet, um eine einzige Zelle des Bandes zu markieren. Zu jedem Zeitpunkt ist jedes simulierte Band so auf seiner Spur positioniert, dass sich der Kopf über der markierten Zelle befindet. Verschiebt die k-bändige TM den Kopf von Band i, dann verschiebt die simulierende einbändige TM den gesamten nicht leeren Inhalt der Spur i um eine Zelle in die entgegengesetzte Richtung, sodass die markierte Zelle noch immer die Zelle enthält, die der Bandkopf i der k-bändigen TM liest. Lässt sich mit dieser Methode die Anzahl der Bandsymbole der einbändigen TM reduzieren? Hat diese Methode gegenüber den anderen vorgestellten Methoden Nachteile?

! **Übung 8.4.9** Eine *k-köpfige* Turing-Maschine ist mit k Köpfen ausgestattet, um Zellen eines Bandes zu lesen. Eine Bewegung dieser TM hängt vom Zustand und dem von jedem Kopf gelesenen Symbol ab. Die TM kann mit einer Bewegung den Zustand ändern, ein Symbol in die von jedem Kopf gelesenen Zellen schreiben sowie jeden Kopf nach links oder rechts bewegen bzw. an der gleichen Position belassen. Da mehrere Köpfe die gleiche Zelle lesen können, nehmen wir an, dass die Köpfe von 1 bis k durchnummeriert sind und nur das Symbol, das der Kopf mit der höchsten Nummer schreibt, tatsächlich in die Zelle geschrieben wird. Beweisen Sie, dass die von TMs mit

k Köpfen akzeptierten Sprachen mit denjenigen übereinstimmen, die von gewöhnlichen TMs akzeptiert werden.

!! **Übung 8.4.10** Eine *zweidimensionale* Turing-Maschine besitzt die übliche endliche Steuerung, deren Band jedoch aus einem zweidimensionalen Zellenraster besteht, das in allen Richtungen unendlich ist. Die Eingabe wird in einer Zeile des Rasters abgelegt, wobei sich, wie üblich, der Kopf am linken Ende der Eingabe und die endliche Steuerung im Startzustand befindet. Diese TM akzeptiert ebenfalls durch den Übergang in einen akzeptierenden Zustand. Beweisen Sie, dass die von zweidimensionalen Turing-Maschinen akzeptierten Sprachen mit denjenigen übereinstimmen, die von gewöhnlichen TMs akzeptiert werden.

8.5 Beschränkte Turing-Maschinen

Wir haben Verallgemeinerungen von Turing-Maschinen kennen gelernt, die jedoch deren Leistungsfähigkeit hinsichtlich Spracherkennung in keiner Weise erweitern. Nun werden wir einige Beispiele offensichtlicher Beschränkungen von Turing-Maschinen betrachten, die ebenfalls die gleiche Leistungsfähigkeit in Bezug auf Spracherkennung aufweisen. Unsere erste Beschränkung ist zwar geringfügig, bei vielen späteren Konstruktionen jedoch nützlich: Wir ersetzen das in beiden Richtungen unendliche TM-Band durch ein Band, das nur nach rechts unendlich ist. Außerdem verbieten wir, dass diese beschränkte TM ein Leerzeichen als ersetzendes Bandsymbol schreibt. Der Wert dieser Beschränkungen besteht darin, dass wir davon ausgehen können, dass die Konfigurationen nur aus nicht leeren Symbolen bestehen und immer am linken Ende der Eingabe beginnen.

Danach werden wir bestimmte Typen mehrbändiger Turing-Maschinen untersuchen, bei denen es sich um generalisierte Pushdown-Automaten (PDA) handelt. Zuerst beschränken wir die Bänder der TM, sodass sie sich wie Stacks verhalten. Dann beschränken wir die Bänder auf die Funktionsweise als »Zähler«; sie können also lediglich eine ganze Zahl repräsentieren, und die TM kann nur die Zahl 0 von einer Zahl ungleich 0 unterscheiden. Die Auswirkungen dieser Diskussion liegen darin, dass es einige sehr einfache Typen von Automaten gibt, die die volle Leistungsfähigkeit von Computern besitzen. Außerdem sind unentscheidbare Probleme hinsichtlich Turing-Maschinen, die in Kapitel 9 behandelt werden, auch auf diese einfachen Maschinen übertragbar.

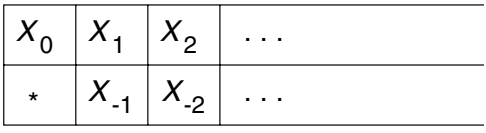
8.5.1 Turing-Maschinen mit semiunendlichen Bändern

Wir haben bisher dem Bandkopf einer Turing-Maschine gestattet, von der Startposition Bewegungen nach links und nach rechts auszuführen, doch es genügt zuzulassen, dass der Kopf nur Positionen am und rechts vom Anfang einnehmen kann. Wir können annehmen, dass das Band *semiunendlich* ist, dass also links von der Anfangsposition des Kopfes keine Zellen existieren. Im nächsten Theorem (Satz) werden wir eine Konstruktion vorstellen, die zeigt, wie eine TM mit einem semiunendlichen Band eine TM simuliert, die wie unser ursprüngliches TM-Modell ein beidseitig unendliches Band besitzt.

Der Trick bei der Konstruktion besteht darin, auf dem semiunendlichen Band zwei Spuren zu verwenden. Die obere Spur repräsentiert die Zellen der Original-TM, die sich rechts von der anfänglichen Kopfposition befinden, einschließlich der Zelle an

dieser Position selbst. Die untere Spur repräsentiert die Positionen links von der anfänglichen Position, jedoch in umgekehrter Reihenfolge. Abbildung 8.17 zeigt die exakte Anordnung. Die obere Spur repräsentiert die Zellen X_0, X_1,…, wobei X_0 die anfängliche Kopfposition bezeichnet; X_1, X_2 und so weiter stehen für die Zellen rechts davon. Die Zellen X_{-1}, X_{-2} und so weiter repräsentieren die Zellen links von der Startposition. Achten Sie auf das Zeichen * in der am weitesten links stehenden Zelle der unteren Spur. Dieses Symbol dient als Endemarkierung und verhindert, dass der Kopf der semiunendlichen TM versehentlich das linke Ende des Bandes verlässt.

Abbildung 8.17: Ein semiunendliches Band kann ein beidseitig unendliches Band simulieren



Wir erlegen unserer Turing-Maschine eine Beschränkung auf: Sie schreibt keine Leerzeichen. Diese einfache Einschränkung bedeutet zusammen mit der Einschränkung, dass das Band semiunendlich ist, dass das Band immer ein Präfix aus nicht leeren Symbolen ist, denen unendlich viele Leerzeichen folgen. Außerdem beginnt die Folge der nicht leeren Symbole immer an der anfänglichen Kopfposition. Wir sehen in Satz 9.19 und erneut in Satz 10.9, wie nützlich die Annahme ist, dass die Konfigurationen diese Form haben.

Satz 8.12 Jede von einer TM M_2 akzeptierte Sprache wird auch von einer TM M_1 akzeptiert, die wie folgt eingeschränkt ist:

1. Der Kopf von M_1 bewegt sich niemals auf eine Position links von dessen anfänglicher Position.
2. M_1 schreibt nie Leerzeichen.

Beweis: Bedingung (2) ist sehr einfach. Kreiert wird ein neues Bandsymbol B', das als Leerzeichen dient, jedoch nicht das Leerzeichen B ist, d. h.:

a) Besitzt M_2 eine Regel der Form $\delta_2(q, X) = (p, B, D)$, dann wird sie in $\delta_2(q, X) = (p, B', D)$ geändert.

b) Für jeden Zustand q soll $\delta_2(q, B')$ mit $\delta_2(q, B)$ übereinstimmen.

Bedingung (1) erfordert mehr Arbeit. Sei

$$M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_2, B, F_2)$$

die TM M_2 mit den oben aufgeführten Änderungen, sodass sie nie das Leerzeichen B schreibt. Wir konstruieren

$$M_1 = (Q_1, \Sigma \times \{B\}, \Gamma_1, \delta_1, q_0, [B, B], F_1)$$

wobei Folgendes gilt:

Q_1: M_1 besitzt die Zustände $\{q_0, q_1\} \cup (Q_2 \times \{U, L\})$. M_1 besitzt also einen Startzustand q_0, einen weiteren Zustand q_1 und alle Zustände von M_2 mit einer zweiten Datenkomponente, entweder U oder L (Upper oder Lower). Die zweite Komponente gibt an, ob die obere oder untere Spur (wie in Abbildung 8.17) von M_2 gelesen wird. Anders ausgedrückt bedeutet U, dass sich der Kopf von M_2 an seiner Anfangsposition oder rechts davon befindet, L dagegen sagt aus, dass er sich links davon befindet.

Γ_1: Die Bandsymbole von M_1 bestehen alle aus Symbolpaaren aus Γ_2, d. h. $\Gamma_2 \times \Gamma_2$. Eingabesymbole von M_1 sind Paare, deren erste Komponente ein Eingabesymbol von M_2 und deren zweite Komponente das Leerzeichen ist, also Paare der Form $[a, B]$, wobei a in Σ enthalten ist. Das Leerzeichen in M_1 hat in beiden Komponenten Leerzeichen. Zusätzlich gibt es für jedes Symbol X in Γ_2 ein Paar $[X, *]$ in Γ_1. $*$ ist ein neues Symbol und nicht in Γ_2 enthalten, und es dient als Markierung des linken Bandendes von M_1.

δ_1: M_1 besitzt folgende Übergänge:

1. $\delta_1(q_0, [a, B]) = (q_1, [a, *], R)$ für jedes a aus Σ. Die erste Bewegung von M_1 speichert die Markierung $*$ in der am weitesten links stehenden Zelle der unteren Spur. Zustand q_1 wird erreicht, und der Kopf bewegt sich nach rechts, da er sich weder nach links bewegen noch stationär bleiben kann.

2. $\delta_1(q_1, [X, B]) = ([q_2, U], [X, B], L)$ für jedes X aus Γ_2. Im Zustand q_1 baut M_1 die anfänglichen Bedingungen von M_2 auf, indem der Kopf zu seiner anfänglichen Position zurückkehrt und zum Zustand $[q_2, U]$ gewechselt wird; es handelt sich also um den anfänglichen Zustand von M_2, wobei sich die Aufmerksamkeit auf die obere Spur von M_1 konzentriert.

3. Ist $\delta_2(q, X) = (p, Y, D)$, dann gilt für jedes Z aus Γ_2:

   a) $\delta_1([q, U], [X, Z]) = ([p, U], [Y, Z], D)$ und

   b) $\delta_1([q, L], [Z, X]) = ([p, L], [Z, Y], \overline{D})$, wobei

   $\overline{D}$ die D entgegengesetzte Richtung bezeichnet. Ist $D = R$, dann ist $\overline{D} = L$ und umgekehrt. Befindet sich M_1 nicht über der am weitesten links stehenden Zelle, dann simuliert sie M_2 auf der entsprechenden Spur, nämlich auf der oberen Spur, falls die zweite Komponente des Zustands U lautet, und auf der unteren Spur, falls die zweite Komponente L lautet. Beachten Sie jedoch, dass sich M_1 bei der Arbeit auf der unteren Spur in die entgegengesetzte Richtung zu M_2 bewegt. Dies ist sinnvoll, da die linke Hälfte des Bandes von M_2 in umgekehrter Richtung auf die untere Spur des Bandes von M_1 abgebildet wurde.

4. Ist $\delta_2(q, X) = (p, Y, R)$, dann gilt:

   $$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, U], [Y, *], R)$$

   Diese Regel behandelt einen Fall, wie mit der linken Endemarkierung $*$ umgegangen wird. Bewegt sich M_2 von der anfänglichen Position nach rechts, dann muss sich M_1 nach rechts bewegen und auf die obere Spur konzentrie-

ren. Dies ist unabhängig davon, ob sich M_2 vorher links oder rechts von der anfänglichen Position befunden hat (und dadurch reflektiert, dass die zweite Komponente des Zustands von M_1 L oder U lautet). M_1 wird sich dann also an der Position befinden, die in Abbildung 8.17 von X_1 repräsentiert wird.

5. Ist $\delta_2(q, X) = (p, Y, L)$, dann gilt:

$$\delta_1([q, L], [X, *]) = \delta_1([q, U], [X, *]) = ([p, L], [Y, *], R)$$

Diese Regel gleicht der vorherigen, behandelt jedoch den Fall, dass sich M_2 von der anfänglichen Position nach links bewegt. M_1 muss sich von ihrer Endemarkierung nach rechts bewegen, wechselt nun aber zur unteren Spur, also zu der Zelle, die in Abbildung 8.17 durch X_{-1} repräsentiert wird.

F_1: Bei den akzeptierenden Zuständen F_1 handelt es sich um die Zustände in $F_2 \times \{U, L\}$, also alle Zustände von M_1, deren erste Komponente ein akzeptierender Zustand von M_2 ist. Wenn M_1 akzeptiert, ist es nicht von Belang, ob sich M_1 zu diesem Zeitpunkt auf die obere oder die untere Spur konzentriert.

Der Beweis des Satzes ist nun im Wesentlichen vollständig. Durch Induktion auf die Anzahl der Bewegungen von M_2 stellen wir fest, dass M_1 die Folge der Konfigurationen von M_2 auf dem eigenen Band simuliert; dabei kehren wir die Reihenfolge der unteren Spur, gefolgt von der oberen Spur, um. Wir stellen außerdem fest, dass M_1 genau dann in einen der akzeptierenden Zustände übergeht, wenn M_2 dies tut. Daher gilt: $L(M_1) = L(M_2)$. ■

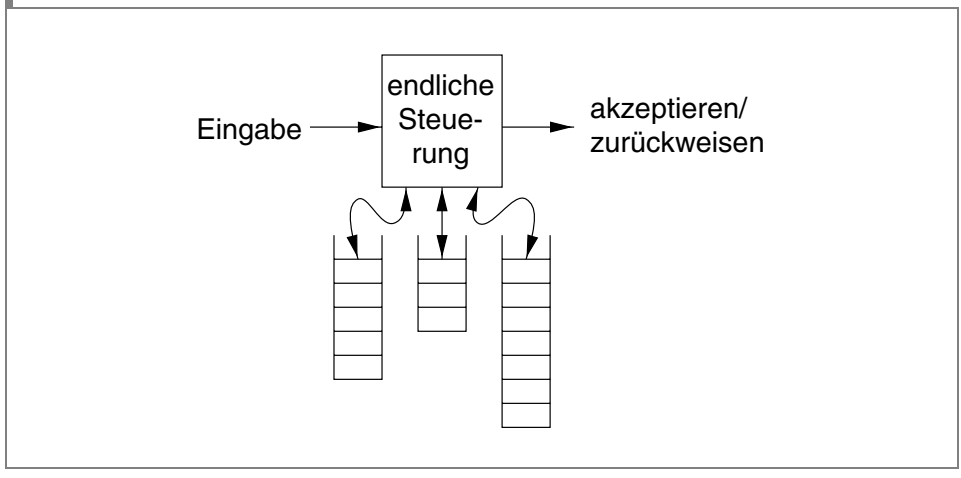
8.5.2 Maschinen mit mehreren Stacks

Wir stellen nun verschiedene Computermodelle vor, die auf Generalisierungen des Pushdown-Automaten (PDA) basieren. Zunächst sehen Sie, was geschieht, wenn der PDA mit mehreren Stacks ausgestattet wird. Wir wissen schon aus Beispiel 8.7, dass eine Turing-Maschine Sprachen akzeptieren kann, die von keinem PDA mit einem einzigen Stack akzeptiert werden. Sie werden aber sehen, dass ein PDA mit zwei Stacks jede Sprache akzeptieren kann, die auch eine Turing-Maschine akzeptieren kann.

Danach sehen wir uns eine Klasse von Maschinen an, die so genannten »Zählermaschinen«. Diese Maschinen besitzen lediglich die Fähigkeit, eine endliche Anzahl von ganzen Zahlen (»Zählern«) zu speichern und verschiedene Bewegungen auszuführen, die davon abhängig sind, welche der Zähler gegebenenfalls den Wert 0 haben. Die Zählermaschine kann nur 1 zu einem Zähler addieren bzw. von ihm subtrahieren und zwei Zähler ungleich 0 nicht voneinander unterscheiden. Ein Zähler entspricht effektiv einem Stack, auf dem wir nur zwei Symbole ablegen können: eine Markierung für das untere Ende des Stacks, die nur dort vorkommt, sowie ein weiteres Symbol, das auf dem Stack abgelegt oder von seinem oberen Ende entfernt werden kann.

Wir werden die mit mehreren Stacks ausgestattete Maschine nicht formal vorstellen. Abbildung 8.18 veranschaulicht zunächst die Idee. Eine k-Stack-Maschine ist ein deterministischer PDA mit k Stacks. Sie erhält ihre Eingabe wie der PDA aus einer Eingabequelle. Die Eingabe wird also nicht wie bei einer TM auf einem Band oder einem Stack abgelegt. Die mit mehreren Stacks ausgestattete Maschine besitzt eine endliche

Abbildung 8.18: Eine Maschine mit drei Stacks



Steuerung, die sich in einem Zustand aus einer endlichen Zustandsmenge befindet. Für alle Stacks wird ein endliches Stackalphabet verwendet. Eine Bewegung dieser Maschine basiert auf Folgendem:

1. Auf dem Zustand der endlichen Steuerung
2. Auf dem gelesenen Eingabesymbol, das aus dem endlichen Eingabealphabet stammt. Alternativ kann die mit mehreren Stacks ausgestattete Maschine unter Verwendung von $\in$ als Eingabe eine Bewegung ausführen, doch damit die Maschine deterministisch ist, darf es in keiner Situation die Möglichkeit einer Wahl zwischen einer ε- und einer Nicht-ε-Bewegung geben.
3. Auf dem obersten Symbol auf jedem der Stacks

In einer Bewegung kann die mit mehreren Stacks ausgestattete Maschine

a) in einen neuen Zustand wechseln.

b) das oberste Symbol auf jedem Stack durch eine Zeichenreihe aus Nullen oder anderen Stacksymbolen ersetzen. Gewöhnlich unterscheiden sich die Ersetzungszeichenreihen der Stacks voneinander.

Eine typische Übergangsregel für eine k-Stack-Maschine sieht folgendermaßen aus:

$$\delta(q, a, X_1, X_2, \ldots, X_k) = (p, \gamma_1, \gamma_2, \ldots, \gamma_k)$$

Diese Regel ist wie folgt zu interpretieren: In Zustand q mit X_i als oberstem Symbol auf dem i-ten Stack für $i = 1, 2, \ldots, k$ kann die Maschine a aus der Eingabe lesen (entweder ein Eingabesymbol oder ε), zu Zustand p wechseln und das oberste Symbol X_i auf dem i-ten Stack durch die Zeichenreihe γ_i für jedes $i = 1, 2, \ldots, k$ ersetzen. Die mit mehreren Stacks ausgestattete Maschine akzeptiert durch Übergang in einen Endzustand.

Wir fügen eine Fähigkeit hinzu, um die Eingabeverarbeitung dieser deterministischen Maschine zu vereinfachen: Wir nehmen an, es existiert ein besonderes Symbol $, die *Endemarkierung*, die nur am Ende der Eingabe auftritt und selbst nicht zur Eingabe gehört. Das Auftreten der Endemarkierung informiert uns, wenn die gesamte verfüg-

bare Eingabe abgearbeitet ist. Sie sehen im nächsten Theorem (Satz), wie es die Endemarkierung einer mit mehreren Stacks ausgestatteten Maschine erleichtert, eine Turing-Maschine zu simulieren. Beachten Sie, dass die konventionelle Turing-Maschine keine Endemarkierung benötigt, da das Eingabeende durch das erste Leerzeichen bestimmt wird.

Satz 8.13 Wird eine Sprache L von einer Turing-Maschine akzeptiert, dann wird L auch von einer Maschine mit zwei Stacks akzeptiert.

Beweis: Die Idee besteht im Wesentlichen darin, dass zwei Stacks ein Band einer Turing-Maschine simulieren können, indem ein Stack alles links vom Kopf und der andere Stack alles rechts davon speichert, abgesehen von den unendlichen Zeichenreihen aus Leerzeichen, die jeweils links und rechts von den Nichtleerzeichen folgen. Detaillierter sei L die Sprache $L(M)$ einer (einbändigen) TM M. Unsere mit zwei Stacks ausgestattete Maschine S arbeitet wie folgt:

1. S beginnt mit einer *Stackanfangsmarkierung* auf jedem Stack. Diese Markierung kann das Startsymbol für die Stacks sein und darf sonst nirgends auf den Stacks erscheinen. Im Folgenden sagen wir, ein »Stack ist leer«, wenn er nur die Stackanfangsmarkierung enthält.

2. Angenommen, $w\$$ befindet sich in der Eingabe von S. S kopiert w auf den ersten Stack und beendet den Kopiervorgang, wenn die Endemarkierung der Eingabe gelesen wird.

3. S entfernt nacheinander jedes Symbol vom ersten Stack und fügt es auf dem zweiten Stack hinzu. Nun ist der erste Stack leer, und der zweite Stack enthält w, mit dem linken Ende von w oben auf dem zweiten Stack.

4. S wechselt in den (simulierten) Startzustand von M. Der erste Stack ist leer und repräsentiert die Tatsache, dass M links von der Zelle, die der Bandkopf liest, nur Leerzeichen besitzt. S hat einen zweiten Stack, der w enthält und die Tatsache repräsentiert, dass w in der Zelle sowie rechts davon gespeichert ist, an der sich der Bandkopf von M befindet.

5. S simuliert eine Bewegung von M wie folgt:

   a) S kennt den Zustand von M, beispielsweise q, da S den Zustand in der eigenen endlichen Steuerung simuliert.

   b) S kennt das vom Bandkopf von M gelesene Symbol X; es befindet sich oben auf dem zweiten Stack von S. Eine Ausnahme tritt auf, wenn M sich gerade zu einem Leerzeichen bewegt hat, denn dann enthält der zweite Stack lediglich die Stackanfangsmarkierung; S interpretiert das von M gelesene Symbol in diesem Fall als das Leerzeichen.

   c) S kennt daher die nächste Bewegung von M.

   d) Der nächste Zustand von M wird in einer Komponente der endlichen Steuerung von S aufgezeichnet und ersetzt den vorherigen Zustand.

   e) Wenn M X durch Y ersetzt und sich nach rechts bewegt, dann fügt S Y auf dem ersten Stack hinzu, was die Tatsache repräsentiert, dass sich Y nun

links vom Kopf von M befindet. X wird vom zweiten Stack von S entfernt. Es gibt jedoch zwei Ausnahmen:

i. Befindet sich auf dem zweiten Stack nur die Stackanfangsmarkierung (und ist X daher ein Leerzeichen), dann wird der zweite Stack nicht geändert; M hat sich zu einem weiteren Leerzeichen nach rechts bewegt.

ii. Ist Y ein Leerzeichen und der erste Stack leer, dann bleibt dieser Stack leer. Der Grund hierfür ist, dass sich links vom Kopf von M noch immer nur Leerzeichen befinden.

f) Wenn M X durch Y ersetzt und sich nach links bewegt, entfernt S das oberste Element des ersten Stacks, sagen wir Z, und ersetzt auf dem zweiten Stack dann X durch ZY. Diese Änderung spiegelt die Tatsache wider, dass die Position, die sich links vom Kopf befand, nun zur neuen Kopfposition geworden ist. Ausgenommen davon ist folgende Situation: Wenn Z die Stackanfangsmarkierung ist, muss M BY dem zweiten Stack hinzufügen und nichts vom ersten Stack entfernen.

6. S akzeptiert, wenn der neue Zustand von M akzeptiert. Im anderen Fall simuliert S auf die gleiche Weise eine weitere Bewegung von M. ■

8.5.3 Zählermaschinen

Zählermaschinen können auf zweierlei Weise betrachtet werden:

1. Die Zählermaschine hat die gleiche Struktur wie die mit mehreren Stacks ausgestattete Maschine (Abbildung 8.18), jeder Stack wird jedoch durch einen Zähler ersetzt. Zähler speichern eine nicht negative ganze Zahl, doch wir können nur zwischen Zahlen gleich oder ungleich 0 unterscheiden. Die Bewegung der Zählermaschine hängt von ihrem Zustand, dem Eingabesymbol und den Zählern ab, die gleich 0 sind (falls vorhanden). Die Zählermaschine kann in einer Bewegung Folgendes ausführen:

   a) Den Zustand wechseln.

   b) Unabhängig voneinander die Zähler um 1 erhöhen oder erniedrigen. Allerdings kann ein Zähler nicht negativ werden, und damit ist die Subtraktion von 1 von einem Zähler, der gerade 0 ist, nicht erlaubt.

2. Eine Zählermaschine kann auch als beschränkte Maschine, die mit mehreren Stacks ausgestattet ist, angesehen werden. Folgende Einschränkungen gelten:

   a) Es gibt nur zwei Stacksymbole, Z_0 (die *Stackanfangsmarkierung*) und X.

   b) Anfänglich befindet sich Z_0 auf jedem Stack.

   c) Wir können Z_0 nur durch eine Zeichenreihe der Form X^iZ_0 für $i \geq 0$ ersetzen.

   d) Wir können X nur durch X^i für $i \geq 0$ ersetzen. Z_0 erscheint also nur als unterstes Element jedes Stacks, und alle anderen Stacksymbole, falls vorhanden, sind X.

Die beiden Definitionen definieren offensichtlich Maschinen äquivalenter Leistungsfähigkeit, doch wir werden Definition (1) für Zählermaschinen verwenden. Der

Grund liegt darin, dass Stack X^iZ_0 mit dem Zählerstand i identifiziert werden kann. In Definition (2) können wir Zählerstand 0 von anderen Zählerständen unterscheiden, da wir dann Z_0 als oberstes Element des Stacks sehen, sonst dagegen X. Allerdings können wir zwei positive Zähler nicht voneinander unterscheiden, da bei beiden X das oberste Element ist.

8.5.4 Die Leistungsfähigkeit von Zählermaschinen

Zu den von Zählermaschinen akzeptierten Sprachen ist einiges zu sagen:

- Jede von einer Zählermaschine akzeptierte Sprache ist rekursiv aufzählbar. Der Grund liegt darin, dass eine Zählermaschine ein Spezialfall einer mit mehreren Stacks ausgestatteten Maschine ist und eine mit mehreren Stacks ausgestattete Maschine wiederum ein Spezialfall einer mehrbändigen Turing-Maschine, die laut Satz 8.9 nur rekursiv aufzählbare Sprachen akzeptiert.
- Jede von einer mit einem Zähler ausgestatteten Maschine akzeptierte Sprache ist kontextfrei. Beachten Sie hierzu, dass ein Zähler in der Auffassung 8.5.3.2 ein Stack ist, und damit ist eine mit einem Zähler ausgestattete Maschine ein Spezialfall einer Maschine, die einen Stack besitzt, also ein PDA. Tatsächlich werden die Sprachen von Maschinen mit einem Zähler auch von deterministischen PDAs akzeptiert, der Beweis ist allerdings überraschend komplex. Die Schwierigkeit des Beweises entspringt der Tatsache, dass die mit mehreren Stacks ausgestatteten Maschinen sowie die Zählermaschinen am Ende der Eingabe eine Endemarkierung $ besitzen. Ein nichtdeterministischer PDA kann annehmen, dass das letzte Eingabesymbol gelesen wurde und nun das Symbol $ folgt; es ist also klar, dass ein nichtdeterministischer PDA ohne die Endemarkierung einen DPDA mit Endemarkierung simulieren kann. Im schwierigen Beweis, den wir hier nicht vorstellen, wird schließlich gezeigt, dass ein DPDA ohne Endemarkierung einen DPDA *mit* Endemarkierung simulieren kann.

Überraschend an Zählermaschinen ist die Tatsache, dass zwei Zähler ausreichen, um eine Turing-Maschine zu simulieren und damit jede rekursiv aufzählbare Sprache zu akzeptieren. Dieses Ergebnis werden wir uns nun näher ansehen. Wir zeigen zuerst, dass drei Zähler ausreichend sind, und simulieren dann drei mit zwei Zählern.

Satz 8.14 Jede rekursiv aufzählbare Sprache wird von einer mit drei Zählern ausgestatteten Maschine akzeptiert.

Beweis: Wir beginnen mit Satz 8.13, der aussagt, dass jede rekursiv aufzählbare Sprache von einer mit zwei Stacks ausgestatteten Maschine akzeptiert wird. Wir müssen dann zeigen, wie ein Stack mit Zählern simuliert wird. Angenommen, es gibt $r - 1$ Bandsymbole, die von der Stackmaschine verwendet werden. Wir können die Symbole mit Ziffern 1 bis $r - 1$ identifizieren und uns einen Stack $X_1X_2\ldots X_n$ als eine ganze Zahl zur Basis r vorstellen. Dieser Stack wird von der ganzen Zahl $X_nr^{n-1} + X_{n-1}r^{n-2} + \ldots + X_2r + X_1$ repräsentiert.

Wir verwenden zwei Zähler zum Speichern der ganzen Zahlen, die jeden der beiden Stacks repräsentieren. Der dritte Zähler dient zur Fortschreibung der beiden anderen Zähler. Genauer gesagt, benötigen wir den dritten Zähler, wenn wir einen Zählerstand durch r dividieren oder mit r multiplizieren.

Die Stackoperationen können in drei Gruppen unterteilt werden: das oberste Symbol wegnehmen, das oberste Symbol ändern und ein Symbol dem Stack hinzufügen. Eine Bewegung der mit zwei Stacks ausgestatteten Maschine könnte verschiedene dieser Operationen involvieren; insbesondere das Ersetzen des obersten Stacksymbols X durch eine Symbolzeichenreihe muss in Ersetzen von X und Hinzufügen zusätzlicher Symbole auf dem Stack unterteilt werden. Wir führen diese Operationen auf einem Stack, der von einem Zählerstand i repräsentiert wird, wie folgt aus. Beachten Sie, dass es möglich ist, die endliche Steuerung der mit mehreren Stacks ausgestatteten Maschine für jede Operation zu verwenden, die das Heraufzählen bis zu einer Zahl = r erfordert.

1. Um Symbole vom Stack zu entfernen, müssen wir i durch i/r ersetzen und den Rest, also X_1, verwerfen. Beginnend mit dem dritten Zähler bei 0, erniedrigen wir den Zählerstand i wiederholt um r und erhöhen den dritten Zähler um 1, falls wir nicht weniger als r vom aktuellen Zählerstand i subtrahieren konnten, bis der Zähler, der ursprünglich i enthielt, 0 erreicht. Dann erhöhen wir den ursprünglichen Zähler wiederholt um 1 und erniedrigen den dritten Zähler um 1, bis der dritte Zähler wiederum 0 ist. An diesem Punkt enthält der Zähler, der ursprünglich i enthielt, nun i/r.

2. Um auf einem Stack, der vom Zählerstand i repräsentiert wird, das oberste Element X in Y zu ändern, erhöhen oder erniedrigen wir i um einen Wert, der r nicht übersteigt. Ist $Y > X$, wird i um $Y - X$ erhöht; ist $Y < X$, wird i um $X - Y$ erniedrigt.

3. Um X auf einem Stack hinzuzufügen, der ursprünglich i enthielt, müssen wir i durch $ir + X$ ersetzen. Wir multiplizieren zunächst mit r. Dazu wird der Zählerstand i wiederholt um 1 erniedrigt und der dritte Zähler (der wie immer bei 0 beginnt) um r erhöht. Hat der ursprüngliche Zähler 0 erreicht, dann enthält der dritte Zähler ir. Wir kopieren den dritten Zähler in den ursprünglichen Zähler. Dazu addieren wir im ursprünglichen Zähler wiederholt 1 bei gleichzeitigem Subtrahieren von 1 im dritten Zähler, bis dieser 0 wird. Zuletzt erhöhen wir den ursprünglichen Zähler um X.

Um die Konstruktion abzuschließen, müssen wir die Zähler initialisieren, um die Stacks in ihrem Startzustand zu simulieren: nur die Startsymbole der mit zwei Stacks ausgestatteten Maschine enthaltend. Dazu werden die beiden Zähler um einen kleinen ganzzahligen Wert aus dem Bereich von 1 bis $r - 1$ erhöht, der dem Startsymbol entspricht. ■

Satz 8.15 Jede rekursiv aufzählbare Sprache wird von einer mit zwei Zählern ausgestatteten Maschine akzeptiert.

Beweis: Nach dem letzten Satz müssen wir lediglich beweisen, wie zwei Zähler drei Zähler simulieren. Der Beweis basiert darauf, die drei Zähler i, j und k durch eine einzelne ganze Zahl zu repräsentieren. Wir wählen die ganze Zahl $2^i3^j5^k$. Ein Zähler speichert diese Zahl, während der andere die Multiplikation oder Division von m mit bzw. durch eine der ersten drei Primzahlen 2, 3 und 5 unterstützt. Zur Simulation der mit drei Zählern ausgestatteten Maschine müssen wir die folgenden Operationen ausführen:

1. Inkrementieren von i, j und/oder k. Um i um 1 zu erhöhen, multiplizieren wir m mit 2. Der Beweis von Satz 8.14 hat schon gezeigt, wie ein Zählerstand mit Hilfe eines zweiten Zählers mit einer Konstanten r multipliziert wird. Entsprechend inkrementieren wir j bzw. k durch eine Multiplikation von m mit 3 bzw. 5.
2. Feststellen, ob i, j oder k gleich 0 ist. Um festzustellen, ob $i = 0$ ist, bestimmen wir, ob m durch 2 teilbar ist. Wir kopieren m in den zweiten Zähler und verwenden den Zustand der Zählermaschine, um festzuhalten, ob m jeweils eine gerade oder ungerade Zahl mal um 1 vermindert wurde. Falls es sich um eine ungerade Anzahl handelt, wenn m den Wert 0 erreicht hat, dann ist $i = 0$. Danach stellen wir m wieder her, indem wir den zweiten Zähler auf den ersten kopieren. In ähnlicher Weise testen wir, ob $j = 0$ bzw. $k = 0$ ist, indem wir die Teilbarkeit durch 3 bzw. 5 bestimmen.
3. Vermindern von i, j und/oder k um 1. Dazu dividieren wir m durch 2, 3 bzw. 5. Der Beweis von Satz 8.14 zeigt, wie die Division durch eine Konstante mit Hilfe eines Zusatzzählers ausgeführt wird. Da die mit drei Zählern ausgestattete Maschine keinen Zählerstand kleiner 0 erreichen kann, ist m in der mit zwei Zählern ausgestatteten Maschine ohne Rest durch die jeweilige Konstante teilbar. Andernfalls liegt ein Fehler vor, und die 2-Zähler-Maschine hält an, ohne zu akzeptieren. ■

Wahl der Konstanten in der 3-in-2-Zähler Konstruktion

Im Beweis des Satzes 8.15 war es wesentlich, dass 2, 3, 5 paarweise verschiedene Primzahlen sind. Hätten wir z.B. $m = 2^i 3^j 4^k$ gewählt, dann könnte $m = 12$ sowohl $i = 0$, $j = 1$, $k = 1$ als auch $i = 2$, $j = 1$, $k = 0$ repräsentieren. Damit könnten wir nicht feststellen, ob i oder $k = 0$ ist, und die 3-Zähler-Maschine nicht zuverlässig simulieren.

8.5.5 Übungen zum Abschnitt 8.5

Übung 8.5.1 Beschreiben Sie informell, aber klar Zählermaschinen, die die folgenden Sprachen akzeptieren. Verwenden Sie in jedem Fall möglichst wenige Zähler, nie jedoch mehr als zwei Zähler.

* a) $\{0^n1^m \mid n \geq m \geq 1\}$

b) $\{0^n1^m \mid 1 \leq m \leq n\}$

*! c) $\{a^ib^jc^k \mid i = j \text{ oder } i = k\}$

!! d) $\{a^ib^jc^k \mid i = j \text{ oder } i = k \text{ oder } j = k\}$

!! **Übung 8.5.2** Diese Übung soll zeigen, dass eine Maschine mit einem Stack und einer Endemarkierung der Eingabe nicht leistungsfähiger als ein deterministischer PDA ist. $L\$$ ist die Verkettung der Sprache L mit der Sprache, die nur die einzige Zeichenreihe \$ enthält; $L\$$ umfasst also die Menge aller Zeichenreihen $w\$$, sodass w in L enthalten ist. Zeigen Sie: Ist $L\$$ eine Sprache, die von einem DPDA akzeptiert wird, wobei \$ das Symbol für die Endemarkierung ist und in keiner Zeichenreihe aus L enthalten ist, dann wird L ebenfalls von einem DPDA akzeptiert. *Hinweis:* Hier soll gezeigt werden,

dass die DPDA-Sprachen unter der in Übung 4.2.2 definierten Operation L / a abgeschlossen sind. Sie müssen dazu den DPDA P für $L\$$ ändern, indem Sie seine Stacksymbole X durch alle möglichen Paare (X, S) ersetzen, wobei S eine Menge von Zuständen ist. Hat P den Stack $X_1X_2\ldots X_n$, dann hat der konstruierte DPDA für L den Stack $(X_1, S_1)(X_2, S_2)\ldots(X_n, S_n)$, wobei S_i die Menge von Zuständen q ist, sodass P beginnend mit der Konfiguration $(q, a, X_iX_{i+1}\ldots X_n)$ akzeptieren wird.

8.6 Turing-Maschinen und Computer

Wir werden nun die Turing-Maschine und das allgemeine Modell eines Computers, wie wir ihn täglich verwenden, miteinander vergleichen. Diese Modelle scheinen sich zwar stark voneinander zu unterscheiden, doch sie können exakt die gleichen Sprachen akzeptieren – die rekursiv aufzählbaren Sprachen. Da der Begriff eines »allgemeinen Computers« mathematisch nicht gut definiert ist, sind die Argumente in diesem Abschnitt notwendigerweise informell. Wir sprechen deshalb Ihre Intuition über die Fähigkeiten eines Computers an, insbesondere wenn die involvierten Zahlen die normale Größenordnung überschreiten, die die Architektur dieser Maschinen vorsieht (beispielsweise 32-Bit-Adressräume). Dieser Abschnitt behandelt zwei Teile:

1. Ein Computer kann eine Turing-Maschine simulieren.
2. Eine Turing-Maschine kann einen Computer simulieren. Sie benötigt dazu einen Zeitaufwand, der höchstens polynomial in der Zahl der Schritte ist, die der Computer ausführt.

8.6.1 Eine Turing-Maschine mit einem Computer simulieren

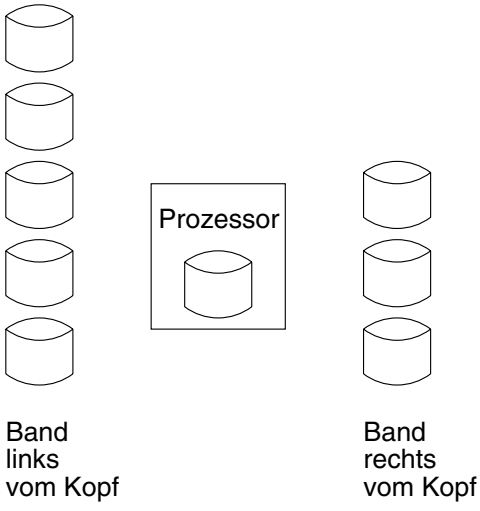
Wir untersuchen zunächst, wie ein Computer eine Turing-Maschine simulieren kann. Ist eine bestimmte TM M gegeben, dann müssen wir ein Programm schreiben, das wie M agiert. Ein Aspekt von M ist deren endliche Steuerung. Da es nur eine endliche Anzahl von Zuständen sowie eine endliche Anzahl von Übergangsregeln gibt, kann unser Programm Zustände als Zeichenreihen kodieren und eine Tabelle mit den Übergängen vorsehen, in der jede Bewegung abgelesen wird. Entsprechend können die Bandsymbole als Zeichenreihen einer festen Länge kodiert werden, da nur eine endliche Anzahl von Bandsymbolen existiert.

Bei der Überlegung, wie unser Programm das Band der Turing-Maschine simulieren soll, stellt sich eine ernste Frage. Dieses Band kann in seiner Länge unbeschränkt anwachsen, doch der Speicher des Computers – Hauptspeicher, Festplatte und andere Speichergeräte – ist endlich. Können wir mit einem Speicher fester Größe ein unbeschränkt langes Band simulieren?

Wenn es nicht möglich ist, Speichergeräte auszutauschen, dann können wir dies tatsächlich nicht; ein Computer wäre dann ein endlicher Automat und würde lediglich reguläre Sprachen akzeptieren. Allerdings sind moderne Computer im Allgemeinen mit Wechselspeichermedien wie etwa einem »Zip-Laufwerk« ausgestattet. Auch die normale Festplatte ist entfernbar und kann gegen eine leere, ansonsten identische Festplatte ausgetauscht werden.

Da uns keine Beschränkung auferlegt ist, wie viele Festplatten wir verwenden können, nehmen wir an, dass so viele Festplatten verfügbar sind, wie der Computer benötigt. Wir können die Festplatten daher in zwei Stacks anordnen, wie es Abbildung 8.19 zeigt. Ein Stack speichert die Daten derjenigen Zellen des Bandes der

Abbildung 8.19: Eine Turing-Maschine mit einem gewöhnlichen Computer simulieren



Turing-Maschine, die sich links vom Bandkopf befinden, der andere Stack speichert die Daten rechts vom Bandkopf. Je weiter unten im Stack, desto weiter sind die Daten vom Bandkopf entfernt.

Bewegt sich der Bandkopf der TM weit nach links und erreicht Zellen, die nicht auf der gerade installierten Festplatte enthalten sind, wird eine Meldung »links austauschen« ausgegeben. Die aktuelle Festplatte wird vom Operator entfernt und oben auf dem rechten Stack abgelegt. Die oberste Festplatte des linken Stacks wird in den Computer eingelegt und die Berechung fortgeführt.

Erreicht der Bandkopf der TM Zellen, die sich so weit rechts befinden, dass sie nicht mehr auf der installierten Festplatte enthalten sind, wird ebenso verfahren und eine Meldung »rechts austauschen« ausgegeben. Der Operator legt die installierte Festplatte oben auf dem linken Stack ab und legt die oberste Festplatte des rechten Stacks in den Computer ein. Ist einer der Stacks leer, wenn der Computer eine Festplatte von diesem Stack verlangt, dann hat die TM einen leeren Abschnitt des Bandes erreicht. In diesem Fall muss der Operator eine neue Festplatte kaufen und installieren.

8.6.2 Einen Computer mit einer Turing-Maschine simulieren

Wir müssen außerdem den umgekehrten Vergleich betrachten: Gibt es etwas, was ein allgemeiner Computer kann, eine Turing-Maschine dagegen nicht? Eine wichtige untergeordnete Frage ist, ob der Computer manches wesentlich schneller als eine Turing-Maschine ausführen kann. In diesem Abschnitt zeigen wir, dass eine TM einen Computer simulieren kann, und in Abschnitt 8.6.3 zeigen wir, dass die Simulation ausreichend schnell ausgeführt werden kann, sodass sich die Ausführungszeiten des Computers und der TM bei einem gegebenen Problem nur polynomial unterscheiden. Der Leser soll wiederum daran erinnert werden, dass es gute Gründe gibt, alle Ausführungszeiten, die sich nur polynomial unterscheiden, als ähnlich hinsichtlich ihrer

Komplexität zu betrachten, während exponentielle Unterschiede »zu viel« sind. Die Theorie polynomialen und exponentiellen Zeitaufwands wird in Kapitel 10 behandelt.

Das Problem sehr umfangreicher Bandalphabete

Ein Argument aus Abschnitt 8.6.1 wird fragwürdig, wenn die Anzahl der Bandsymbole so groß ist, dass der Code für ein Bandsymbol nicht auf eine Festplatte passt. Es gäbe dann wirklich sehr viele Bandsymbole, da eine 30-Gbyte-Festplatte beispielsweise irgendeins von $2^{240000000000}$ Symbolen repräsentieren kann. Ebenso könnte die Anzahl der Zustände so umfangreich sein, dass wir den Zustand nicht auf einer einzigen Festplatte repräsentieren könnten.

Eine Lösung dieses Problems schränkt zunächst die Anzahl der von einer TM verwendeten Bandsymbole ein. Wir können ein beliebiges Bandalphabet immer in Binärdarstellung kodieren. Daher kann jede TM M von einer anderen TM M' simuliert werden, die nur die Bandsymbole 0, 1 und B verwendet. Allerdings benötigt M' viele Zustände, da die TM M' zur Simulation einer einzigen Bewegung von M das eigene Band lesen und in der endlichen Steuerung alle Bits speichern muss, die Informationen darüber liefern, welches Symbol M liest. Auf diese Weise haben wir sehr umfangreiche Zustandsmengen, und der PC, der M' simuliert, erfordert den Austausch mehrerer Festplatten bei der Suche nach dem Zustand und der nächsten Bewegung von M'. Niemand denkt je über Computer nach, die solche Aufgaben erledigen, und daher unterstützt das typische Betriebssystem ein Programm dieses Typs nicht. Wenn wir allerdings wollten, könnten wir den Computer entsprechend programmieren und ihm diese Fähigkeit verleihen.

Glücklicherweise kann die Frage, wie eine TM mit einer riesigen Anzahl von Zuständen oder Bandsymbolen zu simulieren ist, raffinierter beantwortet werden. Wir werden in Abschnitt 9.2.3 sehen, dass eine TM entworfen werden kann, die im Effekt eine für jede TM vorprogrammierte TM ist. Diese »universell« genannte TM übernimmt die Übergangsfunktion jeder beliebigen TM in Binärform auf das eigene Band und simuliert diese TM. Die universelle TM besitzt eine vernünftige Anzahl von Zuständen und Bandsymbolen. Durch die Simulation der universellen TM kann ein allgemeiner Computer so programmiert werden, dass er jede gewünschte rekursiv aufzählbare Sprache akzeptiert, ohne auf die Simulation vieler Zustände zurückgreifen zu müssen, die die Grenzen dessen überschreitet, was auf einer Festplatte gespeichert werden kann.

Zu Anfang unserer Betrachtung, wie eine TM einen Computer simuliert, sehen wir uns ein realistisches, jedoch informelles Modell der Arbeitsweise eines typischen Computers an.

a) Zunächst nehmen wir an, dass der Speicher eines Computers aus einer unendlich langen Folge von *Wörtern* besteht, die alle eine *Adresse* besitzen. Ein realer Computer kennt Wörter der Länge 32 oder 64 Bit, doch wir schränken die Länge eines gegebenen Wortes nicht ein. Adressen sind ganze Zahlen 0, 1, 2 und so weiter. In einem realen Computer würden einzelne Bytes mit aufeinander folgenden ganzen Zahlen durchnummeriert, und die Adressen der Wörter wären ein Vielfaches von 4 oder 8, doch dieser Unterschied ist nicht von Bedeutung. Außerdem wäre die Anzahl der Wörter im »Speicher« begrenzt; doch da wir vom Inhalt einer beliebigen Anzahl von Festplatten oder anderen Speichermedien ausgehen, nehmen wir an, die Anzahl der Wörter sei nicht eingeschränkt.

b) Wir nehmen an, das Programm des Computers sei in einigen der Speicherwörter gespeichert. Diese Wörter repräsentieren jeweils eine einfache Anweisung, entsprechend der Maschinen- oder Assemblersprache eines typischen Computers. Beispiele dafür sind Anweisungen, die Daten von einem Wort zu einem anderen verschieben oder ein Wort zu einem anderen addieren. Wir nehmen an, dass »indirekte Adressierung« erlaubt ist, sodass eine Anweisung auf ein anderes Wort verweisen und den Inhalt dieses Wortes als Adresse des Wortes verwenden kann, auf das die Operation angewendet wird. Diese Fähigkeit stellen alle modernen Computer bereit, und sie ist notwendig, um Arrayzugriffe auszuführen, Verknüpfungen in einer Liste zu folgen oder allgemein Zeigeroperationen auszuführen.

c) Wir nehmen an, dass in jede Operation eine begrenzte (endliche) Anzahl von Wörtern involviert ist und jede Anweisung den Wert mindestens eines Wortes ändert.

d) Ein typischer Computer besitzt *Register*, also Speicherwörter mit besonders schnellem Zugriff. Häufig sind Operationen wie die Addition auf Register beschränkt. Solche Einschränkungen nehmen wir nicht vor; jede Operation kann mit jedem Wort ausgeführt werden. Die relative Ausführungsgeschwindigkeit für verschiedene Wörter muss nicht berücksichtigt werden, wenn wir lediglich die Spracherkennungsfähigkeiten von Computern und Turing-Maschinen vergleichen. Auch wenn der Zeitaufwand innerhalb einer polynomialen Größe liegen soll, ist die relative Geschwindigkeit unterschiedlicher Wortzugriffe nicht von Bedeutung, da es sich bei diesen Unterschieden »nur« um konstante Faktoren handelt.

Abbildung 8.20: Eine Turing-Maschine, die einen Computer simuliert

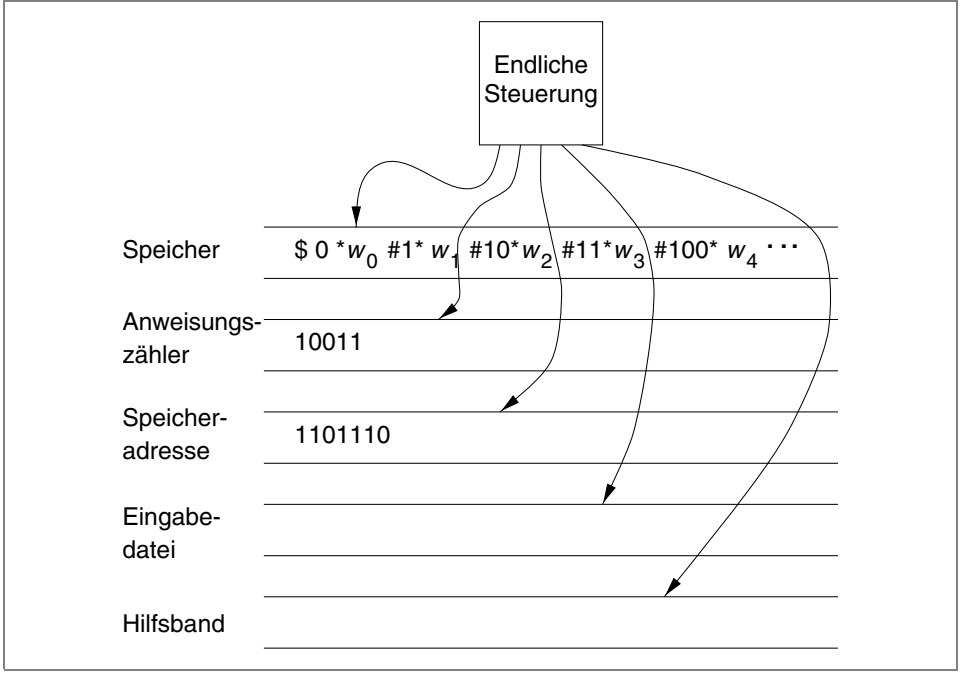


Abbildung 8.20 veranschaulicht, wie die Turing-Maschine zur Simulation eines Computers entworfen werden könnte. Diese TM verwendet verschiedene Bänder, könnte aber unter Verwendung der Konstruktion aus Abschnitt 8.4.1 in eine einbändige TM konvertiert werden. Das erste Band repräsentiert den gesamten Speicher des Computers. Wir verwenden einen Code, in dem sich Adressen von Speicherwörtern, in numerischer Reihenfolge, mit den Inhalten dieser Speicherwörter abwechseln. Sowohl Adressen als auch Inhalte sind binär dargestellt. Die Markierungssymbole * und # erleichtern die Suche nach dem Ende von Adressen und Inhalten und kennzeichnen eine Binärzeichenreihe als Adresse oder Inhalt. Eine weitere Markierung, $, zeigt den Anfang der Folge von Adressen und Inhalten an.

Das zweite Band ist der »Anweisungszähler«. Dieses Band speichert eine ganze Zahl in Binärdarstellung, die einen der Speicherorte auf Band 1 repräsentiert. Der an diesem Ort gespeicherte Wert wird als nächste auszuführende Computeranweisung interpretiert.

Das dritte Band speichert eine »Speicheradresse« oder den Inhalt dieser Adresse, nachdem sie auf Band 1 gefunden wurde. Um eine Anweisung auszuführen, muss die TM nach den Inhalten einer oder mehrerer Speicheradressen suchen, die die Daten enthalten, die für die Anweisung benötigt werden. Zuerst wird die gewünschte Adresse auf Band 3 kopiert und mit den Adressen auf Band 1 verglichen, bis eine Übereinstimmung gefunden wird. Der Inhalt dieser Adresse wird auf das dritte Band kopiert und dorthin gebracht, wo er benötigt wird, normalerweise zu einer der niedrigen Adressen, die den Registern des Computers entsprechen.

Unsere TM simuliert den *Anweisungszyklus* des Computers wie folgt:

1. Das erste Band wird nach einer Adresse durchsucht, die mit der Anweisungsnummer auf Band 2 übereinstimmt. Wir beginnen bei dem Symbol $ auf dem ersten Band und suchen rechtsgerichtet, indem wir jede Adresse mit dem Inhalt auf Band 2 vergleichen. Der Vergleich von Adressen auf den beiden Bändern ist einfach, da wir lediglich die Bandköpfe synchron nach rechts verschieben und prüfen, ob die gelesenen Symbole immer übereinstimmen.

2. Ist die Anweisungsadresse gefunden, wird ihr Wert untersucht. Wir nehmen an, dass die ersten Bits eines Wortes, wenn es eine Anweisung ist, die auszuführende Aktion repräsentieren (z. B. Kopieren, Addieren, Verzweigen), während die restlichen Bits eine oder mehrere Adressen kodieren, die in die Aktion involviert sind.

3. Erfordert die Anweisung den Wert einer Adresse, dann ist diese Adresse Teil der Anweisung. Wir kopieren sie auf das dritte Band und markieren deren Position, indem wir eine zweite Spur des ersten Bandes verwenden (Abbildung 8.22 zeigt diese Spur nicht). Auf diese Weise finden wir, falls notwendig, zur Anweisung zurück. Nun suchen wir auf dem ersten Band nach der Speicheradresse und kopieren deren Wert auf Band 3, auf dem die Speicheradresse abgelegt ist.

4. Die Anweisung bzw. der Teil der Anweisung, die diesen Wert benötigt, wird ausgeführt. Wir können hier nicht alle möglichen Maschinenanweisungen vorstellen. Hier also einige Beispiele dafür, wie der neue Wert verarbeitet werden könnte:

   a) Der Wert kann in eine neue Adresse kopiert werden. Wir erhalten die zweite Adresse aus der Anweisung, speichern sie auf Band 3 und suchen,

wie schon beschrieben, auf Band 1 nach der Adresse. Ist die zweite Adresse gefunden, kopieren wir den Wert in den Bereich, der für den Wert dieser Adresse reserviert ist. Benötigt der neue Wert mehr oder weniger Speicherplatz als der alte Wert, wird der verfügbare Platz durch *Verschieben* geändert:

i. Das gesamte nicht leere Band rechts vom Speicherort des neuen Werts wird auf ein Hilfsband kopiert.

ii. Der neue Wert wird unter Ausnutzung des korrekten Speicherplatzes geschrieben.

iii. Das Hilfsband wird unmittelbar rechts vom neuen Wert wieder auf Band 1 kopiert.

Ein Spezialfall liegt vor, wenn die Adresse noch nicht auf dem ersten Band vorhanden ist, weil sie vorher noch nicht vom Computer verwendet wurde. In diesem Fall suchen wir auf dem ersten Band nach dem korrekten Ort, sorgen mit VERSCHIEBEN für einen adäquaten Speicherplatz und speichern dann sowohl die Adresse als auch den neuen Wert.

b) Der gerade gefundene Wert wird zum Wert einer anderen Adresse addiert. Wir gehen zur Anweisung zurück, um die andere Adresse auf Band 1 zu suchen. Der Wert dieser Adresse sowie der auf Band 3 gespeicherte Wert werden binär addiert. Die TM liest die beiden Werte von rechts und kann daher eine einfache Addition ausführen. Sollte das Ergebnis zusätzlichen Speicherplatz benötigen, wird mit der Verschiebetechnik freier Speicherplatz auf Band 1 geschaffen.

c) Bei der Anweisung handelt es sich um einen »Sprung«, also um eine Anweisung, die nächste Anweisung aus der Adresse zu lesen, die nun als Wert auf Band 3 gespeichert ist. Wir kopieren einfach Band 3 auf Band 2 und führen den Anweisungszyklus erneut aus.

5. Nach Ausführung der Anweisung und Feststellung, dass es sich bei dieser Anweisung nicht um einen Sprung handelt, wird der Anweisungszähler auf Band 2 um 1 erhöht und der Anweisungszyklus erneut ausgeführt.

Die Simulation eines typischen Computers durch eine TM enthält viele weitere Details. Abbildung 8.22 zeigt ein viertes Band, auf dem die simulierte Eingabe des Computers gespeichert ist, da der Computer seine Eingabe (das Wort, das auf Zugehörigkeit zu einer Sprache geprüft wird) aus einer Datei liest. Stattdessen kann die TM dieses Band lesen.

Darunter sehen Sie ein Hilfsband. Die Simulationen einiger Computeranweisungen könnten ein oder mehrere Hilfsbänder zur Berechnung arithmetischer Operationen wie der Multiplikation nutzen.

Zuletzt nehmen wir an, dass der Computer ausgibt, ob er die Eingabe akzeptiert. Um diese Aktion in Anweisungen zu übersetzen, welche die Turing-Maschine ausführen kann, nehmen wir an, dass eine AKZEPTIEREN-Anweisung des Computers existiert, die z. B. dem Funktionsaufruf des Computers entspricht, der `ja` in eine Ausgabedatei schreibt. Wenn die TM die Ausführung dieser Computeranweisung simuliert, wechselt sie in einen eigenen akzeptierenden Zustand und hält an.

Diese Beschreibung ist weit von einem vollständigen formalen Beweis entfernt, dass eine TM einen typischen Computer simulieren kann, doch sie sollte Ihnen genügend Informationen bieten, um Sie davon zu überzeugen, dass eine TM ein vollständiges Modell dafür ist, was ein Computer leisten kann. Im Folgenden werden wir daher nur die Turing-Maschine als formales Modell dafür verwenden, was von einem beliebigen Rechengerät berechnet werden kann.

8.6.3 Ein Vergleich der Ausführungszeiten von Computern und Turing-Maschinen

Wir müssen nun die Rechenzeit der Turing-Maschine ansprechen, die einen Computer simuliert. Folgendes wurde schon angemerkt:

- Die Rechenzeit ist wichtig, da wir die TM nicht nur zur Untersuchung der Frage einsetzen wollen, was überhaupt berechnet werden kann, sondern auch, was ausreichend effizient berechnet werden kann, sodass die Computer-basierte Lösung eines Problems praktisch einsetzbar ist.
- Die Trennungslinie zwischen handhabbaren, also effizient lösbaren, Problemen und nicht handhabbaren Problemen, also solchen, die zwar lösbar sind, deren Zeitaufwand für die Lösung jedoch für eine Nutzbarkeit zu hoch ist, liegt im Allgemeinen zwischen dem, was in polynomialer Zeit berechenbar ist, und allem, was eine höhere Ausführungszeit erfordert.
- Wir müssen uns daher vergewissern, dass ein von einem Computer in polynomialer Zeit lösbares Problem von einer Turing-Maschine ebenfalls mit polynomialem Zeitaufwand lösbar ist und umgekehrt. Durch diese polynomiale Äquivalenz sind unsere Schlussfolgerungen darüber, was eine Turing-Maschine mit adäquater Effizienz ausführen kann und was nicht, ebenso auf einen Computer anwendbar.

In Abschnitt 8.4.3 wurde beschrieben, dass die Differenz in der Rechenzeit zwischen ein- und mehrbändigen TMs polynomial ist – genauer gesagt, quadratisch. Daher muss nur gezeigt werden, dass alles, was der Computer kann, auch von der in Abschnitt 8.6.2 beschriebenen mehrbändigen TM in einem Zeitraum erledigt werden kann, der gegenüber dem Zeitraum, den der Computer benötigt, polynomial ist. Wir wissen dann, dass das Gleiche für eine einbändige TM gilt.

Vor dem Beweis, dass die oben beschriebene Turing-Maschine n Schritte eines Computers in der Zeit $O(n^3)$ simulieren kann, müssen wir uns mit der Multiplikation als Computeranweisung befassen. Das Problem liegt darin, dass wir die Anzahl der Bits eines Computerwortes nicht eingeschränkt haben. Würde der Computer beispielsweise mit einem Wort beginnen, das die ganze Zahl 2 enthält, und dieses Wort in n aufeinander folgenden Schritten mit sich selbst multiplizieren, dann enthielte dieses Wort die Zahl 2^{2^n}. Diese Zahl beansprucht $2^n + 1$ Bit, und damit würde die Turing-Maschine zur Simulation dieser n Anweisungen einen mindestens in n exponentiellen Zeitraum benötigen.

Wir könnten auf einer festen Maximallänge der Wörter bestehen, beispielsweise 64 Bit. Führte nun eine Multiplikation (oder eine andere Operation) zu einem längeren Wort, dann würde der Computer anhalten, und die Turing-Maschine müsste ihn nicht weiter simulieren. Wir verwenden jedoch einen liberaleren Ansatz. Der Computer kann Wörter beliebiger Länge verwenden, doch eine Computeranweisung darf nur ein Wort produzieren, das um ein Bit länger als das längste der Argumente ist.

Beispiel 8.16 Unter der obigen Einschränkung ist die Addition erlaubt, da das Ergebnis nur ein Bit länger als die Maximallänge der Summanden sein kann. Die Multiplikation ist nicht erlaubt, da zwei m-Bit-Wörter ein Produkt der Länge $2m$ besitzen können. Wir können die Multiplikation von zwei m-Bit-Wörtern jedoch durch eine Folge von m Additionen simulieren, denen jeweils ein Verschieben des Multiplikanden um ein Bit nach links folgt (auch diese Operation erhöht die Wortlänge nur um 1). Wir können also beliebig lange Wörter multiplizieren, doch die vom Computer benötigte Rechenzeit ist proportional zum Quadrat der Länge der Operanden. ■

Unter der Annahme, dass das maximale Wachstum 1 Bit pro ausgeführter Computeranweisung beträgt, können wir die polynomiale Beziehung zwischen den beiden Ausführungszeiten beweisen. Die Idee des Beweises besteht darin, nach der Ausführung von n Anweisungen festzustellen, dass die Anzahl der auf dem Speicherband angesprochenen Wörter der TM $O(n)$ ist und jedes Computerwort $O(n)$ Zellen der Turing-Maschine zur Darstellung erfordert. Das Band ist daher $O(n^2)$ Zellen lang, und die Turing-Maschine kann die endliche Anzahl der von einer Computeranweisung benötigten Wörter in der Zeit $O(n^2)$ finden.

Allerdings muss den Anweisungen eine weitere Beschränkung auferlegt werden. Auch wenn die Anweisungen als Ergebnis kein langes Wort produzieren, könnte es sehr viel Zeit in Anspruch nehmen, das Ergebnis zu berechnen. Wir nehmen daher zusätzlich an, dass die Anweisung selbst, auf Wörter einer Länge bis zu k angewandt, von einer mehrbändigen Turing-Maschine in $O(k^2)$ Schritten ausgeführt werden kann. Mit Sicherheit sind die typischen Computeroperationen wie Addition, Verschieben und Wertevergleich von einer mehrbändigen TM in $O(k)$ Schritten ausführbar, und somit sind wir überaus liberal mit dem, was wir dem Computer zugestehen, in einer Anweisung auszuführen.

Satz 8.17 Verwendet ein Computer

1. nur Anweisungen, die die maximale Wortlänge um höchstens 1 erhöhen, und
2. nur Anweisungen, die eine mehrbändige TM mit Wörtern der Länge k in $O(k^2)$ oder weniger Schritten ausführen kann,

dann kann die in Abschnitt 8.6.2 beschriebene Turing-Maschine n Schritte des Computers in $O(n^3)$ eigenen Schritten simulieren.

Beweis: Wir gehen davon aus, dass das erste (Speicher-)Band der TM in Abbildung 8.20 nur das Programm des Computers enthält. Dieses Programm kann lang sein, doch es ist von konstanter Länge und von der Anzahl n der Anweisungsschritte, die der Computer ausführt, unabhängig. Es gibt also zwei Konstanten c und d mit folgenden Bedeutungen: Die Konstante c entspricht dem längsten der Wörter (einschließlich der Adressen) im Programm des Computers. Die Konstante d entspricht der Anzahl der Wörter des Programms.

Nach der Ausführung von n Schritten kann der Computer also keine Wörter kreiert haben, die die Länge von $c + n$ überschreiten, und kann auch keine Adressen kreiert oder verwendet haben, die länger als $c + n$ sind. Jede Anweisung kreiert höchstens eine neue Adresse, die einen Wert erhält, und somit liegt die Gesamtzahl der Adressen nach n ausgeführten Anweisungen bei höchstens $d + n$. Da jede Adresse-Wort-Kombination einschließlich der Adresse, des Inhalts und zweier Markierungssymbole als Separatoren höchstens $2(c + n) + 2$ Bit erfordert, ist die Gesamtzahl belegter TM-Band-

zellen nach n simulierten Anweisungen höchstens $2(d + n)(c + n + 1)$. Da es sich bei c und d um Konstanten handelt, ist die Anzahl der Zellen $O(n^2)$.

Wir wissen nun, dass jedes Nachschlagen von Adressen, das während einer einzelnen Computeranweisung notwendig ist, in der Zeit $O(n^2)$ ausgeführt werden kann. Da die Wörter die Länge $O(n)$ besitzen, wissen wir durch die zweite Annahme, die wir im Satz gemacht haben, dass die Anweisungen selbst von einer TM in der Zeit $O(n^2)$ ausgeführt werden können. Signifikant für die Ausführung einer Anweisung ist nun nur noch die Zeit, welche die TM benötigt, um auf dem Band Raum für ein neues oder expandiertes Wort zu schaffen. Verschieben erfordert das Kopieren von höchsten $O(n^2)$ Daten von Band 1 auf das Hilfsband und zurück. Für das Verschieben ist also auch nur ein Zeitaufwand von $O(n^2)$ pro Computeranweisung notwendig.

Wir stellen also abschließend fest, dass die TM einen von n Schritten des Computers in $O(n^2)$ eigenen Schritten ausführt. Daher können n Schritte des Computers in $O(n^3)$ Schritten der Turing-Maschine ausgeführt werden. ■

Nun wissen wir außerdem aus Abschnitt 8.4.3, dass eine einbändige TM eine mehrbändige TM simulieren kann und dazu höchstens das Quadrat der Anzahl der Schritte der mehrbändigen TM notwendig ist. Daher gilt:

Satz 8.18 *n Schritte eines Computers des in Satz 8.17 beschriebenen Typs können durch eine einbändige Turing-Maschine in höchstens $O(n^6)$ eigenen Schritten simuliert werden.* ■

8.7 Zusammenfassung von Kapitel 8

- *Die Turing-Maschine*: Die TM ist eine abstrakte Rechenmaschine mit der Leistungsfähigkeit sowohl realer Computer als auch anderer mathematischer Definitionen dessen, was berechnet werden kann. Die TM besteht aus einer Steuerung mit endlich vielen Zuständen und einem unendlichen, in Zellen unterteilten Band. Jede Zelle speichert ein Symbol aus einer endlichen Anzahl von Bandsymbolen, und eine Zelle ist die aktuelle Position des Bandkopfes. Die Bewegungen der TM basieren auf dem aktuellen Zustand und dem Bandsymbol in der Zelle, die der Bandkopf liest. Mit einer Bewegung der TM wird der Zustand geändert, ein Bandsymbol in die gelesene Zelle geschrieben und der Kopf um eine Zelle nach links oder rechts bewegt.
- *Akzeptieren durch eine Turing-Maschine*: Die TM startet mit der Eingabe, einer endlichen Zeichenreihe aus Eingabesymbolen, auf dem Band. Alle Zellen des restlichen Bandes enthalten das Symbol Leerzeichen. Das Leerzeichen gehört zu den Bandsymbolen, und die Eingabesymbole bilden eine Teilmenge der Bandsymbole. Das Leerzeichen gehört dabei nicht zu den Eingabesymbolen. Die TM akzeptiert eine Eingabe, falls sie irgendwann in einen akzeptierenden Zustand übergeht.
- *Rekursiv aufzählbare Sprachen* : Die von TMs akzeptierten Sprachen werden rekursiv aufzählbare Sprachen genannt. Diese Sprachen werden ebenso von jedem Computer erkannt oder akzeptiert.
- *Konfigurationen einer TM*: Wir können die aktuelle Konfiguration einer TM mit einer Zeichenreihe endlicher Länge beschreiben, die alle Bandzellen zwischen den am weitesten links und rechts stehenden Zeichen umfasst, die keine Leerzeichen sind. Zustand und Position des Kopfes sind ersichtlich, indem der Zustand inner-

halb der Folge von Bandsymbolen unmittelbar links von der aktuellen Zelle angegeben wird.

- *Speichern in der endlichen Steuerung*: Manchmal ist der Entwurf einer TM für eine bestimmte Sprache einfacher, wenn wir uns vorstellen, der Zustand würde aus zwei oder mehr Komponenten bestehen. Eine davon ist die eigentliche Steuerkomponente, und die anderen speichern Daten, die von der TM in diesem Zustand benötigt werden.
- *Mehrere Spuren*: Häufig hilft es, dass wir uns die Bandsymbole als Vektoren mit einer festen Anzahl von Komponenten vorstellen. Jede Komponente kann als separate Spur auf dem Band betrachtet werden.
- *Mehrbändige Turing-Maschinen* : Ein erweitertes TM-Modell ist mit einer festen Zahl von Bändern ausgestattet. Eine Bewegung dieser TM basiert auf dem Zustand sowie dem Vektor mit den Symbolen, die die Köpfe aller Bänder lesen. Mit einer Bewegung ändert die mehrbändige TM den Zustand, sie überschreibt Symbole in den Zellen unter den jeweiligen Bandköpfen und bewegt einige oder alle Bandköpfe um eine Zelle unabhängig voneinander nach links oder rechts. Die mehrbändige TM kann bestimmte Sprachen zwar schneller als die konventionelle einbändige TM erkennen, sie ist jedoch nicht in der Lage, Sprachen zu erkennen, die nicht rekursiv aufzählbar sind.
- *Nichtdeterministische Turing-Maschinen*: Die NTM besitzt eine endliche Anzahl von Auswahlmöglichkeiten für die nächste Bewegung (Zustand, neues Symbol und Kopfbewegung) für jeden Zustand und jedes gelesene Symbol. Sie akzeptiert eine Eingabe, wenn eine Folge von Auswahlmöglichkeiten zu einer Konfiguration mit einem akzeptierenden Zustand führt. Die NTM scheint zwar leistungsfähiger als die deterministische TM zu sein, kann jedoch keine Sprachen erkennen, die nicht rekursiv aufzählbar sind.
- *Turing-Maschinen mit einem semiunendlichen Band*: Wir können eine TM auf ein Band beschränken, das lediglich nach rechts unendlich ist und links von der anfänglichen Kopfposition keine Zellen besitzt. Eine solche TM akzeptiert jede rekursiv aufzählbare Sprache.
- *Turing-Maschinen mit mehreren Stacks*: Wir können die Bänder einer mehrbändigen TM einschränken, sodass sie sich wie ein Stack verhalten. Die Eingabe befindet sich auf einem separaten Band, das einmal von links nach rechts gelesen wird; dies imitiert den Eingabemodus eines endlichen Automaten oder PDA. Eine Maschine mit einem Stack ist tatsächlich ein DPDA, während eine Maschine mit zwei Stacks jede rekursiv aufzählbare Sprache akzeptieren kann.
- *Zählermaschinen* : Wir können die Stacks einer mit mehreren Stacks ausgestatteten Maschine weiter einschränken, sodass sie außer der Stackanfangsmarkierung nur ein weiteres Symbol enthalten. Jeder Stack fungiert daher als Zähler und ermöglicht, eine nicht negative ganze Zahl zu speichern sowie diese Zahl auf 0 zu prüfen. Mehr ist jedoch nicht möglich. Eine Maschine mit zwei Zählern ist ausreichend, um jede rekursiv aufzählbare Sprache zu akzeptieren.
- *Eine Turing-Maschine mit einem realen Computer simulieren*: Prinzipiell besteht die Möglichkeit, eine TM mit einem realen Computer zu simulieren, falls wir akzeptieren, dass ein potenziell unendlicher Vorrat an Wechselspeichermedien wie Fest-

platten zur Verfügung steht, um den nicht leeren Abschnitt des TM-Bandes zu simulieren. Da die physikalischen Ressourcen für Festplatten nicht unendlich sind, ist diese Aussage fragwürdig. Da allerdings nicht bekannt ist, wo die Grenzen für Speicher im Universum liegen, und davon ausgegangen werden kann, dass sie jedenfalls sehr groß sind, ist die Annahme einer unendlichen Ressource wie ein TM-Band in der Praxis realistisch und allgemein akzeptiert.

- *Einen Computer mit einer Turing-Maschine simulieren*: Eine TM kann den Speicher und die Steuerung eines realen Computers simulieren, indem ein Band alle Adressen von Zellen und deren Inhalte (Register, Hauptspeicher, Festplatten und andere Speichermedien) speichert. Wir können daher darauf vertrauen, dass etwas, was eine TM nicht leisten kann, auch von einem realen Computer nicht geleistet werden kann.

LITERATURANGABEN ZU KAPITEL 8

Die Turing-Maschine ist [8] entnommen. Etwa zur gleichen Zeit erschienen mehrere weniger an Maschinen orientierte Vorschläge zur Charakterisierung dessen, was berechnet werden kann, darunter die Arbeiten von Church [1], Kleene [5] und Post [7]. Diesen ging die Arbeit von Gödel [3] voraus, die erfolgreich zeigte, dass ein Computer nicht in der Lage ist, alle mathematischen Fragen zu beantworten.

Das Studium der mehrbändigen Turing-Maschinen, insbesondere des Themas, wie deren Ausführungszeit im Vergleich zu der des einbändigen Modells ausfällt, begann mit Hartmanis und Stearns [4]. Die Untersuchung von Maschinen mit mehreren Stacks und Zählermaschinen stammt aus [6], die hier vorgestellte Konstruktion dagegen aus [2].

Der Ansatz in Abschnitt 8.1, »hello, world« als Ersatz für Akzeptieren oder Anhalten einer Turing-Maschine zu verwenden, erschien in unveröffentlichten Notizen von S. Rudich.

1. A. Church [1936]. »An undecidable problem in elementary number theory«, *American J. Math.* **58**, 345–363.
2. P. C. Fischer [1966]. »Turing machines with restricted memory access«, *Information and Control* **9**:4, 364–379.
3. K. Gödel [1931]. »Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme«, *Monatshefte für Mathematik und Physik* **38**, 173–198.
4. J. Hartmanis und R. E. Stearns [1965]. »On the computational complexity of algorithms«, *Transactions of the AMS* **117**, 285–306.
5. S. C. Kleene [1936]. »General recursive functions of natural numbers«, *Mathematische Annalen* **112**, 727–742.
6. M. L. Minsky [1961]. »Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines«, *Annals of Mathematics* **74**:3, 437–455.
7. E. Post [1936]. »Finite combinatory processes-formulation«, *J. Symbolic Logic* **1**, 103–105.
8. A. M. Turing [1936]. »On computable numbers with an application to the Entscheidungsproblem«, *Proc. London Math. Society* **2**:42, 230–265. Korrektur ebd., **2**:43, 544–546.

KAPITEL 9

Unentscheidbarkeit

In diesem Kapitel wird zunächst die auf plausible Argumente gegründete Behauptung aus Abschnitt 8.1 im Kontext der Turing-Maschinen wiederholt: Es gibt Probleme, die nicht von Computern gelöst werden können. Das Problem beim »Beweis« lag darin, dass wir gezwungen waren, die realen Beschränkungen jeder Implementierung von C (oder jeder anderen Programmiersprache) auf einem realen Computer zu ignorieren. Diese Beschränkungen wie etwa die Größe des Adressraums sind allerdings nicht grundlegend. Zukünftig erwarten wir die Entwicklung von Computern, deren Adressräume, Hauptspeicher und sonstige Speichermedien nahezu unbeschränkt wachsen werden.

Wir konzentrieren uns auf Turing-Maschinen, die solchen Einschränkungen nicht unterliegen, und können daher die eigentliche Idee besser fassen, nämlich wozu Rechengeräte, wenn nicht heute, dann zumindest künftig in der Lage sein werden. In diesem Kapitel beweisen wir formal die Existenz eines Problems im Zusammenhang mit Turing-Maschinen, das keine Turing-Maschine lösen kann. Wir wissen aus Abschnitt 8.6, dass Turing-Maschinen reale Computer simulieren können, auch solche, die nicht den heutzutage bekannten Einschränkungen unterliegen, und folgern daher rigoros, dass das Problem

- Akzeptiert eine beliebig vorgegebene Turing-Maschine (den Code von) sich selbst als Eingabe?

nicht von einem Computer gelöst werden kann, unabhängig davon, wie locker wir die praktischen Grenzen handhaben.

Wir werden dann Probleme, die von einer Turing-Maschine gelöst werden können, in zwei Klassen einteilen: in solche, für die es einen *Algorithmus* gibt (etwa eine Turing-Maschine, die unabhängig davon anhält, ob sie die Eingabe akzeptiert oder nicht), und in solche, die nur von Turing-Maschinen gelöst werden können, die unendlich lang mit einigen oder allen Eingaben arbeiten, die sie nicht akzeptieren. Diese zweite Form der Akzeptanz ist problematisch, da wir für manche Eingaben, die nicht akzeptiert werden, zu keinem Zeitpunkt wissen, ob sie akzeptiert werden oder nicht, unabhängig davon, wie lange die TM arbeitet. Wir werden uns daher auf Techniken konzentrieren, die zeigen, dass Probleme »unentscheidbar« sind; für diese gibt es keinen Algorithmus, auch wenn sie von einer Turing-Maschine akzeptiert werden, die aber bei einigen Eingaben nicht anhält.

Wir beweisen das folgende Problem als unentscheidbar:

- Akzeptiert eine beliebig vorgegebene Turing-Maschine eine beliebig vorgegebene Eingabe?

Danach nützen wir dieses Ergebnis der Unentscheidbarkeit aus, um weitere unentscheidbare Probleme anzugeben. Beispielsweise zeigen wir, dass alle nicht trivialen Probleme bezüglich einer von einer Turing-Maschine akzeptierten Sprache ebenso unentscheidbar sind wie einige Probleme, die in keiner Beziehung zu Turing-Maschinen, Programmen oder Computern stehen.

9.1 Eine nicht rekursiv aufzählbare Sprache

Wir wissen, dass eine Sprache L rekursiv aufzählbar ist, wenn $L = L(M)$ für eine TM M gilt. In Abschnitt 9.2 stellen wir »rekursive« oder »entscheidbare« Sprachen vor, die nicht nur rekursiv aufzählbar sind, sondern auch von einer TM akzeptiert werden, die unabhängig von der Akzeptanz immer anhält.

Unser langfristiges Ziel ist der Beweis, dass die aus Paaren (M, w) bestehende Sprache unentscheidbar ist, wobei gilt:

1. M ist eine Turing-Maschine (passend binär kodiert) mit dem Eingabealphabet $\{0, 1\}$.
2. w ist eine Zeichenreihe aus den Zeichen 0 und 1.
3. M akzeptiert die Eingabe w.

Falls dieses Problem bei einer Eingabe, die auf das Binäralphabet beschränkt ist, unentscheidbar ist, dann ist das allgemeinere Problem, wobei TMs jedes Alphabet zur Verfügung steht, sicher unentscheidbar.

Im ersten Schritt stellen wir die Frage nach der Zugehörigkeit zu einer bestimmten Sprache. Dazu benötigen wir einen Code für Turing-Maschinen, der unabhängig von der Anzahl der Zustände der TM nur die Zeichen 0 und 1 verwendet. Sobald wir diese Kodierung haben, können wir jede binäre Zeichenreihe behandeln, als sei sie eine Turing-Maschine. Ist die Zeichenreihe keine wohlgeformte Darstellung einer TM, dann sehen wir sie als Darstellung einer TM ohne Bewegungen an. Daher können wir jede Binärzeichenreihe als eine TM betrachten.

Ein Zwischenziel, das Thema dieses Abschnitts, involviert die Sprache L_d, die »Diagonalisierungssprache«, die aus all jenen Zeichenreihen w besteht, für die die von w repräsentierte TM die Eingabe w nicht akzeptiert. Wir werden zeigen, dass keine Turing-Maschine L_d akzeptiert. Wir wissen, wenn gezeigt wird, dass es für eine Sprache keine Turing-Maschine gibt, dann ist dies stärker als zu zeigen, dass die Sprache unentscheidbar ist (d. h. dass es für sie keinen Algorithmus bzw. keine TM gibt, die immer anhält).

Die Sprache L_d spielt eine Rolle analog zum hypothetischen Programm H_2 aus Abschnitt 8.1.2, das immer dann `hello, world` ausgibt, wenn seine Eingabe *nicht* `hello, world` ausgibt, wenn es sich selbst als Eingabe erhält. Präziser gesagt, so wie es H_2 nicht geben kann, da seine Reaktion, wenn es sich selbst als Eingabe erhält, paradox ist, kann L_d nicht von einer Turing-Maschine akzeptiert werden; denn würde die Sprache akzeptiert, dann würde diese Turing-Maschine mit sich selbst in Widerspruch stehen, falls sie den eigenen Code als Eingabe erhielte.

9.1.1 Binärzeichenreihen aufzählen

Im Folgenden müssen wir allen Binärzeichenreihen ganze Zahlen zuweisen, sodass jede Zeichenreihe jeweils einer ganzen Zahl und jede ganze Zahl einer Zeichenreihe zugeordnet ist. Ist w eine Binärzeichenreihe, dann behandeln wir $1w$ als die binäre ganze Zahl i. Wir nennen w dann die i-te Zeichenreihe. Also ist $\in$ die erste Zeichenreihe, 0 die zweite, 1 die dritte, 00 die vierte, 01 die fünfte und so weiter. Auf diese Weise werden Zeichenreihen nach der Länge und gleich lange Zeichenreihen lexikografisch sortiert. Von nun an nennen wir die i-te Zeichenreihe w_i.

9.1.2 Codes für Turing-Maschinen

Unser nächstes Ziel ist die Entwicklung eines Binärcodes für Turing-Maschinen, sodass jede TM mit Eingabealphabet {0, 1} als Binärzeichenreihe angesehen werden kann. Da wir gerade gesehen haben, wie Binärzeichenreihen aufzuzählen sind, haben wir dann eine Identifikation der Turing-Maschinen mit ganzen Zahlen und können über »die i-te Turing-Maschine M_i« sprechen. Um eine TM $M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F)$ als Binärzeichenreihe zu repräsentieren, müssen wir den Zuständen, Bandsymbolen und Richtungen L und R zunächst ganze Zahlen zuweisen.

- Wir nehmen an, es existieren die Zustände $q_1, q_2, \ldots, q_k$ für ein k. q_1 ist immer der Startzustand, und q_2 ist der einzige akzeptierende Zustand. Beachten Sie, dass nur ein einziger akzeptierender Zustand erforderlich ist, da wir annehmen können, dass die TM anhält, wann immer sie in einen akzeptierenden Zustand übergeht[1].
- Wir nehmen an, es existieren die Bandsymbole $X_1, X_2, \ldots, X_m$ für ein m. X_1 ist immer das Symbol 0, X_2 immer das Symbol 1 und X_3 immer B, das Leerzeichen. Alle weiteren Bandsymbole werden $X_4, X_5, \ldots, X_m$ beliebig zugeordnet.
- Wir bezeichnen die Richtung L als D_1 und die Richtung R als D_2.

Da den Zuständen und Bandsymbolen jeder TM M ganze Zahlen in vielen unterschiedlichen Reihenfolgen zugeordnet werden können, wird es im Allgemeinen mehr als eine Kodierung für eine TM geben. Allerdings ist diese Tatsache im Folgenden nicht von Bedeutung, da wir nun zeigen, dass keine Kodierung eine TM M repräsentieren kann, sodass $L(M) = L_d$ ist.

Sobald jeder Zustand, jedes Symbol und jede Richtung von einer ganzen Zahl repräsentiert werden, können wir die Übergangsfunktion δ kodieren. Angenommen, eine Übergangsfunktion sei $\delta(q_i, X_j) = (q_k, X_l, D_m)$ für ganze Zahlen i, j, k, l und m. Wir kodieren diese Regel mit der Zeichenreihe $0^i10^j10^k10^l10^m$. Beachten Sie, dass im Code eines einzelnen Übergangs die 1 nie mehrfach hintereinander auftritt, da i, j, k, l und m mindestens den Wert 1 besitzen.

Der Code für die gesamte TM M besteht aus allen Codes für die Übergänge, die beliebig geordnet und jeweils durch 11 voneinander getrennt sind:

$$C_1 11 C_2 11 \ldots C_{n-1} 11 C_n$$

Jedes C repräsentiert den Code eines Übergangs von M.

1. Mit dieser Konvention verzichten wir darauf, Informationen über den Verlauf der Berechnungen der TM bis zum Anhalten durch den akzeptierenden Zustand zu übermitteln. Dieser Verzicht beeinträchtigt die folgenden Überlegungen allerdings nicht.

Beispiel 9.1 Die fragliche TM sei

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$$

wobei δ aus den folgenden Regeln besteht:

$$\delta(q_1, 1) = (q_3, 0, R)$$
$$\delta(q_3, 0) = (q_1, 1, R)$$
$$\delta(q_3, 1) = (q_2, 0, R)$$
$$\delta(q_3, B) = (q_3, 1, L)$$

Der Code für jede der Regeln lautet wie folgt:

0100100010100
0001010100100
00010010010100
0001000100010010

Beispielsweise kann die erste Regel als $\delta(q_1, X_2) = (q_3, X_1, D_2)$ ausgedrückt werden, da $1 = X_2$, $0 = X_1$ und $R = D_2$ ist. Der Code lautet also $0^1 1 0^2 1 0^3 1 0^1 1 0^2$. Ein Code für M lautet:

0100100010100110001010100100110001001001010011000100010001001 0

Beachten Sie, dass es noch viele weitere mögliche Codes für M gibt. Insbesondere die Codes für die vier Übergänge könnten in jeder der 4! möglichen Reihenfolgen geschrieben werden, und damit erhielten wir 24 Codes für M. ■

In Abschnitt 9.2.3 müssen wir Paare (M, w) kodieren, die aus einer TM und einer Zeichenreihe bestehen. Für dieses Paar verwenden wir den Code von M, gefolgt von 111, dem w folgt. Beachten Sie, dass wir sicher sein können, dass das erste Auftreten von 111 die Codes von M und w trennt, da kein gültiger Code einer TM dreimal die 1 hintereinander enthält. Handelte es sich bei M beispielsweise um die TM aus Beispiel 9.1 und hätte w den Wert 1011, dann bestünde der Code von (M, w) aus der Zeichenreihe am Ende des Beispiels, gefolgt von 1111011.

9.1.3 Die Diagonalisierungssprache

In Abschnitt 9.1.2 haben wir Turing-Maschinen kodiert, sodass wir nun eine konkrete Vorstellung von M_i, der »i-ten Turing-Maschine«, besitzen: die TM M, deren Code w_i ist, die i-te Binärzeichenreihe. Viele ganze Zahlen entsprechen keiner TM. Beispielsweise beginnt 11001 nicht mit 0, und in 0010111010010100 folgt einmal die 1 dreimal hintereinander. Ist w_i kein gültiger TM-Code, dann nehmen wir an, M_i sei die TM mit einem Zustand und keinen Übergängen. Für diese Werte von i ist M_i eine Turing-Maschine, die bei jeder Eingabe sofort anhält, ohne zu akzeptieren. $L(M_i)$ ist daher $\varnothing$, falls w_i kein gültiger TM-Code ist.

Nun können wir eine wichtige Definition formulieren:

- Die Sprache L_d, die *Diagonalisierungssprache*, ist die Menge der Zeichenreihen w_i, sodass w_i nicht in $L(M_i)$ enthalten ist.

L_d besteht also aus allen Zeichenreihen w, sodass die TM M, deren Code w ist, die Eingabe w nicht akzeptiert.

Abbildung 9.1: Die Tabelle repräsentiert die Akzeptanz von Zeichenreihen durch Turing-Maschinen

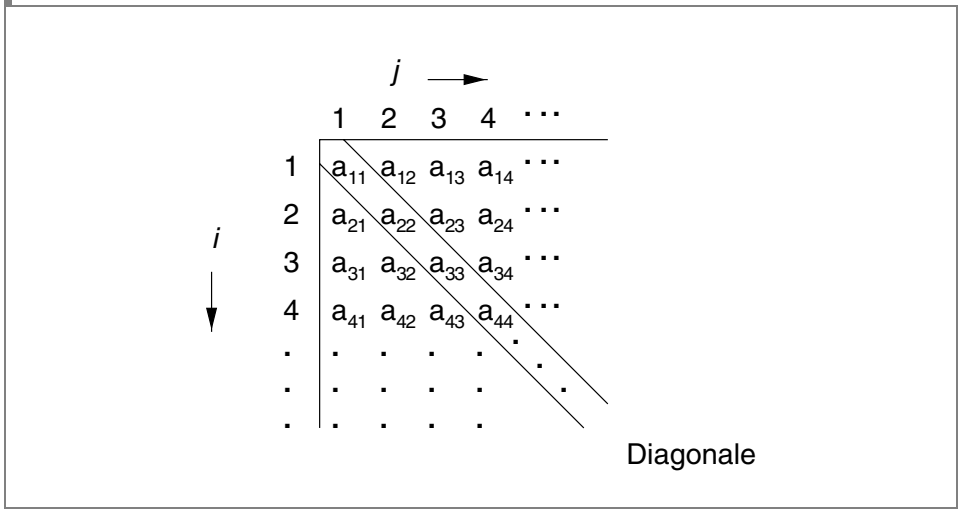


Abbildung 9.1 zeigt, warum L_d eine Diagonalisierungssprache genannt wird. Diese Tabelle sagt für alle i und j aus, ob die TM M_i die Eingabezeichenreihe w_j akzeptiert; $a_{ij} = 1$ bedeutet »sie akzeptiert«, $a_{ij} = 0$ dagegen »nein, sie akzeptiert nicht«. Wir können uns die i-te Zeile als den *charakteristischen Vektor* der Sprache $L(M_i)$ vorstellen; die Einsen in dieser Zeile zeigen die Zeichenreihen an, die in dieser Sprache enthalten sind.

Die Werte in der Diagonalen sagen aus, ob M_i w_i akzeptiert. Kehrt man also die Werte der Diagonalen um, d. h. $\overline{a}_{ii} = 1$, falls $a_{ii} = 0$, und $\overline{a}_{ii} = 0$, falls $a_{ii} = 1$ ist, dann stellen die $\overline{a}_{ii}$ den charakteristischen Vektor der Sprache L_d dar.

Dieser Trick, die Diagonale umzukehren, um den charakteristischen Vektor einer Sprache zu konstruieren, der in keiner Zeile auftreten kann, wird *Diagonalisierung* genannt. Er kann angewendet werden, weil es sich beim Komplement der Diagonalen selbst um einen charakteristischen Vektor handelt, der die Zugehörigkeit zu einer Sprache beschreibt, hier zu L_d. Dieser charakteristische Vektor stimmt mit keiner Zeile der in Abbildung 9.1 dargestellten Tabelle überein. Also kann das Komplement der Diagonalen nicht der charakteristische Vektor einer Turing-Maschine sein.

9.1.4 Der Beweis, dass L_d nicht rekursiv aufzählbar ist

Nach der obigen intuitiven Behandlung charakteristischer Vektoren und der Diagonalen werden wir nun eine grundlegende Aussage über Turing-Maschinen formal beweisen: Es existiert keine Turing-Maschine, welche die Sprache L_d akzeptiert.

Satz 9.2 L_d ist keine rekursiv aufzählbare Sprache; das heißt es gibt keine Turing-Maschine, die L_d akzeptiert.

Beweis: Angenommen, L_d sei $L(M)$ für eine TM M. Da L_d eine Sprache über dem Alphabet {0, 1} ist, wäre M in der konstruierten Liste der Turing-Maschinen enthalten, da sie alle TMs mit Eingabealphabet {0, 1} umfasst. Daher existiert mindestens ein Code i für M, d. h. $M = M_i$.

Nun fragen wir, ob w_i in L_d enthalten ist.

- Ist w_i in L_d enthalten, dann akzeptiert M_i w_i. Doch nach Definition von L_d ist w_i nicht in L_d enthalten, da L_d nur solche w_j enthält, sodass M_j w_j *nicht* akzeptiert.
- Ist w_i nicht in L_d enthalten, dann akzeptiert M_i w_i nicht. Nach der Definition von L_d ist w_i daher in L_d enthalten.

Da w_i weder in L_d enthalten noch nicht darin enthalten sein kann, schließen wir, dass dies ein Widerspruch zu unserer Aussage ist, dass M existiert. L_d ist also keine rekursiv aufzählbare Sprache. ■

9.1.5 Übungen zum Abschnitt 9.1

Übung 9.1.1 Um welche Zeichenreihen handelt es sich bei

* a) w_{37}?

b) w_{100}?

Übung 9.1.2 Schreiben Sie einen der möglichen Codes für die Turing-Maschine aus Tabelle 8.1.

! **Übung 9.1.3** Es folgen zwei Definitionen von Sprachen, die der Definition von L_d ähneln, sich aber von L_d unterscheiden sind. Zeigen Sie für beide Sprachen, dass sie von keiner Turing-Maschine akzeptiert werden, und verwenden Sie dazu ein der Diagonalisierung ähnliches Argument. Beachten Sie, dass Ihr Beweis nicht auf der Diagonalen selbst entwickelt werden kann, sondern dass Sie nach einer anderen unendlichen Folge von Komponenten in der Matrix (Abb. 9.1) suchen müssen.

* a) Die Menge aller w_i, sodass w_i nicht von M_{2i} akzeptiert wird.

b) Die Menge aller w_i, sodass w_{2i} nicht von M_i akzeptiert wird.

! **Übung 9.1.4** Wir haben nur Turing-Maschinen betrachtet, die das Eingabealphabet {0, 1} besitzen. Angenommen, wir möchten allen Turing-Maschinen unabhängig vom Eingabealphabet eine ganze Zahl zuweisen. Dies ist nicht so ohne weiteres möglich, da zwar die Namen der Zustände und der Bandsymbole, die keine Eingabesymbole sind, beliebig sind, dies jedoch nicht für die eigentlichen Eingabesymbole gilt. Beispielsweise sind die beiden Sprachen $\{0^n1^n \mid n \geq 1\}$ und $\{a^nb^n \mid n \geq 1\}$ zwar äquivalent, doch handelt es sich *nicht* um die gleiche Sprache, und sie werden von verschiedenen TMs akzeptiert. Wir nehmen jedoch an, wir hätten eine unendliche Menge von Symbolen $\{a_1, a_2, \ldots\}$, aus der alle TM-Eingabealphabete ausgewählt werden. Zeigen Sie, wie wir allen TMs eine ganze Zahl zuweisen können, deren Eingabealphabet eine endliche Teilmenge dieser Symbole ist.

9.2 Ein unentscheidbares Problem, das rekursiv aufzählbar ist

Wir kennen nun ein nicht rekursiv aufzählbares Problem, die Diagonalisierungssprache L_d, die von keiner Turing-Maschine akzeptiert wird. Unser nächstes Ziel besteht darin, die rekursiv aufzählbaren Sprachen (diejenigen, die von TMs akzeptiert wer-

den) in zwei Klassen aufzuteilen. Eine der Klassen, die dem entsprechen, was wir uns allgemein als Algorithmus vorstellen, hat eine TM, die nicht nur die Sprache erkennt, sondern auch mitteilt, ob die Eingabezeichenreihe nicht in der Sprache enthalten ist. Eine solche Turing-Maschine hält immer an, unabhängig davon, ob ein akzeptierender Zustand erreicht wurde oder nicht.

Die zweite Sprachklasse besteht aus jenen rekursiv aufzählbaren Sprachen, die von keiner Turing-Maschine akzeptiert werden, die garantiert anhält. Diese Sprachen werden auf eine unbequeme Weise akzeptiert: Ist die Eingabe in der Sprache enthalten, dann erfahren wir dies irgendwann nach endlicher Zeit, im anderen Fall jedoch könnte die Turing-Maschine endlos laufen, und wir wären nie sicher, ob die Eingabe schließlich nicht doch akzeptiert würde. Ein Beispiel für diesen Sprachtyp ist, wie wir sehen werden, die Menge der kodierten Paare (M, w), sodass TM M Eingabe w akzeptiert.

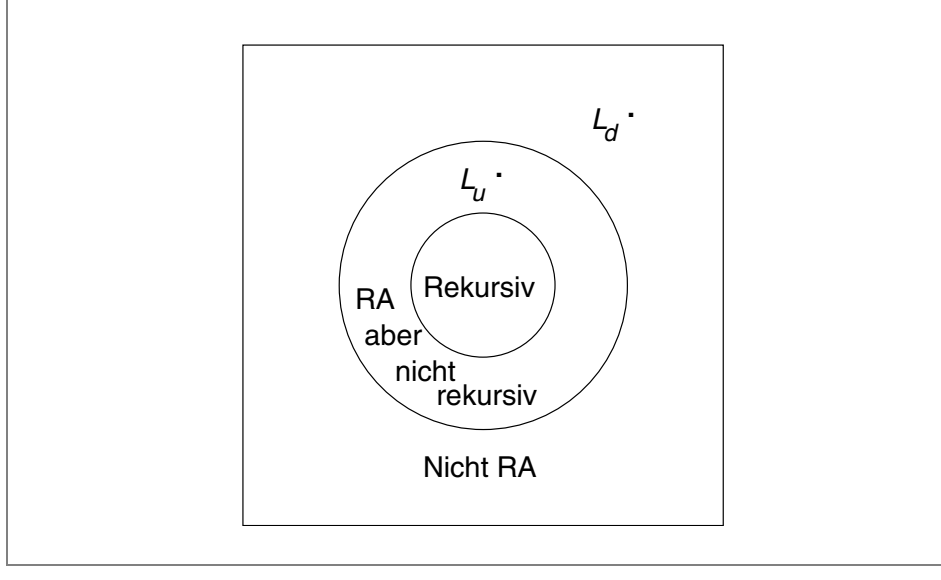
9.2.1 Rekursive Sprachen

Wir nennen eine Sprache L rekursiv, wenn $L = L(M)$ für eine Turing-Maschine M, sodass Folgendes gilt:

1. Ist w in L enthalten, dann akzeptiert M (und hält daher an).
2. Ist w nicht in L enthalten, dann hält M schließlich an, geht jedoch nie in einen akzeptierenden Zustand über.

Eine TM dieses Typs entspricht unserer informellen Definition eines »Algorithmus«, einer wohldefinierten Folge von Schritten, die immer terminiert und zu einer Antwort führt. Stellen wir uns die Sprache L als »Problem« vor, wie es häufig der Fall sein wird, dann wird das Problem L *entscheidbar* genannt, wenn es eine rekursive Sprache ist, und *unentscheidbar*, wenn es keine rekursive Sprache ist.

Abbildung 9.2: Die Beziehung zwischen rekursiven, rekursiv aufzählbaren (RA) und nicht rekursiv aufzählbaren Sprachen



Häufig ist die Frage wichtiger, ob es einen Algorithmus zur Lösung eines Problems gibt, als die Frage nach der Existenz einer TM zur Lösung des Problems. Wie oben angemerkt, erhalten wir von den Turing-Maschinen, die nicht garantiert halten, eventuell nicht ausreichende Informationen, um darauf zu schließen, dass eine Zeichenreihe nicht in der Sprache enthalten ist. In einem gewissen Sinn haben sie also »das Problem nicht gelöst«. Daher ist es oft wichtiger, Probleme oder Sprachen in entscheidbar – sie werden von einem Algorithmus gelöst – und unentscheidbar zu unterteilen, statt eine Klassifizierung nach rekursiv aufzählbaren Sprachen (die eine TM besitzen) und nicht rekursiv aufzählbaren Sprachen (die keine TM besitzen) vorzunehmen. Abbildung 9.2 zeigt die Beziehungen zwischen den drei Sprachklassen:

1. Die rekursiven Sprachen
2. Die rekursiv aufzählbaren, jedoch nicht rekursiven Sprachen
3. Die nicht rekursiv aufzählbaren Sprachen

Abbildung 9.2 zeigt außerdem die Einordnung der nicht rekursiv aufzählbaren Sprache L_d sowie der Sprache L_u oder »universellen Sprache«, die wir im Folgenden als nicht rekursiv, aber als rekursiv aufzählbar beweisen.

9.2.2 Komplemente rekursiver und rekursiv aufzählbarer Sprachen

Ein leistungsfähiges Werkzeug für den Beweis, dass Sprachen im zweiten Ring in Abbildung 9.2 angesiedelt sind (dass sie rekursiv aufzählbar, aber nicht rekursiv sind), ist die Betrachtung des Komplements einer Sprache. Wir werden zeigen, dass die rekursiven Sprachen unter der Komplementbildung abgeschlossen sind. Ist eine Sprache L also rekursiv aufzählbar, das Komplement $\overline{L}$ von L jedoch nicht, dann wissen wir, dass L nicht rekursiv sein kann. Wäre L rekursiv, dann wäre $\overline{L}$ ebenfalls rekursiv und damit auch rekursiv aufzählbar. Wir beweisen nun diese wichtige Eigenschaft der Abgeschlossenheit für die rekursiven Sprachen.

Warum »rekursiv«?

Programmierer sind heutzutage mit rekursiven Funktionen vertraut. Allerdings scheint zwischen diesen rekursiven Funktionen und Turing-Maschinen, die immer anhalten, kein Zusammenhang zu bestehen. Schlimmer noch, »nicht rekursiv« bezieht sich auf Sprachen, die von keinem Algorithmus erkannt werden, doch Programmierer sind daran gewöhnt, dass sich »nicht rekursiv« auf Berechungen bezieht, die so einfach sind, dass keine rekursiven Funktionsaufrufe nötig sind.

Der Begriff »rekursiv« als Synonym für »entscheidbar« stammt aus der Mathematik zu einer Zeit, als noch keine Computer existierten. Damals wurden Formalismen für Berechungen, die auf Rekursion basierten (nicht jedoch auf Iteration oder Schleifen), allgemein zur Darstellung von Berechnungen verwendet. Diese Darstellungen, auf die wir hier nicht weiter eingehen, waren etwa mit Berechnungen in funktionalen Programmiersprachen wie LISP oder ML vergleichbar. In diesem Sinne bedeutet die Aussage, ein Problem sei »rekursiv« so viel wie »Es ist ausreichend einfach, sodass ich eine rekursive Funktion schreiben kann, um es zu lösen, und diese Funktion terminiert immer«. Dies ist exakt die Bedeutung, die der Begriff heute im Zusammenhang mit Turing-Maschinen besitzt.

→

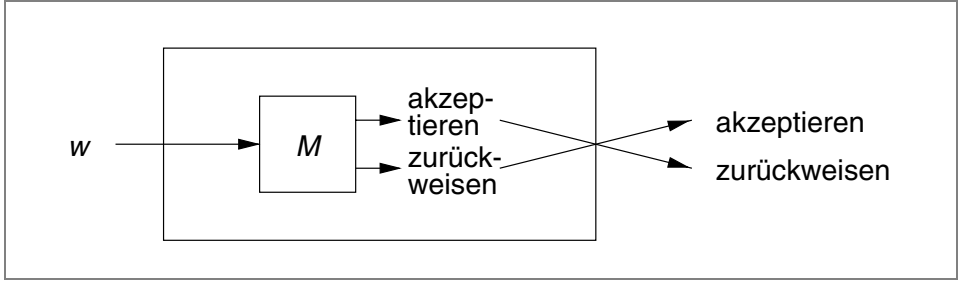
→ Der Begriff »rekursiv aufzählbar« stammt aus der gleichen Konzeptfamilie. Eine Funktion könnte alle Elemente einer Sprache in irgendeiner Reihenfolge auflisten; sie könnten also »aufgezählt« werden. Die Sprachen, deren Elemente in irgendeiner Reihenfolge auflistbar sind, stimmen mit den Sprachen überein, die von einer TM akzeptiert werden, wobei diese TM aber bei Eingaben, die sie nicht akzeptiert, nicht notwendigerweise zum Halten kommen muss.

Satz 9.3 Wenn L eine rekursive Sprache ist, dann ist auch $\overline{L}$ eine rekursive Sprache.

Beweis: Sei $L = L(M)$ für eine TM M, die immer anhält. Wir konstruieren entsprechend der Abbildung 9.3 eine TM $\overline{M}$, sodass $\overline{L} = L(\overline{M})$. $\overline{M}$ verhält sich im Wesentlichen wie M. Allerdings wird M wie folgt modifiziert, um $\overline{M}$ zu konstruieren:

1. Die akzeptierenden Zustände von M werden zu nicht akzeptierenden Zuständen von $\overline{M}$ ohne weitere Übergänge; $\overline{M}$ hält also in diesen Zuständen an, ohne zu akzeptieren.
2. $\overline{M}$ besitzt einen neuen akzeptierenden Zustand r; von r aus gibt es keine weiteren Übergänge.
3. Für jede Kombination aus einem nicht akzeptierenden Zustand von M und einem Bandsymbol von M, sodass M dafür keinen Übergang besitzt (M hält an, ohne zu akzeptieren), fügen wir in $\overline{M}$ einen Übergang zum akzeptierenden Zustand r hinzu.
4. Da M garantiert anhält, wissen wir, dass $\overline{M}$ ebenfalls anhält. Außerdem akzeptiert $\overline{M}$ exakt die Zeichenreihen, die M nicht akzeptiert. Daher akzeptiert $\overline{M}$ $\overline{L}$.

Abbildung 9.3: Konstruktion einer TM, die das Komplement einer rekursiven Sprache akzeptiert



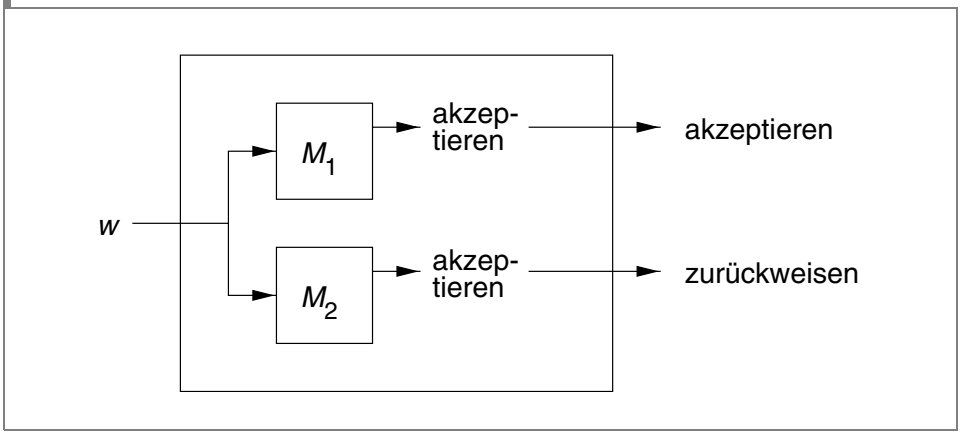
Es gibt eine weitere wichtige Feststellung über Komplemente von Sprachen, die in gewissen Fällen einschränkt, wo eine Sprache und ihr Komplement in das Diagramm in Abbildung 9.2 einzuordnen sind. Wir legen diese Einschränkung im nächsten Satz dar.

Satz 9.4 Ist sowohl eine Sprache L als auch ihr Komplement rekursiv aufzählbar, dann ist L rekursiv. Beachten Sie, dass nach Satz 9.3 dann auch $\overline{L}$ rekursiv ist.

Beweis: Abbildung 9.4 legt den Beweis nahe. Sei $L = L(M_1)$ und $\overline{L} = L(M_2)$. M_1 und M_2 werden parallel von einer TM M simuliert. Wir richten M als zweibändige TM ein, die wir dann in eine einbändige TM konvertieren, um die Simulation einfach und über-

sichtlich zu gestalten. Ein Band von M simuliert das Band von M_1, das andere dagegen das Band von M_2. Die Zustände von M_1 und M_2 sind jeweils Komponenten des Zustands von M.

Abbildung 9.4: Simulation zweier TMs, die eine Sprache und deren Komplement akzeptieren



Ist die Eingabe w von M in L enthalten, dann wird M_1 akzeptieren. In diesem Fall akzeptiert M und hält an. Ist w nicht in L enthalten, dann in $\overline{L}$, und in diesem Fall akzeptiert M_2. Wenn M_2 akzeptiert, hält M an, ohne zu akzeptieren. M hält also bei allen Eingaben an, und $L(M)$ stimmt mit L überein. Da M immer anhält und $L(M) = L$, schließen wir, dass L rekursiv ist. ■

Wir können die Sätze 9.3 und 9.4 zusammenfassen. Von den neun Möglichkeiten, eine Sprache L und deren Komplement $\overline{L}$ im Diagramm aus Abbildung 9.2 zu platzieren, sind nur die folgenden vier möglich:

1. Sowohl L als auch $\overline{L}$ ist rekursiv; beide befinden sich also im inneren Ring.
2. Weder L noch $\overline{L}$ ist rekursiv aufzählbar; beide befinden sich im äußeren Ring.
3. L ist rekursiv aufzählbar, aber nicht rekursiv, und $\overline{L}$ ist nicht rekursiv aufzählbar; eine befindet sich also im mittleren Ring und die andere im äußeren.
4. $\overline{L}$ ist rekursiv aufzählbar, aber nicht rekursiv, und L ist nicht rekursiv; hier haben wir die gleiche Situation wie in (3), jedoch tauschen L und $\overline{L}$ die Positionen.

Beim Beweis dieser Zusammenfassung eliminiert Satz 9.3 die Möglichkeit, dass eine der Sprachen (L oder $\overline{L}$) rekursiv ist und die andere einer der beiden weiteren Klassen angehört. Satz 9.4 eliminiert die Möglichkeit, dass beide zwar rekursiv aufzählbar, aber nicht rekursiv sind.

Beispiel 9.5 Als Beispiel sehen wir uns die Sprache L_d an, von der wir wissen, dass sie nicht rekursiv aufzählbar ist. Daher kann $\overline{L}_d$ nicht rekursiv sein. Allerdings wäre es möglich, dass $\overline{L}_d$ entweder nicht rekursiv aufzählbar oder rekursiv aufzählbar, aber nicht rekursiv ist. Tatsächlich ist diese Sprache rekursiv aufzählbar, aber nicht rekursiv.

$\overline{L}_d$ ist die Menge der Zeichenreihen w_i, sodass M_i w_i akzeptiert. Diese Sprache ähnelt der universellen Sprache L_u, die aus allen Paaren (M, w) besteht, sodass M w akzeptiert. Wir zeigen in Abschnitt 9.2.3, dass L_u rekursiv aufzählbar ist. Mit der gleichen Schlussfolgerung kann man zeigen, dass $\overline{L}_d$ ebenfalls rekursiv aufzählbar ist. ■

9.2.3 Die universelle Sprache

Wir haben schon im Abschnitt 8.6.2 informell diskutiert, wie eine Turing-Maschine eingesetzt werden könnte, um einen Computer zu simulieren, der mit einem beliebigen Programm geladen wurde. Eine einzelne TM kann also als »gespeichertes Computerprogramm« verwendet werden, wobei sie ihr Programm wie auch ihre Daten von einem oder mehreren Bändern erhält, auf denen sich die Eingabe befindet. In diesem Abschnitt werden wir diese Idee wieder aufgreifen und formalisieren, wie es bei der Diskussion über die Turing-Maschine als unsere Repräsentation eines gespeicherten Computerprogramms notwendig ist.

Wir definieren L_u, die *universelle Sprache*, als die Menge der Binärzeichenreihen, die in der Notation von Abschnitt 9.1.2 ein Paar (M, w) kodieren, wobei M eine TM mit dem binären Eingabealphabet ist und w eine Zeichenreihe aus $(\mathbf{0 + 1})^*$, sodass w in $L(M)$ enthalten ist. L_u ist also die Menge der Zeichenreihen, die eine TM und eine Eingabe repräsentieren, wobei diese Eingabe von der TM akzeptiert wird. Wir werden zeigen, dass eine TM U, häufig die *universelle Turing-Maschine* genannt, existiert, sodass $L_u = L(U)$. Da es sich bei der Eingabe von U um eine Binärzeichenreihe handelt, ist U tatsächlich eine TM M_j aus der Liste der Turing-Maschinen mit binären Eingaben, die wir in Abschnitt 9.1.2 entwickelt haben.

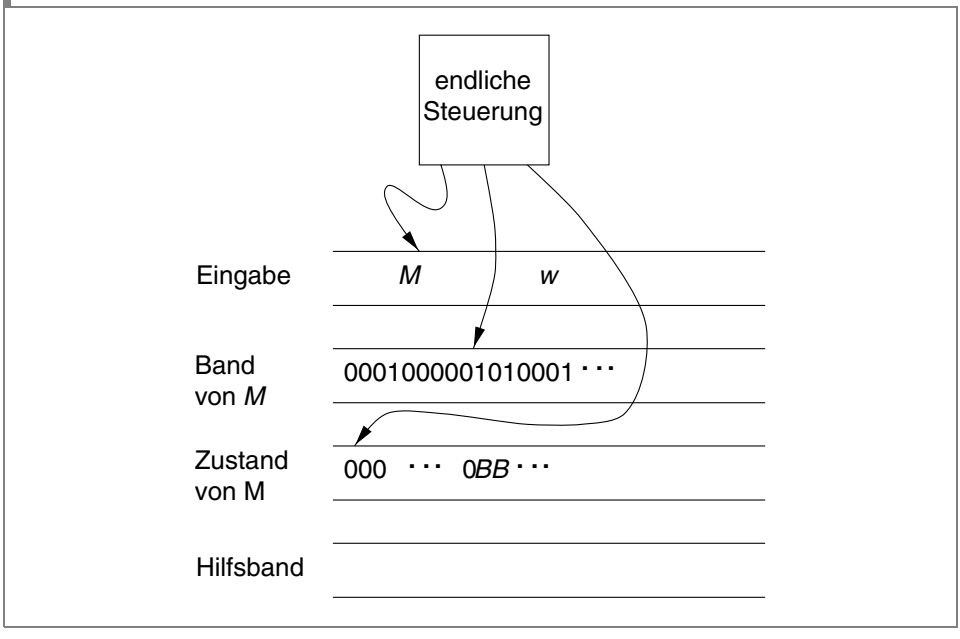
U ist am einfachsten als mehrbändige Turing-Maschine im Sinn von Abbildung 8.20 zu beschreiben. In diesem Fall werden die Übergänge von U zusammen mit der Zeichenreihe w anfänglich auf dem ersten Band gespeichert. Auf einem zweiten Band wird das simulierte Band von M gespeichert, wobei das gleiche Format wie das des Codes von M verwendet wird. Das heißt das Bandsymbol X_i von M wird durch 0^i repräsentiert, und eine einzelne 1 dient als Trennzeichen der Bandsymbole. Das dritte Band von U speichert den Zustand von M, wobei Zustand q_i von i Nullen repräsentiert wird. Abbildung 9.5 zeigt eine Skizze für U.

Die Arbeitsweise von U kann folgendermaßen zusammengefasst werden:

1. Untersuche die Eingabe, um sicherzustellen, dass der Code für M ein gültiger Code für eine TM ist. Ist das nicht der Fall, hält U an ohne zu akzeptieren. Da von ungültigen Codes angenommen wird, dass sie die TM ohne Bewegungen repräsentieren, und eine solche TM keine Eingaben akzeptiert, ist diese Aktion korrekt.

2. Initialisiere das zweite Band, um die Eingabe w in kodierter Form zu speichern. Das heißt, speichere für jede 0 von w eine 10 auf dem zweiten Band, für jede 1 dagegen 100. Beachten Sie, dass die Leerzeichen auf dem simulierten Band von M, die von 1000 repräsentiert werden, nicht auf diesem Band erscheinen; alle Zellen außerhalb derjenigen, die für w verwendet werden, enthalten das Leerzeichen von U. Allerdings weiß U das; sucht U also nach einem simulierten Symbol von M und trifft dabei das eigene Leerzeichen, dann ist dieses Leerzeichen durch die Folge 1000 zu ersetzen, um das Leerzeichen von M zu simulieren.

Abbildung 9.5: Organisation einer universellen Turing-Maschine



3. Speichere 0, den Startzustand von M, auf dem dritten Band, und bewege den Kopf von Us zweitem Band zur ersten simulierten Zelle.

4. Um eine Bewegung von M zu simulieren, sucht U auf dem ersten Band nach einem Übergang $0^i10^j10^k10^l10^m$, sodass 0^i der Zustand auf Band 3 und 0^j das Bandsymbol von M ist, das an der Position auf Band 2 beginnt, die U liest. Dies ist der nächste Übergang von M. U sollte Folgendes ausführen:

   a) Den Inhalt von Band 3 in 0^k ändern, also die Zustandsänderung von M simulieren. Dazu ändert U zuerst alle Nullen auf Band 3 in Leerzeichen und kopiert dann 0^k von Band 1 auf Band 3.

   b) 0^j auf Band 2 durch 0^l ersetzen, also das Bandsymbol von M ändern. Ist mehr oder weniger Raum nötig (wenn $i \neq l$), werden das Hilfsband und die in Abschnitt 8.6.2 beschriebene Verschiebetechnik verwendet, um die Anpassung durchzuführen.

   c) Den Kopf von Band 2 auf die Position der nächsten 1 nach links bzw. rechts zu bewegen, abhängig davon, ob $m = 1$ (Bewegung nach links) oder $m = 2$ (Bewegung nach rechts) ist. U simuliert also die Bewegung von M nach links oder rechts.

5. Besitzt M keinen Übergang, der zu simuliertem Zustand und Bandsymbol gehört, dann wird in Schritt (4) kein Übergang gefunden. M hält daher in der simulierten Konfiguration an, und auch U muss anhalten.

6. Geht M in den akzeptierenden Zustand über, dann akzeptiert auch U.

Eine effizientere universelle TM

Bei einer effizienten Simulation von M durch U, die kein Verschieben von Symbolen auf dem Band erfordert, würde U zuerst die Anzahl der von M verwendeten Bandsymbole bestimmen. Sind es zwischen $2^{k-1} + 1$ und 2^k Symbole, dann kann U einen k-Bit-Binärcode verwenden, um die verschiedenen Bandsymbole eindeutig zu repräsentieren. Die Bandzellen von M könnten von k Bandzellen von U simuliert werden. Zur weiteren Vereinfachung könnte U die gegebenen Übergänge von M umschreiben und einen Binärcode fester Länge statt des hier vorgestellten Unärcodes variabler Länge einsetzen.

Auf diese Weise simuliert U M mit Eingabe w. U akzeptiert das kodierte Paar (M, w) ausschließlich dann, wenn M w akzeptiert.

9.2.4 Unentscheidbarkeit der universellen Sprache

Wir stellen nun ein Problem vor, das rekursiv aufzählbar, nicht aber rekursiv ist, die Sprache L_u. Das Wissen, dass L_u unentscheidbar ist (d. h. L_u ist keine rekursive Sprache), ist in vieler Hinsicht wertvoller als unsere vorherige Entdeckung, dass L_d nicht rekursiv aufzählbar ist. Der Grund liegt darin, dass die Reduktion von L_u auf ein anderes Problem P bei dem Beweis eingesetzt werden kann, dass es keinen Algorithmus gibt, um P zu lösen, unabhängig davon, ob P rekursiv aufzählbar ist oder nicht. Dagegen ist die Reduktion von L_d auf P nur möglich, wenn P nicht rekursiv aufzählbar ist, und daher kann anhand von L_d die Unentscheidbarkeit solcher Probleme nicht bewiesen werden, die zwar rekursiv aufzählbar, nicht aber rekursiv sind. Wollen wir allerdings ein Problem als nicht rekursiv aufzählbar beweisen, dann kann nur L_d verwendet werden; L_u ist dafür nutzlos, da es rekursiv aufzählbar *ist*.

Das Halteproblem

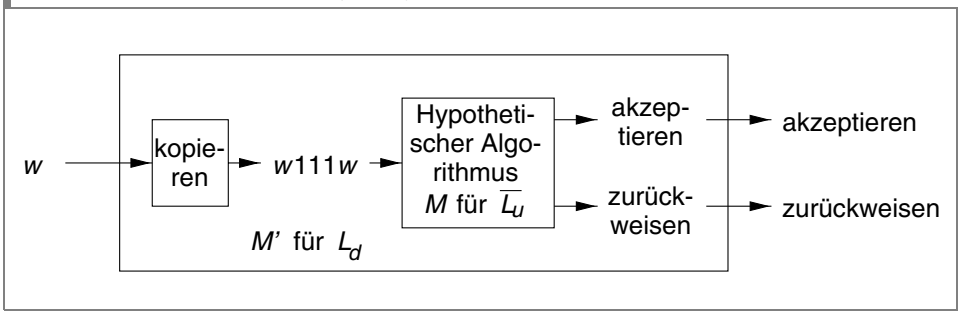
Das *Halteproblem* von Turing-Maschinen wird oft als L_u ähnlich bezeichnet – das Problem ist rekursiv aufzählbar, jedoch nicht rekursiv. Tatsächlich akzeptierte die ursprüngliche Turing-Maschine von A. M. Turing durch Anhalten und nicht durch den Übergang in einen Endzustand. Wir könnten $H(M)$ für die TM M als die Menge der Eingaben w definieren, sodass M bei gegebener Eingabe w anhält, unabhängig davon, ob M w akzeptiert. Dann bestünde das Halteproblem aus der Menge von Paaren (M, w), sodass w in $H(M)$ enthalten ist. Dieses Problem (diese Sprache) ist ein weiteres Beispiel für ein Problem, das zwar rekursiv aufzählbar, aber nicht rekursiv ist.

Satz 9.6 L_u ist rekursiv aufzählbar, jedoch nicht rekursiv.

Beweis: Wir haben in Abschnitt 9.2.3 bewiesen, dass L_u rekursiv aufzählbar ist. Angenommen, L_u wäre rekursiv; dann wäre nach Satz 9.3 $\overline{L}_u$, das Komplement von L_u, ebenfalls rekursiv. Haben wir allerdings eine stets haltende TM M, die $\overline{L}_u$ akzeptiert, dann können wir eine TM konstruieren, die L_d akzeptiert (die Methode wird im Folgenden erläutert). Da wir schon wissen, dass L_d nicht rekursiv aufzählbar ist, ist dies ein Widerspruch zu unserer Annahme, dass L_u rekursiv sei.

Angenommen, $L(M) = \overline{L}_u$. Wie in Abbildung 9.6 dargestellt, ändern wir TM M in eine TM M', die L_d akzeptiert.

Abbildung 9.6: Reduktion von L_d auf $\overline{L}_u$



1. Bei einer gegebenen Zeichenreihe w als Eingabe ändert M' die Eingabe in $w111w$. Sie können als Übung ein TM-Programm schreiben, das diesen Schritt auf einem einzigen Band ausführt. Allerdings ist der Beweis, dass dies möglich ist, einfacher, wenn ein weiteres Band eingesetzt wird, um w zu kopieren. Danach wird die zweibändige TM in eine einbändige TM konvertiert.
2. M' simuliert M mit der neuen Eingabe. Ist w nach unserer Nummerierung w_i, dann stellt M' fest, ob M_i w_i akzeptiert oder nicht, und akzeptiert w_i genau dann, wenn M_i w_i nicht akzeptiert; w_i ist also in L_d enthalten.

Daher akzeptiert M' genau dann, wenn w in L_d enthalten ist. Da wir wissen, dass M' nach Satz 9.2 nicht existieren kann, schließen wir, dass L_u nicht rekursiv ist. ■

9.2.5 Übungen zum Abschnitt 9.2

Übung 9.2.1 Zeigen Sie, dass das Halteproblem – also die Menge der Paare (M, w), sodass M bei Eingabe w (mit oder ohne Akzeptieren) anhält – rekursiv aufzählbar, jedoch nicht rekursiv ist (siehe Kasten »Das Halteproblem« in Abschnitt 9.2.4).

Übung 9.2.1 Im Kasten »Warum rekursiv?« in Abschnitt 9.2.1 wurde darauf hingewiesen, dass es einen Begriff »rekursive Funktion« gibt, der mit der Turing-Maschine als ein Modell für das konkurriert, was berechnet werden kann. In dieser Übung untersuchen wir ein Beispiel einer rekursiven Funktion. Eine *rekursive Funktion* ist eine Funktion F, die von einer endlichen Menge von Regeln definiert wird. Jede Regel spezifiziert den Wert der Funktion F für bestimmte Argumente; die Spezifikation kann Variablen, nicht negative ganzzahlige Konstanten, die Funktion »Nachfolger« (addiere 1), die Funktion F selbst sowie Ausdrücke verwenden, die durch Funktionskomposition aus diesen Komponenten erstellt werden. Beispielsweise wird die *Ackermann-Funktion* von folgenden Regeln definiert:

1. $A(0, y) = 1$ für $y \geq 0$
2. $A(1, 0) = 2$
3. $A(x, 0) = x + 2$ für $x \geq 2$
4. $A(x + 1, y + 1) = A(A(x, y + 1), y)$ für $x \geq 0$ und $y \geq 0$

Lösen Sie die folgenden Aufgaben:

* a) Berechnen Sie $A(2, 1)$.

! b) Welche Funktion von x ist $A(x, 2)$?

! c) Berechnen Sie $A(4, 3)$.

Übung 9.2.3 Beschreiben Sie informell mehrbändige Turing-Maschinen, die die folgenden Mengen ganzer Zahlen derart *aufzählen* , dass sie mit leeren Bändern beginnen und dann auf einem der Bänder $10^{i_1}10^{i_2}1\ldots$ schreiben, um die Menge $\{i_1, i_2,\ldots\}$ zu repräsentieren.

* a) Die Menge aller Quadratzahlen $\{1, 4, 9,\ldots\}$

b) Die Menge aller Primzahlen $\{2, 3, 5, 7, 11,\ldots\}$

! c) Die Menge aller i, sodass M_i w_i akzeptiert. *Hinweis:* Es ist nicht möglich, all diese i in numerischer Reihenfolge zu generieren. Der Grund liegt darin, dass diese Sprache, d. h. $\overline{L}_d$, rekursiv aufzählbar, aber nicht rekursiv ist. Tatsächlich lautet eine Definition der Sprachen, die zwar rekursiv aufzählbar, aber nicht rekursiv sind, dass sie aufgezählt werden können, nicht jedoch in numerischer Reihenfolge. Der »Trick«, sie überhaupt aufzuzählen, liegt darin, dass wir alle M_i mit w_i simulieren müssen, doch wir können keiner M_i erlauben, endlos zu laufen, da dies verhindern würde, eine weitere M_j für $j \neq i$ zu prüfen, sobald wir eine M_i gefunden haben, die bei w_i nicht anhält. Wir müssen daher in Runden arbeiten, wobei wir in der k-ten Runde nur eine begrenzte Menge von M_i prüfen, denen jeweils eine begrenzte Anzahl von Schritten erlaubt ist. Jede Runde wird daher in endlicher Zeit ausgeführt. Solange für jede TM M_i und für jede Anzahl von Schritten s eine Runde existiert, sodass M_i mit mindestens s Schritten simuliert wird, werden wir schließlich jede M_i finden, die w_i akzeptiert, und i aufzählen.

* **Übung 9.2.4** Sei $L_1, L_2,\ldots, L_k$ eine Gruppe von Sprachen über dem Alphabet Σ, sodass Folgendes gilt:

1. Für alle $i \neq j$ ist $L_i \cap L_j = \emptyset$; d. h. keine Zeichenreihe ist in zweien der Sprachen enthalten.
2. $L_1 \cup L_2 \cup\ldots\cup L_k = \Sigma^*$; d. h. jede Zeichenreihe ist in einer der Sprachen enthalten.
3. Jede der Sprachen L_i für $i = 1, 2,\ldots, k$ ist rekursiv aufzählbar.

Beweisen Sie, dass jede der Sprachen daher rekursiv ist.

*! **Übung 9.2.5** Sei L rekursiv aufzählbar und $\overline{L}$ nicht rekursiv aufzählbar. Sei L' die folgende Sprache:

$$L' = \{0w \mid w \text{ ist in } L \text{ enthalten}\} \cup \{1w \mid w \text{ ist nicht in } L \text{ enthalten}\}$$

Können Sie mit Sicherheit sagen, ob L' oder deren Komplement rekursiv, rekursiv aufzählbar oder nicht rekursiv aufzählbar ist? Begründen Sie Ihre Antwort.

! **Übung 9.2.6** Bis auf die Diskussion des Komplements in Abschnitt 9.2.2 haben wir uns nicht mit Abgeschlossenheitseigenschaften der rekursiven oder der rekursiv aufzählbaren Sprachen befasst. Untersuchen Sie, ob die rekursiven Sprachen und/oder die rekursiv aufzählbaren Sprachen unter den folgenden Operationen abgeschlossen sind. Zeigen Sie die Abgeschlossenheit mit informellen, aber klaren Konstruktionen.

* a) Vereinigung

b) Durchschnitt

c) Verkettung

d) Kleenesche Hülle (Stern)

* e) Homomorphismus

f) Inverser Homomorphismus

9.3 Turing-Maschinen und unentscheidbare Probleme

Wir werden nun die Sprachen L_u und L_d verwenden, deren Status bezüglich Entscheidbarkeit und rekursiver Aufzählbarkeit wir kennen, um andere Sprachen als unentscheidbar oder nicht rekursiv aufzählbar aufzuzeigen. In jedem dieser Beweise wird die Reduktionstechnik eingesetzt. Die ersten unentscheidbaren Probleme betreffen durchweg Turing-Maschinen. Die Diskussion in diesem Abschnitt findet ihren Höhepunkt im Beweis des »Satzes von Rice«, der aussagt, dass jede nicht triviale Eigenschaft von Turing-Maschinen, die nur von der Sprache abhängt, die die TM akzeptiert, unentscheidbar sein muss. In Abschnitt 9.4 untersuchen wir einige unentscheidbare Probleme, die weder Turing-Maschinen noch deren Sprachen involvieren.

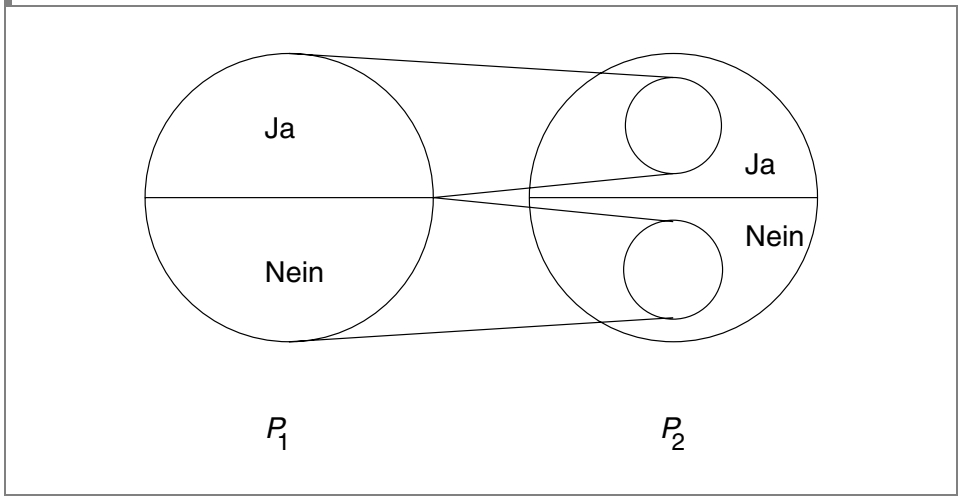
9.3.1 Reduktionen

Wir haben den Begriff der Reduktion in Abschnitt 8.1.3 vorgestellt. Allgemein ausgedrückt: Wenn wir einen Algorithmus haben, um Instanzen eines Problems P_1 in Instanzen eines Problems P_2 zu überführen, die die gleiche Antwort besitzen, dann wird P_1 *auf* P_2 *reduziert*. Wir können mit diesem Beweis zeigen, dass P_2 mindestens so hart wie P_1 ist. Ist daher P_1 nicht rekursiv, dann kann P_2 nicht rekursiv sein. Ist P_1 nicht rekursiv aufzählbar, dann kann P_2 nicht rekursiv aufzählbar sein. Wie in Abschnitt 8.1.3 angemerkt, müssen Sie darauf achten, ein als hart bekanntes Problem auf ein Problem zu reduzieren, das Sie als mindestens ebenso hart beweisen wollen; Sie dürfen niemals umgekehrt vorgehen.

Wie Abbildung 9.7 zeigt, muss eine Reduktion jede Instanz von P_1, die die Antwort »ja« besitzt, in eine Instanz von P_2 mit der Antwort »ja« überführen; ebenso werden Instanzen von P_1 mit der Antwort »nein« in Instanzen von P_2 mit der Antwort »nein« überführt. Beachten Sie, dass es nicht wesentlich ist, dass jede Instanz von P_2 ein Ziel einer oder mehrerer Instanzen von P_1 ist. Häufig ist nur ein Teil von P_2 Ziel der Reduktion.

Formal ist eine Reduktion von P_1 auf P_2 eine Turing-Maschine, die mit einer Instanz von P_1 auf dem Band beginnt und mit einer Instanz von P_2 auf dem Band anhält. In der Praxis werden wir Reduktionen von P_1 auf P_2 im Allgemeinen beschreiben, als seien sie Computerprogramme, die eine Instanz von P_1 als Eingabe erhalten und eine Instanz von P_2 als Ausgabe produzieren. Die Äquivalenz zwischen Turing-

Abbildung 9.7: Reduktionen überführen positive in positive Instanzen und negative in negative Instanzen



Maschinen und Computerprogrammen ermöglicht uns, die Reduktion auf beide Arten zu beschreiben. Die Bedeutung von Reduktionen hebt der folgende Satz hervor, von dem wir zahlreiche Anwendungen sehen werden.

Satz 9.7 Existiert eine Reduktion von P_1 auf P_2, dann gilt:

a) Ist P_1 unentscheidbar, dann ist P_2 ebenfalls unentscheidbar.

b) Ist P_1 nicht rekursiv aufzählbar, dann ist P_2 ebenfalls nicht rekursiv aufzählbar.

Beweis: Zuerst nehmen wir an, P_1 sei unentscheidbar. Wenn es möglich wäre, P_2 zu entscheiden, dann könnten wir die Reduktion von P_1 auf P_2 mit dem Algorithmus kombinieren, der P_2 entscheidet, um einen Algorithmus zu konstruieren, der P_1 entscheidet. Abbildung 9.7 zeigt diese Idee. Genauer ausgedrückt, wir nehmen an, wir haben eine Instanz w von P_1. Wir wenden auf w den Algorithmus an, der w in eine Instanz x von P_2 konvertiert. Danach wird der Algorithmus angewendet, der P_2 bezüglich x entscheidet. Antwortet dieser Algorithmus mit »ja«, dann ist x in P_2 enthalten. Da wir P_1 auf P_2 reduziert haben, wissen wir, dass die Antwort darauf, ob w in P_1 enthalten ist, »ja« lautet. Ist entsprechend x nicht in P_2 enthalten, dann ist auch w nicht in P_1 enthalten. Wie auch immer die Frage »Ist x in P_2 enthalten?« beantwortet wird, diese Antwort gilt auch für »Ist w in P_1 enthalten?«.

Wir haben also einen Widerspruch zur Annahme, P_1 sei unentscheidbar. Die Schlussfolgerung lautet, dass P_2 ebenfalls unentscheidbar ist, wenn P_1 unentscheidbar ist.

Nun sehen wir uns Teil (b) an. Angenommen, P_1 sei nicht rekursiv aufzählbar, doch P_2 sei rekursiv aufzählbar. Wir haben also einen Algorithmus, um P_1 auf P_2 zu reduzieren, und eine Methode, um P_2 zu erkennen; es gibt also eine TM, die mit »ja« antwortet, wenn deren Eingabe in P_2 enthalten ist, eventuell jedoch nicht hält, wenn deren Eingabe nicht in P_2 enthalten ist. Wie bei Teil (a) beginnen wir mit einer Instanz w von P_1 und konvertieren sie mit dem Reduktionsalgorithmus in eine Instanz x von

P_2. Dann wenden wir die TM für P_2 auf x an. Wird x akzeptiert, dann wird auch w akzeptiert.

Diese Methode beschreibt eine TM (die eventuell nicht anhält), deren Sprache P_1 ist. Ist w in P_1 enthalten, dann ist auch x in P_2 enthalten, und diese TM wird w akzeptieren. Ist w nicht in P_1 enthalten, dann ist x nicht in P_2 enthalten. Wir wissen dann nicht, ob die TM anhält, aber sie wird w sicherlich nicht akzeptieren. Da wir angenommen haben, dass keine TM für P_1 existiert, haben wir durch einen Widerspruch gezeigt, dass auch für P_2 keine TM existiert; ist P_1 also nicht rekursiv aufzählbar, dann ist auch P_2 nicht rekursiv aufzählbar. ■

9.3.2 Turing-Maschinen, die die leere Sprache akzeptieren

Als Beispiel für Reduktionen, die Turing-Maschinen involvieren, untersuchen wir zwei Sprachen, L_e und L_{ne}. Jede besteht aus Binärzeichenreihen. Ist w eine Binärzeichenreihe, dann repräsentiert sie eine TM M_i (in der Aufzählung aus Abschnitt 9.1.2).

Ist $L(M_i) = \emptyset$, akzeptiert M_i also keine Eingabe, dann ist w in L_e enthalten. L_e ist daher die Sprache, die aus allen kodierten TMs besteht, deren Sprache leer ist. Ist $L(M_i)$ allerdings nicht die leere Sprache, dann ist w in L_{ne} enthalten. L_{ne} ist daher die Sprache aller Codes von Turing-Maschinen, die mindestens eine Eingabezeichenreihe akzeptieren.

Im Folgenden sollen Zeichenreihen als die Turing-Maschinen angesehen werden, die sie repräsentieren. Die oben genannten Sprachen können somit wie folgt definiert werden:

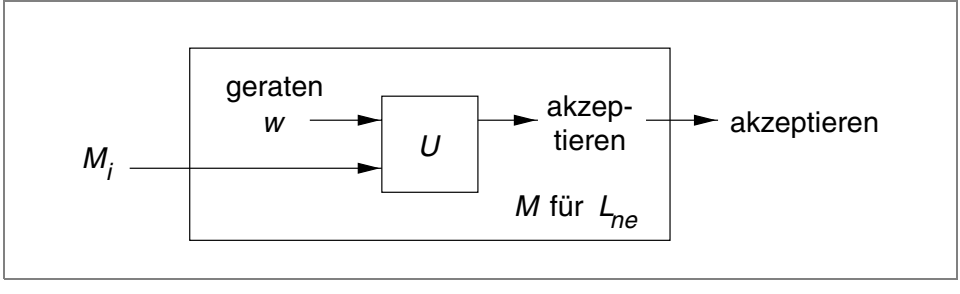
- $L_e = \{M \mid L(M) = \emptyset\}$
- $L_e = \{M \mid L(M) \neq \emptyset\}$

Beachten Sie, dass sowohl L_e als auch L_{ne} Sprachen über dem Binäralphabet {0, 1} und jeweils das Komplement der anderen sind. Wir werden sehen, dass L_{ne} die »einfachere« der beiden Sprachen ist; sie ist rekursiv aufzählbar, jedoch nicht rekursiv. L_e dagegen ist nicht rekursiv aufzählbar.

Satz 9.8 L_{ne} ist rekursiv aufzählbar.

Beweis: Wir müssen lediglich zeigen, dass eine TM L_{ne} akzeptiert. Es ist am einfachsten, eine nichtdeterministische TM M zu beschreiben, deren Entwurf Abbildung 9.8 zeigt. Nach Satz 8.11 kann M in eine deterministische TM überführt werden.

Abbildung 9.8: Konstruktion einer NTM, die L_{ne} akzeptiert



M arbeitet wie folgt:

1. M übernimmt einen TM-Code M_i als Eingabe.
2. Mithilfe der nichtdeterministischen Fähigkeiten rät M zu einer Eingabe w, die M_i akzeptieren könnte.
3. M testet, ob M_i w akzeptiert. Dazu kann M die universelle TM U simulieren, die L_u akzeptiert.
4. Akzeptiert M_i w, dann akzeptiert M die Eingabe M_i.

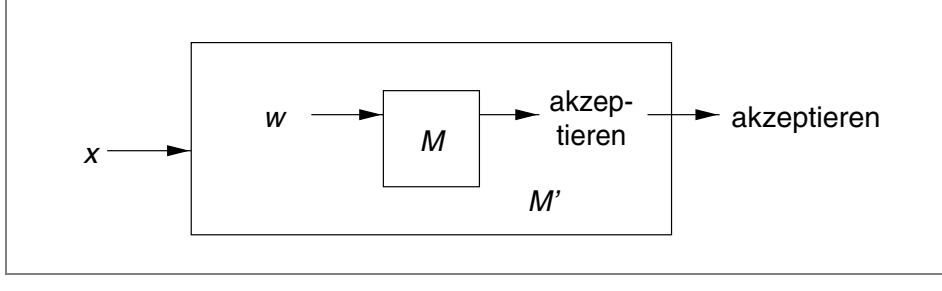
Wenn M_i also auch nur eine Zeichenreihe akzeptiert, wird M zu dieser Zeichenreihe (unter allen anderen) raten und M_i akzeptieren. Ist allerdings $L(M_i) = \emptyset$, dann führt kein Raten von w dazu, dass M_i akzeptiert, und daher akzeptiert M M_i nicht. Daher gilt $L(M) = L_{ne}$. ■

Im nächsten Schritt beweisen wir, dass L_{ne} nicht rekursiv ist. Dazu reduzieren wir L_{ne} auf L_u. Wir werden einen Algorithmus beschreiben, der eine Eingabe (M, w) in eine Ausgabe M' überführt, also in den Code einer anderen Turing-Maschine, sodass w wirklich nur in dem Fall in $L(M)$ enthalten ist, wenn $L(M')$ nicht leer ist. M akzeptiert w also nur dann, wenn M' mindestens eine Zeichenreihe akzeptiert. Der Clou besteht darin, dass M' ihre Eingabe ignoriert und stattdessen M mit der Eingabe w simuliert. Akzeptiert M, dann akzeptiert M' ihre eigene Eingabe; die Akzeptanz von w durch M ist daher gleichbedeutend damit, dass $L(M')$ nicht leer ist. Wäre L_{ne} rekursiv, dann hätten wir einen Algorithmus, der uns mitteilen würde, ob M w akzeptiert oder nicht: Konstruiere M' und entscheide, ob $L(M') \neq \emptyset$ ist oder nicht.

Satz 9.9 L_{ne} ist nicht rekursiv.

Beweis: Wir folgen der oben stehenden Skizze des Beweises. Wir müssen einen Algorithmus entwerfen, der eine aus einem binär kodierten Paar (M, w) bestehende Eingabe in eine TM M' konvertiert, sodass $L(M') \neq \emptyset$ nur dann, wenn M die Eingabe w akzeptiert. Abbildung 9.9 zeigt die Konstruktion von M'. Wir werden sehen, dass M' keine Eingabe akzeptiert, wenn M w nicht akzeptiert; dann ist also $L(M') = \emptyset$. Falls M allerdings w akzeptiert, dann akzeptiert M' jede Eingabe, und $L(M')$ ist in diesem Fall sicher nicht $\emptyset$.

Abbildung 9.9: Entwurf der TM M', die in Satz 9.9 aus (M, w) konstruiert wird; M' akzeptiert beliebige Eingaben genau dann, wenn M w akzeptiert.



M' führt Folgendes aus:

1. M' ignoriert die eigene Eingabe x und ersetzt sie durch die Zeichenreihe, welche die TM M und die Eingabezeichenreihe w repräsentiert. Da M' für ein bestimmtes Paar (M, w), dessen Länge n sei, entworfen wurde, können wir M' so konstruieren, dass sie eine Folge von Zuständen $q_0, q_1, \ldots, q_n$ besitzt, wobei q_0 der Startzustand ist.

   a) In Zustand q_i für $i = 0, 1, \ldots, n - 1$ schreibt M' das $(i + 1)$-te Bit des Codes von (M, w), wechselt in Zustand q_{i+1} und führt eine Bewegung nach rechts aus.

   b) In Zustand q_n geht M', falls notwendig, nach rechts und ersetzt alle nicht leeren Zeichen (dies wäre der letzte Teil von x, falls diese Eingabe von M' länger als n ist) durch Leerzeichen.

2. Erreicht M' im Zustand q_n ein Leerzeichen, dann verwendet M' eine ähnliche Sammlung von Zuständen, um den Kopf wieder am linken Ende des Bandes zu positionieren.

3. Unter Verwendung weiterer Zustände simuliert M' auf ihrem Band eine universelle TM U.

4. Akzeptiert U, dann akzeptiert auch M'. Akzeptiert U dagegen niemals, dann akzeptiert auch M' nie.

Die obige Beschreibung von M' sollte ausreichen, um Sie von Folgendem zu überzeugen. Sie können eine Turing-Maschine entwerfen, die den Code von M und der Zeichenreihe w in den Code von M' umwandelt. Es existiert also ein Algorithmus, der die Reduktion von L_{ne} auf L_u ausführt. Außerdem ist offensichtlich, dass M' jede Eingabe x akzeptiert, die sich ursprünglich auf ihrem Band befand, wenn M w akzeptiert. Dabei ist irrelevant, dass x ignoriert wurde, weil die Definition der Akzeptanz durch eine TM besagt, dass das, was sich zu Beginn auf dem Band befand, von der TM akzeptiert wird. Wenn also M w akzeptiert, dann ist der Code von M' in L_{ne} enthalten.

Wenn M umgekehrt w nicht akzeptiert, dann akzeptiert M' unabhängig von der Eingabe nie. In diesem Fall ist der Code von M' nicht in L_{ne} enthalten. Wir haben also mit dem Algorithmus, der M' aus M und w konstruiert, L_{ne} erfolgreich auf L_u reduziert; wir können daraus schließen, dass mit L_u auch L_{ne} nicht rekursiv ist. Die Existenz dieser Reduktion ist ausreichend, um den Beweis abzuschließen. Wir wollen jedoch den Einfluss der Reduktion besonders verdeutlichen und gehen noch einen Schritt weiter. Wäre L_{ne} rekursiv, dann könnten wir für L_u den folgenden Algorithmus entwickeln:

1. Konvertiere (M, w) wie oben in die TM M'.

2. Verwende den hypothetischen Algorithmus für L_{ne}, um festzustellen, ob $L(M') = \varnothing$ ist oder nicht; wenn ja, akzeptiert M w nicht; wenn nein, dann akzeptiert M w.

Da wir aus Satz 9.6 wissen, dass ein solcher Algorithmus für L_u nicht existiert, haben wir einen Widerspruch zur Annahme, L_{ne} sei rekursiv, und schließen, dass L_{ne} nicht rekursiv ist. ■

Wir kennen nun auch den Status von L_e. Wäre L_e rekursiv aufzählbar, dann wäre nach Satz 9.4 sowohl L_e selbst als auch L_{ne} rekursiv. Da L_{ne} nach Satz 9.9 nicht rekursiv ist, schließen wir daraus:

Satz 9.10 L_e ist nicht rekursiv aufzählbar. ■

9.3.3 Der Satz von Rice und Eigenschaften der rekursiv aufzählbaren Sprachen

Die Tatsache, dass Sprachen wie L_e und L_{ne} unentscheidbar sind, ist eigentlich ein Spezialfall eines weit allgemeineren Satzes: Alle nicht trivialen Eigenschaften der rekursiv aufzählbaren Sprachen sind in dem Sinne unentscheidbar, als eine Turing-Maschine solche Binärzeichenreihen nicht erkennen kann, die Codes einer TM sind, deren Sprache die Eigenschaft besitzt. Ein Beispiel für eine Eigenschaft der rekursiv aufzählbaren Sprachen ist »Die Sprache ist kontextfrei«. Es ist unentscheidbar, ob eine gegebene TM eine kontextfreie Sprache akzeptiert, und dies ist ein Spezialfall des allgemeinen Prinzips, dass alle nicht trivialen Eigenschaften der rekursiv aufzählbaren Sprachen unentscheidbar sind.

Bei einer *Eigenschaft* der rekursiv aufzählbaren Sprachen handelt es sich einfach um eine Menge rekursiv aufzählbarer Sprachen. Die Eigenschaft »kontextfrei« entspricht daher formal der Menge aller kontextfreien Sprachen. Die Eigenschaft »leer« entspricht der Menge $\{\emptyset\}$, die nur die leere Sprache enthält.

Warum sich Probleme und ihre Komplemente unterscheiden

Intuitiv halten wir ein Problem sowie dessen Komplement für das gleiche Problem. Um ein Problem zu lösen, können wir einen Algorithmus für dessen Komplement verwenden und kehren dann im letzten Schritt die Ausgabe um: »ja« statt »nein« und umgekehrt. Diese Intuition ist richtig, solange das Problem und dessen Komplement rekursiv sind.

Wie in Abschnitt 9.2.2 beschrieben, gibt es allerdings noch zwei weitere Möglichkeiten. Zunächst kann weder das Problem noch dessen Komplement auch nur rekursiv aufzählbar sein. Dann sind beide von keinem Typ einer Turing-Maschine zu lösen, und damit gleichen sich die beiden Probleme wieder in einem gewissen Sinn. Der interessante Fall, verkörpert von L_e und L_{ne}, liegt vor, wenn ein Problem rekursiv aufzählbar und sein Komplement nicht rekursiv aufzählbar ist.

Für eine rekursiv aufzählbare Sprache können wir eine TM entwerfen, die eine Eingabe w übernimmt und nach einem Grund sucht, warum w gegebenenfalls in der Sprache enthalten ist. Für L_{ne} mit einer TM M als Eingabe sucht unsere TM daher nach Zeichenreihen, die von M akzeptiert werden. Sobald wir eine solche Zeichenreihe finden, akzeptieren wir M. Handelt es sich bei M um eine TM mit einer leeren Sprache, dann wissen wir nie mit Bestimmtheit, dass M nicht in L_{ne} enthalten ist, aber wir akzeptieren M auch niemals, und das ist die korrekte Antwort der TM.

Andererseits gibt es für das nicht rekursiv aufzählbare Komplement L_e keine Möglichkeit, jemals alle Zeichenreihen aus L_e zu akzeptieren. Angenommen, wir haben eine Zeichenreihe M für eine TM, deren Sprache leer ist. Wir können Eingaben der TM M prüfen und werden dabei nie eine finden, die M akzeptiert, doch wir wären im Allgemeinen auch nie sicher, dass nicht doch eine Eingabe existiert, die M akzeptieren würde. Daher kann M nie akzeptiert werden, obwohl sie akzeptiert werden sollte.

Eine Eigenschaft ist *trivial*, wenn sie entweder leer ist (d. h. sie kommt keiner Sprache zu) oder aus allen rekursiv aufzählbaren Sprachen besteht. Anderenfalls ist sie *nicht trivial*.

- Beachten Sie, dass die leere Eigenschaft, $\emptyset$, und die Eigenschaft, eine leere Sprache zu sein, $\{\emptyset\}$, verschieden sind.

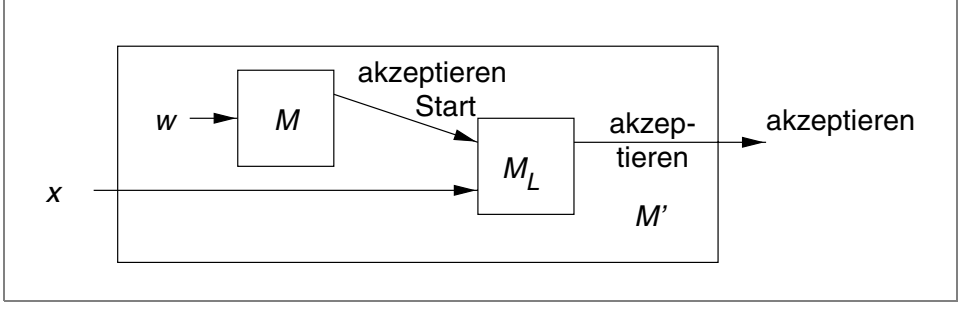
Wir können eine Menge von Sprachen nicht als die Sprachen selbst erkennen. Das liegt daran, dass eine typische Sprache, die unendlich ist, nicht als eine Zeichenreihe endlicher Länge geschrieben werden kann, die Eingabe einer TM sein könnte. Stattdessen müssen wir die Turing-Maschinen erkennen, die diese Sprachen akzeptieren; der TM-Code selbst ist endlich, auch wenn die akzeptierte Sprache unendlich ist. Ist $\mathcal{P}$ daher eine Eigenschaft der rekursiv aufzählbaren Sprachen, dann besteht die Sprache $L_{\mathcal{P}}$ aus der Menge der Codes für Turing-Maschinen M_i, sodass $L(M_i)$ eine Sprache aus $\mathcal{P}$ ist. Wenn wir über die Entscheidbarkeit einer Eigenschaft P sprechen, meinen wir die Entscheidbarkeit der Sprache $L_{\mathcal{P}}$.

Satz 9.11 (Der Satz von Rice) Jede nicht triviale Eigenschaft der rekursiv aufzählbaren Sprachen ist unentscheidbar.

Beweis: Sei $\mathcal{P}$ eine nicht triviale Eigenschaft der rekursiv aufzählbaren Sprachen. Zunächst nehmen wir an, die leere Sprache $\emptyset$ sei nicht in $\mathcal{P}$ enthalten; wir werden später den umgekehrten Fall betrachten. Da $\mathcal{P}$ nicht trivial ist, muss eine nicht leere Sprache L existieren, die in $\mathcal{P}$ enthalten ist. M_L sei eine TM, die L akzeptiert.

Wir werden L_u auf $L_{\mathcal{P}}$ reduzieren und damit beweisen, dass $L_{\mathcal{P}}$ unentscheidbar ist, da L_u unentscheidbar ist. Der Algorithmus für die Reduktion übernimmt ein Paar (M, w) als Eingabe und produziert eine TM M'. Abbildung 9.10 zeigt den Entwurf von M'; $L(M')$ ist $\emptyset$, wenn M w nicht akzeptiert, und $L(M') = L$, wenn M w akzeptiert.

Abbildung 9.10: Konstruktion von *M'* für den Beweis des Satzes von Rice



M' ist eine zweibändige TM. Ein Band wird verwendet, um M mit w zu simulieren. Sie wissen, dass der Reduktionsalgorithmus M und w als Eingabe erhält und diese zum Entwurf der Übergänge von M' verwenden kann. Daher ist die Simulation von M mit w in M' »eingebaut«; die TM M' muss also die Übergänge von M nicht auf einem eigenen Band lesen.

Das andere Band von M' dient, falls erforderlich, zur Simulation von M_L mit der Eingabe x von M'. Auch hier sind dem Reduktionsalgorithmus die Übergänge von M_L bekannt und können in die Übergänge von M' »eingebaut« werden. Die TM M' soll Folgendes ausführen:

1. M mit Eingabe w simulieren. Beachten Sie, dass w nicht die Eingabe von M' ist; stattdessen schreibt M' M und w auf eines der eigenen Bänder und simuliert wie im Beweis von Satz 9.8 mit diesem Paar die universelle TM U.
2. Akzeptiert M w nicht, dann führt M' weiter nichts aus. M' akzeptiert die eigene Eingabe x niemals, und damit ist $L(M') = \emptyset$. Da wir annehmen, dass $\emptyset$ nicht in der Eigenschaft $\mathcal{P}$ enthalten ist, ist der Code von M' nicht in $L_{\mathcal{P}}$ enthalten.
3. Akzeptiert M w, dann beginnt M' mit der Simulation von M_L mit der eigenen Eingabe x. M' wird daher exakt die Sprache L akzeptieren. Da L in $\mathcal{P}$ enthalten ist, ist der Code von M' in $L_{\mathcal{P}}$ enthalten.

Sie sollten beachten, dass ein Algorithmus die Konstruktion von M' aus M und w ausführen kann. Da dieser Algorithmus (M, w) in eine M' umwandelt, die nur dann in $L_{\mathcal{P}}$ enthalten ist, wenn (M, w) in L_u enthalten ist, ist dieser Algorithmus eine Reduktion von L_u auf $L_{\mathcal{P}}$ und beweist, dass die Eigenschaft $\mathcal{P}$ unentscheidbar ist.

Wir sind noch nicht ganz fertig, denn noch ist der Fall zu behandeln, dass $\emptyset$ in $\mathcal{P}$ enthalten *ist*. In diesem Fall sehen wir uns die Komplementeigenschaft $\overline{\mathcal{P}}$ an, die Menge der rekursiv aufzählbaren Sprachen, die die Eigenschaft $\mathcal{P}$ nicht besitzen. Nach den vorstehenden Aussagen ist $\overline{\mathcal{P}}$ unentscheidbar. Da allerdings jede TM eine rekursiv aufzählbare Sprache akzeptiert, stimmt die Menge $\overline{L}_{\mathcal{P}}$ der (Codes für) Turing-Maschinen, die eine Sprache aus $\mathcal{P}$ nicht akzeptieren, mit der Menge $L_{\overline{\mathcal{P}}}$ der TMs überein, die eine Sprache aus $\overline{\mathcal{P}}$ akzeptieren. Angenommen, $L_{\mathcal{P}}$ sei entscheidbar, dann wäre auch $L_{\overline{\mathcal{P}}}$ im Widerspruch zur obigen Aussage entscheidbar, da das Komplement einer rekursiven Sprache ebenfalls rekursiv ist (Satz 9.3). ■

9.3.4 Probleme bezüglich Spezifikationen von Turing-Maschinen

Alle Probleme bezüglich Turing-Maschinen, die nur die von der TM akzeptierte Sprache involvieren, sind nach Satz 9.11 unentscheidbar. Einige dieser Probleme sind sehr interessant. Beispielsweise sind die folgenden Probleme unentscheidbar:

1. Ist die von einer TM akzeptierte Sprache leer? (Dies wissen wir aus den Sätzen 9.9 und 9.3.)
2. Ist die von einer TM akzeptierte Sprache endlich?
3. Ist die von einer TM akzeptierte Sprache eine reguläre Sprache?
4. Ist die von einer TM akzeptierte Sprache eine kontextfreie Sprache?

Der Satz von Rice sagt allerdings nicht aus, dass alles im Bereich einer TM unentscheidbar ist. Beispielsweise können Fragen über die Zustände der TM durchaus entscheidbar sein.

Beispiel 9.12 Es ist entscheidbar, ob eine TM fünf Zustände besitzt. Der Algorithmus, der diese Frage entscheidet, durchsucht einfach den Code der TM und zählt die Anzahl der Zustände in den Übergängen.

Als weiteres Beispiel ist entscheidbar, ob eine Eingabe existiert, sodass die TM mindestens fünf Bewegungen ausführt. Der Algorithmus ist offensichtlich, wenn wir uns daran erinnern, dass eine TM lediglich die neun Bandzellen um die anfängliche Position des Kopfes prüft, wenn sie fünf Bewegungen ausführt. Wir können die fünf Bewegungen der TM daher auf einem beliebigen der endlichen Anzahl von Bändern

simulieren, die aus maximal fünf Symbolen bestehen, die links und rechts von Leerzeichen umgeben sind. Führt eine dieser Simulationen nicht zum Anhalten, dann können wir schließen, dass die TM mindestens fünf Bewegungen bei einer Eingabe ausführt. ■

9.3.5 Übungen zum Abschnitt 9.3

* **Übung 9.3.1** Zeigen Sie, dass die Menge der Turing-Maschinencodes von TMs, die alle aus Palindromen bestehenden Eingaben (möglicherweise zusammen mit anderen Eingaben) akzeptieren, unentscheidbar ist.

Übung 9.3.2 Big Computer Corp. möchte die sinkenden Marktanteile durch die Herstellung einer High-Tech-Version der Turing-Maschine, GPTM genannt, aufbessern. Diese neue TM ist mit *Glocken* und *Pfeifen* ausgestattet. Die GPTM entspricht grundlegend der gewöhnlichen Turing-Maschine, jedoch ist jeder Zustand der Maschine entweder als »Glockenzustand« oder »Pfeifenzustand« ausgezeichnet. Wann immer die GPTM in einen neuen Zustand wechselt, läutet sie entweder die Glocke oder bläst die Pfeife, je nach Typ des neuen Zustands. Beweisen Sie, dass unentscheidbar ist, ob eine gegebene GPTM M bei gegebener Eingabe w jemals pfeift.

Übung 9.3.3 Zeigen Sie, dass die Sprache der Codes von TMs M, die mit einem leeren Band beginnen und irgendwann eine 1 auf das Band schreiben, unentscheidbar ist.

! **Übung 9.3.4** Wir wissen aus dem Satz von Rice, dass keines der folgenden Probleme entscheidbar ist. Doch sind sie rekursiv aufzählbar oder nicht rekursiv aufzählbar?

a) Enthält $L(M)$ mindestens zwei Zeichenreihen?

b) Ist $L(M)$ unendlich?

c) Ist $L(M)$ eine kontextfreie Sprache?

d) Ist $L(M) = (L(M))^R$?

! **Übung 9.3.5** Sei L die Sprache, die aus Paaren von TM-Codes sowie einer ganzen Zahl, (M_1, M_2, k), besteht, sodass $L(M_1) \cap L(M_2)$ mindestens k Zeichenreihen enthält. Zeigen Sie, dass L rekursiv aufzählbar, nicht jedoch rekursiv ist.

Übung 9.3.6 Zeigen Sie, dass die folgenden Fragen entscheidbar sind:

* a) Die Menge der Codes der TMs M, die mit einem leeren Band beginnen und irgendwann ein nicht leeres Symbol auf ihr Band schreiben. *Hinweis:* Besitzt M m Zustände, betrachten Sie die ersten m Übergänge von M.

! b) Die Menge der Codes von TMs, die niemals eine Bewegung nach links ausführen.

! c) Die Menge der Paare (M, w), sodass eine TM M, die mit Eingabe w beginnt, eine Bandzelle niemals öfter als einmal liest.

! **Übung 9.3.7** Zeigen Sie, dass die folgenden Probleme nicht rekursiv aufzählbar sind:

! a) Die Menge der Paare (M, w), sodass TM M mit Eingabe w nicht anhält.

! b) Die Menge der Paare (M_1, M_2), sodass $L(M_1) \cap L(M_2) = \emptyset$.

! c) Die Menge der Tripel (M_1, M_2, M_3), sodass $L(M_1) = L(M_2)\ L(M_3)$; die Sprache der ersten TM ist also die Verkettung der Sprachen der anderen beiden TMs.

!! **Übung 9.3.8** Welche der folgenden Probleme sind rekursiv oder rekursiv aufzählbar, aber nicht rekursiv, oder nicht rekursiv aufzählbar?

* a) Die Menge aller TM-Codes für TMs, die bei jeder Eingabe anhalten.

b) Die Menge aller TM-Codes für TMs, die bei keiner Eingabe anhalten.

c) Die Menge aller TM-Codes für TMs, die bei mindestens einer Eingabe anhalten.

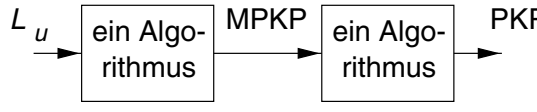
* d) Die Menge aller TM-Codes für TMs, die nicht bei mindestens einer Eingabe anhalten.

9.4 Das Postsche Korrespondenzproblem

In diesem Abschnitt werden wir damit beginnen, unentscheidbare Fragen bezüglich Turing-Maschinen auf unentscheidbare Fragen bezüglich »realer« Tatsachen zu reduzieren, die in keinem Zusammenhang mit der Abstraktion der Turing-Maschine stehen. Wir beginnen mit dem so genannten »Postschen Korrespondenzproblem« (*PKP*), das zwar noch immer abstrakt ist, aber Zeichenreihen statt Turing-Maschinen involviert. Wir wollen beweisen, dass dieses Zeichenreihenproblem unentscheidbar ist, und dann dessen Unentscheidbarkeit als Basis verwenden, um weitere Probleme als unentscheidbar zu beweisen, indem wir das PKP auf diese Probleme reduzieren.

Wir beweisen die Unentscheidbarkeit des PKP, indem wir L_u auf PKP reduzieren. Dazu verwenden wir ein »modifiziertes« PKP und reduzieren dieses modifizierte Problem auf das ursprüngliche PKP. Dann reduzieren wir L_u auf das modifizierte PKP. Abbildung 9.11 zeigt die Reduktionsfolge. Da die ursprüngliche L_u als unentscheidbar bekannt ist, schließen wir, dass das PKP unentscheidbar ist.

Abbildung 9.11: Reduktionen, die die Unentscheidbarkeit des Postschen Korrespondenzproblems beweisen



9.4.1 Definition des Postschen Korrespondenzproblems

Eine Instanz des *Postschen Korrespondenzproblems* (PKP) besteht aus zwei Listen von Zeichenreihen über einem Alphabet Σ; die beiden Listen müssen von gleicher Länge sein. Wir verweisen allgemein auf die Listen A und B mit der Schreibweise $A = w_1, w_2, \ldots, w_k$ und $B = x_1, x_2, \ldots, x_k$ für eine ganze Zahl k. Das Paar (w_i, x_i) wird für jedes i ein *korrespondierendes* Paar genannt.

Wir sagen, diese Instanz des PKP *besitzt eine Lösung* , wenn es eine Folge von einer oder mehreren ganzen Zahlen $i_1, i_2, \ldots, i_m$ gibt, die die gleiche Zeichenreihe ergeben, wenn sie in den Listen A und B als Indizes von Zeichenreihen interpretiert werden. Das heißt $w_{i_1}w_{i_2}w_{i_3}\ldots w_{i_m} = x_{i_1}x_{i_2}x_{i_3}\ldots x_{i_m}$. In diesem Fall sagen wir, die Folge $i_1, i_2, \ldots, i_m$ ist eine *Lösung* dieser Instanz des PKP. Das Postsche Korrespondenzproblem lautet folgendermaßen:

- Gegeben sei eine Instanz des PKP. Besitzt diese Instanz eine Lösung?

Beispiel 9.13 Sei Σ = {0, 1} und A und B Listen, wie in Tabelle 9.1 definiert. In diesem Fall besitzt das PKP eine Lösung. Beispielsweise seien $m = 4$, $i_1 = 2$, $i_2 = 1$, $i_3 = 1$ und $i_4 = 3$; die Lösung ist also die Liste 2, 1, 1, 3. Wir verifizieren dies, indem wir die korrespondierenden Zeichenreihen aus den beiden Listen jeweils in dieser Reihenfolge verketten. Wir erhalten $w_2w_1w_1w_3 = x_2x_1x_1x_3 = 101111110$. Beachten Sie, dass diese Lösung nicht eindeutig ist; eine weitere Lösung lautet 2, 1, 1, 3, 2, 1, 1, 3. ■

Tabelle 9.1: Eine Instanz des PKP

	Liste A	Liste B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Beispiel 9.14 Dieses Beispiel hat keine Lösung. Auch hier ist Σ = {0, 1}, doch nun besteht die Instanz aus den beiden Listen aus Tabelle 9.2.

Tabelle 9.2: Eine weitere Instanz des PKP

	Liste A	Liste B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

Das PKP als Sprache

Da wir das Entscheidbarkeitsproblem diskutieren, ob eine gegebene Instanz des PKP eine Lösung besitzt, müssen wir dieses Problem als Sprache ausdrücken. Da das PKP den Instanzen erlaubt, beliebige Alphabete zu besitzen, besteht die Sprache PKP eigentlich aus einer Menge von Zeichenreihen über einem festen Alphabet, die Instanzen des PKP kodieren, ähnlich unserer Kodierung von Turing-Maschinen in Abschnitt 9.1.2, die beliebige Mengen von Zuständen und Bandsymbolen besitzen. Beispielsweise können wir bei einer PKP-Instanz mit einem Alphabet mit bis zu 2^k Symbolen für die Symbole unterschiedliche k-Bit-Binärcodes verwenden.

Da jede PKP-Instanz ein endliches Alphabet besitzt, können wir für jede Instanz ein k finden und dann alle Instanzen in einem Alphabet aus den drei Symbolen 1, 0 und »Komma« als Trennzeichen zwischen Zeichenreihen kodieren. Wir beginnen die Kodierung mit k in Binärdarstellung, gefolgt von einem Komma. Danach folgen alle Zeichenreihenpaare, wobei die Zeichenreihen durch Kommas getrennt und ihre Symbole in einem k-Bit-Binärcode kodiert werden.

Angenommen, die PKP-Instanz aus Tabelle 9.2 besitzt eine Lösung, beispielsweise i_1, i_2,…, i_m für $m \geq 1$. Wir müssen fordern: $i_1 = 1$. Wäre $i_1 = 2$, dann müsste eine Zeichenreihe, die mit $w_2 = 011$ beginnt, mit einer Zeichenreihe übereinstimmen, die mit $x_2 = 11$ beginnt. Doch diese Gleichheit ist nicht möglich, da die ersten Symbole dieser beiden Zeichenreihen 0 bzw. 1 lauten. Ebenso ist $i_1 = 3$ nicht möglich, denn dann müsste eine mit $w_3 = 101$ beginnende Zeichenreihe mit einer anderen Zeichenreihe übereinstimmen, die mit $x_3 = 011$ beginnt.

Ist $i_1 = 1$, dann müssten die beiden korrespondierenden Zeichenreihen aus den Listen A und B wie folgt beginnen:

A: 10…
B: 101…

Sehen wir uns nun an, wie i_2 lauten könnte:

1. Wäre $i_2 = 1$, dann hätten wir ein Problem, da keine mit $w_1w_1 = 1010$ beginnende Zeichenreihe mit einer Zeichenreihe übereinstimmen kann, die mit $x_1x_1 = 101101$ beginnt; auf der vierten Position ist keine Übereinstimmung möglich.
2. Wäre $i_2 = 2$, hätten wir erneut ein Problem, da keine mit $w_1w_2 = 10011$ beginnende Zeichenreihe mit einer Zeichenreihe übereinstimmen kann, die mit $x_1x_2 = 10111$ beginnt; sie unterscheiden sich in der dritten Position.
3. Nur $i_2 = 3$ ist möglich.

Wählen wir $i_2 = 3$, dann lauten die entsprechenden Zeichenreihen zur Liste der ganzen Zahlen i_1, i_3:

A: 10101…
B: 101011…

Nichts an diesen Zeichenreihen deutet sofort darauf hin, dass die Liste 1, 3 nicht bis hin zu einer Lösung erweiterbar ist. Allerdings können wir schließen, dass dies nicht möglich ist. Der Grund liegt darin, dass wir uns wieder in der gleichen Situation wie nach der Wahl von $i_1 = 1$ befinden: Wenn man davon absieht, dass sich in Liste B eine

weitere 1 am Ende befindet, entspricht die Zeichenreihe zur Liste B genau derjenigen zur Liste A. Wir sind also dazu gezwungen, $i_3 = 3$, $i_4 = 3$ und so weiter zu wählen. Zeichenreihe A und Zeichenreihe B sind so niemals in Übereinstimmung zu bringen, und es existiert daher auch keine Lösung. ■

9.4.2 Das »modifizierte« PKP

Die Reduktion von L_u auf PKP ist einfacher, wenn wir zuerst eine Zwischenversion des PKP vorstellen, das *Modifizierte Postsche Korrespondenzproblem* oder MPKP. In diesem MPKP wird an eine Lösung die zusätzliche Anforderung gestellt, dass das erste Paar in den Listen A und B auch das erste Paar der Lösung sein muss. Formal ausgedrückt besteht eine Instanz des MPKP aus zwei Listen, $A = w_1, w_2, \ldots, w_k$ und $B = x_1, x_2, \ldots, x_k$, und eine Lösung besteht aus einer Liste mit 0 oder mehr ganzen Zahlen $i_1, i_2, \ldots, i_m$, sodass $w_1 w_{i_1} w_{i_2} \ldots w_{i_m} = x_1 x_{i_1} x_{i_2} \ldots x_{i_m}$.

Beachten Sie, dass das Paar (w_1, x_1) am Anfang der beiden Zeichenreihen stehen muss, auch wenn der Index 1 nicht am Anfang der Lösungsliste erwähnt wird. Im Gegensatz zu PKP, bei der die Lösung mindestens eine ganze Zahl in der Lösungsliste besitzen muss, könnte in MPKP auch die leere Liste eine Lösung sein, falls $w_1 = x_1$ (doch solche Instanzen sind uninteressant und werden in unserer Verwendung des MPKP nicht vorkommen).

Partielle Lösungen

In Beispiel 9.14 haben wir eine Technik zur Analyse von PKP-Instanzen verwendet, die häufig eingesetzt wird. Wir haben überlegt, wie die möglichen *partiellen Lösungen* lauten könnten, also Folgen von Indizes $i_1, i_2, \ldots, i_r$, sodass eine der Listen $w_{i_1} w_{i_2} \ldots w_{i_r}$ bzw. $x_{i_1} x_{i_2} \ldots x_{i_r}$ Präfix der anderen ist, obwohl die beiden Listen nicht übereinstimmen. Beachten Sie, dass jedes Präfix einer Folge ganzer Zahlen eine partielle Lösung sein muss, wenn diese Folge eine Lösung ist. Kennen wir also die partiellen Lösungen, dann können wir auf mögliche Lösungen schließen.

Beachten Sie aber, dass das PKP unentscheidbar ist und es daher keinen Algorithmus gibt, um alle partiellen Lösungen zu berechnen. Es kann unendlich viele davon geben, und außerdem existiert keine Obergrenze für die unterschiedlichen Längen der Zeichenreihen $w_{i_1} w_{i_2} \ldots w_{i_r}$ und $x_{i_1} x_{i_2} \ldots x_{i_r}$, selbst wenn eine partielle Lösung zu einer Lösung führt.

Beispiel 9.15 Die Listen aus Tabelle 9.1 können als eine Instanz des MPKP angesehen werden. Diese Instanz besitzt jedoch keine Lösung. Zum Beweis dessen beachten Sie, dass jede partielle Lösung mit dem Index 1 beginnen muss, und daher würde der Anfang der beiden Lösungszeichenreihen folgendermaßen aussehen:

A: 1…
B: 111…

Die nächste ganze Zahl kann weder 2 noch 3 sein, da sowohl w_2 als auch w_3 mit 10 beginnen und daher nicht zu einer Übereinstimmung auf der dritten Position führen würden. Daher muss der nächste Index 1 lauten, und wir erhalten Folgendes:

A: 11…
B: 111111…

Auf diese Weise könnten wir unendlich lange folgern. Nur eine weitere 1 in der Lösung kann zu Übereinstimmungen führen, doch wenn wir lediglich Index 1 auswählen können, bleibt Zeichenreihe B stets dreimal so lang wie Zeichenreihe A, und beide werden daher niemals übereinstimmen. ■

Ein wichtiger Schritt beim Beweis der Unentscheidbarkeit des PKP ist die Reduktion von MPKP auf PKP. Später zeigen wir, dass MPKP unentscheidbar ist, indem wir L_u auf MPKP reduzieren. Zu diesem Zeitpunkt haben wir den Beweis, dass PKP ebenfalls unentscheidbar ist; wäre PKP entscheidbar, dann könnten wir MPKP entscheiden und damit auch L_u.

Gegeben sei eine Instanz des MPKP mit dem Alphabet Σ. Wir konstruieren eine Instanz des PKP. Zuerst führen wir ein neues Symbol $*$ ein, das in der PKP-Instanz zwischen allen Symbolen in den Zeichenreihen der MPKP-Instanz steht. Allerdings folgen die Symbole $*$ in der Liste A den Symbolen aus Σ, und in der Liste B stehen sie ihnen voran. Die einzige Ausnahme ist ein neues Paar, das auf dem ersten Paar der MPKP-Instanz basiert; dieses Paar ist durch einen zusätzlichen $*$ am Anfang von w_1 ausgezeichnet und kann daher als Anfang der PKP-Lösung verwendet werden. Die PKP-Instanz wird um ein Endpaar ($, *$) ergänzt. Dieses Paar dient als letztes Paar in einer PKP-Lösung, die eine Lösung der MPKP-Instanz simuliert.

Wir formalisieren nun die obige Konstruktion. Wir haben eine Instanz des MPKP mit den Listen $A = w_1, w_2, \ldots, w_k$ und $B = x_1, x_2, \ldots, x_k$. Wir nehmen an, $*$ und $ seien Symbole, die nicht im Alphabet Σ dieser MPKP-Instanz enthalten sind. Wir konstruieren PKP-Instanzen $C = y_0, y_1, \ldots, y_{k+1}$ sowie $D = z_0, z_1, \ldots, z_{k+1}$ wie folgt:

1. Für $i = 1, 2, \ldots, k$ sei y_i gleich w_i, wobei nach jedem Symbol von w_i ein $*$ folgt, und z_i sei x_i, wobei jedem Symbol von x_i ein $*$ voransteht.
2. $y_0 = *y_1$ und $z_0 = z_1$. Das Paar 0 sieht also aus wie Paar 1, doch am Anfang der ersten Zeichenreihen aus der ersten Liste steht ein zusätzlicher $*$. Beachten Sie, dass das Paar 0 das einzige Paar der PKP-Instanz sein wird, in dem beide Zeichenreihen mit dem gleichen Symbol beginnen, und damit muss jede Lösung dieser PKP-Instanz mit Index 0 beginnen.
3. $y_{k+1} = \$$ und $z_{k+1} = *\$$.

Beispiel 9.16 Angenommen, Tabelle 9.1 zeigt eine MPKP-Instanz. Dann zeigt Tabelle 9.3 die durch die obigen Schritte konstruierte PKP-Instanz. ■

Satz 9.17 MPKP kann auf PKP reduziert werden.

Beweis: Die obige Konstruktion ist der Kern des Beweises. Zuerst nehmen wir an, $i_1, i_2, \ldots, i_m$ sei eine Lösung der gegebenen MPKP-Instanz mit den Listen A und B. Dann wissen wir, dass $w_1 w_{i_1} w_{i_2} \ldots w_{i_m} = x_1 x_{i_1} x_{i_2} \ldots x_{i_m}$ gilt. Würden wir jeweils jedes w_j durch y_j und jedes x_j durch z_j ersetzen, dann erhielten wir zwei Zeichenreihen, die sich fast gleichen: $y_1 y_{i_1} y_{i_2} \ldots y_{i_m}$ und $z_1 z_{i_1} z_{i_2} \ldots z_{i_m}$. Der Unterschied liegt darin, dass der ersten Zeichenreihe ein $*$ am Anfang und der zweiten Zeichenreihe ein $*$ am Ende fehlen würde, das heißt:

$$*y_1 y_{i_1} y_{i_2} \ldots y_{i_m} = z_1 z_{i_1} z_{i_2} \ldots z_{i_m} *$$

Tabelle 9.3: Konstruktion einer PKP-Instanz aus einer MPKP-Instanz

	Liste C	Liste D
I	y_i	z_i
0	*1*	*1*1*1
1	1*	*1*1*1
2	1*0*1*1*1*	*1*0
3	1*0*	*0
4	$	*$

Da allerdings $y_0 = *y_1$ und $z_0 = z_1$ gilt, können wir den ersten $*$ einfügen, indem wir den ersten Index durch 0 ersetzen. Daraus folgt:

$$y_0 y_{i_1} y_{i_2} \ldots y_{i_m} = z_1 z_{i_1} z_{i_2} \ldots z_{i_m} *$$

Wir kümmern uns um den letzten $*$, indem wir den Index $k + 1$ hinten anfügen. Da y_{k+1} = \$ und z_{k+1} = *\$ gilt, erhalten wir:

$$y_0 y_{i_1} y_{i_2} \ldots y_{i_m} y_{k+1} = z_0 z_{i_1} z_{i_2} \ldots z_{i_m} z_{k+1}$$

Wir haben also gezeigt, dass 0, i_1, i_2,…, i_m, i_{k+1} eine Lösung der Instanz des PKP ist.

Nun müssen wir das Umgekehrte zeigen. Besitzt die konstruierte Instanz des PKP eine Lösung, dann besitzt die ursprüngliche MPKP-Instanz ebenfalls eine. Wir beobachten, dass eine Lösung der PKP-Instanz mit Index 0 beginnen und mit Index $k + 1$ enden muss, da nur das 0-te Paar die Zeichenreihen y_0 und z_0 enthält, die mit dem gleichen Symbol beginnen, und nur das $(k + 1)$-te Paar enthält Zeichenreihen, die mit dem gleichen Symbol enden. Daher lautet die PKP-Lösung 0, i_1, i_2, …, i_m, i_{k+1}.

Wir zeigen, dass i_1, i_2, …, i_m eine Lösung der MPKP-Instanz ist. Der Grund dafür liegt darin, dass wir die Zeichenreihe $w_1 w_{i_1} w_{i_2} \ldots w_{i_m}$ erhalten, wenn wir die Symbole $*$ und das letzte \$ aus der Zeichenreihe $y_0 y_{i_1} y_{i_2} \ldots y_{i_m} y_{k+1}$ entfernen. Das Gleiche führen wir mit der Zeichenreihe $z_0 z_{i_1} z_{i_2} \ldots z_{i_m} z_{k_1}$ aus und erhalten $x_1 x_{i_1} x_{i_2} \ldots x_{i_m}$. Wir wissen, dass

$$y_0 y_{i_1} y_{i_2} \ldots y_{i_m} y_{k+1} = z_0 z_1 z_{i_1} z_{i_2} \ldots z_{i_m} z_{k+1}$$

gilt, und somit folgt:

$$w_1 w_{i_1} w_{i_2} \ldots w_{i_m} = x_1 x_{i_1} x_{i_2} \ldots x_{i_m}$$

Daher impliziert eine Lösung der PKP-Instanz auch eine Lösung der MPKP-Instanz.

Wir sehen nun, dass die Konstruktion, wie sie vor diesem Satz beschrieben wurde, ein Algorithmus ist, der eine Instanz des MPKP mit einer Lösung in eine Instanz des PKP mit einer Lösung konvertiert; dies gilt ebenso für eine Instanz des MPKP ohne Lösung, die in eine Instanz des PKP ohne Lösung konvertiert wird. Es existiert daher eine Reduktion von MPKP auf PKP, die bestätigt, dass MPKP entscheidbar wäre, wenn PKP entscheidbar wäre. ■

9.4.3 Abschluss des Beweises der PKP-Unentscheidbarkeit

Wir vervollständigen nun die Kette der Reduktionen aus Abbildung 9.11, indem wir L_u auf MPKP reduzieren. Gegeben sei ein Paar (M, w). Wir konstruieren eine Instanz (A, B) des MPKP, sodass TM M die Eingabe w nur dann akzeptiert, wenn (A, B) eine Lösung besitzt.

Die grundlegende Idee besteht darin, dass die MPKP-Instanz (A, B) in ihren partiellen Lösungen die Berechnung von M mit Eingabe w simuliert. Die partiellen Lösungen bestehen aus Zeichenreihen, die Präfixe der Folgen von Konfigurationen von M sind: $\#\alpha_1\#\alpha_2\#\alpha_3\#\ldots$, wobei α_1 die anfängliche Konfiguration von M mit Eingabe w und $\alpha_i \vdash \alpha_{i+1}$ für alle i ist. Die Zeichenreihe aus Liste B wird der Zeichenreihe aus Liste A immer um eine Konfiguration voraus sein, bis M in einen akzeptierenden Zustand wechselt. In diesem Fall werden Paare zur Verfügung stehen, die der Liste A ermöglichen, die Liste B »einzuholen« und eine Lösung zu produzieren. Wird allerdings kein akzeptierender Zustand erreicht, so gibt es keine Möglichkeit, dass diese Paare verwendet werden können, und es existiert keine Lösung.

Um die Konstruktion einer MPKP-Instanz zu vereinfachen, nutzen wir Satz 8.12, der aussagt, dass wir Folgendes annehmen können: Unsere TM gibt niemals ein Leerzeichen aus und bewegt sich nie links von der anfänglichen Position des Kopfes. In diesem Fall ist eine Konfiguration der Turing-Maschine immer eine Zeichenreihe der Form $\alpha q \beta$, wobei α und β Zeichenreihen aus nicht leeren Bandsymbolen sind und q ein Zustand ist. Wir sollten β allerdings erlauben, leer zu sein, wenn sich der Kopf über dem Leerzeichen unmittelbar rechts von α befindet, statt ein Leerzeichen rechts vom Zustand einzufügen. Daher entsprechen die Symbole von α und β exakt den Inhalten der Zellen, die die Eingabe enthielten, sowie der Zellen rechts davon, die der Kopf schon gelesen hat.

Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ eine TM, die Satz 8.12 genügt, und w aus Σ^* sei eine Eingabezeichenreihe. Wir konstruieren daraus eine Instanz des MPKP. Sie werden die Motivation für unsere Auswahl der Paare besser verstehen, wenn Sie sich daran erinnern, dass das Ziel darin liegt, dass die erste Liste so lange um eine Konfiguration hinter der zweiten Liste liegt, bis M akzeptiert.

1. Das erste Paar ist

Liste A	Liste B
#	$\#q_0w\#$

   Dieses Paar, mit dem nach den Regeln von MPKP jede Lösung beginnen muss, startet die Simulation von M mit Eingabe w. Beachten Sie, dass die Liste B der Liste A anfänglich um eine Konfiguration voraus ist.

2. An beide Listen können Bandsymbole und der Separator # angehängt werden. Die Paare

Liste A	Liste B	
X	X	für jedes X in Γ
#	#	

erlauben es, Symbole zu »kopieren«, in die der Zustand nicht involviert ist. Die Wahl solcher Paare ermöglicht die Ausdehnung der Zeichenreihe A, um eine Übereinstimmung mit Zeichenreihe B zu erhalten und gleichzeitig Teile der vorherigen Konfiguration an das Ende der Zeichenreihe B zu kopieren. Dies unterstützt den Aufbau der nächsten Konfiguration in der Folge von Bewegungen von M am Ende der Zeichenreihe B.

3. Um eine Bewegung von M zu simulieren, stehen uns bestimmte Paare zur Verfügung, die solche Bewegungen reflektieren. Für alle q aus $Q - F$ (q ist also ein nicht akzeptierender Zustand), p aus Q und X, Y sowie Z aus Γ ergibt sich also Folgendes:

Liste A	Liste B	
qX	Yp	wenn $\delta(q, X) = (p, Y, R)$
ZqX	PZY	wenn $\delta(q, X) = (p, Y, L)$; Z ist ein beliebiges Bandsymbol
$q\#$	$Yp\#$	wenn $\delta(q, B) = (p, Y, R)$
$Zq\#$	$pZY\#$	wenn $\delta(q, B) = (p, Y, L)$; Z ist ein beliebiges Bandsymbol

Ähnlich den Paaren aus (2) unterstützen auch diese Paare die Ausdehnung der Zeichenreihe B, um die nächste Konfiguration hinzuzufügen, indem die Zeichenreihe A ausgedehnt wird, sodass sie mit der Zeichenreihe B übereinstimmt. Allerdings verwenden diese Paare den Zustand, um die Änderung in der aktuellen Konfiguration zu bestimmen, die für die Produktion der nächsten Konfiguration benötigt wird. Diese Änderungen – ein neuer Zustand, ein Bandsymbol und eine Kopfbewegung – spiegeln sich in der Konfiguration wider, die am Ende der Zeichenreihe B konstruiert wird.

4. Besitzt die Konfiguration am Ende der Zeichenreihe B einen akzeptierenden Zustand, dann müssen wir es ermöglichen, dass aus der partiellen eine vollständige Lösung wird. Dazu erweitern wir mit »Konfigurationen«, die nicht wirkliche Konfigurationen von M sind. Sie repräsentieren jedoch, was geschehen würde, wenn der akzeptierende Zustand beiderseits alle Bandsymbole konsumieren könnte. Ist daher q ein akzeptierender Zustand, dann existieren für alle Bandsymbole X und Y Paare:

Liste A	Liste B
XqY	q
Xq	q
qY	q

5. Wenn der akzeptierende Zustand schließlich alle Bandsymbole konsumiert hat, steht er als Konfiguration isoliert in der Zeichenreihe B. Wir verwenden als letztes Paar

Liste A	Liste B
$q\#\#$	$\#$

um die Lösung zu vervollständigen.

Im Folgenden beziehen wir uns auf die oben generierten Paare als die Paare von Regel (1), von Regel (2) und so weiter.

Beispiel 9.18 Wir konvertieren die TM

$$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_3\})$$

wobei δ durch

q_i	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, B)$
q_1	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
q_3	–	–	–

gegeben ist, mit der Eingabezeichenreihe $w = 01$ in eine Instanz des MPKP. Zur Vereinfachung schreibt M niemals ein Leerzeichen, und somit enthält keine Konfiguration ein B. Wir können daher alle Paare, die ein B enthalten, weglassen. Tabelle 9.4 zeigt die gesamte Paarliste sowie eine Erläuterung, woher jedes Paar stammt.

Beachten Sie, dass M die Eingabe 01 mit folgender Bewegungsfolge akzeptiert:

$$q_101 \vdash 1q_21 \vdash 10q_1 \vdash 1q_201 \vdash q_3101$$

Sehen wir uns die Folge der partiellen Lösungen an, die diese Berechung von M simulieren und zu einer Lösung führen. Wir beginnen mit dem ersten Paar, wie es in jeder MPKP-Lösung erforderlich ist:

$$A\colon \#$$
$$B\colon \#q_101\#$$

Das Verfahren, die partielle Lösung zu erweitern, muss so erfolgen, dass die Zeichenreihe aus Liste A auch weiterhin Präfix des Restes $q_101\#$ von B ist. Wir müssen daher als Nächstes das Paar $(q_10, 1q_2)$ wählen, das wir aus Regel (3) erhalten. Die partielle Lösung lautet danach:

$$A\colon \#q_10$$
$$B\colon \#q_101\#1q_2$$

Tabelle 9.4: Eine MPKP-Instanz, konstruiert aus der TM *M* aus Beispiel 9.18

Regel	Liste *A*	Liste *B*	Quelle
(1)	#	$\#q_1 01\#$	
(2)	0	0	
	1	1	
	#	#	
(3)	$q_1 0$	$1q_2$	aus $\delta(q_1, 0) = (q_2, 1, R)$
	$0q_1 1$	$q_2 00$	aus $\delta(q_1, 1) = (q_2, 0, L)$
	$1q_1 1$	$q_2 10$	aus $\delta(q_1, 1) = (q_2, 0, L)$
	$0q_1\#$	$q_2 01\#$	aus $\delta(q_1, B) = (q_2, 1, L)$
	$1q_1\#$	$q_2 11\#$	aus $\delta(q_1, B) = (q_2, 1, L)$
	$0q_2 0$	$q_3 00$	aus $\delta(q_2, 0) = (q_3, 0, L)$
	$1q_2 0$	$q_3 10$	aus $\delta(q_2, 0) = (q_3, 0, L)$
	$q_2 1$	$0q_1$	aus $\delta(q_2, 1) = (q_1, 0, R)$
	$q_2\#$	$0q_2\#$	aus $\delta(q_2, B) = (q_2, 0, R)$
(4)	$0q_3 0$	q_3	
	$0q_3 1$	q_3	
	$1q_3 0$	q_3	
	$1q_3 1$	q_3	
	$0q_3$	q_3	
	$1q_3$	q_3	
	$q_3 0$	q_3	
	$q_3 1$	q_3	
(5)	$q_3\#\#$	#	

Wir können die partielle Lösung nun mit den »kopierenden« Paaren aus Regel (2) bis ausschließlich des Zustands (q_2) in der letzten (zweiten) Konfiguration von *B* erweitern. Die partielle Lösung ist dann:

$$A\colon \#q_1 01\#1$$
$$B\colon \#q_1 01\#1q_2 1\#1$$

Nun verwenden wir ein weiteres Paar aus Regel (3), das Paar ($q_2 1$, $0 q_1$), um die nächste Bewegung zu simulieren, und erhalten die folgende partielle Lösung:

$$A\colon \#q_1 01\#1q_2 1$$
$$B\colon \#q_1 01\#1q_2 1\#10q_1$$

Wir könnten nun Paare aus Regel (2) verwenden, um die nächsten drei Symbole #, 1 und 0 zu »kopieren«. Allerdings wäre es ein Fehler, so weit zu gehen, da die nächste Bewegung von M den Kopf nach links bewegt und die 0 vor dem Zustand für das nächste Paar aus Regel (3) benötigt wird. Daher »kopieren« wir lediglich die nächsten beiden Symbole und erhalten folgende partielle Lösung:

$$A\colon \#q_1 01\#1q_2 1\#1$$
$$B\colon \#q_1 01\#1q_2 1\#10q_1\#1$$

Als nächstes Paar aus Regel (3) wird ($0q_1\#$, $q_2 01\#$) verwendet, und wir erhalten:

$$A\colon \#q_1 01\#1q_2 1\#10q_1\#$$
$$B\colon \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#$$

Wir können nun ein weiteres Paar aus Regel (3) verwenden, ($1q_2 0$, $q_3 10$), das zur Akzeptanz führt:

$$A\colon \#q_1 01\#1q_2 1\#10q_1\#1q_2 0$$
$$B\colon \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 10$$

Nun setzen wir Paare aus Regel (4) ein, um alles bis auf q_3 aus der letzten Konfiguration von B zu entfernen. Außerdem benötigen wir Paare aus Regel (2), um Symbole zu kopieren. Die Fortsetzung der partiellen Lösung ergibt:

$$A\colon \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#$$
$$B\colon \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#$$

Da nur q_3 in der letzten Konfiguration von B übrig ist, können wir das Paar ($q_3\#\#$, #) aus Regel (5) verwenden, um die Lösung abzuschließen:

$$A\colon \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#\#$$
$$B\colon \#q_1 01\#1q_2 1\#10q_1\#1q_2 01\#q_3 101\#q_3 01\#q_3 1\#q_3\#\# \qquad \blacksquare$$

Satz 9.19 Das Postsche Korrespondenzproblem ist unentscheidbar.

Beweis: Wir haben die in Abbildung 9.11 dargestellte Reduktionskette fast abgeschlossen. Die Reduktion von MPKP auf PKP hat Satz 9.17 gezeigt. Die Konstruktion in diesem Abschnitt zeigt, wie L_u auf MPKP reduziert wird. Wir schließen daher den Beweis der Unentscheidbarkeit von PKP ab, indem wir die Korrektheit der Konstruktion beweisen:

- M akzeptiert w nur dann, wenn die konstruierte MPKP-Instanz eine Lösung besitzt.

(Nur-wenn-Teil) Beispiel 9.18 zeigt die grundlegende Idee. Ist w in $L(M)$ enthalten, dann können wir mit dem Paar aus Regel (1) beginnen und die Berechnung von M mit w simulieren. Wir verwenden ein Paar aus Regel (3), um eine Bewegung von M zu

simulieren, und die Paare aus Regel (2), um Bandsymbole sowie die Markierung # nach Bedarf zu kopieren. Erreicht M einen akzeptierenden Zustand, dann dienen die Paare aus Regel (4) sowie zum Schluss das Paar aus Regel (5) dazu, die Übereinstimmung der Zeichenreihen A und B zu erreichen und eine Lösung zu erhalten.

(Wenn-Teil) Wir müssen argumentieren: Falls die MPKP-Instanz eine Lösung besitzt, dann akzeptiert M w. Da wir mit MPKP arbeiten, muss jede Lösung mit dem ersten Paar beginnen. Eine partielle Lösung beginnt also wie folgt:

$$A\text{: }\#$$
$$B\text{: }\#q_0w\#$$

Solange die partielle Lösung keinen akzeptierenden Zustand enthält, sind die Paare aus den Regeln (4) und (5) nutzlos. Zustände und ein oder zwei der umgebenden Bandsymbole können nur mit Paaren der Regel (3) behandelt werden. Um alle weiteren Bandsymbole sowie # kümmern sich die Paare aus Regel (2). Solange M keinen akzeptierenden Zustand erreicht hat, sind daher alle partiellen Lösungen von der Form

$$A\text{: }x$$
$$B\text{: }xy$$

x ist hierbei eine Folge von Konfigurationen von M, die eine Berechung von M mit Eingabe w repräsentieren, möglicherweise gefolgt von # und dem Anfang der nächsten Konfiguration α. Der Rest y entspricht dem Rest von α, einem weiteren # und dem Anfang der Konfiguration, die auf α folgt, bis zu dem Punkt, an dem x selbst in α endet.

Solange M also nicht in einen akzeptierenden Zustand übergeht, stellt die partielle Lösung keine Lösung dar, weil Zeichenreihe B länger ist als Zeichenreihe A. Wenn eine Lösung existiert, muss M also irgendwann in einen akzeptierenden Zustand übergehen; das heißt M akzeptiert w. ■

9.4.4 Übungen zum Abschnitt 9.4

Übung 9.4.1 Besitzt jede der folgenden Instanzen des PKP eine Lösung? Jede Instanz wird von zwei Listen, A und B, repräsentiert, und die i-ten Zeichenreihen der beiden Listen korrespondieren für jedes $i = 1, 2, \ldots$.

* a) $A = (01, 001, 10)$; $B = (011, 10, 00)$

b) $A = (01, 001, 10)$; $B = (011, 01, 00)$

c) $A = (ab, a, bc, c)$; $B = (bc, ab, ca, a)$

! **Übung 9.4.2** Wir haben gezeigt, dass PKP unentscheidbar ist, doch wir nahmen an, das Alphabet Σ könne beliebig sein. Zeigen Sie, dass PKP auch unentscheidbar ist, wenn wir das Alphabet auf $\Sigma = \{0, 1\}$ beschränken, indem Sie PKP auf diesen Spezialfall des PKP reduzieren.

*! **Übung 9.4.3** Angenommen, wir beschränken PKP auf ein Alphabet aus einem Symbol, beispielsweise $\Sigma = \{0\}$. Wäre dieser eingeschränkte Fall des PKP noch immer unentscheidbar?

! **Übung 9.4.4** Ein »*post tag system*« besteht aus einer Menge von Zeichenreihenpaaren, die aus einem endlichen Alphabet Σ gewählt sind, sowie einer Startzeichenreihe. Ist (w, x) ein Paar und y eine Zeichenreihe über Σ, dann sagen wir $wy \vdash yx$. Wir können also mit einer Bewegung ein Präfix w der »aktuellen« Zeichenreihen wy entfernen und an das Ende die Zeichenreihe x anfügen, die mit w ein Paar bildet. Definieren Sie $\vdash^*$ als null oder mehr Schritte von $\vdash$, wie bei Ableitungen in einer kontextfreien Grammatik. Gegeben sei eine Menge von Paaren P sowie eine Startzeichenreihe z. Zeigen Sie, dass unentscheidbar ist, ob $z \vdash^* \in$. *Hinweis:* Für jede TM M und Eingabe w sei z die anfängliche Konfiguration von M mit Eingabe w, gefolgt von einem Trennzeichen #. Wählen Sie die Paare P so, dass jede Konfiguration von M schließlich zur Konfiguration werden muss, die auf eine Bewegung von M folgt. Wechselt M in einen akzeptierenden Zustand, dann richten Sie es so ein, dass die aktuelle Zeichenreihe schließlich gelöscht, also auf $\in$ reduziert werden kann.

9.5 Weitere unentscheidbare Probleme

Wir werden uns nun verschiedene andere Probleme ansehen, die wir als unentscheidbar nachweisen können. Prinzipiell besteht die Technik darin, PKP auf das Problem zu reduzieren, das wir als unentscheidbar nachweisen wollen.

9.5.1 Probleme bei Programmen

Zuerst beobachten wir, dass wir in jeder konventionellen Sprache ein Programm schreiben können, das eine Instanz des PKP als Eingabe übernimmt und systematisch nach Lösungen sucht, etwa in der Reihenfolge der *Länge* (Anzahl der Paare) potenzieller Lösungen. Da PKP beliebige Alphabete erlaubt, sollten wir die Symbole des Alphabets binär kodieren oder ein anderes festes Alphabet verwenden, wie im Kasten »Das PKP als Sprache« in Abschnitt 9.4.1 erläutert.

Unser Programm kann außerdem jede gewünschte Aufgabe ausführen, z. B. anhalten oder `hello, world` ausgeben, wenn es eine Lösung findet. Im anderen Fall wird das Programm diese besondere Aktion niemals ausführen. Es ist daher unentscheidbar, ob ein Programm `hello, world` ausgibt, ob es anhält, eine bestimmte Funktion aufruft, eine Glocke anschlägt oder eine andere nicht triviale Aktion ausführt. Tatsächlich gibt es in Analogie zum Satz von Rice einen Satz für Programme: Jede nicht triviale Eigenschaft, in die die Aktionen des Programms involviert sind (nicht jedoch eine lexikalische oder syntaktische Eigenschaft des Programms selbst), muss unentscheidbar sein.

9.5.2 Unentscheidbarkeit der Mehrdeutigkeit bei kontextfreien Grammatiken

Programme sind Turing-Maschinen hinreichend ähnlich, sodass die Beobachtungen aus Abschnitt 9.5.1 nicht überraschen. Nun werden wir sehen, wie wir PKP auf ein Problem reduzieren, das keiner Frage über Computer ähnelt: Ist eine gegebene kontextfreie Grammatik mehrdeutig?

Die Idee basiert auf der Untersuchung von Zeichenreihen, die eine Liste von Indizes repräsentieren, sowie auf korrespondierenden Zeichenreihen entsprechend einer der Listen einer PKP-Instanz. Diese Zeichenreihen können von einer Grammatik generiert werden, ebenso wie die entsprechenden Zeichenreihen für die andere Liste

der PKP-Instanz. Nehmen wir die Vereinigung dieser Grammatiken, dann gibt es genau dann eine Zeichenreihe, die von jeder der beiden ursprünglichen Grammatiken erzeugt wird, wenn diese PKP-Instanz eine Lösung besitzt. Es existiert also genau dann eine Lösung, wenn die Grammatik der Vereinigung mehrdeutig ist.

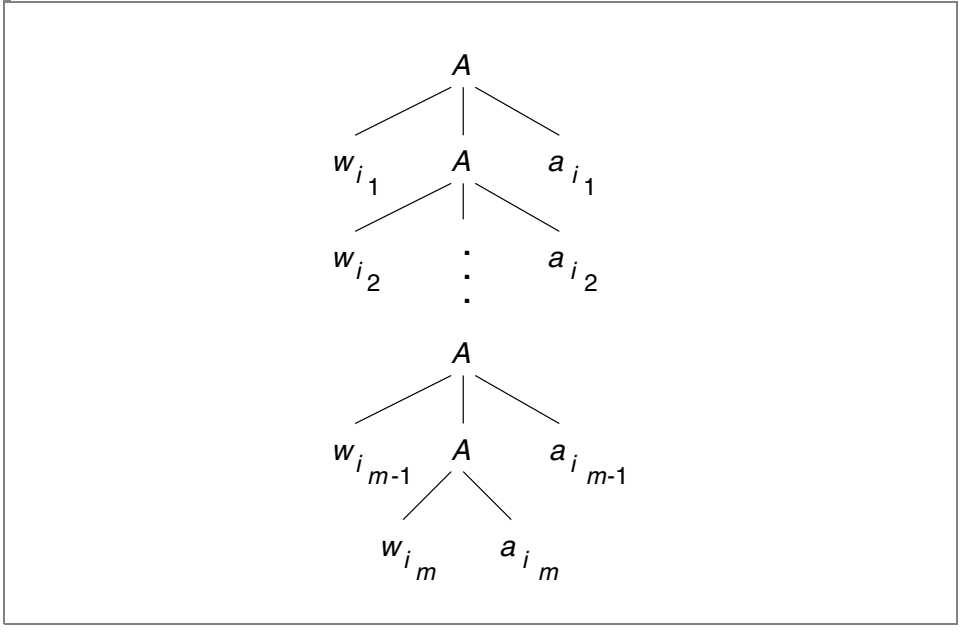
Wir legen diese Idee nun präziser dar. Die PKP-Instanz soll aus den Listen $A = w_1, w_2, \ldots, w_k$ und $B = x_1, x_2, \ldots, x_k$ bestehen. Für Liste A werden wir eine kfG mit A als einziger Variablen konstruieren. Die Terminale bestehen aus allen Symbolen des Alphabets Σ dieser PKP-Instanz sowie einer davon abweichenden Menge von *Indexsymbolen* $a_1, a_2, \ldots, a_k$, die die Auswahl der Zeichenreihenpaare in einer Lösung der PKP-Instanz repräsentieren. Das Indexsymbol a_i repräsentiert also die Wahl von w_i aus der Liste A oder von x_i aus der Liste B. Die Produktionen der kfG für die Liste A lauten wie folgt:

$$A \rightarrow w_1Aa_1 \mid w_2Aa_2 \mid \ldots \mid w_kAa_k \mid w_1a_1 \mid w_2a_2 \mid \ldots \mid w_ka_k$$

Wir nennen diese Grammatik G_A und ihre Sprache L_A. Künftig werden wir auf eine Sprache wie L_A als die *Sprache der Liste A* verweisen.

Beachten Sie, dass die von G_A abgeleiteten Terminal-Zeichenreihen alle von der Form $w_{i_1}w_{i_2}\ldots w_{i_m}\ a_{i_m}\ldots a_{i_2}a_{i_1}$ für $m \geq 1$ und Listen ganzer Zahlen $i_1, i_2, \ldots, i_m$ sind; jede ganze Zahl stammt aus dem Bereich 1 bis k. Die Satzformen von G_A besitzen alle ein einziges A zwischen den Zeichenreihen (w) und den Indexsymbolen (a), bis wir eine aus der letzten Gruppe von k Produktionen verwenden, die alle kein A im Rumpf besitzen. Abbildung 9.12 zeigt, wie die Ableitungsbäume aussehen.

Abbildung 9.12: Die Gestalt der Ableitungsbäume der Grammatik G_A



Beachten Sie außerdem, dass jede aus A in G_A ableitbare Zeichenreihe eine eindeutige Ableitung besitzt. Die Indexsymbole am Ende der Zeichenreihe bestimmen eindeutig, welche Produktion bei jedem Schritt einzusetzen ist. Das heißt nur zwei Produktions-

rümpfe enden auf ein gegebenes Indexsymbol a_i: $A \to w_i A a_i$ und $A \to w_i a_i$. Wir müssen die erste dieser beiden Produktionen verwenden, falls es sich nicht um den letzten Ableitungsschritt handelt, und im anderen Fall die zweite.

Wir sehen uns nun den zweiten Teil der gegebenen PKP-Instanz an, die Liste $B = x_1, x_2, \ldots, x_k$. Für diese Liste entwickeln wir die folgende analoge Grammatik G_B:

$$B \to x_1 B a_1 \mid x_2 B a_2 \mid \ldots \mid x_k B a_k \mid x_1 a_1 \mid x_2 a_2 \mid \ldots \mid x_k a_k$$

Wir bezeichnen die Sprache dieser Grammatik als L_B. Auf G_B sind die gleichen Beobachtungen wie auf G_A anwendbar. Insbesondere besitzt eine Terminal-Zeichenreihe in L_B eine eindeutige Ableitung, die über die Indexsymbole in der Endzeichenreihe bestimmt werden kann.
Zuletzt kombinieren wir die Sprachen und Grammatiken der beiden Listen und erhalten eine Grammatik G_{AB} für die gesamte PKP-Instanz. G_{AB} ist folgendermaßen zusammengesetzt:

1. Variable A, B und S; S ist das Startsymbol
2. Produktionen $S \to A \mid B$
3. Alle Produktionen aus G_A
4. Alle Produktionen aus G_B

Wir behaupten, dass G_{AB} genau dann mehrdeutig ist, wenn die Instanz (A, B) des PKP eine Lösung besitzt; diese Aussage bildet den Kern des folgenden Satzes.

Satz 9.20 Es ist unentscheidbar, ob eine kfG mehrdeutig ist.

Beweis: Wir haben schon den größten Teil der Reduktion von PKP auf die Frage, ob eine kfG mehrdeutig ist, geliefert; diese Reduktion beweist, dass das Problem der Mehrdeutigkeit einer kfG unentscheidbar ist, da PKP unentscheidbar ist. Wir müssen lediglich zeigen, dass die obige Konstruktion korrekt ist und damit Folgendes gilt:

- G_{AB} ist genau dann mehrdeutig, wenn die Instanz (A, B) des PKP eine Lösung besitzt.

(Wenn-Teil) Angenommen, $i_1, i_2, \ldots, i_m$ sei eine Lösung dieser Instanz des PKP. Sehen wir uns die folgenden beiden Ableitungen in G_{AB} an:

$$S \Rightarrow A \Rightarrow w_{i_1} A a_{i_1} \Rightarrow w_{i_1} w_{i_2} A a_{i_2} a_{i_1} \Rightarrow \ldots \Rightarrow$$

$$w_{i_1} w_{i_2} \ \ldots \ w_{i_{m-1}} A a_{i_{m-1}} \ \ldots \ a_{i_2} a_{i_1} \Rightarrow w_{i_1} w_{i_2} \ \ldots w_{i_m} a_{i_m} \ \ldots \ a_{i_2} a_{i_1}$$

$$S \Rightarrow B \Rightarrow x_{i_1} B a_{i_1} \Rightarrow x_{i_1} x_{i_2} B a_{i_2} a_{i_1} \Rightarrow \ldots \Rightarrow$$

$$x_{i_1} x_{i_2} \ \ldots \ x_{i_{m-1}} B a_{i_{m-1}} \ \ldots \ a_{i_2} a_{i_1} \Rightarrow x_{i_1} x_{i_2} \ \ldots x_{i_m} a_{i_m} \ \ldots \ a_{i_2} a_{i_1}$$

Da $i_1, i_2, \ldots, i_m$ eine Lösung ist, wissen wir, dass $w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$ ist. Diese beiden Ableitungen sind daher Ableitungen der gleichen Terminal-Zeichenreihe. Da die Ableitungen selbst offensichtlich zwei verschiedene Linksabteilungen dieser Terminal-Zeichenreihen sind, folgern wir, dass G_{AB} mehrdeutig ist.

(Nur-wenn-Teil) Wir haben schon festgestellt, dass eine gegebene Terminal-Zeichenreihe nicht mehr als eine Ableitung in G_A und nicht mehr als eine in G_B besitzen

kann. Eine Terminal-Zeichenreihe kann daher nur dann zwei Linksabteilungen in G_{AB} besitzen, wenn die eine mit $S \Rightarrow A$ beginnt und mit einer Ableitung in G_A fortfährt, während die andere mit $S \Rightarrow B$ beginnt und mit einer Ableitung der gleichen Zeichenreihe in G_B fortfährt.

Die Zeichenreihe mit zwei Ableitungen endet auf Indizes $a_{i_m} \ldots a_{i_2} a_{i_1}$ für $m \geq 1$. Diese Indexfolge muss in umgekehrter Reihenfolge eine Lösung der PKP-Instanz sein, da der Anfang der Zeichenreihe mit zwei Ableitungen sowohl $w_{i_1} w_{i_2} \ldots w_{i_m}$ als auch $x_{i_1} x_{i_2} \ldots x_{i_m}$ ist. ■

9.5.3 Das Komplement einer Listensprache

Da wir nun kontextfreie Sprachen (kfS) wie L_A für die Liste A haben, können wir einige Probleme bei kfS als unentscheidbar beweisen. Wir erhalten weitere Fakten über die Unentscheidbarkeit von kontextfreien Sprachen, wenn wir die Komplementsprache $\overline{L}_A$ betrachten. Beachten Sie, dass die Sprache $\overline{L}_A$ aus allen Zeichenreihen über dem Alphabet $\Sigma \cup \{a_1, a_2, \ldots, a_k\}$ besteht, die nicht in L_A enthalten sind, wobei Σ das Alphabet einer Instanz des PKP ist und die a_i Symbole sind, die sich davon unterscheiden und die Indizes von Paaren dieser PKP-Instanz repräsentieren.

Interessante Elemente aus $\overline{L}_A$ sind solche Zeichenreihen, die ein Präfix aus Σ^* besitzen, nämlich eine Verkettung von Zeichenreihen aus der Liste A, sowie ein Suffix aus Indexsymbolen, die *nicht* zur Verkettung der Zeichenreihen aus A im Präfix passen. Allerdings enthält $\overline{L}_A$ auch viele Zeichenreihen, die einfach die falsche Form haben und z. B. nicht in der Sprache der regulären Ausdrücke $\Sigma^* (a_1 + a_2 + \ldots + a_k)^*$ enthalten sind.

Wir behaupten, $\overline{L}_A$ sei eine kfS. Im Gegensatz zu L_A ist es nicht einfach, eine Grammatik für $\overline{L}_A$ zu entwerfen, doch wir können einen PDA, genauer gesagt, einen deterministischen PDA für $\overline{L}_A$ entwerfen. Der nächste Satz zeigt die Konstruktion.

Satz 9.21 Ist L_A die Sprache für die Liste A, dann ist $\overline{L}_A$ eine kontextfreie Sprache.

Beweis: Sei Σ das Alphabet der Zeichenreihen aus Liste $A = w_1, w_2, \ldots, w_k$ und $I = \{a_1, a_2, \ldots, a_k\}$ die Menge der Indexsymbole. Wir entwerfen einen DPDA P, der $\overline{L}_A$ akzeptiert und wie folgt arbeitet:

1. Solange P Symbole aus Σ sieht, werden diese auf dem Stack gespeichert. Da alle Zeichenreihen aus Σ^* in $\overline{L}_A$ enthalten sind, werden sie im weiteren Verlauf von P akzeptiert.

2. Sobald P ein Indexsymbol aus I sieht, etwa a_i, werden Symbole vom Stack entfernt, und P prüft, ob die obersten Symbole w_i^R bilden, also die Spiegelung der korrespondierenden Zeichenreihen.

   a) Falls nicht, dann ist die bisherige Eingabe sowie jede Fortsetzung dieser Eingabe in $\overline{L}_A$ enthalten. P wechselt daher in einen akzeptierenden Zustand und verarbeitet alle weiteren Eingaben, ohne dabei den Stack zu verändern.

   b) Wurde w_i^R vom Stack entfernt, doch ist die Stackanfangsmarkierung noch nicht auf dem Stack sichtbar, dann akzeptiert P, erinnert sich in diesem Zustand jedoch daran, dass nur nach Symbolen aus I gesucht wird und eine Zeichenreihe aus L_A auftreten könnte (die P *nicht* akzeptiert). P wie-

derholt Schritt (2) so lange, wie die Frage nicht gelöst ist, ob die Eingabe in L_A enthalten ist.

c) Wurde w_i^R vom Stack entfernt und ist die Stackanfangsmarkierung auf dem Stack sichtbar, dann hat P eine in L_A enthaltene Eingabe gesehen und akzeptiert diese Eingabe nicht. Folgen allerdings weitere Eingaben, wechselt P in einen Zustand, in dem alle weiteren Eingaben akzeptiert werden und der Stack unverändert bleibt.

3. Sieht P nach einem oder mehreren Symbolen aus I ein weiteres Symbol aus Σ, dann besitzt die Eingabe nicht die korrekte Form einer Zeichenreihe aus L_A. P wechselt daher in einen Zustand, indem er diese sowie alle weiteren Eingaben akzeptiert, ohne den Stack zu verändern. ■

Wir können L_A, L_B und deren Komplemente einsetzen, um die Unentscheidbarkeit einiger Eigenschaften kontextfreier Sprachen zu zeigen. Der nächste Satz fasst einige dieser Fakten zusammen.

Satz 9.22 G_1 und G_2 seien kontextfreie Grammatiken, und R sei ein regulärer Ausdruck. Dann ist Folgendes unentscheidbar:

a) Ist $L(G_1) \cap L(G_2) = \emptyset$?

b) Ist $L(G_1) = L(G_2)$?

c) Ist $L(G_1) = L(R)$?

d) Ist $L(G_1) = T^*$ für ein Alphabet T?

e) Ist $L(G_1) \subseteq L(G_2)$?

f) Ist $L(R) \subseteq L(G_1)$?

Beweis: Jeder der Beweise ist eine Reduktion von PKP. Wir zeigen, wie eine Instanz (A, B) aus PKP in eine Frage über kontextfreie Grammatiken und/oder reguläre Ausdrücke konvertiert wird, die einzig und allein dann positiv beantwortet werden kann, wenn die PKP-Instanz eine Lösung besitzt. In einigen Fällen reduzieren wir PKP auf die Frage, wie sie im Theorem gestellt wird, in anderen Fällen dagegen in das Komplement. Dies ist nicht von Bedeutung, denn wenn wir zeigen, dass das Komplement eines Problems unentscheidbar ist, dann ist es nicht möglich, dass das Problem selbst entscheidbar ist, da die rekursiven Sprachen unter dem Komplement abgeschlossen sind (Satz 9.3).

Wir bezeichnen das Alphabet der Zeichenreihen dieser Instanz als Σ und das Alphabet der Indexsymbole als I. Unsere Reduktionen basieren darauf, dass L_A, L_B, $\overline{L}_A$ und $\overline{L}_B$ alle kontextfreie Grammatiken besitzen. Wir konstruieren diese Grammatiken entweder wie in Abschnitt 9.5.2 direkt oder durch die Konstruktion eines PDA für die in Satz 9.21 verwendeten Komplementsprachen zusammen mit der Konvertierung eines PDA in eine kfG (Satz 6.14).

a) Sei $L(G_1) = L_A$ und $L(G_2) = L_B$; dann ist $L(G_1) \cap L(G_2)$ die Menge der Lösungen dieser PKP-Instanz. Der Durchschnitt ist dann und nur dann leer, wenn keine Lösung existiert. Beachten Sie, dass wir PKP technisch gesehen auf die Sprache der Paare von kontextfreien Grammatiken reduziert haben, deren Durch-

schnitt nicht leer ist; wir haben also gezeigt, dass das Problem »Ist der Durchschnitt von zwei kontextfreien Grammatiken nicht leer?« unentscheidbar ist. Wie allerdings in der Einführung des Beweises erwähnt, ist der Beweis, dass das Komplement eines Problems unentscheidbar ist, gleichbedeutend mit dem Beweis, dass das Problem selbst unentscheidbar ist.

b) Da kontextfreie Grammatiken unter der Vereinigung abgeschlossen sind, können wir für $\overline{L}_A \cup \overline{L}_B$ eine kfG G_3 konstruieren. Da $(\Sigma \cup I)^*$ eine reguläre Menge ist, können wir dafür mit Sicherheit eine kfG G_4 konstruieren. Nun gilt: $\overline{L}_A \cup \overline{L}_B = \overline{L_A \cap L_B}$. $L(G_3)$ fehlen daher alle Zeichenreihen, die Lösungen der PKP-Instanz repräsentieren. $L(G_4)$ dagegen fehlen in $(\Sigma \cup I)^*$ keine Zeichenreihen. Daher sind deren Sprachen nur dann gleich, wenn die PKP-Instanz keine Lösung besitzt.

c) Die Beweisführung ist mit b) identisch, jedoch sei R der reguläre Ausdruck $(\Sigma \cup I)^*$.

d) Die Beweisführung von c) ist ausreichend, da $\Sigma \cup I$ das einzige Alphabet ist, von dem $\overline{L}_A \cup \overline{L}_B$ die Hülle sein könnte.

e) G_4 sei eine kfG für $(\Sigma \cup I)^*$, und G_3 sei eine kfG für $\overline{L}_A \cup \overline{L}_B$. Dann gilt $L(G_4) \subseteq L(G_3)$, jedoch nur dann, wenn $\overline{L}_A \cup \overline{L}_B = (\Sigma \cup I)^*$, also nur dann, wenn die PKP-Instanz keine Lösung besitzt.

f) Die Beweisführung ist mit e) identisch, jedoch sei R der reguläre Ausdruck $(\Sigma \cup I)^*$, und $L(G_4)$ sei $\overline{L}_A \cup \overline{L}_B$. ■

9.5.4 Übungen zum Abschnitt 9.5

* **Übung 9.5.1** Sei L die Menge der (Codes zu) kontextfreien Grammatiken G, sodass $L(G)$ mindestens ein Palindrom enthält. Zeigen Sie, dass L unentscheidbar ist. *Hinweis*: Reduzieren Sie PKP auf L, indem Sie aus jeder Instanz des PKP eine Grammatik konstruieren, deren Sprache dann und nur dann ein Palindrom enthält, wenn die PKP-Instanz eine Lösung besitzt.

! **Übung 9.5.2** Zeigen Sie, dass die Sprache $\overline{L}_A \cup \overline{L}_B$ nur dann eine reguläre Sprache ist, wenn sie die Menge aller Zeichenreihen über ihrem Alphabet ist, wenn also die Instanz (A, B) des PKP keine Lösung besitzt. Beweisen Sie so, dass unentscheidbar ist, ob eine kfG eine reguläre Sprache generiert oder nicht. *Hinweis*: Angenommen, PKP besäße eine Lösung und eine Zeichenreihe wx würde dementsprechend in $\overline{L}_A \cup \overline{L}_B$ fehlen, wobei w eine Zeichenreihe aus dem Alphabet Σ dieser PKP-Instanz und x die Spiegelung der entsprechenden Zeichenreihen aus Indexsymbolen ist. Definieren Sie einen Homomorphismus $h(0) = w$ und $h(1) = x$. Was ist dann $h^{-1}(\overline{L}_A \cup \overline{L}_B)$? Verwenden Sie die Tatsache, dass reguläre Mengen unter inversen Homomorphismen und Komplementbildung abgeschlossen sind, sowie das Pumping-Lemma für reguläre Mengen, um zu zeigen, dass $\overline{L}_A \cup \overline{L}_B$ nicht regulär ist.

!! **Übung 9.5.3** Es ist unentscheidbar, ob das Komplement einer kfS ebenfalls eine kfS ist. Die Übung 9.5.2 kann zum Beweis verwendet werden, dass unentscheidbar ist, ob das Komplement einer kfS regulär ist, doch dies ist nicht das Gleiche. Um unsere ursprüngliche Behauptung zu beweisen, müssen wir eine andere Sprache definieren,

die die Nichtlösungen einer Instanz (A, B) des PKP repräsentiert. Sei L_{AB} die Menge der Zeichenreihen von der Form $w\#x\#y\#z$, sodass Folgendes gilt:

1. w und x sind Zeichenreihen über dem Alphabet Σ der PKP-Instanz.
2. y und z sind Zeichenreihen über dem Indexalphabet I dieser Instanz.
3. # ist ein Symbol, das weder in Σ noch in I enthalten ist.
4. $w \neq x^R$.
5. $y \neq z^R$.
6. x^R entspricht *nicht* dem, was die Indexzeichenreihe y gemäß der Liste B generiert.
7. w entspricht nicht dem, was die Indexzeichenreihe z^R gemäß der Liste A generiert.

Beachten Sie, dass L_{AB} aus allen Zeichenreihen in $\Sigma^*\#\Sigma^*\#I^*\#I^*$ besteht, sofern die Instanz (A, B) keine Lösung besitzt, doch $\overline{L}_{AB}$ ist nichtsdestoweniger eine kfS. Beweisen Sie, dass $\overline{L}_{AB}$ nur dann eine kfS ist, wenn keine Lösung existiert. *Hinweis*: Verwenden Sie den Trick des inversen Homomorphismus aus Übung 9.5.2 und Ogdens Lemma, um wie im Hinweis zu Übung 7.2.5 (b) Gleichheit der Längen bestimmter Teilzeichenreihen zu erzwingen.

9.6 Zusammenfassung von Kapitel 9

- *Rekursive und rekursiv aufzählbare Sprachen* : Die Sprachen, die von Turing-Maschinen akzeptiert werden, werden rekursiv aufzählbar genannt, und die Teilmenge der rekursiv aufzählbaren Sprachen, die von einer Turing-Maschine akzeptiert werden, die immer anhält, wird rekursiv genannt.
- *Komplemente von rekursiven und rekursiv aufzählbaren Sprachen* : Die rekursiven Sprachen sind unter der Komplementbildung abgeschlossen. Ist eine Sprache sowie deren Komplement rekursiv aufzählbar, dann sind beide Sprachen sogar rekursiv. Daher kann das Komplement einer zwar rekursiv aufzählbaren, jedoch nicht rekursiven Sprache niemals rekursiv aufzählbar sein.
- *Entscheidbarkeit und Unentscheidbarkeit*: »Entscheidbar« ist ein Synonym für »rekursiv«, doch wir tendieren dazu, Sprachen als »rekursiv« und Probleme (wobei es sich um Sprachen handelt, die als Fragen interpretiert werden) als »entscheidbar« zu bezeichnen. Ist eine Sprache nicht rekursiv, dann nennen wir das von dieser Sprache ausgedrückte Problem »unentscheidbar«.
- *Die Sprache L_d*: Diese Sprache ist die Menge der Zeichenreihen aus den Symbolen 0 und 1, die bei der Interpretation als TM *nicht* in der Sprache dieser TM enthalten sind. Die Sprache L_d ist ein gutes Beispiel für eine Sprache, die nicht rekursiv aufzählbar ist; das heißt L_d wird von keiner Turing-Maschine akzeptiert.
- *Die universelle Sprache*: Die Sprache L_u besteht aus Zeichenreihen, die als eine TM M gefolgt von einer Eingabe w für diese TM interpretiert werden. Eine Zeichenreihe ist genau dann in L_u enthalten, wenn die TM M die Eingabe w akzeptiert. L_u ist ein gutes Beispiel für eine Sprache, die rekursiv aufzählbar, jedoch nicht rekursiv ist.

- *Der Satz von Rice*: Jede nicht triviale Eigenschaft der Sprachen, die von Turing-Maschinen akzeptiert werden, ist unentscheidbar. Beispielsweise ist die Menge der Codes für Turing-Maschinen, deren Sprache leer ist, nach dem Satz von Rice unentscheidbar. Tatsächlich ist diese Sprache nicht rekursiv aufzählbar; dagegen ist ihr Komplement – die Menge der Codes von Turing-Maschinen, die mindestens eine Zeichenreihe akzeptieren – rekursiv aufzählbar, jedoch nicht rekursiv.
- *Das Postsche Korrespondenzproblem*: Dieses Problem fragt, ob aus zwei Listen, die die gleiche Anzahl von Zeichenreihen enthalten, eine Folge korrespondierender Zeichenreihen entnommen und so aus beiden Listen jeweils durch Verkettung die gleiche Zeichenreihe gebildet werden kann. PKP ist ein wichtiges Beispiel für unentscheidbare Probleme. PKP ist auch eine gute Wahl, um auf andere Probleme zu reduzieren und sie damit als unentscheidbar nachzuweisen.
- *Unentscheidbare Probleme kontextfreier Sprachen*: Durch eine Reduktion von PKP können wir zeigen, dass eine Reihe von Fragen zu kontextfreien Sprachen oder deren Grammatiken unentscheidbar ist. Beispielsweise ist unentscheidbar, ob eine kfG mehrdeutig ist, ob eine kfS in einer anderen enthalten ist oder ob der Durchschnitt von zwei kontextfreien Sprachen leer ist.

LITERATURANGABEN ZU KAPITEL 9

Die Unentscheidbarkeit der universellen Sprache ist im Wesentlichen ein Ergebnis von Turing [9], allerdings wurde dies dort durch die Berechnung arithmetischer Funktionen und Anhalten statt durch Sprachen und Akzeptanz durch einen Endzustand ausgedrückt. Der Satz von Rice stammt aus [8].

Die Unentscheidbarkeit des Postschen Korrespondenzproblems wurde in [7] gezeigt, der hier verwendete Beweis stammt jedoch aus unveröffentlichten Notizen von R. W. Floyd. Die Unentscheidbarkeit von *post tag systems* (in Übung 9.4.4 definiert) ist [6] entnommen.

[1] und [5] sind die grundlegenden Veröffentlichungen über die Unentscheidbarkeit von Fragen zu kontextfreien Sprachen. Die Tatsache, dass unentscheidbar ist, ob eine kfG mehrdeutig ist, wurde unabhängig von Cantor [2], Floyd [4] sowie Chomsky und Schützenberger [3] entdeckt.

1. Y. Bar-Hillel, M. Perles und E. Shamir [1961]. »On formal properties of simple phrase-structure grammars«, *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* **14**, 143–172.
2. D. C. Cantor [1962]. »On the ambiguity problem in Backus systems«, *J. ACM* **9**:4, 477–479.
3. N. Chomsky und M. P. Schützenberger [1963]. »The algebraic theory of context-free languages«, *Computer Programming and Formal Systems*, 118–161, North Holland, Amsterdam.
4. R. W. Floyd [1962]. »On ambiguity in phrase structure languages«, *Communications of the ACM* **5**:10, 526–534.
5. S. Ginsburg, und G. F. Rose [1963]. »Some recursively unsolvable problems in ALGOL-like languages«, *J. ACM* **10**:1, 29–47.
6. M. L. Minsky [1961]. »Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines«, *Annals of Mathematics* **74**:3, 437–455.

→

7. E. Post [1946]. »A variant of a recursively unsolvable problem«, *Bulletin of the AMS* **52**, 264–268.

8. H. G. Rice [1953]. »Classes of recursively enumerable sets and their decision problems«, *Transactions of the AMS* **89**, 25–59.

9. A. M. Turing [1936]. »On computable numbers with an application to the Entscheidungsproblem«, *Proc. London Math Society* **2**:42, 230–265.

KAPITEL 10

Nicht handhabbare Probleme

Unsere Diskussion der Frage, was berechnet werden kann und was nicht, wird nun auf die Effizienz konzentriert. Wir untersuchen dazu entscheidbare Probleme und fragen, welche dieser Probleme von Turing-Maschinen berechnet werden können, die in der Länge der Eingabe polynomiale Ausführungszeit benötigen. Sie sollten sich dazu zwei wichtige Aspekte aus Abschnitt 8.6.3 in Erinnerung rufen:

- Probleme, die von einem typischen Computer in polynomialer Zeit lösbar sind, sind exakt die Probleme, die auch von einer Turing-Maschine in polynomialer Zeit gelöst werden.
- Die Erfahrung hat gezeigt, dass die Grenze zwischen Problemen, die in polynomialer Zeit lösbar sind, und solchen, die einen exponentiellen oder noch höheren Zeitaufwand erfordern, grundlegend ist. Praktische Probleme, die einen polynomialen Aufwand erfordern, sind fast immer in einem Zeitrahmen lösbar, den wir tolerieren können, während solche, die einen exponentiellen Aufwand erfordern, mit Ausnahme kleiner Instanzen nicht lösbar sind.

Dieses Kapitel bietet eine Einführung in die Theorie der »Nichthandhabbarkeit« (engl. *intractablility*), d. h. Techniken zur Beweisführung, dass gewisse Probleme nicht in polynomialer Zeit lösbar sind. Wir beginnen mit einem speziellen Problem: der Frage, ob ein Boolescher Ausdruck *erfüllt* werden kann, also bei einer bestimmten Zuweisung der Wahrheitswerte FALSE und TRUE an die Variablen wahr ist. Dieses Problem spielt bei nicht handhabbaren Problemen die gleiche Rolle wie L_u oder PKP bei unentscheidbaren Problemen. Wir beginnen mit dem »Satz von Cook«, der aussagt, dass die Erfüllbarkeit Boolescher Formeln nicht in polynomialer Zeit entschieden werden kann. Danach zeigen wir, wie dieses Problem auf viele weitere Probleme reduziert werden kann, die damit ebenfalls als nicht behandelbar bewiesen werden.

Da im Mittelpunkt steht, ob Probleme in polynomialer Zeit lösbar sind, muss der Begriff der Reduktion geändert werden. Es reicht nicht mehr aus, dass es einen Algorithmus gibt, der Instanzen eines Problems in Instanzen eines anderen Problems transformiert. Der Algorithmus selbst darf nur polynomiale Zeit benötigen, sonst können wir nicht über die Reduktion darauf schließen, dass das Zielproblem nicht behandelbar ist, auch wenn das Quellproblem diese Eigenschaft besitzt. Im ersten Abschnitt führen wir daher den Begriff der »Reduktion mit polynomialem Zeitaufwand« ein.

Es gibt einen weiteren wichtigen Unterschied zwischen unseren Schlussfolgerungen in der Theorie der Unentscheidbarkeit und denjenigen, die uns die Theorie der

Nichthandhabbarkeit erlaubt. Die Beweise der Unentscheidbarkeit aus Kapitel 9 sind unanfechtbar; sie basieren lediglich auf der Definition einer Turing-Maschine und allgemeiner Mathematik. Im Gegensatz dazu basieren die hier vorgestellten Resultate zu nicht handhabbaren Problemen auf einer nicht bewiesenen Annahme, die jedoch eine hohe Glaubwürdigkeit besitzt. Diese Annahme wird häufig durch $\mathcal{P} \neq \mathcal{NP}$ ausgedrückt.

Das heißt wir nehmen an, dass die Klasse der Probleme, die von nichtdeterministischen TMs in polynomialer Zeit lösbar sind, zumindest einige Probleme enthält, die nicht von deterministischen TMs in polynomialer Zeit lösbar sind (auch wenn wir der deterministischen TM einen höheren Grad an polynomialer Zeit zugestehen). Es gibt buchstäblich viele tausend Probleme, die dieser Kategorie anzugehören scheinen, da sie leicht in polynomialer Zeit von einer NTM gelöst werden können, doch keine DTM (oder Computerprogramm, was das Gleiche ist) bekannt ist, die es in polynomialer Zeit lösen könnte. Überdies lautet eine wichtige Konsequenz aus der Theorie der Nichthandhabbarkeit, dass entweder alle diese Probleme deterministisch-polynomiale Lösungen besitzen (die bisher nicht entdeckt wurden) oder keines dieser Probleme ist deterministisch-polynomial; sie erfordern dann tatsächlich einen exponentiellen Zeitaufwand.

10.1 Die Klassen $\mathcal{P}$ und $\mathcal{NP}$

Dieser Abschnitt stellt die grundlegenden Konzepte der Theorie der Nichthandhabbarkeit vor: Die Klassen $\mathcal{P}$ und $\mathcal{NP}$ der Probleme, die von deterministischen bzw. nichtdeterministischen TMs in polynomialer Zeit lösbar sind, sowie die Technik der Reduktion in polynomialer Zeit. Wir definieren außerdem den Begriff der »NP-Vollständigkeit«, eine Eigenschaft, die einige Probleme in $\mathcal{NP}$ *haben*; sie sind mindestens so hart (in Hinsicht auf polynomialen Zeitaufwand) wie jedes beliebige Problem in $\mathcal{NP}$.

10.1.1 Mit polynomialem Zeitaufwand lösbare Probleme

Eine Turing-Maschine M besitzt die *Zeitkomplexität* $T(n)$ (oder die »Ausführungszeit $T(n)$«), wenn M bei gegebener Eingabe w mit der Länge n immer nach höchstens $T(n)$ Bewegungen anhält, unabhängig davon, ob M akzeptiert oder nicht. Diese Definition kann auf jede Funktion $T(n)$ wie etwa $T(n) = 50n^2$ oder $T(n) = 3^n + 5n^4$ angewendet werden; wir interessieren uns bevorzugt für den Fall, in dem $T(n)$ in n polynomial ist. Wir sagen, eine Sprache L ist in der Klasse $\mathcal{P}$ enthalten, wenn ein polynomiales $T(n)$ existiert, sodass $L = L(M)$ für eine deterministische TM M mit der Zeitkomplexität $T(n)$ gilt.

Gibt es etwas zwischen polynomial und exponentiell?

In der Einführung und auch in den folgenden Ausführungen tun wir häufig so, als sei die Ausführungszeit aller Programme entweder polynomial ($O(n^k)$ für eine ganze Zahl k) oder exponentiell ($O(2^{cn})$ für eine Konstante $c > 0$) bzw. höher. In der Praxis lassen sich die bekannten Algorithmen für allgemeine Probleme gewöhnlich in eine dieser beiden Kategorien einordnen. Allerdings gibt es auch Ausführungszeiten, die zwischen polynomial und exponentiell einzuordnen sind. Wir verwenden exponentiell in dem Sinn, dass damit »jede Ausführungsdauer gemeint ist, die größer ist als polynomial«.

→

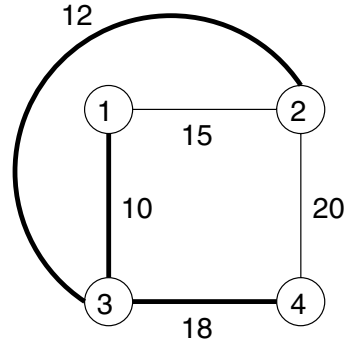
→ $n^{\log_2 n}$ ist ein Beispiel für eine Funktion mit einer Ausführungszeit, die zwischen exponentiell und polynomial liegt. Diese Funktion wächst in n schneller als jedes Polynom, da log n (für große n) schließlich größer als jede Konstante k wird. Auf der anderen Seite gilt $n^{\log_2 n} = 2^{(\log_2 n)^2}$ ($n = 2^{\log_2 n}$). Diese Funktion wächst langsamer als 2^{cn} für jedes $c > 0$. Wie klein die positive Konstante c auch ist, cn wird schließlich auf jeden Fall größer als $(\log_2 n)^2$.

10.1.2 Beispiel: Der Kruskal-Algorithmus

Sie sind wahrscheinlich mit vielen Problemen vertraut, die effiziente Lösungen besitzen. Zum Beispiel haben Sie einige in einer Vorlesung über Datenstrukturen und Algorithmen kennen gelernt. Diese Probleme sind im Allgemeinen in $\mathcal{P}$ enthalten. Wir werden eines dieser Probleme betrachten: die Suche nach einem einen Graphen aufspannenden Baum mit minimalem Gewicht (Minimum Weight Spanning Tree, *MWST*).

Informell stellen wir uns Graphen als Diagramme wie dasjenige in Abbildung 10.1 vor. Der Graph enthält Knoten, die im Beispiel von 1 bis 4 durchnummeriert sind, sowie Kanten zwischen einigen Knotenpaaren. Jede Kante hat ein *Gewicht*, durch eine ganze Zahl gekennzeichnet. Ein *aufspannender Baum* ist eine Teilmenge der Kanten, sodass alle Knoten durch diese Kanten miteinander verbunden sind; es gibt jedoch keine Zyklen. Abbildung 10.1 zeigt ein Beispiel für einen aufspannenden Baum; er besteht aus den drei fetten Kanten. Ein aufspannender Baum mit *minimalem Gewicht* hat das kleinstmögliche Gesamtkantengewicht aller aufspannenden Bäume.

Abbildung 10.1: Ein Graph; der aufspannende Baum mit minimalem Gewicht ist durch fette Linien hervorgehoben



Ein bekannter »sparsamer (greedy)« Algorithmus, der *Kruskal-Algorithmus*[1], dient zur Suche nach einem MWST. Hier nun eine informelle Beschreibung des grundlegenden Gedankengangs:

1. Kruskal, J. B. [1956]. »On the shortest spanning subtree of a graph and the travelling salesman problem«, *Proc. AMS* **7**:1, 48–50.

1. Speichere für jeden Knoten die *Zusammenhangskomponente*, in der der Knoten erscheint, und verwende dazu alle bisher ausgewählten Kanten des Baumes. Anfänglich sind keine Kanten ausgewählt, und somit steht jeder Knoten in einer Zusammenhangskomponente für sich allein.
2. Wähle eine Kante mit geringstem Gewicht aus denen, die bisher noch nicht betrachtet wurden. Verbindet diese Kante zwei Knoten, die sich zum aktuellen Zeitpunkt in verschiedenen Zusammenhangskomponenten befinden, dann
   a) wähle die Kante für den aufspannenden Baum und
   b) vereinige die beiden betroffenen Zusammenhangskomponenten durch die Änderung der Komponentennummer aller Knoten in beiden Zusammenhangskomponenten, sodass ihre Nummern übereinstimmen.

   Verbindet die gewählte Kante jedoch zwei Knoten der gleichen Zusammenhangskomponente, dann gehört sie nicht in den aufspannenden Baum; diese Kante würde zu einem Zyklus führen.
3. Betrachte weiterhin Kanten, bis entweder alle Kanten berücksichtigt wurden oder die Anzahl der für den aufspannenden Baum ausgewählten Kanten um 1 niedriger als die Anzahl der Knoten ist. Beachten Sie, dass im zweiten Fall alle Knoten in der gleichen Zusammenhangskomponente liegen und wir daher keine weiteren Kanten berücksichtigen müssen.

Beispiel 10.1 Im Graph aus Abbildung 10.1 betrachten wir zuerst die Kante (1, 3), da sie mit 10 das geringste Gewicht hat. Da sich 1 und 3 anfänglich in verschiedenen Komponenten befinden, akzeptieren wir diese Kante und weisen 1 und 3 die gleiche Komponentennummer zu, beispielsweise »Komponente 1«. Die nächste Kante ist (2, 3) mit dem Gewicht 12. Da sich 2 und 3 in verschiedenen Komponenten befinden, akzeptieren wir diese Kante und nehmen Knoten 2 in »Komponente 1« auf. Die dritte Kante ist (1, 2) mit dem Gewicht 15. Allerdings befinden sich 1 und 2 schon in der gleichen Komponente, und daher weisen wir diese Kante zurück und fahren mit der vierten Kante (3, 4) fort. Da 4 nicht in »Komponente 1« vorhanden ist, akzeptieren wir diese Kante. Nun haben wir drei Kanten des aufspannenden Baumes eines Graphen mit 4 Knoten und beenden daher die Auswahl. ■

Dieser Algorithmus kann (unter Verwendung eines Computers, nicht aber einer Turing-Maschine) für einen Graphen mit m Knoten und e Kanten im Zeitraum $O(m + e \log e)$ implementiert werden. Eine einfachere, leichter zu verfolgende Implementierung führt e Durchläufe aus. Die aktuelle Zusammenhangskomponente jedes Knotens wird in einer Tabelle gespeichert. Wir wählen die aktuelle Kante mit dem geringsten Gewicht in der Zeit $O(e)$ und finden die Zusammenhangskomponenten der beiden durch diese Kante verbundenen Knoten in der Zeit $O(m)$. Befinden sie sich in verschiedenen Zusammenhangskomponenten, werden alle Knoten aus diesen beiden über das Durchsuchen der Knotentabelle in der Zeit $O(m)$ vereinigt. Dieser Algorithmus benötigt die Gesamtzeit $O(e(e + m))$. Diese Ausführungszeit ist in Bezug auf die »Größe« der Eingabe, die wir informell als die Summe aus e und m annehmen, polynomial.

Wenn wir diese Ideen auf Turing-Maschinen übertragen, müssen wir uns mit verschiedenen Themen befassen:

- Wenn wir Algorithmen studieren, treffen wir auf »Probleme«, die unterschiedliche Ausgaben erfordern, beispielsweise eine Liste der Kanten eines MWST. Bei Turing-Maschinen stellen wir uns Probleme lediglich als Sprachen vor, und die einzig mögliche Ausgabe besteht aus `yes` oder `no`, also aus Akzeptieren oder Zurückweisen. Beispielsweise könnte das MWST-Problem folgendermaßen ausgedrückt werden: »Gegeben seien der Graph G und der Grenzwert W; besitzt G einen aufspannenden Baum mit dem Gewicht W oder weniger?« Dieses Problem scheint einfacher zu beantworten zu sein als das MWST-Problem, mit dem wir vertraut sind, da der aufspannende Baum nicht konstruiert werden muss. Allerdings wollen wir in der Theorie der Nichthandhabbarkeit im Allgemeinen argumentieren, dass ein Problem hart und nicht einfach ist. Die Tatsache, dass eine Ja-Nein-Version eines Problems hart ist, impliziert, dass eine eher standardisierte Version, für die die vollständige Antwort berechnet werden muss, ebenfalls hart ist.
- Wir können uns die »Größe« eines Graphen als die Anzahl seiner Knoten oder Kanten vorstellen, doch die Eingabe einer TM ist eine Zeichenreihe über einem endlichen Alphabet. Problemelemente wie Knoten und Kanten sind daher passend zu kodieren. Diese Anforderung wirkt sich dahingehend aus, dass Eingaben von Turing-Maschinen generell länger als die intuitive »Größe« der Eingabe sind. Allerdings sprechen zwei Gründe dafür, dass die Differenz nicht signifikant ist:

  1. Der Unterschied zwischen der Größe als TM-Eingabezeichenreihe und als informelles Problem überschreitet niemals einen kleinen Faktor, gewöhnlich den Logarithmus der Eingabegröße. Was daher mit der einen Methode in polynomialer Zeit erledigt werden kann, ist mit der anderen Methode ebenso in polynomialer Zeit auszuführen.
  2. Die Länge einer Zeichenreihe, die die Eingabe repräsentiert, ist tatsächlich eine genauere Angabe für die Anzahl der Bytes, die ein realer Computer lesen muss, um seine Eingabe zu bekommen. Wird ein Knoten beispielsweise durch eine ganze Zahl repräsentiert, dann ist die Anzahl der Bytes, die zur Repräsentation dieser ganzen Zahl notwendig sind, proportional zum Logarithmus der Größe dieser ganzen Zahl und nicht »1 Byte für jeden Knoten«, wie wir bei einer informellen Betrachtung der Eingabegröße annehmen könnten.

Beispiel 10.2 Sehen wir uns einen möglichen Code für die Graphen und Gewichtsgrenzen an, die als Eingabe für das MWST-Problem dienen könnten. Der Code besitzt die fünf Symbole 0, 1, die linke und die rechte Klammer sowie das Komma.

1. Weise den Knoten die ganzen Zahlen 1 bis m zu.
2. Der Code beginnt mit dem binären Wert von m und der binären Gewichtsgrenze W, mit einem Komma als Trennzeichen zwischen den beiden Werten.
3. Existiert eine Kante zwischen den Knoten i und j mit dem Gewicht w, dann nehme (i, j, w) in den Code auf. Die ganzen Zahlen i, j und w sind binär kodiert. Die Reihenfolge von i und j innerhalb einer Kante sowie die Reihenfolge der Kanten innerhalb des Codes ist belanglos.

Daher lautet einer der möglichen Codes für den in Abbildung 10.1 dargestellten Graphen mit Gewicht $W = 40$:

100,101000(1,10,1111)(1,11,1010)(10,11,1100)(10,100,10100)(11,100,10010) ■

Repräsentieren wir Eingaben des MWST-Problems wie in Beispiel 10.2, dann kann eine Eingabe der Länge n höchstens $O(n/\log n)$ Kanten repräsentieren. Möglicherweise ist m, die Anzahl der Knoten, in n exponentiell, wenn nur wenige Kanten existieren. Falls die Anzahl e der Kanten allerdings nicht mindestens $m - 1$ ist, kann der Graph nicht zusammenhängend sein und besitzt damit unabhängig von seinen Kanten keinen MWST. Daraus folgt, dass der Kruskal-Algorithmus nicht verwendet werden muss, wenn die Anzahl der Knoten nicht mindestens einem Bruchteil von $n/\log n$ entspricht; wir sagen einfach: »Nein, es existiert kein aufspannender Baum mit diesem Gewicht.«

Haben wir also für die Ausführungszeit des Kruskal-Algorithmus eine obere Schranke in Form einer Funktion von m und e, etwa die oben entwickelte Schranke $O(e(e + m))$, dann können wir sowohl e als auch m konservativ durch n ersetzten. Die Ausführungszeit als Funktion der Eingabelänge n ist dann $O(n(n + n))$ oder $O(n^2)$. Tatsächlich benötigt eine bessere Implementierung des Kruskal-Algorithmus die Ausführungszeit $O(n \log n)$, doch wir werden diese Verbesserung an dieser Stelle nicht berücksichtigen.

Wir setzen natürlich eine Turing-Maschine als unser Berechnungsmodell ein, während der oben beschriebene Algorithmus in einer Programmiersprache mit nützlichen Datenstrukturen wie Arrays und Zeiger implementiert werden sollte. Wir behaupten jedoch, die oben beschriebene Version des Kruskal-Algorithmus kann in $O(n^2)$ Schritten auf einer mehrbändigen TM implementiert werden. Die zusätzlichen Bänder erfüllen verschiedene Aufgaben:

1. Ein Band speichert die Knoten und deren aktuelle Zusammenhangskomponenten-Nummern. Die Länge dieser Tabelle beträgt $O(n)$.

2. Ein Band speichert das aktuelle geringste Kantengewicht aller Kanten, die noch nicht als »benutzt« markiert sind, während wir die Kanten auf dem Eingabeband durchsuchen. Wir könnten eine zweite Spur des Eingabebandes verwenden, um die Kanten zu markieren, die in einem früheren Durchgang des Algorithmus ausgewählt wurden. Die Suche nach der unmarkierten Kante mit dem geringsten Gewicht erfordert einen Zeitaufwand von $O(n)$, da jede Kante nur einmal ausgewertet wird, und der Gewichtsvergleich kann über ein lineares Durchsuchen der ganzen Zahlen von rechts nach links erfolgen.

3. Wird eine Kante während eines Durchlaufs ausgewählt, dann werden deren Knoten auf einem Band gespeichert. Wir durchsuchen die Tabelle der Knoten und Zusammenhangskomponenten nach den Zusammenhangskomponenten dieser beiden Knoten. Diese Aufgabe erfordert einen Zeitaufwand von $O(n)$.

4. Ein Band speichert die beiden Zusammenhangskomponenten i und j, die zusammengefasst werden, wenn eine Kante zwischen den beiden bisher nicht verbundenen Zusammenhangskomponenten entdeckt wird. Wir durchsuchen dann die Tabelle der Knoten und Zusammenhangskomponenten und ändern die Zusammenhangskomponenten-Nummer jedes Knotens in Zusammenhangskomponente i in die neue Nummer j. Auch diese Aufgabe erfordert einen Zeitaufwand von $O(n)$.

Sie sollten nun die Behauptung, dass ein Durchlauf auf einer mehrbändigen TM in der Zeit $O(n)$ ausgeführt werden kann, abschließen können. Da die Anzahl der Durchläufe, e, höchstens n beträgt, schließen wir, dass der Zeitaufwand auf einer mehrbändigen TM $O(n^2)$ ist. Satz 8.10 sagt aus, dass alles, was eine mehrbändige TM in s Schritten ausführen kann, ebenso von einer einbändigen TM in $O(s^2)$ Schritten ausgeführt werden kann. Benötigt die mehrbändige TM daher $O(n^2)$ Schritte, dann können wir eine einbändige TM konstruieren, die die gleiche Aufgabe in $O((n^2)^2) = O(n^4)$ ausführt. Wir folgern, dass die Ja-Nein-Version des MWST-Problems »Besitzt G einen MWST mit Gesamtgewicht W oder weniger?« in $\mathcal{P}$ enthalten ist.

10.1.3 Nichtdeterministischer polynomialer Zeitaufwand

Bei der Untersuchung der Nichtbehandelbarkeit bilden solche Probleme eine wichtige Klasse, die von einer nichtdeterministischen TM in polynomialer Zeit gelöst werden können. Formal ausgedrückt ist eine Sprache L in der Klasse $\mathcal{NP}$ (nichtdeterministisch polynomial) enthalten, wenn eine nichtdeterministische TM M und eine polynomiale Zeitkomplexität $T(n)$ existieren, sodass $L = L(M)$ gilt, und wenn M eine Eingabe der Länge n erhält, gibt es keine Folgen mit mehr als $T(n)$ Bewegungen von M.

Zunächst beobachten wir, dass $\mathcal{P} \subseteq \mathcal{NP}$, da jede deterministische TM eine nichtdeterministische TM ist, die nie eine Auswahlmöglichkeit bei den Bewegungen hat. Anscheinend enthält $\mathcal{NP}$ jedoch viele Probleme, die nicht in $\mathcal{P}$ enthalten sind. Die Ursache liegt darin, dass eine NTM mit polynomialer Ausführungszeit die Fähigkeit besitzt, eine exponentielle Anzahl möglicher Lösungen eines Problems zu raten und jede dieser Lösungen in polynomialer Zeit, »parallel«, prüfen kann. Allerdings ist Folgendes zu beachten:

- Eine der tiefgreifendsten offenen Fragen der Mathematik lautet, ob $\mathcal{P} = \mathcal{NP}$ ist, ob also tatsächlich alles, was eine NTM in polynomialer Zeit erledigt, ebenso von einer DTM in polynomialer Zeit (vielleicht höheren Grades) erledigt werden kann.

10.1.4 Ein $\mathcal{NP}$-Beispiel: Das Problem des Handlungsreisenden

Damit Sie ein Gefühl für die Leistungsfähigkeit von $\mathcal{NP}$ bekommen, werden wir uns ein Beispiel für ein Problem ansehen, das in $\mathcal{NP}$, anscheinend jedoch nicht in $\mathcal{P}$ enthalten ist: das *Problem des Handlungsreisenden* (Traveling Salesman Problem, TSP). Die Eingabe des TSP stimmt mit der des MWST überein, ein Graph wie der in Abbildung 10.1 mit ganzzahligen Gewichtungen der Kanten sowie einer Gewichtsgrenze W. Hier wird gefragt, ob der Graph einen »Hamiltonschen Kreis« mit einem Gesamtgewicht von höchstens W besitzt. Ein *Hamiltonscher Kreis* ist eine Menge von Kanten, die die Knoten in einem einzigen Zyklus verbinden, wobei jeder Knoten exakt einmal enthalten ist. Beachten Sie, dass die Anzahl der Kanten eines Hamiltonschen Kreises mit der Gesamtzahl der Knoten im Graphen übereinstimmen muss.

Beispiel 10.3 Der in Abbildung 10.1 dargestellte Graph besitzt nur einen Hamiltonschen Kreis, den Pfad (1, 2, 4, 3, 1). Das Gesamtgewicht dieses Pfades ist 15 + 20 + 18 + 10 = 63. Ist W daher 63 oder höher, dann lautet die Antwort »ja«, falls jedoch $W < 63$, lautet die Antwort »nein«.

Allerdings ist das TSP bei Graphen mit vier Knoten irreführend einfach, da es nie mehr als zwei verschiedene Hamiltonsche Kreise geben kann, wenn wir von den unterschiedlichen Knoten, an denen der Pfad beginnen kann, sowie von der Richtung absehen, in der wir den Pfad durchlaufen. Bei Graphen mit m Knoten wächst die Zahl

Eine Variante nichtdeterministischer Akzeptanz

Beachten Sie, dass wir von unserer NTM fordern, entlang aller Zweige nach polynomialer Dauer anzuhalten, ob sie nun akzeptiert oder nicht. Wir hätten die polynomiale Zeitgrenze $T(n)$ genauso gut nur den Zweigen auferlegen können, die zur Akzeptanz führen; wir hätten also $\mathcal{NP}$ als die Sprachen definiert, die von einer NTM akzeptiert werden, sodass im Akzeptanzfall mindestens eine Folge von höchstens $T(n)$ Bewegungen für ein polynomiales $T(n)$ existiert.

Allerdings hätten wir damit die gleiche Sprachklasse erhalten. Da wir wissen, dass M innerhalb von $T(n)$ Bewegungen akzeptiert, falls sie überhaupt akzeptiert, könnten wir M modifizieren, sodass auf einer separaten Spur des Bandes ein Zähler bis $T(n)$ zählt. M würde dann anhalten, ohne zu akzeptieren, wenn dieser Grenzwert überschritten wird. Die modifizierte M könnte $O(T^2(n))$ Schritte erfordern, doch $T^2(n)$ ist polynomial, wenn $T(n)$ polynomial ist.

Tatsächlich hätten wir auch $\mathcal{P}$ über die Akzeptanz von TMs definieren können, die innerhalb des Zeitraums $T(n)$ für polynomiale $T(n)$ akzeptieren. Unter Umständen halten solche TMs nicht an, wenn sie nicht akzeptieren. Mit der gleichen Konstruktion wie für NTMs könnten wir allerdings die DTM modifizieren, sodass sie bis $T(n)$ zählt und anhält, wenn der Grenzwert überschritten wird. Die DTM würde die Ausführungszeit $O(T^2(n))$ benötigen.

unterschiedlicher Pfade wie $O(m!)$, die Fakultät von m, die für jede Konstante c und genügend großes m den Wert 2^{cm} übersteigt. ■

Es scheint, als wären alle Möglichkeiten zur Lösung des TSP im wesentlichen Versuche, alle Zyklen zu prüfen und deren Gesamtgewichtung zu berechnen. Wenn wir durchdacht vorgehen, können wir einige offensichtlich schlechte Wahlmöglichkeiten eliminieren. Doch anscheinend müssen wir trotz aller Optimierungsversuche eine exponentielle Anzahl von Pfaden prüfen, bevor wir schließen können, dass kein Pfad mit der gewünschten Gewichtungsgrenze W existiert, oder wir finden im ungünstigen Fall eine Lösung hinsichtlich der Reihenfolge, in der wir die Pfade prüfen.

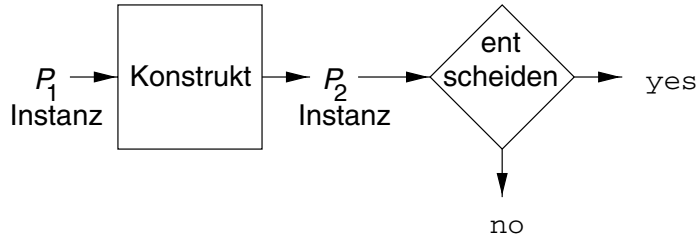
Hätten wir andererseits einen nichtdeterministischen Computer, könnten wir zuerst die Permutation der Knoten schätzen und dann das Gesamtgewicht des Knotenpfads berechnen. Gäbe es einen realen nichtdeterministischen Computer, würde kein Zweig mehr als $O(n)$ Schritte in Anspruch nehmen, falls die Eingabe die Länge n besitzt. Mit einer mehrbändigen NTM können wir eine Permutation in $O(n^2)$ Schritten schätzen und deren Gesamtgewicht mit ähnlichem Zeitaufwand prüfen. Eine einbändige NTM kann das TSP daher mit einem Zeitaufwand von höchstens $O(n^4)$ lösen. Wir folgern, dass das TSP in $\mathcal{NP}$ enthalten ist.

10.1.5 Reduktionen mit polynomialem Zeitaufwand

Unsere prinzipielle Methode zum Beweis, dass ein Problem P_2 nicht mit polynomialem Zeitaufwand gelöst werden kann (dass P_2 also nicht in $\mathcal{P}$ enthalten ist), ist die Reduktion eines Problems P_1, von dem bekannt ist, dass es nicht in $\mathcal{P}$ enthalten ist, auf P_2[2]. Diesen Ansatz stellt Abbildung 8.5 dar; Abbildung 10.2 zeigt ihn erneut.

2. Diese Aussage ist nicht ganz korrekt. In der Praxis *nehmen wir nur an*, dass P_1 nicht in $\mathcal{P}$ enthalten ist, und verwenden die starke Evidenz, dass P_1 »NP-vollständig« ist; dieses Konzept wird in Abschnitt 10.1.6 vorgestellt. Wir beweisen dann, dass auch P_2 »NP-vollständig« ist, und folgern, dass P_2 nicht in $\mathcal{P}$ enthalten ist.

Abbildung 10.2: Erneute Darstellung einer Reduktion



Angenommen, wir möchten die Aussage »Wenn P_2 in $\mathcal{P}$ enthalten ist, dann ist auch P_1 darin enthalten« beweisen. Da wir annehmen, dass P_1 *nicht* in $\mathcal{P}$ enthalten ist, könnten wir schließen, dass P_2 ebenfalls nicht in $\mathcal{P}$ enthalten ist. Jedoch ist die bloße Existenz des in Abbildung 10.2 als »Konstrukt« bezeichneten Algorithmus nicht ausreichend, um die gewünschte Aussage zu beweisen.

Nehmen wir beispielsweise an, der Algorithmus würde bei einer gegebenen Instanz von P_1 der Länge m eine Ausgabezeichenreihe der Länge 2^m produzieren, die als Eingabe des hypothetischen polynomialen Algorithmus von P_2 dienen würde. Falls nun die Ausführungsdauer dieses Entscheidungsalgorithmus $O(n^k)$ betragen würde, käme bei einer Eingabe der Länge 2^m eine Ausführungsdauer von $O(2^{km})$ zustande, die in m exponentiell ist. Der Entscheidungsalgorithmus für P_1 benötigt bei einer Eingabe der Länge m daher eine Dauer, die in m exponentiell ist. Diese Tatsachen sind konsistent mir der Situation, in den P_2 in $\mathcal{P}$ und P_1 nicht in $\mathcal{P}$ enthalten ist.

Auch wenn der Algorithmus, der aus einer P_1-Instanz eine P_2-Instanz konstruiert, immer eine Instanz produziert, die zur Eingabegröße polynomial ist, kommen wir unter Umständen nicht zur gewünschten Schlussfolgerung. Nehmen wir beispielsweise an, die konstruierte Instanz von P_2 hätte die gleiche Größe m wie die P_1-Instanz, doch der Konstruktionsalgorithmus selbst würde eine in m exponentielle Ausführungsdauer beanspruchen, etwa $O(2^m)$. Nun impliziert ein Entscheidungsalgorithmus für P_2, der bei einer Eingabelänge n eine polynomiale Dauer beansprucht, lediglich, dass ein Entscheidungsalgorithmus für P_1 existiert, der bei der Eingabelänge m $O(2^m + m^k)$ benötigt. Diese Grenze der Ausführungsdauer berücksichtigt die Tatsache, dass wir sowohl die Übersetzung in P_2 durchführen als auch die resultierende P_2-Instanz lösen müssen. Wiederum wäre es möglich, dass P_2 in $\mathcal{P}$ enthalten ist und P_1 nicht.

Der Übersetzung von P_1 in P_2 muss deshalb die Einschränkung auferlegt werden, dass dieser Vorgang einen zur Länge der Eingabe polynomialen Zeitaufwand erfordert. Beachten Sie, dass bei einer Übersetzungsdauer von $O(m^j)$ bei Eingabelänge m die Ausgabeinstanz von P_2 nicht länger als die Anzahl der benötigten Schritte sein kann, also höchstens cm^j für eine Konstante c. Wir können nun beweisen, dass mit P_2 auch P_1 in $\mathcal{P}$ enthalten ist.

Für den Beweis nehmen wir an, dass wir mit einem Zeitaufwand von $O(n^k)$ entscheiden können, ob eine Zeichenreihe der Länge n in P_2 enthalten ist. Dann entscheiden wir in der Zeit $O(m^j + (cm^j)^k)$, ob eine Zeichenreihe der Länge m in P_1 enthalten ist; der Term m^j entspricht der Übersetzungsdauer und der Term $(cm^j)^k$ der Dauer, bis die resultierende Instanz von P_2 entschieden ist. Wenn wir den Ausdruck vereinfachen, dann ist ersichtlich, dass P_1 mit Zeitaufwand $O(m^j + cm^{jk})$ gelöst werden kann. Da c, j

und k Konstanten sind, ist diese Dauer in m polynomial, und wir folgern, dass P_1 in $\mathcal{P}$ enthalten ist.

In der Theorie der Nichtbehandelbarkeit verwenden wir daher lediglich *Reduktionen mit polynomialem Zeitaufwand*. Eine Reduktion von P_1 auf P_2 ist von polynomialer Dauer, wenn diese Dauer zur Länge der P_1-Instanz polynomial ist. Beachten Sie, dass dies die Konsequenz hat, dass die P_2-Instanz von einer Länge ist, die zur Länge der P_1-Instanz polynomial ist.

10.1.6 NP-vollständige Probleme

Wir behandeln nun die Familie der Probleme, die wohlbekannte Kandidaten dafür sind, zwar in $\mathcal{NP}$, nicht aber in $\mathcal{P}$ enthalten zu sein. Sei L eine Sprache (ein Problem) in $\mathcal{NP}$. Wir sagen, L ist *NP-vollständig* , wenn die folgenden Aussagen über L wahr sind:

1. L ist in $\mathcal{NP}$ enthalten.
2. Für jede Sprache L' in $\mathcal{NP}$ existiert eine polynomiale Reduktion von L' auf L.

Das in Abschnitt 10.1.4 vorgestellte Problem des Handlungsreisenden ist, wie wir sehen werden, ein Beispiel für ein NP-vollständiges Problem. Da anscheinend $\mathcal{P} \neq \mathcal{NP}$ gilt und insbesondere alle NP-vollständigen Probleme in $\mathcal{NP} - \mathcal{P}$ enthalten sind, sehen wir den Beweis der NP-Vollständigkeit eines Problems generell als Beweis dafür an, dass das Problem nicht in $\mathcal{P}$ enthalten ist.

Wir beweisen unser erstes Problem SAT (Boolean Satisfiability, Boolesche Erfüllbarkeit) als NP-vollständig, indem wir zeigen, dass die Sprache jeder polynomialen NTM eine Reduktion auf SAT in polynomialer Zeit besitzt. Wenn wir einige NP-vollständige Probleme kennen, können wir ein neues Problem als NP-vollständig beweisen, indem wir ein bekanntes NP-vollständiges Problem auf das neue Problem polynomial reduzieren. Der folgende Satz zeigt, warum eine solche Reduktion das Zielproblem als NP-vollständig beweist.

NP-harte Probleme

Einige Probleme L sind so hart, dass wir zwar die Bedingung (2) der Definition der NP-Vollständigkeit beweisen können (jede Sprache in $\mathcal{NP}$ kann mit einer polynomialen Reduktion auf L reduziert werden), nicht jedoch Bedingung (1), dass L in $\mathcal{NP}$ enthalten ist. In diesem Fall nennen wir *L NP-hart*. Wir haben schon den informellen Begriff »nicht behandelbar« verwendet und damit Probleme beschrieben, die einen exponentiellen Zeitaufwand zu erfordern scheinen. Es ist allgemein akzeptabel, »nicht behandelbar« im Sinne von »NP-hart« zu verwenden, obwohl prinzipiell Probleme existieren könnten, die einen exponentiellen Zeitaufwand erfordern, obwohl sie im formalen Sinn nicht NP-hart sind.

Ein Beweis dafür, dass L NP-hart ist, ist ausreichend, um zu zeigen, dass L wahrscheinlich einen exponentiellen oder noch höheren Zeitaufwand erfordert. Ist L allerdings nicht in $\mathcal{NP}$ enthalten, dann unterstützt diese offensichtliche Schwierigkeit nicht die Behauptung, dass alle NP-vollständigen Probleme schwierig sind. Es könnte sich erweisen, dass $\mathcal{P} = \mathcal{NP}$ und L trotzdem einen exponentiellen Zeitaufwand erfordert.

Satz 10.4 Wenn P_1 NP-vollständig und P_2 in NP ist und eine polynomiale Reduktion von P_1 auf P_2 existiert, dann ist P_2 NP-vollständig.

Beweis: Wir müssen zeigen, dass jede Sprache L in $\mathcal{NP}$ mit polynomialem Zeitaufwand auf P_2 reduziert werden kann. Wir wissen, dass eine solche Reduktion von L auf P_1 existiert; diese Reduktion erfordert die polynomiale Dauer $p(n)$. Daher wird eine Zeichenreihe w der Länge n in L in eine Zeichenreihe x in P_1 konvertiert, der eine Länge von höchstens $p(n)$ besitzt.

Wir wissen außerdem, dass eine polynomiale Reduktion von P_1 auf P_2 existiert; diese Reduktion erfordere also einen Zeitaufwand von $q(m)$. Die Reduktion überführt x in eine Zeichenreihe y aus P_2 und erfordert eine Dauer von höchstens $q(p(n))$. Die Transformation von w in y erfordert daher höchstens $p(n) + q(p(n))$, und dieser Wert ist polynomial. Wir folgern, dass L in polynomialer Zeit auf P_2 reduzierbar ist. Da es sich bei L um jede Sprache aus $\mathcal{NP}$ handeln kann, haben wir gezeigt, dass alle in $\mathcal{NP}$ enthaltenen Sprachen polynomial auf P_2 reduzierbar sind; P_2 ist daher NP-vollständig. ■

Es ist noch ein weiterer wichtiger Satz über NP-vollständige Probleme zu beweisen: Ist eines von ihnen in $\mathcal{P}$ enthalten, dann sind alle Probleme aus $\mathcal{NP}$ in $\mathcal{P}$ enthalten. Da wir als sicher annehmen, dass viele Probleme in $\mathcal{NP}$ existieren, die *nicht* in $\mathcal{P}$ enthalten sind, erachten wir einen Beweis, dass ein Problem NP-vollständig ist, als gleichbedeutend mit dem Beweis, dass es keinen polynomialen Algorithmus besitzt und daher keine praktisch brauchbare Computerlösung dafür existiert.

Satz 10.5 Ist ein NP-vollständiges Problem P in $\mathcal{P}$ enthalten, dann ist $\mathcal{P} = \mathcal{NP}$.

Beweis: Angenommen, P ist sowohl NP-vollständig als auch in $\mathcal{P}$ enthalten. Dann können alle Sprachen L aus $\mathcal{NP}$ polynomial auf P reduziert werden. Ist P in $\mathcal{P}$ enthalten, dann ist auch L in $\mathcal{P}$ enthalten, wie in Abschnitt 10.1.5 diskutiert.

Andere Begriffe der NP-Vollständigkeit

Das Ziel des Studiums der NP-Vollständigkeit ist Satz 10.5, also die Identifikation von Problemen *P*, deren Präsenz in der Klasse $\mathcal{P} = \mathcal{NP}$ impliziert. Die von uns verwendete Definition von »NP-vollständig«, die häufig *Karp-Vollständigkeit* genannt wird, da sie zuerst in einer grundlegenden Veröffentlichung von R. Karp zu diesem Thema verwendet wurde, ist dazu adäquat, jedes Problem zu erfassen, von dem wir annehmen können, dass es Satz 10.5 erfüllt. Allerdings gibt es weitere, umfassendere Begriffe der NP-Vollständigkeit, die ebenfalls die Behauptung von Satz 10.5 erfüllen.

Beispielsweise definierte S. Cook in seiner Originalveröffentlichung über das Thema ein Problem *P* als »NP-vollständig«, wenn es ein *Orakel* für das Problem *P* (einen Mechanismus, der in einer Zeiteinheit jede Frage über die Zugehörigkeit einer gegebenen Zeichenreihe zu $\mathcal{P}$ beantwortet) gibt, sodass jede in $\mathcal{NP}$ enthaltene Sprache mit polynomialem Zeitaufwand erkannt werden kann. Dieser Typ der NP-Vollständigkeit wird *Cook-Vollständigkeit* genannt. In einem gewissen Sinne ist Karp-Vollständigkeit der Spezialfall, in welchem dem Orakel nur eine einzige Frage gestellt wird. Allerdings erlaubt die Cook-Vollständigkeit auch das Komplement der Antwort; Sie könnten dem Orakel z. B. eine Frage stellen und mit dem Gegenteil dessen, was das Orakel sagt, antworten. Als Konsequenz aus Cooks Definition folgt, dass die Komplemente der NP-vollständigen Probleme ebenfalls NP-vollständig wären. Da wir die strengere Definition der Karp-Vollständigkeit verwenden, können wir in Abschnitt 11.1 eine wichtige Unterscheidung zwischen NP-vollständigen Problemen (im Sinn von Karp) und deren Komplementen vornehmen.

10.1.7 Übungen zum Abschnitt 10.1

Übung 10.1.1 Angenommen, wir ändern die Gewichtung der Kanten in Abbildung 10.1 wie folgt. Welcher MWST würde daraus resultieren?

a) Ändere das Gewicht 10 von Kante (1, 3) in 25.

b) Ändere stattdessen das Gewicht der Kante (2, 4) in 16.

Übung 10.1.2 Ergänzen Sie den Graphen in Abbildung 10.1 um eine Kante des Gewichts 19 zwischen den Knoten 1 und 4. Welchen Hamiltonschen Kreis mit minimalem Gewicht hat der Graph dann?

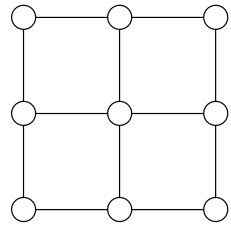
*! **Übung 10.1.3** Angenommen, es existiert ein NP-vollständiges Problem mit einer deterministischen Lösung, die einen Zeitaufwand von $O(n^{\log_2 n})$ erfordert. Beachten Sie, dass diese Funktion zwischen polynomial und exponentiell liegt und keiner der beiden Klassen angehört. Was können wir dann über die Ausführungsdauer eines beliebigen Problems in $\mathcal{NP}$ aussagen?

!! **Übung 10.1.4** Betrachten Sie Graphen, deren Knoten Rasterpunkte in einem n-dimensionalen Kubus mit Seitenlänge m sind; die Knoten entsprechen also Vektoren $(i_1, i_2, \ldots, i_n)$, wobei jedes i_j im Bereich 0 bis m liegt. Es existiert nur dann eine Kante zwischen zwei Knoten, wenn sie sich in exakt einer Dimension um 1 unterscheiden. Beispielsweise entspricht der Fall $n = 2$ und $m = 1$ einem Quadrat, $n = 3$ und $m = 1$ einem Würfel sowie $n = 2$ und $m = 2$ dem Graphen in Abbildung 10.5. Einige dieser Graphen besitzen einen Hamiltonschen Kreis, andere dagegen nicht. Beispielsweise ist dies beim Quadrat offensichtlich, und auch der Würfel besitzt einen Hamiltonschen Kreis, z. B. (0, 0, 0) zu (0, 0, 1), (0, 1, 1), (0, 1, 0), (1, 1, 0), (1, 1, 1), (1, 0, 1) und über (1, 0, 0) zurück zu (0, 0, 0). Der Graph in Abbildung 10.3 besitzt keinen Hamiltonschen Kreis.

a) Beweisen Sie, dass der in Abbildung 10.5 dargestellte Graph keinen Hamiltonschen Kreis besitzt. *Hinweis:* Überlegen Sie, was geschieht, wenn ein hypothetischer Hamiltonscher Kreis den zentralen Knoten passiert. Woher kann er kommen und wohin kann er führen, ohne einen Teil des Graphen aus dem Kreis auszuschließen?

b) Für welche Werte von n und m existiert ein Hamiltonscher Kreis?

Abbildung 10.3: Ein Graph mit $n = 2$ und $m = 2$



! **Übung 10.1.5** Angenommen, es liegt eine Kodierung von kontextfreien Grammatiken über einem endlichen Alphabet vor. Beurteilen Sie die folgenden beiden Sprachen:

1. $L_1 = \{(G, A, B) \mid G$ ist eine (kodierte) kfG, A und B sind (kodierte) Variable von G, und die Mengen der aus A und B abgeleiteten Terminalzeichenreihen sind identisch$\}$.
2. $L_2 = \{(G_1, G_2) \mid G_1$ und G_2 sind (kodierte) kontextfreie Grammatiken, und $L(G_1) = L(G_2)\}$.

Beantworten Sie die folgenden Fragen:

* a) Zeigen Sie, dass L_1 mit polynomialem Aufwand auf L_2 zu reduzieren ist.

b) Zeigen Sie, dass L_2 mit polynomialem Aufwand auf L_1 zu reduzieren ist.

* c) Was sagen (a) und (b) darüber aus, ob L_1 und L_2 NP-vollständig sind?

Übung 10.1.6 Als Sprachklassen besitzen $\mathcal{P}$ und $\mathcal{NP}$ einige Abgeschlossenheitseigenschaften. Zeigen Sie, dass $\mathcal{P}$ unter jeder der folgenden Operationen abgeschlossen ist:

a) Spiegelung

* b) Vereinigung

*! c) Konkatenation

! d) Hüllenbildung (Stern)

e) Inverser Homomorphismus

f) Komplementbildung

Übung 10.1.7 Auch $\mathcal{NP}$ ist unter jeder der in Übung 10.1.6 aufgelisteten Operationen mit (der vermuteten) Ausnahme von (f), dem Komplement, abgeschlossen. Es ist nicht bekannt, ob $\mathcal{NP}$ unter der Komplementbildung abgeschlossen ist. Dieses Thema wird in Abschnitt 11.1 erörtert. Beweisen Sie, dass $\mathcal{NP}$ unter den Operationen (a) bis (e) aus Übung 10.1.6 abgeschlossen ist.

10.2 Ein NP-vollständiges Problem

Sie lernen nun das erste NP-vollständige Problem kennen. Dieses Problem – die Erfüllbarkeit eines Booleschen Ausdrucks – wird durch die explizite Reduktion der Sprache einer beliebigen nichtdeterministisch polynomialen TM auf das Problem der Erfüllbarkeit als NP-vollständig nachgewiesen.

10.2.1 Das Problem der Erfüllbarkeit

Boolesche Ausdrücke bestehen aus folgenden Elementen:

1. Variablen, die Boolesche Werte besitzen; sie haben entweder den Wert 1 (wahr) oder den Wert 0 (falsch)
2. den Binäroperatoren $\wedge$ und $\vee$, die das logische UND und ODER von zwei Ausdrücken repräsentieren

3. dem Unäroperator $\neg$, der die logische Negation repräsentiert
4. Klammern zum Gruppieren von Operatoren und Operanden soweit zur Änderung des normalen Vorrangs von Operatoren nötig: $\neg$ hat die höchste Priorität, danach folgt $\wedge$ und dann $\vee$.

Beispiel 10.6 Ein Beispiel für einen Booleschen Ausdruck ist $x \wedge \neg(y \vee z)$. Der Teilausdruck $y \vee z$ ist wahr, wenn eine der Variablen y oder z wahr ist, und er ist falsch, wenn sowohl y als auch z falsch sind. Der umfangreichere Teilausdruck $\neg(y \vee z)$ ist genau dann wahr, wenn $y \vee z$ falsch ist, wenn also sowohl y als auch z falsch sind. Sind y oder z oder beide wahr, dann ist $\neg(y \vee z)$ falsch.

Sehen wir uns nun den gesamten Ausdruck an. Da es sich um die logische UND-Verknüpfung zweier Teilausdrücke handelt, ist der Ausdruck genau dann wahr, wenn beide Teilausdrücke wahr sind. $x \wedge \neg(y \vee z)$ ist also genau dann wahr, wenn x wahr, y falsch und z falsch ist. ■

Bei einem gegebenen Booleschen Ausdruck E weist eine *Belegung* jeder der Variablen in E entweder wahr oder falsch zu. Der *Wert* des Ausdrucks E bei einer Belegung T, $E(T)$ genannt, ist das Ergebnis der Auswertung von E, wobei jede Variable x durch den Wahrheitswert $T(x)$ (wahr oder falsch) ersetzt wird, den T an x zuweist.

Eine Belegung T *erfüllt* einen Booleschen Ausdruck E, wenn $E(T) = 1$ ist; nach der Belegung T ist der Ausdruck E also wahr. Ein Boolescher Ausdruck E ist *erfüllbar*, wenn mindestens eine Belegung T existiert, die E erfüllt.

Beispiel 10.7 Der Ausdruck $x \wedge \neg(y \vee z)$ aus Beispiel 10.6 ist erfüllbar. Sie haben gesehen, dass die durch $T(x) = 1$, $T(y) = 0$ und $T(z) = 0$ definierte Belegung T diesen Ausdruck erfüllt, da er den Wert wahr (1) annimmt. Außerdem ist T die *einzige* erfüllende Belegung für diesen Ausdruck, da die sieben anderen Wertekombinationen der drei Variablen dem Ausdruck den Wert `falsch` (0) zuweisen.

Als weiteres Beispiel sehen wir uns den Ausdruck $E = x \wedge (\neg x \vee y) \wedge \neg y$ an. Wir behaupten, dass E nicht erfüllbar ist. Da dieser Ausdruck nur zwei Variablen enthält, gibt es $2^2 = 4$ Belegungen. Sie können also einfach alle vier Belegungen testen und verifizieren, dass E jeweils den Wert 0 annimmt. Wir können allerdings auch anders argumentieren. E ist nur dann wahr, wenn alle drei mit $\wedge$ verknüpften Terme wahr sind. Dies bedeutet, dass x wahr sein muss (wegen des ersten Terms) und y falsch (wegen des letzten Terms). Jedoch ist der mittlere Term $\neg x \vee y$ unter dieser Belegung falsch. E kann daher niemals wahr sein und ist somit nicht erfüllbar.

Sie haben nun ein Beispiel kennen gelernt, in dem ein Ausdruck genau eine erfüllende Belegung besitzt, und ein weiteres Beispiel, in dem keine erfüllende Belegung möglich ist. Es gibt auch viele Beispiele für Ausdrücke, die mehrere erfüllende Belegungen besitzen. Als einfaches Beispiel soll $F = x \vee \neg y$ dienen. Bei drei Zuweisungen besitzt F den Wert 1:

1. $T_1(x) = 1$; $T_1(y) = 1$.
2. $T_2(x) = 1$; $T_2(y) = 0$.
3. $T_3(x) = 0$; $T_3(y) = 0$.

F hat nur bei der vierten Zuweisung mit $x = 0$ und $y = 1$ den Wert 0. Daher ist F erfüllbar. ■

Das *Erfüllbarkeitsproblem* lautet:

- Gegeben sei ein Boolescher Ausdruck; ist er erfüllbar?

Wir werden das Erfüllbarkeitsproblem (engl. *Satisfiability Problem*) im Folgenden *SAT* nennen.

Als Sprache ausgedrückt ist das Problem SAT die Menge der (kodierten) Booleschen Ausdrücke, die erfüllbar sind. Zeichenreihen, die entweder kein gültiger Code eines Booleschen Ausdrucks oder Code eines nicht erfüllbaren Booleschen Ausdrucks sind, sind nicht in SAT enthalten.

10.2.2 SAT-Instanzen repräsentieren

Boolesche Ausdrücke enthalten die Symbole $\wedge$, $\vee$, $\neg$, die öffnende und die schließende Klammer sowie Symbole, die Variable repräsentieren. Die Erfüllbarkeit eines Ausdrucks hängt nicht von den Variablennamen ab, sondern nur davon, ob es sich bei einem Auftreten von zwei Variablen um die gleiche oder verschiedene Variable handelt. Wir können daher annehmen, dass x_1, x_2, … die Variablen sind, werden in den Beispielen jedoch weiterhin Variablennamen wie x, y und z einsetzen. Wir können außerdem Variable umbenennen und verwenden dafür die kleinstmöglichen Indizes. Beispielsweise werden wir x_5 nicht verwenden, wenn wir im gleichen Ausdruck nicht schon x_1 bis x_4 eingesetzt haben.

Da in einem Booleschen Ausdruck prinzipiell unbeschränkt viele Symbole auftreten können, haben wir ein bekanntes Problem mit der Entwicklung eines Codes mit einem festen, endlichen Alphabet, um Ausdrücke mit unbeschränkt vielen Variablen zu repräsentieren. Nur dann können wir über SAT als »Problem« sprechen, also als einer Sprache über einem festen Alphabet, die aus den Codes solcher Boolescher Ausdrücke besteht, die erfüllbar sind. Wir werden den folgenden Code verwenden:

1. Die Symbole $\wedge$, $\vee$, $\neg$, (und) repräsentieren sich selbst.
2. Die Variable x_i wird vom Symbol x gefolgt von der in den Symbolen 0 und 1 kodierten Binärzahl i repräsentiert.

Das Alphabet für das SAT-Problem (die SAT-Sprache) besteht also nur aus acht Symbolen. Alle Instanzen von SAT sind Zeichenreihen über diesem festen, endlichen Alphabet.

Beispiel 10.8 Wir sehen uns den Ausdruck $x \wedge \neg(y \vee z)$ aus Beispiel 10.6 an. Im ersten Kodierungsschritt ersetzen wir die Variablen durch indizierte Variablen x_i. Da der Ausdruck drei Variablen enthält, verwenden wir x_1, x_2 und x_3. Uns steht frei, welche der Variablen x, y und z wir durch die x_i ersetzen, doch wir verwenden $x = x_1$, $y = x_2$ und $z = x_3$. Der Ausdruck lautet nun $x_1 \wedge \neg(x_2 \vee x_3)$. Der Code für diesen Ausdruck lautet:

$$x1 \wedge \neg(x10 \vee x11)$$ ■

Beachten Sie, dass die Länge eines kodierten Booleschen Ausdrucks in etwa der Anzahl der Positionen im Ausdruck entspricht, wenn jedes Auftreten einer Variablen als 1 gezählt wird. Der Grund für die Differenz besteht darin, dass der Ausdruck $O(m)$ Variablen enthalten kann, wenn er m Positionen hat, und damit können Variable beim Kodieren $O(\log m)$ Symbole erfordern. Ein Ausdruck mit einer Länge von m Positionen kann also einen Code besitzen, der aus $n = O(m \log m)$ Symbolen besteht.

Nun ist aber der Unterschied zwischen m und $m \log m$ polynomial begrenzt. Solange wir uns lediglich damit befassen, ob ein Problem mit polynomialem Zeitaufwand bezüglich der Eingabelänge lösbar ist, müssen wir nicht zwischen der Länge des Codes eines Ausdrucks und der Anzahl der Positionen im Code selbst unterscheiden.

10.2.3 NP-Vollständigkeit des SAT-Problems

Wir beweisen nun den »Satz von Cook«, also die Tatsache, dass SAT NP-vollständig ist. Zuerst müssen wir zeigen, dass das Problem SAT in $\mathcal{NP}$ enthalten ist, und dann, dass jede Sprache in $\mathcal{NP}$ auf das fragliche Problem reduziert werden kann. Im Allgemeinen zeigen wir den zweiten Teil mithilfe einer polynomialen Reduktion aus einem anderen NP-vollständigen Problem und wenden dann Satz 10.5 an. Allerdings kennen wir noch keine NP-vollständigen Probleme, die wir auf SAT reduzieren könnten. Uns steht daher nur die absolute Strategie zur Verfügung, jedes Problem in $\mathcal{NP}$ auf SAT zu reduzieren.

Satz 10.9 (Satz von Cook) SAT ist NP-vollständig.

Beweis: Im ersten Teil des Beweises zeigen wir, dass SAT in $\mathcal{NP}$ enthalten ist. Dieser Teil ist einfach:

1. Die nichtdeterministische Eigenschaft einer NTM wird verwendet, um eine Belegung T für den gegebenen Ausdruck E zu raten. Ist der kodierte Ausdruck E von der Länge n, dann reicht $O(n)$ Zeit auf einer mehrbändigen NTM aus. Beachten Sie, dass diese NTM viele Auswahlmöglichkeiten für Bewegungen besitzt und am Ende des Rateprozesses unter Umständen bis zu 2^n verschiedene Konfigurationen erreicht hat, wobei jeder Zweig das Erraten einer anderen Belegung repräsentiert.
2. E wird für die Belegung T ausgewertet. Ist $E(T) = 1$, dann wird akzeptiert. Beachten Sie, dass dieser Teil deterministisch ist. Die Tatsache, dass andere Zweige der NTM nicht zur Akzeptanz führen, hat keinen Einfluss auf das Ergebnis, da die NTM akzeptiert, wenn nur eine einzige Belegung gefunden wird.

Die Auswertung kann mit einer mehrbändigen NTM leicht in der Zeit $O(n^2)$ erfolgen. Daher erfordert die gesamte Erkennung von SAT durch eine mehrbändige NTM den Zeitaufwand $O(n^2)$. Die Konvertierung auf eine einbändige NTM kann den Zeitaufwand quadrieren, und somit ist $O(n^4)$ ausreichend.

Nun müssen wir den schwierigen Teil beweisen: wenn L eine Sprache aus $\mathcal{NP}$ ist, dann existiert eine polynomiale Reduktion von L auf SAT. Wir können annehmen, dass eine einbändige NTM M und ein polynomiales $p(n)$ existieren, sodass M bei einer Eingabe der Länge n entlang jedes Zweiges nicht mehr als $p(n)$ Schritte benötigt. Außerdem können die Einschränkungen aus Satz 8.12, die wir für DTMs bewiesen haben, auf die gleiche Weise auch für NTMs bewiesen werden. Wir können daher annehmen, dass M nie ein Leerzeichen schreibt und den Kopf nie auf eine Position links von der anfänglichen Kopfposition bewegt.

Wenn M daher eine Eingabe w akzeptiert und $|w| = n$ ist, dann existiert eine Folge von Bewegungen von M, sodass Folgendes gilt:

1. α_0 ist die anfängliche Konfiguration von M bei Eingabe w.
2. $\alpha_0 \vdash \alpha_1 \vdash \ldots \vdash \alpha_k$, mit $k \leq p(n)$.

3. α_k ist eine Konfiguration mit einem akzeptierenden Zustand.
4. Jedes α_i enthält kein einziges Leerzeichen (ausgenommen, α_i endet in einem Zustand und einem Leerzeichen) und erstreckt sich von der anfänglichen Kopfposition, dem am weitesten links stehenden Eingabesymbol, nach rechts.

Unsere Strategie lässt sich wie folgt zusammenfassen:

a) Jedes α_i kann als Folge von Symbolen $X_{i0}X_{i1}\ldots X_{i,p(n)}$ geschrieben werden. Eines dieser Symbole ist ein Zustand, die anderen sind Bandsymbole. Wie immer nehmen wir an, dass die Zustände und Bandsymbole disjunkt sind, und wir somit wissen, welches X_{ij} der Zustand ist und wo sich damit der Bandkopf befindet. Beachten Sie, dass es keinen Grund gibt, Symbole rechts von den ersten $p(n)$ Symbolen auf dem Band zu repräsentieren (zusammen mit dem Zustand ergibt das eine Konfiguration von der Länge $p(n) + 1$), da sie keinen Einfluss auf eine Bewegung von M haben, wenn M garantiert nach höchstens $p(n)$ Bewegungen anhält.

b) Um die Folge der Konfigurationen in Begriffen Boolescher Variablen zu beschreiben, verwenden wir die Variablen x_{ijA}, die jeweils die Behauptung $X_{ij} = A$ repräsentieren. i und j sind ganzzahlige Werte aus dem Bereich 0 bis $p(n)$, und A ist entweder ein Bandsymbol oder ein Zustand.

c) Wir repräsentieren die Bedingung, dass die Folge der Konfigurationen die Akzeptanz einer Eingabe w wiedergibt, indem wir einen Booleschen Ausdruck verwenden, der nur dann erfüllbar ist, wenn M w mit einer Folge von höchstens $p(n)$ Bewegungen akzeptiert. Die Erfüllungsbelegung wird diejenige sein, die »die Wahrheit« über die Konfigurationen aussagt; x_{ijA} ist ausschließlich dann wahr, wenn $X_{ij} = A$ gilt. Um sicherzustellen, dass die polynomiale Reduktion von $L(M)$ auf SAT korrekt ist, erstellen wir diesen Ausdruck so, dass er über die Berechnung aussagt:

   i. *Startet richtig*, d. h. die anfängliche Konfiguration lautet q_0w, gefolgt von Leerzeichen.

   ii. *Nächste Bewegung ist richtig* (die Bewegung folgt den Regeln der TM), d. h. jede nachfolgende Konfiguration folgt der vorherigen nach einer der möglichen zulässigen Bewegungen von M.

   iii. *Endet richtig*, d. h. es gibt eine Konfiguration mit einem akzeptierenden Zustand.

Bevor wir die Konstruktion unseres Booleschen Ausdrucks präzisieren, sind einige Details zu klären.

- Wir haben Konfigurationen so spezifiziert, dass sie enden, wenn der unendlich lange Rest aus Leerzeichen beginnt. Es ist aber bei der Simulation einer Berechnung mit polynomialem Zeitaufwand günstiger, sich alle Konfigurationen von der gleichen Länge $p(n) + 1$ vorzustellen. Daher kann eine Konfiguration einen Rest aus Leerzeichen enthalten.
- Es ist von Vorteil, wenn wir annehmen, dass alle Berechungen exakt $p(n)$ Bewegungen umfassen (und daher aus $p(n) + 1$ Konfigurationen bestehen), auch wenn eine Akzeptanz schon früher eintritt. Wir erlauben daher jeder Konfiguration mit einem akzeptierenden Zustand, ihr eigener Nachfolger zu sein. Besitzt α also

einen akzeptierenden Zustand, dann erlauben wir die »Bewegung« $\alpha \vdash \alpha$. Falls es sich also um eine akzeptierende Berechung handelt, dann gehört zu $\alpha_{p(n)}$ ein akzeptierender Zustand, und damit ist dies das Einzige, was für die Bedingung »Endet richtig« zu prüfen ist.

Tabelle 10.1 zeigt ein Beispiel für eine polynomiale Berechnung von *M*. Die Zeilen entsprechen den Folgen von Konfigurationen, und die Spalten bezeichnen die Zellen des Bandes, die in der Berechnung eingesetzt werden. Beachten Sie, dass die Anzahl der Vierecke in Tabelle 10.1 dem Wert $(p(n) + 1)^2$ entspricht. Außerdem ist die Anzahl der verschiedenen Variablen, die in der Tabelle auftreten, endlich und nur von *M* abhängig; es ist die Summe der Anzahl der Zustände sowie der Bandsymbole von *M*.

Tabelle 10.1: Konstruktion des Arrays der Zellen/Konfigurationsfakten

Konfiguration	0	1	...				...	p(n)
α_0	X_{00}	X_{01}						$X_{0,p(n)}$
α_1	X_{10}	X_{11}						$X_{1,p(n)}$
α_i				$X_{i,j-1}$	$X_{i,j}$	$X_{i,j+1}$		
α_{i+1}				$X_{i+1,j-1}$	$X_{i+1,j}$	$X_{i+1,j+1}$		
$\alpha_{p(n)}$	$X_{p(n),0}$	$X_{p(n),1}$						$X_{p(n),p(n)}$

Wir werden nun einen Algorithmus vorstellen, der aus *M* und *w* einen Booleschen Ausdruck $E_{M,w}$ konstruiert. Das Format von $E_{M,w}$ entspricht $S \wedge N \wedge F$, wobei *S*, *N* und *F* Ausdrücke sind, die aussagen, dass *M* richtig startet (*S*), die nächste Bewegung richtig ausführt (*N*) bzw. richtig endet (*F*).

Startet richtig

X_{00} muss der Startzustand q_0 von *M* sein, X_{01} bis X_{0n} entsprechen *w* (wobei *n* die Länge von *w* ist), und die restlichen X_{0j} sind Leerzeichen *B*. Ist $w = a_1a_2\ldots a_n$, dann gilt:

$$S = y_{00q_0} \wedge y_{01a_1} \wedge y_{02a_2} \wedge \cdots \wedge y_{0na_n} \wedge y_{0,n+1,B} \wedge y_{0,n+2,B} \wedge \cdots \wedge y_{0,p(n),B}$$

Mit Sicherheit können wir *S* bei gegebener Kodierung von *M* und gegebenem *w* mit dem Zeitaufwand $O(p(n))$ auf ein zweites Band einer mehrbändigen TM schreiben.

Endet richtig

Da wir annehmen, dass eine akzeptierende Konfiguration immer wiederholt wird, entspricht die Akzeptanz durch *M* dem Auffinden eines akzeptierenden Zustands in $\alpha_{p(n)}$. Wir hatten angenommen, *M* sei eine NTM, die innerhalb von *p(n)* Schritten akzeptiert, falls sie überhaupt akzeptiert. *F* ist daher die ODER-Verknüpfung von Ausdrücken F_j für $j = 0, 1, \ldots, p(n)$, wobei F_j aussagt, dass $X_{p(n),j}$ ein akzeptierender

Zustand ist. F_j ist daher $y_{p(n),j,a_1} \vee y_{p(n),j,a_2} \vee \ldots \vee y_{p(n),j,a_k}$, wobei $a_1, a_2, \ldots, a_k$ alle akzeptierenden Zustände von M sind. Schließlich ist

$$F = F_0 \vee F_1 \vee \ldots \vee F_{p(n)}$$

Beachten Sie, dass die Anzahl der in den F_j auftretenden Symbole eine Konstante ist, die von M abhängt, nicht jedoch von der Länge n der Eingabe w. F besitzt daher die Länge $O(p(n))$. Noch wichtiger ist, dass der Zeitaufwand dafür, F bei einer gegebenen Kodierung für M und der Eingabe w zu schreiben, in n polynomial ist; F kann auf einer mehrbändigen TM in der Zeit $O(p(n))$ geschrieben werden.

Nächste Bewegung ist richtig

Sicherzustellen, dass die Bewegungen von M korrekt sind, ist bei weitem der schwierigste Teil. Der Ausdruck N entspricht der UND-Verknüpfung der Ausdrücke N_i für $i = 0, 1, \ldots, p(n) - 1$, wobei jedes N_i sicherstellt, dass die Konfiguration α_{i+1} eine der Konfigurationen ist, die in M auf α_i folgen können. Wir beginnen die Erläuterung, wie N_i zu schreiben ist, indem wir uns in Tabelle 10.1 das Symbol $X_{i+1,j}$ ansehen. Dieses Symbol können wir immer aus Folgendem bestimmen:

1. aus den drei darüber stehenden Symbolen $X_{i,j-1}$, X_{ij} und $X_{i,j+1}$ sowie,
2. falls eines dieser Symbole der Zustand von α_i ist, aus der von der NTM M gewählten Bewegung.

Wir schreiben N_i als $(A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \ldots \wedge (A_{i,p(n)} \vee B_{i,p(n)})$.

- Ausdruck A_{ij} sagt Folgendes aus:
  a) Der Zustand von α_i befindet sich auf der Position j (X_{ij} ist also der Zustand).
  b) M kann eine Bewegung wählen, wobei X_{ij} der Zustand und $X_{i,j+1}$ das gelesene Symbol ist, sodass diese Bewegung die Folge der Symbole $X_{i,j-1}$, X_{ij} und $X_{i,j+1}$ in $X_{i+1,j-1}$, $X_{i+1,j}$ und $X_{i+1,j+1}$ transformiert. Falls X_{ij} allerdings ein akzeptierender Zustand ist, besteht auch die »Möglichkeit«, überhaupt keine Bewegung auszuführen, sodass alle folgenden Konfigurationen mit derjenigen übereinstimmen, die zuerst zur Akzeptanz führte.
- Ausdruck B_{ij} sagt Folgendes aus:
  a) Der Zustand α_i ist ausreichend weit von X_{ij} entfernt, sodass er $X_{i+1,j}$ nicht beeinflusst (also ist weder $X_{i,j-1}$ noch X_{ij} noch $X_{i,j+1}$ ein Zustand).
  b) $X_{i+1,j} = X_{ij}$

B_{ij} ist einfacher zu schreiben. Seien $q_1, q_2, \ldots, q_m$ die Zustände von M und $Z_1, Z_2, \ldots, Z_r$ die Bandsymbole. Dann folgt:

$$\begin{aligned} B_{ij} = {} & (y_{i,j-1,Z_1} \vee y_{i,j-1,Z_2} \vee \ldots \vee y_{i,j-1,Z_r}) \wedge \\ & (y_{i,j,Z_1} \vee y_{i,j,Z_2} \vee \ldots \vee y_{i,j,Z_r}) \wedge \\ & (y_{i,j+1,Z_1} \vee y_{i,j+1,Z_2} \vee \ldots \vee y_{i,j+1,Z_r}) \wedge \\ & (y_{i,j,Z_1} \wedge y_{i+1,j,Z_1}) \vee (y_{i,j,Z_2} \wedge y_{i+1,j,Z_2}) \vee \ldots \vee y_{i,j,Z_r} \wedge y_{i+1,j,Z_r}) \end{aligned}$$

Die erste Zeile von B_{ij} sagt aus, dass $X_{i,j-1}$ eines der Bandsymbole ist, und die zweite bzw. dritte Zeile sagt das Gleiche über X_{ij} bzw. $X_{i,j+1}$ aus. Die letzte Zeile sagt aus, dass $X_{ij} = X_{i+1,j}$, indem alle möglichen Bandsymbole ausgewertet werden; entweder sind beide Z_1 oder beide sind Z_2 und so weiter.

Es gibt zwei wichtige Sonderfälle: $j = 0$ und $j = p(n)$. Im ersten Fall existieren keine Variablen $y_{i,j-1,Z}$ und im zweiten Fall keine Variablen $y_{i,j+1,Z}$. Allerdings wissen wir, dass sich der Kopf nie über die anfängliche Position nach links hinaus bewegt und dass er nicht die Zeit hat, mehr als $p(n)$ Zellen rechts von der Anfangsposition zu lesen. Wir können daher einige Terme aus B_{i0} und $B_{i,p(n)}$ entfernen; diese Vereinfachung sei Ihnen überlassen.

Sehen wir uns nun die Ausdrücke A_{ij} an. Diese Ausdrücke spiegeln alle möglichen Beziehungen zwischen den Symbolen eines 2x3-Rechtecks im Array in Tabelle 10.1 wider: $X_{i,j-1}$, X_{ij}, $X_{i,j+1}$, $X_{i+1,j-1}$, $X_{i+1,j}$ und $X_{i+1,j+1}$. Eine Symbolzuordnung an diese sechs Variablen ist korrekt, falls Folgendes gilt:

1. X_{ij} ist ein Zustand, doch $X_{i,j-1}$ und $X_{i,j+1}$ sind Bandsymbole.
2. Genau eine der Variablen $X_{i+1,j-1}$, $X_{i+1,j}$ und $X_{i+1,j+1}$ ist ein Zustand.
3. Es existiert eine Bewegung von M, die erklärt, wie $X_{i,j-1}X_{ij}X_{i,j+1}$ in

   $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1}$
   transformiert wird.

Es existiert also eine endliche Anzahl von Symbolordnungen an die sechs Variablen, die korrekt sind. Sei A_{ij} die ODER-Verknüpfung der Terme, von denen jeder jeweils eine der Mengen von sechs Variablen repräsentiert, die eine korrekte Zuordnung bilden.

Wir nehmen beispielsweise an, dass eine Bewegung von M auf der Tatsache beruht, dass $\delta(q, A)$ (p, C, L) enthält. Sei D ein Bandsymbol von M. Dann ist $X_{i,j-1}X_{i,j}X_{i,j+1} = DqA$ und $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = pDC$ eine korrekte Zuordnung. Beachten Sie, wie diese Zuordnung die Änderung der Konfiguration reflektiert, die von dieser Bewegung von M verursacht wird. Dieser Möglichkeit entspricht der folgende Term:

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,p} \wedge y_{i+1,j,D} \wedge y_{i+1,j+1,C}$$

Wenn $\delta(q, A)$ etwa (p, C, R) enthält, dann lautet die entsprechende korrekte Zuordnung $X_{i,j-1}X_{i,j}X_{i,j+1} = DqA$ und $X_{i+1,j-1}X_{i+1,j}X_{i+1,j+1} = DCp$.

Der Term für diese korrekte Zuordnung ist:

$$y_{i,j-1,D} \wedge y_{i,j,q} \wedge y_{i,j+1,A} \wedge y_{i+1,j-1,D} \wedge y_{i+1,j,C} \wedge y_{i+1,j+1,p}$$

A_{ij} ist die ODER-Verknüpfung aller Terme für gültige Zuordnungen. In den Sonderfällen $j = 0$ und $j = p(n)$ müssen wir wie bei B_{ij} einiges ändern, um die Nichtexistenz der Variablen y_{ijZ} für $j < 0$ oder $j > p(n)$ zu reflektieren. Schließlich ergibt sich

$$N_i = (A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \ldots \wedge (A_{i,p(n)} \vee B_{i,p(n)})$$

und dann

$$N = N_0 \wedge N_1 \wedge \ldots \wedge N_{p(n)-1}$$

Wenn M viele Zustände und/oder Bandsymbole besitzt, können A_{ij} und B_{ij} zwar sehr groß werden, doch ist deren Größe in Bezug auf die Länge der Eingabe w eine Kon-

stante; die Größe ist also von n, der Länge von w, unabhängig. Daher ist $O(p(n))$ die Länge von N_i für jedes i und $O(p^2(n))$ die Länge von N. Noch wichtiger ist, dass wir N auf ein Band einer mehrbändigen TM mit einem Zeitaufwand schreiben können, der proportional zur Länge von N ist, und dieser Aufwand ist in n, der Länge von w, polynomial.

Abschluss des Beweises des Satzes von Cook

Zunächst sei auf Folgendes hingewiesen:

Wir haben die Konstruktion des Ausdrucks

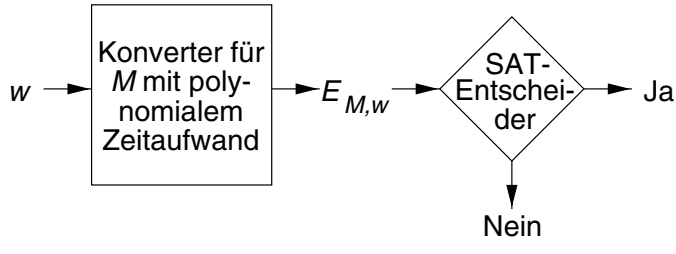
$$E_{M,w} = S \wedge N \wedge F$$

zwar als Funktion sowohl von M als auch von w beschrieben, doch lediglich der Teilausdruck S, »Startet richtig«, ist von w abhängig, und dies nur auf einfache Weise (w befindet sich auf dem Band der anfänglichen Konfiguration). Die weiteren Abschnitte N und F sind lediglich von M und n, der Länge von w, abhängig.

Abschließend stellen wir fest: Wir können für jede NTM M, die in polynomialer Zeit $p(n)$ akzeptiert, einen Algorithmus konstruieren, der eine Eingabe w der Länge n übernimmt und $E_{M,w}$ produziert. Bei einer mehrbändigen deterministischen TM beträgt die Ausführungszeit dieses Algorithmus $O(p^2(n))$, und diese mehrbändige TM kann in eine einbändige TM konvertiert werden, die $O(p^4(n))$ benötigt. Dieser Algorithmus gibt einen Booleschen Ausdruck $E_{M,w}$ aus, der nur dann erfüllbar ist, wenn M mit höchstens $p(n)$ Bewegungen w akzeptiert. ■

Um die Bedeutung des Satzes von Cook hervorzuheben, sehen wir uns an, wie Satz 10.5 darauf anzuwenden ist. Wir nehmen an, es gäbe eine deterministische TM für SAT, die ihre Instanzen in polynomialer Zeit, etwa $p(n)$, erkennt. Dann würde jede von einer polynomialen NTM M akzeptierte Sprache von der in Abbildung 10.4 skizzierten DTM in polynomialer Zeit akzeptiert. Abbildung 10.4 zeigt die Arbeitsweise dieser DTM. Die Eingabe w von M wird in einen Booleschen Ausdruck $E_{M,w}$ transformiert. Dieser Ausdruck ist die Eingabe des SAT-Entscheiders, und die Reaktion des Entscheiders auf $E_{M,w}$ ist genau die gleiche wie die von M auf w.

Abbildung 10.4: Ist SAT in $\mathcal{P}$ enthalten, dann kann von einer auf diese Weise entworfenen DTM gezeigt werden, dass jede Sprache in $\mathcal{NP}$ auch in $\mathcal{P}$ enthalten ist



10.2.4 Übungen zum Abschnitt 10.2

Übung 10.2.1 Wie viele erfüllbare Belegungen besitzen die folgenden Booleschen Ausdrücke? Welche sind in SAT enthalten?

* a) $x \wedge (y \vee \neg x) \wedge (z \vee \neg y)$

b) $(x \vee y) \wedge (\neg(x \vee z) \vee (\neg z \wedge \neg y))$

! ! **Übung 10.2.2** Angenommen, der Graph G bestehe aus vier Knoten 1, 2, 3 und 4. x_{ij} sei für $1 \leq i < j \leq 4$ eine Aussagenvariable, die wir als »Es existiert eine Kante zwischen den Knoten i und j« interpretieren. Jeder Graph mit diesen vier Knoten kann durch eine Belegung repräsentiert werden. Beispielsweise wird der in Abbildung 10.1 dargestellte Graph repräsentiert, wenn x_{14} falsch ist und die anderen fünf Variablen wahr sind. Jede Eigenschaft eines Graphen, die lediglich die Existenz oder Nichtexistenz von Kanten involviert, kann als Boolescher Ausdruck ausgedrückt werden, der nur dann wahr ist, wenn die Belegung der Variablen einen Graphen beschreibt, der diese Eigenschaft besitzt. Schreiben Sie für die folgenden Eigenschaften Ausdrücke:

* a) G besitzt einen Hamiltonschen Kreis.

b) G ist zusammenhängend.

c) G enthält eine Clique der Größe 3, also eine Menge aus drei Knoten, von denen je zwei durch eine Kante verbunden werden (das ist ein Dreieck in G).

d) G enthält mindestens einen isolierten Knoten, also einen Knoten ohne Kanten.

10.3 Ein eingeschränktes Erfüllbarkeitsproblem

Wir planen, eine Vielzahl von Problemen wie etwa das in Abschnitt 10.1.4 erwähnte TSP-Problem als NP-vollständig nachzuweisen. Prinzipiell suchen wir dazu nach polynomialen Reduktionen des SAT-Problems auf jedes Problem, das uns interessiert. Allerdings gibt es ein wichtiges Zwischenproblem, das 3SAT-Problem, das wesentlich einfacher auf typische Probleme zu reduzieren ist als SAT. 3SAT ist zwar auch ein Problem der Erfüllbarkeit Boolescher Ausdrücke, doch besitzen diese Ausdrücke eine sehr reguläre Form: Sie sind die UND-Verknüpfung von »Klauseln«, die wiederum die ODER-Verknüpfung von exakt drei Variablen oder negierten Variablen sind.

In diesem Abschnitt lernen Sie wichtige terminologische Begriffe im Zusammenhang mit Booleschen Ausdrücken kennen. Dann reduzieren wir die Erfüllbarkeit jedes Ausdrucks auf die Erfüllbarkeit von Ausdrücken in der Normalform des 3SAT-Problems. Es ist sehr interessant, dass zwar jeder Boolesche Ausdruck E einen äquivalenten Ausdruck F in der 3SAT-Normalform besitzt, die Größe von F jedoch exponentiell in der Größe von E sein kann. Unsere polynomiale Reduktion von SAT auf 3SAT muss daher subtiler als eine einfache Umformung in Boolescher Algebra sein. Wir müssen jeden SAT-Ausdruck E in einen anderen Ausdruck F in 3SAT-Normalform konvertieren. F ist allerdings nicht notwendigerweise zu E äquivalent. Es gilt nur, dass F ausschließlich dann erfüllbar ist, wenn auch E erfüllbar ist.

10.3.1 Normalformen Boolescher Ausdrücke

Die folgenden Definitionen sind wesentlich:

- Ein *Literal* ist entweder eine Variable oder eine negierte Variable. Beispiele dafür sind x und $\neg y$. Aus Platzgründen werden wir häufig $\overline{y}$ statt des Literals $\neg y$ verwenden.
- Eine *Klausel* ist die logische ODER-Verknüpfung eines oder mehrerer Literale. Beispiele hierfür sind x, $x \vee y$ und $x \vee \overline{y} \vee z$.
- Ein Boolescher Ausdruck hat die *konjunktive Normalform*[3] oder *KNF*, wenn er die UND-Verknüpfung von Klauseln ist.

Um unsere Ausdrücke weiter zu komprimieren, werden wir die alternative Notation verwenden, wobei $\vee$ unter Verwendung des Operators + wie eine Summe und $\wedge$ wie ein Produkt behandelt wird. Bei Produkten verwenden wir normalerweise das Nebeneinandersetzen, also keinen Operator, entsprechend der Verkettung in regulären Ausdrücken. Dann kann eine Klausel als eine »Summe von Literalen« und ein KNF-Ausdruck als »Produkt von Klauseln« bezeichnet werden.

Beispiel 10.10 Der Ausdruck $(x \vee \neg y) \wedge (\neg x \vee z)$ wird in unserer komprimierten Notation als $(x + \overline{y})(\overline{x} + z)$ dargestellt. Er liegt in konjunktiver Normalform vor, da er die UND-Verknüpfung (Produkt) der Klauseln $(x + \overline{y})$ und $(\overline{x} + z)$ ist.

Der Ausdruck $(x + y\overline{z})(x + y + z)(\overline{y} + \overline{z})$ ist nicht in KNF. Er stellt die UND-Verknüpfung der drei Teilausdrücke $(x + y\overline{z})$, $(x + y + z)$ und $(\overline{y} + \overline{z})$ dar. Bei den letzten beiden Ausdrücken handelt es sich um Klauseln, nicht jedoch beim ersten; er ist die Summe eines Literals und eines Produkts zweier Literale.

Der Ausdruck xyz ist in KNF. Er ist das Produkt der drei Klauseln (x), (y) und (z). ■

Ein Ausdruck liegt in *k-konjunktiver Normalform* (k-KNF) vor, wenn er aus dem Produkt von Klauseln besteht. Jede dieser Klauseln ist die Summe von exakt k paarweise verschiedenen Literalen. Beispielsweise ist der Ausdruck $(x + \overline{y})(y + \overline{z})(z + \overline{x})$ in 2-KNF, da jede der Klauseln genau zwei verschiedene Literale enthält.

All diese Einschränkungen von Booleschen Ausdrücken führen zu eigenen Problemen bei der Erfüllbarkeit von Ausdrücken, die den Einschränkungen genügen. Wir werden uns daher mit den folgenden Problemen befassen:

- CSAT ist das Problem: Ist ein in KNF gegebener Boolescher Ausdruck erfüllbar?
- kSAT ist das Problem: Ist ein in k-KNF gegebener Boolescher Ausdruck erfüllbar?

Wir werden sehen, dass CSAT, 3SAT und kSAT für alle $k > 3$ NP-vollständig sind. Für 1SAT und 2SAT allerdings existieren Algorithmen mit linearem Zeitaufwand.

10.3.2 Ausdrücke in KNF konvertieren

Zwei Boolesche Ausdrücke heißen *äquivalent* , wenn sie bei jeder Belegung ihrer Variablen das gleiche Ergebnis besitzen. Sind zwei Ausdrücke äquivalent, dann sind entweder beide oder keiner der beiden erfüllbar. Die Konvertierung beliebiger Ausdrücke in äquivalente KNF-Ausdrücke ist daher ein viel versprechender Ansatz für die

3. »Konjunktion« ist eine Bezeichnung für die logische UND-Verknüpfung.

Umgang mit inkorrekten Eingaben

Jedes der diskutierten Probleme – SAT, CSAT, 3SAT und so weiter – ist eine Sprache über einem festen Alphabet aus acht Zeichen, deren Zeichenreihen wir zum Teil als Boolesche Ausdrücke interpretieren können. Ist eine Zeichenreihe nicht als Ausdruck interpretierbar, dann kann sie nicht in der Sprache SAT enthalten sein. Wenn wir Ausdrücke einer eingeschränkten Form betrachten, so ist eine Zeichenreihe, die zwar ein wohlgeformter Ausdruck, aber kein Ausdruck in der geforderten Form ist, nicht in der Sprache enthalten. Daher antwortet ein Algorithmus, der beispielsweise das CSAT-Problem entscheidet, mit »nein«, wenn ein Boolescher Ausdruck gegeben ist, der zwar erfüllbar ist, jedoch nicht in KNF vorliegt.

Entwicklung einer polynomialen Reduktion von SAT auf CSAT. Diese Reduktion würde CSAT als NP-vollständig nachweisen.

Allerdings sind die Verhältnisse nicht so einfach. Wir können zwar jeden Ausdruck in KNF konvertieren, doch der Zeitaufwand ist im Allgemeinen nicht mehr polynomial. Insbesondere die Länge des KNF-Ausdrucks kann exponentiell in der Länge des ursprüngichen Ausdrucks sein und dann sicherlich einen exponentiellen Aufwand zur Generierung der Ausgabe erfordern.

Glücklicherweise ist die Konvertierung eines beliebigen Booleschen Ausdrucks in einen Ausdruck in KNF nicht die einzige Möglichkeit, die sich bietet, um SAT auf CSAT zu reduzieren und damit CSAT als NP-vollständig zu beweisen. Wir *brauchen* lediglich eine SAT-Instanz E in eine CSAT-Instanz F zu konvertieren, sodass F nur dann erfüllbar ist, wenn E erfüllbar ist. Es ist nicht notwendig, dass E und F äquivalent sind. Es ist nicht einmal notwendig, dass E und F die gleiche Variablenmenge enthalten, und tatsächlich enthält F im Allgemeinen eine Obermenge der Variablen von E.

Die Reduktion von SAT auf CSAT besteht aus zwei Teilen. Zuerst wird die Negation $\neg$ im Ausdrucksbaum nach »unten durchgereicht«, sodass nur noch Variablen negiert werden. Der Boolesche Ausdruck wird also zu einer UND- und ODER-Verknüpfung von Literalen. Diese Transformation produziert einen äquivalenten Ausdruck und erfordert einen Zeitaufwand, der höchstens quadratisch in der Größe des Ausdrucks ist. Der Vorgang erfordert bei sorgfältig entworfenen Datenstrukturen auf einem konventionellen Computer sogar lediglich einen linearen Zeitaufwand.

Im zweiten Schritt schreiben wir einen Ausdruck, der ein Produkt von Klauseln ist; dies ist also die Überführung in KNF. Diese Transformation gelingt durch die Einführung neuer Variablen mit einem Zeitaufwand, der in Bezug auf die Größe des gegebenen Ausdrucks polynomial ist. Der neue Ausdruck F ist im Allgemeinen nicht zum alten Ausdruck E äquivalent. Allerdings ist F lediglich dann erfüllbar, wenn E ebenfalls erfüllbar ist. Genauer gesagt: Ist T eine Belegung, durch die E wahr wird, dann existiert eine *Erweiterung* von T, S genannt, durch die F wahr wird; S ist eine Erweiterung von T, wenn S jeder in T auftretenden Variablen den gleichen Wert wie T zuweist, doch S kann Werte auch Variablen zuweisen, die in T nicht auftreten.

Im ersten Schritt reichen wir die Negation $\neg$ »nach unten durch«. Wir benötigen die folgenden Regeln:

1. $\neg(E \wedge F) \Rightarrow \neg(E) \vee \neg(F)$. Diese Regel, eine der *DeMorganschen Regeln*, ermöglicht uns, die Negation dem $\wedge$ unterzuordnen. Beachten Sie den Nebeneffekt, dass $\wedge$ in $\vee$ geändert wird.

2. $\neg(E \vee F) \Rightarrow \neg(E) \wedge \neg(F)$. Diese DeMorgansche Regel ordnet die Negation dem $\vee$ unter. Als Nebeneffekt wird $\vee$ in $\wedge$ geändert.
3. $\neg(\neg(E)) \Rightarrow E$. Dieses Gesetz der *doppelten Negation* eliminiert ein Paar der Symbole $\neg$, die sich auf den gleichen Ausdruck beziehen.

Beispiel 10.11 Sehen wir uns den Ausdruck $E = \neg((\neg(x + y))(\overline{x} + y))$ an. Beachten Sie, dass er eine Kombination aus beiden Notationen enthält, wobei der Operator $\neg$ explizit verwendet wird, wenn der zu negierende Ausdruck mehr als eine einzelne Variable enthält. Tabelle 10.3 zeigt die Schritte, mit denen die Negation im Ausdruck E nach unten durchgereicht wird, bis sie nur noch in Literalen auftritt.

Tabelle 10.2: Die Negation $\neg$ wird im Ausdrucksbaum nach unten durchgereicht, sodass sie nur noch in Literalen auftritt

Ausdruck	Regel
$\neg((\neg(x + y))(\overline{x} + y))$	Start
$\neg(\neg(x + y)) + \neg(\overline{x} + y)$	(1)
$x + y + \neg(\overline{x} + y)$	(3)
$x + y + (\neg(\overline{x}))\overline{y}$	(2)
$x + y + x\overline{y}$	(3)

Der resultierende Ausdruck ist ein ODER/UND-Ausdruck aus Literalen und mit dem Original äquivalent. Er könnte weiter bis zum Ausdruck $x + y$ vereinfacht werden, doch dies ist für unsere Behauptung, dass jeder Ausdruck umgeformt werden kann, sodass die Negation nur noch in Literalen auftritt, nicht von Bedeutung. ■

Satz 10.12 Jeder Boolesche Ausdruck E ist mit einem Ausdruck F äquivalent, in dem die Negation nur in Literalen auftritt; d. h. sie ist nur noch direkt auf Variable anzuwenden. Außerdem ist die Länge von F in der Anzahl der Symbole von E linear, und F kann mit polynomialem Zeitaufwand aus E konstruiert werden.

Beweis: Der Beweis ist eine Induktion über die Anzahl n der Operatoren ($\wedge$, $\vee$ und $\neg$) in E. Wir zeigen, dass ein äquivalenter Ausdruck F existiert, der die Negation nur in Literalen enthält. Besitzt E $n \geq 1$ Operatoren, dann enthält F höchstens $2n - 1$ Operatoren.

Da F nicht mehr als ein Klammerpaar pro Operator haben muss und die Anzahl der Variablen in einem Ausdruck die Anzahl der Operatoren nicht um mehr als 1 übersteigen kann, folgern wir, dass die Länge von F proportional zur Länge von E ist. Noch wichtiger ist: Da durch die relativ einfache Konstruktion von F der Zeitaufwand proportional zur Länge von F ist, ist er auch proportional zur Länge von E.

Induktionsbeginn: Besitzt E einen Operator, dann muss der Ausdruck von der Form $\neg x$, $x \vee y$ oder $x \wedge y$ für Variable x und y sein. In jedem Fall befindet sich E schon in der geforderten Form, und somit ist $F = E$. Beachten Sie, dass die Beziehung »F besitzt

höchstens doppelt so viele Operatoren wie E minus 1« gilt, da E und F jeweils einen Operator besitzen.

INDUKTIONSSCHRITT: Angenommen, die Aussage ist für alle Ausdrücke wahr, die weniger Operatoren als E enthalten. Ist $\neg$ nicht der höchste Operator von E, dann muss E von der Form $(E_1) \vee (E_2)$ oder $(E_1) \wedge (E_2)$ sein. In jedem Fall kann die Induktionshypothese auf E_1 und E_2 angewendet werden; sie sagt aus, dass äquivalente Ausdrücke F_1 bzw. F_2 existieren, in denen die Negation nur in Literalen auftritt. Dann ist $F = (F_1) \vee (F_2)$ bzw. $F = (F_1) \wedge (F_2)$ äquivalent zu E. Angenommen, E_1 und E_2 besitzen a bzw. b Operatoren, dann besitzt E $a + b + 1$ Operatoren. Nach der Induktionshypothese besitzen F_1 und F_2 höchstens $2a - 1$ bzw. $2b - 1$ Operatoren. F besitzt höchstens $2a + 2b - 1$ Operatoren, und dieser Wert entspricht $2(a + b + 1) - 1$, also der doppelten Anzahl der Operatoren von E minus 1.

Wir betrachten nun den Fall, dass E von der Form $\neg(E_1)$ ist. Es gibt drei Möglichkeiten, die vom obersten Operator von E abhängen. Beachten Sie, dass E_1 einen Operator besitzen muss, da im Induktionsschritt $n > 1$ ist.

1. $E_1 = \neg(E_2)$. Nach dem Gesetz der doppelten Negation ist $E = \neg(\neg(E_2))$ zu E_2 äquivalent. Da E_2 weniger Operatoren als E besitzt, ist die Induktionshypothese anwendbar. Wir suchen nach einem äquivalenten F für E_2, in dem die Negation lediglich in Literalen enthalten ist. F dient ebenso wie E_2 für E. Da F gegenüber E_2 höchstens die doppelte Anzahl von Operatoren minus 1 besitzt, enthält F auch nicht mehr als die doppelte Anzahl der Operatoren von E minus 1.

2. $E_1 = (E_2) \vee (E_3)$. Nach DeMorgans Regel ist $E = \neg((E_2) \vee (E_3))$ zu $(\neg(E_2)) \wedge (\neg(E_3))$ äquivalent. Sowohl $\neg(E_2)$ als auch $\neg(E_3)$ besitzen weniger Operatoren als E, und daher existieren nach der Induktionshypothese äquivalente F_2 bzw. F_3, in denen die Negation lediglich in Literalen auftritt. $F = (F_2) \wedge (F_3)$ ist also äquivalent zu E. Wir behaupten außerdem, dass die Anzahl der Operatoren in F nicht zu hoch ist. Angenommen, E_1 und E_2 besitzen a bzw. b Operatoren. Da $\neg(E_2)$ und $\neg(E_3)$ $a + 1$ bzw. $b + 1$ Operatoren besitzen und F_2 sowie F_3 aus diesen Ausdrücken konstruiert werden, wissen wir nach der Induktionshypothese, dass F_1 und F_2 höchstens $2(a + 1) - 1$ bzw. $2(b + 1) - 1$ Operatoren enthalten. F besitzt daher höchstens $2a + 2b + 3$ Operatoren. Diese Anzahl ist genau das Doppelte der Operatoren von E minus 1.

3. $E_1 = (E_2) \wedge (E_3)$. Die Beweisführung ist unter Verwendung der zweiten DeMorganschen Regel analog zu (2).

10.3.3 NP-Vollständigkeit von CSAT

Nun müssen wir einen Ausdruck E, die UND- und ODER-Verknüpfung von Literalen, in KNF konvertieren. Wie schon angemerkt, müssen wir eine Transformation verwenden, bei der wir davon absehen, die Äquivalenz von E und F zu erhalten, um mit polynomialem Zeitaufwand einen Ausdruck F aus E zu produzieren, wobei F nur dann erfüllbar ist, wenn E ebenfalls erfüllbar ist. Dazu ist es erforderlich, einige neue Variable in F aufzunehmen, die in E nicht vorhanden sind. Wir werden diesen »Trick« im Beweis des Satzes verwenden, dass CSAT NP-vollständig ist, und die Konstruktion im Beweis dann anhand eines Beispiels erläutern.

Beschreibungen von Algorithmen

Formal entspricht die Ausführungszeit einer Reduktion der Zeit, die zur Ausführung auf einer einbändigen Turing-Maschine notwendig ist, doch diese Algorithmen sind unnötig kompliziert. Wir wissen, dass die Mengen der Probleme, die mit konventionellen Computern, mit mehrbändigen TMs oder mit einbändigen TMs mit polynomialem Zeitaufwand gelöst werden können, identisch sind, wenn sich auch die Grade des polynomialen Aufwands unterscheiden können. Wenn wir nun einige komplizierte Algorithmen entwickeln, die wir für die Reduktion eines NP-vollständigen Problems auf ein anderes Problem benötigen, einigen wir uns darauf, dass der Zeitaufwand durch effiziente Implementierungen auf einem konventionellen Computer gemessen wird. Dadurch vermeiden wir Details, wie z. B. die Manipulation von Bändern, und können uns auf die wesentlichen Ideen der Algorithmen konzentrieren.

Satz 10.13 CSAT ist NP-vollständig.

BEWEIS: Wir zeigen, wie SAT mit polynomialem Zeitaufwand auf CSAT reduziert wird. Zuerst verwenden wir die Methode aus Satz 10.12, um eine gegebene SAT-Instanz in einen Ausdruck E zu konvertieren, dessen Negationen lediglich in Literalen auftreten. Dann zeigen wir, wie E mit polynomialem Zeitaufwand in einen KNF-Ausdruck F konvertiert wird, der nur dann erfüllbar ist, wenn auch E erfüllbar ist. Die Konstruktion von F erfolgt durch eine Induktion über die Länge von E. Dabei ist F etwas spezieller, als von uns benötigt. Präziser ausgedrückt, zeigen wir durch Induktion über die Länge von E Folgendes:

- Es gibt eine Konstante c, sodass zu einem Booleschen Ausdruck E mit Länge n, der Negationen nur in Literalen enthält, ein Ausdruck F mit folgenden Eigenschaften existiert:
  a) F liegt in KNF und besteht aus höchstens n Klauseln.
  b) F ist mit einem Zeitaufwand von höchstens cn^2 aus E konstruierbar.
  c) Durch eine Belegung T für E wird E nur dann wahr, wenn eine Erweiterung S von T existiert, durch die F wahr wird.

INDUKTIONSBEGINN: Besteht E aus einem oder zwei Symbolen, dann handelt es sich um ein Literal. Ein Literal ist eine Klausel, und daher befindet sich E schon in KNF.

INDUKTIONSSCHRITT: Angenommen, jeder Ausdruck, der kürzer als E ist, kann in ein Produkt von Klauseln konvertiert werden, und diese Konvertierung erfordert bei einem Ausdruck der Länge n höchstens den Zeitaufwand cn^2. Abhängig vom obersten Operator im Baum für E gibt es zwei Fälle:

Fall 1: $E = E_1 \wedge E_2$. Nach der Induktionshypothese existieren zu E_1 und E_2 abgeleitete Ausdrücke F_1 bzw. F_2 in KNF. Alle diese Ausdrücke und nur die erfüllenden Belegungen für E_1 können in eine erfüllende Belegung für F_1 erweitert werden, und das Gleiche gilt für E_2 und F_2. Ohne Beschränkung der Allgemeinheit (o.B.d.A.) können wir annehmen, dass die Variablen von F_1 und F_2 disjunkt sind, ausgenommen die Variablen, die in E auftreten; wenn wir also Variablen in F_1 und/oder F_2 aufnehmen müssen, verwenden wir dafür paarweise verschiedene Variablen.

Sei $F = F_1 \wedge F_2$. Offensichtlich handelt es sich bei $F_1 \wedge F_2$ um einen KNF-Ausdruck, wenn auch F_1 und F_2 KNF-Ausdrücke sind. Wir müssen zeigen, dass eine Belegung T

für E nur dann in eine erfüllende Belegung für F erweitert werden kann, wenn T E erfüllt.

(Wenn-Teil) Angenommen, T erfülle E. T_1 sei das nur auf die in E_1 auftretenden Variablen eingeschränkte T; T_2 sei das Gleiche für E_2. Nach der Induktionshypothese können T_1 und T_2 dann in Belegungen S_1 und S_2 erweitert werden, die F_1 bzw. F_2 erfüllen. S soll mit S_1 und S_2 für jede definierte Variable übereinstimmen. Beachten Sie, dass S immer konstruiert werden kann, denn F_1 und F_2 haben nur solche gemeinsamen Variablen, die aus E stammen, und S_1 sowie S_2 müssen für diese Variablen übereinstimmen. S ist dann aber eine Erweiterung von T, die F erfüllt.

(Nur-wenn-Teil) Umgekehrt nehmen wir an, T besitze eine Erweiterung S, die F erfüllt. T_1 (bzw. T_2) sei auf die Variablen von E_1 (bzw. E_2) beschränkt. Sei S, auf die Variablen von F_1 (bzw. F_2) beschränkt, S_1 (bzw. S_2); dann ist S_1 eine Erweiterung von T_1 und S_2 eine Erweiterung von T_2. Da F die UND-Verknüpfung von F_1 und F_2 ist, muss S_1 F_1 und ebenso S_2 F_2 erfüllen. Nach der Induktionshypothese muss T_1 (bzw. T_2) E_1 (bzw. E_2) erfüllen. T erfüllt also E.

Fall 2: $E = E_1 \vee E_2$. Wie in Fall 1 verwenden wir die Induktionshypothese und stellen fest, dass KNF-Ausdrücke F_1 und F_2 mit folgenden Eigenschaften existieren:

1. Eine Belegung für E_1 (bzw. E_2) erfüllt E_1 (bzw. E_2) ausschließlich dann, wenn sie in eine erfüllende Belegung für F_1 (bzw. F_2) erweitert werden kann.
2. Die Variablen von F_1 und F_2 sind bis auf die Variablen, die in E auftreten, disjunkt.
3. F_1 und F_2 sind in KNF.

Wir können nicht einfach die ODER-Verknüpfung von F_1 und F_2 nehmen und das gewünschte F konstruieren, da der resultierende Ausdruck nicht in KNF wäre. Wir verwenden stattdessen eine etwas komplizierte Konstruktion, die sich die Tatsache zunutze macht, dass wir lediglich die Erfüllbarkeit, aber nicht die Äquivalenz erhalten wollen. Angenommen,

$$F_1 = g_1 \wedge g_2 \wedge \ldots \wedge g_p$$

und $F_2 = h_1 \wedge h_2 \wedge \ldots \wedge h_p$, wobei die g_i und h_j Klauseln sind. Wir führen eine neue Variable y ein und definieren:

$$F = (y + g_1) \wedge (y + g_2) \wedge \ldots \wedge (y + g_p) \wedge (\overline{y} + h_1) \wedge (\overline{y} + h_2) \wedge \ldots \wedge (\overline{y} + h_q)$$

Wir müssen beweisen, dass eine Belegung T für E E nur dann erfüllt, wenn T in eine Belegung S erweitert werden kann, die F erfüllt.

(Nur-wenn-Teil) Angenommen, T erfülle E. Wie in Fall 1 sei T_1 (bzw. T_2) das auf die Variablen von E_1 (bzw. E_2) beschränkte T. Da $E = E_1 \vee E_2$, erfüllt T E_1 oder E_2. Wir nehmen an, T erfülle E_1. Dann kann T_1, das dem auf die Variablen von E_1 beschränkten T entspricht, in S_1 erweitert werden, das F_1 erfüllt. Wir konstruieren eine Erweiterung S für T wie folgt; S wird den oben definierten Ausdruck F erfüllen:

1. Für alle Variablen x in F_1 wird definiert: $S(x) = S_1(x)$.
2. $S(y) = 0$. Durch diese Definition sind alle aus F_2 abgeleiteten Klauseln in F wahr.
3. Für alle Variablen x, die in F_2, jedoch nicht in F_1 enthalten sind, kann $S(x)$ entweder 0 oder 1 sein.

Mit Regel 1 werden alle aus den g abgeleiteten Klauseln wahr. Mit Regel 2 werden auch alle aus den h abgeleiteten Klauseln wahr. S erfüllt daher F.

Wenn T E_2 erfüllt, nicht aber E_1, dann ist der Beweis im Wesentlichen der gleiche, abgesehen von $S(y) = 1$ in Regel 2. Außerdem muss $S(x)$ mit $S_2(x)$ übereinstimmen, wann immer $S_2(x)$ definiert ist, doch ist $S(x)$ für Variablen, die nur in S_1 erscheinen, beliebig. Wir folgern, dass S auch in diesem Fall F erfüllt.

(Wenn-Teil) Angenommen, die Belegung T für E sei in die Belegung S für F erweitert und S erfülle F. Es gibt zwei Fälle, abhängig davon, welcher Wahrheitswert y zugewiesen wird. Zuerst nehmen wir an, dass $S(y) = 0$ ist. Dann sind alle von den h abgeleiteten Klauseln von F wahr. Allerdings bietet y keine Hilfe bei Klauseln der Form $(y + g_i)$, die von den g abgeleitet sind. S muss also dafür sorgen, dass jedes g_i selbst und damit F_1 wahr wird.

Präziser gesagt, sei S_1 das auf die Variablen von F_1 eingeschränkte S. Dann erfüllt S_1 F_1. Auf Grund der Induktionshypothese muss T_1, das dem auf die Variablen von E_1 eingeschränkten T entspricht, E_1 erfüllen. Der Grund liegt darin, dass S_1 eine Erweiterung von T_1 ist. Da T_1 F_1 erfüllt, muss T $E = E_1 \vee E_2$.

Wir müssen außerdem den Fall $S(y) = 1$ betrachten, doch dieser Fall ist zum gerade Behandelten analog und sei daher dem Leser überlassen. Wir folgern, dass T E erfüllt, wenn S F erfüllt.

Nun müssen wir zeigen, dass der Zeitaufwand zur Konstruktion von F aus E höchstens quadratisch in n, der Länge von E, ist. Unabhängig davon, welcher Fall vorliegt, erfordert die Aufteilung von E in E_1 und E_2 sowie die Konstruktion von F aus F_1 und F_2 jeweils einen Zeitaufwand, der linear in der Größe von E ist. Sei dn also eine Obergrenze für die Zeit zur Konstruktion von E_1 und E_2 aus E zuzüglich der Zeit zur Konstruktion von F aus F_1 und F_2. Dann gibt es eine Rekursionsgleichung für $T(n)$, den Zeitaufwand für die Konstruktion von F aus einem E der Länge n:

$$T(1) = T(2) \leq e \text{ für eine Konstante } e$$
$$T(n) \leq dn + c\max_{0<i<n-1}(T(i) + T(n-1-i)) \text{ für } n \geq 3$$

Hierin ist c eine Konstante, die noch so zu bestimmen ist, dass wir zeigen können, dass $T(n) \leq cn^2$ gilt. Die Anfangsregel für $T(1)$ und $T(2)$ sagt einfach aus, dass wir keine Rekursion benötigen, wenn E aus einem einzelnen Symbol oder einem Paar aus Symbolen besteht und der gesamte Prozess einen Zeitaufwand von e erfordert. Die rekursive Regel basiert auf der Tatsache, dass E_2 die Länge $n - i - 1$ hat, wenn E aus Teilausdrücken E_1 und E_2 besteht, die mit dem Operator $\wedge$ oder $\vee$ miteinander verknüpft sind, und wenn E_1 die Länge i hat. Zudem besteht die gesamte Konvertierung von E nach F aus den zwei einfachen Schritten – E in E_1 und E_2 sowie F_1 und F_2 in F ändern –, von denen wir wissen, dass sie höchstens einen Zeitaufwand von dn erfordern, sowie aus den beiden rekursiven Konvertierungen von E_1 in F_1 und E_2 in F_2.

Wir müssen durch Induktion über n zeigen, dass eine Konstante c existiert, sodass für alle n $T(n) \leq cn^2$ gilt.

INDUKTIONSBEGINN: Für $n = 1$ und $n = 2$ wählen wir ein c, das mindestens so groß wie $4d$ ist.

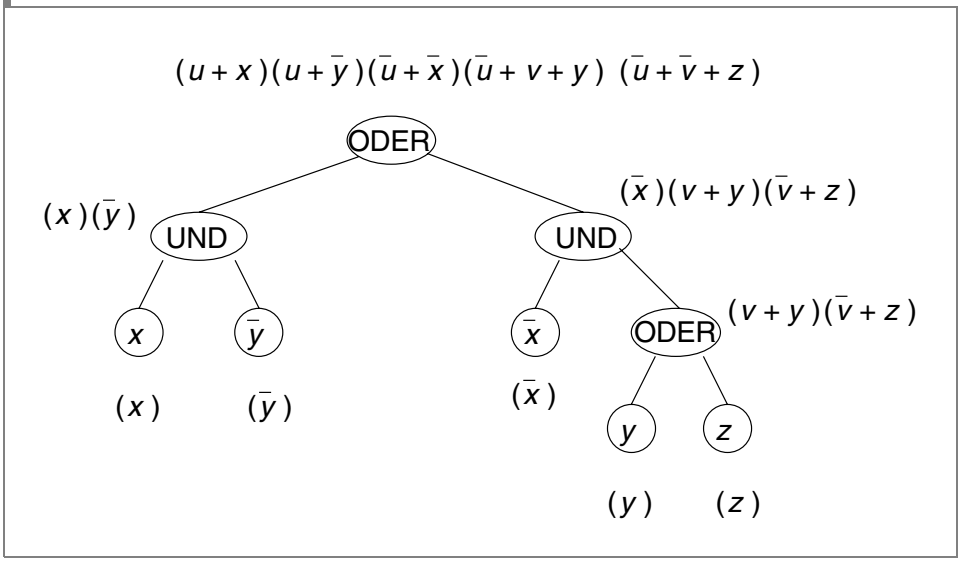
INDUKTIONSSCHRITT: Wir nehmen an, $n \geq 3$ und die Aussage sei richtig für jede Länge kleiner als n. Dann ist $T(i) \leq ci^2$ und $T(n - i - 1) \leq c(n - i - 1)^2$. Daher gilt:

$$T(i) + T(n - i - 1) \leq c\,*(n^2 - 2i(n - i) - 2(n - i) + 1) \qquad (10.1)$$

Da $n \geq 3$ und $0 < i < n - 1$, ist $2i(n - i)$ mindestens n und $2(n - i)$ mindestens 2. Die rechte Seite von (10.1) ist also kleiner als $c*(n^2 - n)$ für jedes i im erlaubten Bereich. Die rekursive Regel in der Definition von $T(n)$ führt daher zu $T(n) \leq dn + cn^2 - cn$. Wählen wir $c \geq d$, dann können wir schließen, dass $T(n) \leq cn^2$ für n gilt, und damit ist die Induktion abgeschlossen. Die Konstruktion von F aus E erfordert also den Zeitaufwand $O(n^2)$. ■

Beispiel 10.14 Wir zeigen, wie die Konstruktion aus Satz 10.13 auf den einfachen Ausdruck $E = x\bar{y} + \bar{x}(y + z)$ anzuwenden ist. Abbildung 10.9 zeigt den Ableitungsbaum dieses Ausdrucks. Bei jedem Knoten ist der KNF-Ausdruck angegeben, der für den von diesem Knoten repräsentierten Ausdruck konstruiert wurde.

Abbildung 10.5: Einen Booleschen Ausdruck in KNF überführen



Die Blätter entsprechen den Literalen, und bei jedem Blatt besteht der KNF-Ausdruck aus einer Klausel, die aus dem Literal selbst besteht. Das Blatt $\bar{y}$ besitzt beispielsweise den zugehörigen KNF-Ausdruck $(\bar{y})$. Die Klammern sind nicht notwendig, doch wir verwenden sie in KNF-Ausdrücken, um Sie daran zu erinnern, dass wir über ein Produkt von Klauseln sprechen.

Bei UND-Knoten wird der KNF-Ausdruck einfach aus dem Produkt (UND) aller Klauseln der beiden Teilausdrücke konstruiert. Der Knoten des Teilausdrucks $\bar{x}(y + z)$ besitzt beispielsweise einen zugehörigen KNF-Ausdruck, der aus dem Produkt der Klausel für $\bar{x}$, also $(\bar{x})$, und den beiden Klauseln für $y + z$, also $(v + y)(\bar{v} + z)$, besteht.[4]

Bei einem ODER-Knoten müssen wir eine neue Variable einführen. Wir fügen sie in alle Klauseln des linken Operanden ein und nehmen deren Negation in die Klauseln des rechten Operanden auf. Sehen wir uns beispielsweise den Wurzelknoten in

4. In diesem Spezialfall ist der Teilausdruck $y + z$ bereits eine Klausel. Wir müssten die allgemeine Konstruktion des ODER von Ausdrücken nicht ausführen und könnten $(y + z)$ als das Produkt von Klauseln produzieren, das zu $y + z$ äquivalent ist. In diesem Beispiel halten wir uns jedoch an die allgemeinen Regeln.

Abbildung 10.9 an. Dies ist die ODER-Verknüpfung der Ausdrücke $x\overline{y}$ und $\overline{x}(y + z)$, deren KNF-Ausdrücke $(x)(\overline{y})$ bzw. $(\overline{x})(v + y)(\overline{v} + z)$ lauten. Wir führen eine neue Variable u ein, die nicht negiert in die erste und negiert in die zweite Klauselgruppe eingefügt wird. Daraus resultiert:

$$F = (u + x)(u + \overline{y})(\overline{u} + \overline{x})(\overline{u} + v + y)(\overline{u} + \overline{v} + z)$$

Satz 10.13 sagt aus, dass jede Belegung T, die E erfüllt, in eine Belegung S erweitert werden kann, die F erfüllt. Beispielsweise wird E von der Zuweisung $T(x) = 0$, $T(y) = 1$ und $T(z) = 1$ erfüllt. Wir können T in S erweitern, indem wir $S(u) = 1$ und $S(v) = 0$ zu den durch T vorgegebenen $S(x) = 0$, $S(y) = 1$ und $S(z) = 1$ hinzufügen. Überprüfen Sie, ob S F erfüllt.

Beachten Sie, dass wir bei der Wahl von S $S(u) = 1$ wählen mussten, da durch T nur der zweite Teil von E wahr wird, also $\overline{x}(y + z)$. Daher ist $S(u) = 1$ erforderlich, damit die Klauseln $(u + x)(u + \overline{y})$, die aus dem ersten Teil von E stammen, wahr werden. Für v könnten wir allerdings jeden Wert wählen, da im Teilausdruck $y + z$ mit T beide Seiten des ODER-Ausdrucks wahr sind. ■

10.3.4 NP-Vollständigkeit von 3SAT

Wir stellen nun eine noch kleinere Klasse von Booleschen Ausdrücken mit einem NP-vollständigen Erfüllbarkeitsproblem vor. Das schon früher erwähnte 3SAT-Problem lautet wie folgt:

- Gegeben sei ein Boolescher Ausdruck E aus dem Produkt von Klauseln, die jeweils die Summe von drei paarweise verschiedenen Variablen sind. Ist E erfüllbar?

Die 3-KNF-Ausdrücke sind zwar nur eine kleine Teilmenge der KNF-Ausdrücke, doch sie sind komplex genug, um ihren Erfüllbarkeitstest NP-vollständig zu machen, wie der nächste Satz zeigt.

Satz 10.15 3SAT ist NP-vollständig.

Beweis: 3SAT ist in *NP* enthalten, da SAT in *NP* enthalten ist. Zum Beweis der NP-Vollständigkeit werden wir CSAT wie folgt auf 3SAT reduzieren. Gegeben sei ein KNF-Ausdruck $E = e_1 \wedge e_2 \wedge \ldots \wedge e_k$, in dem wir jede Klausel e_i wie folgt ersetzen, um einen neuen Ausdruck F zu konstruieren. Der Zeitaufwand bei der Konstruktion von F ist linear zur Länge von E, und wir werden sehen, dass eine Belegung E nur dann erfüllt, wenn sie in eine erfüllende Belegung für F erweitert werden kann.

1. Ist e_i ein einzelnes Literal, etwa (x)[5], dann verwenden wir zwei neue Variablen u und v. Wir ersetzen (x) durch ein Produkt von vier Klauseln $(x + u + v)(x + u + \overline{v})(x + \overline{u} + v)(x + \overline{u} + \overline{v})$. Da u und v in allen Kombinationen vorkommen, können alle vier Klauseln nur dann erfüllt werden, wenn x wahr ist. Daher können alle und nur die erfüllenden Belegungen für E in erfüllende Belegungen für F erweitert werden.

2. Angenommen, e_i sei die Summe von zwei Literalen $(x + y)$. Wir verwenden eine neue Variable z und ersetzen e_i durch das Produkt von zwei Klauseln $(x +$

5. Aus Gründen der Bequemlichkeit sehen wir Literale als nicht negierte Variablen wie etwa x an. Die Konstruktion ist jedoch genauso gut anzuwenden, wenn einige oder auch alle Literale wie etwa $\overline{x}$ negiert wären.

$y + z)(x + y + \overline{z})$. Wie in Fall 1 können die beiden Klauseln nur dann erfüllt werden, wenn $(x + y)$ erfüllt wird.

3. Ist e_i die Summe von drei Literalen, dann liegt der Ausdruck schon in der für 3-KNF erforderlichen Form vor, und wir belassen e_i in dem zu konstruierenden Ausdruck F.

4. Sei $e_i = (x_1 + x_2 + \ldots + x_m)$ für $m \geq 4$. Wir verwenden die neuen Variablen y_1, y_2, …, y_{m-3} und ersetzen e_i durch das Produkt der Klauseln

$$(x_1 + x_2 + y_1)(x_3 + \overline{y_1} + y_2)(x_4 + \overline{y_2} + y_3)\cdots$$
$$(x_{m-2} + \overline{y_{m-4}} + y_{m-3})(x_{m-1} + x_m + \overline{y_{m-3}}) \qquad (10.2)$$

Durch eine Belegung T, die E erfüllt, muss mindestens ein Literal von e_i wahr sein; sagen wir, x_j ist wahr (Sie wissen, dass x_j eine negierte oder nicht negierte Variable sein kann). Wenn dann $y_1, y_2, \ldots, y_{j-2}$ wahr und $y_{j-1}, y_j, \ldots, y_{m-3}$ falsch ist, sind alle Klauseln aus (10.2) erfüllt. Daher kann T erweitert werden, sodass diese Klauseln erfüllt werden. Wenn umgekehrt alle x_k durch T falsch sind, dann kann T nicht erweitert werden, sodass (10.2) wahr wird. Dies liegt daran, dass $m - 2$ Klauseln existieren und jedes der $m - 3$ y_l nur eine Klausel wahr machen kann, unabhängig davon, ob y_l wahr oder falsch ist.

Wir haben also gezeigt, wie jede Instanz E von CSAT auf eine Instanz F von 3SAT reduziert wird, sodass F nur dann erfüllbar ist, wenn E erfüllbar ist. Die Konstruktion erfordert offensichtlich einen Zeitaufwand, der linear zur Länge von E ist, da keiner der vier obigen Fälle eine Klausel um mehr als den Faktor 32/3 (der Quotient der Symbolanzahlen in Fall 1) expandiert, und es ist einfach, die benötigten Symbole von F mit einem Zeitaufwand zu berechnen, der proportional zur Anzahl dieser Symbole ist. Da CSAT NP-vollständig ist, folgt daraus, dass 3SAT ebenfalls NP-vollständig ist. ■

10.3.5 Übungen zum Abschnitt 10.3

Übung 10.3.1 Transformieren Sie die folgenden Booleschen Ausdrücke in 3-KNF:

* a) $xy + \overline{x}z$

b) $wxyz + u + v$

c) $wxy + \overline{x}uv$

Übung 10.3.2 Das Problem *4TA-SAT* ist wie folgt definiert: Gegeben sei ein Boolescher Ausdruck E; besitzt E mindestens vier erfüllende Belegungen? Zeigen Sie, dass 4TA-SAT NP-vollständig ist.

Übung 10.3.3 In dieser Übung werden wir eine Familie von 3-KNF-Ausdrücken definieren. Der Ausdruck E_n besitzt n Variablen $x_1, x_2, \ldots, x_n$. E_n besitzt für jede Menge $\{i_1, i_2, i_3\}$ aus drei paarweise verschiedenen ganzen Zahlen zwischen 1 und n die Klauseln $(x_{i_1} + x_{i_2} + x_{i_3})$ und $(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$. Ist E_n erfüllbar für:

*! a) $n = 4$?

!! b) $n = 5$?

! **Übung 10.3.4** Entwickeln Sie einen polynomialen Algorithmus zur Lösung des Problems 2SAT, also die Erfüllbarkeit von Booleschen Ausdrücken in KNF, die nur zwei Literale pro Klausel besitzen. *Hinweis:* Ist eines der beiden Literale einer Klausel falsch, dann muss das Andere wahr sein. Beginnen Sie mit einer Annahme über die Belegung einer Variablen, und verfolgen Sie dann die Konsequenzen für andere Variablen.

10.4 Weitere NP-vollständige Probleme

Sie lernen nun ein kleines Beispiel für den Prozess kennen, mit dessen Hilfe ein NP-vollständiges Problem zu Beweisen führt, dass andere Probleme ebenfalls NP-vollständig sind. Dieser Prozess des Entdeckens neuer NP-vollständiger Probleme hat zwei wichtige Effekte:

- Wenn wir ein Problem als NP-vollständig erkennen, dann wissen wir, dass nur eine geringe Wahrscheinlichkeit besteht, einen effizienten Algorithmus zu entwickeln, der das Problem löst. Wir sollten also nach Heuristiken, partiellen Lösungen, Näherungswerten oder anderen Möglichkeiten suchen, um zu vermeiden, das Problem in voller Allgemeinheit direkt anzugehen. Wir können das guten Gewissens tun, weil eben eine praktisch brauchbare allgemeine Lösung fast sicher nicht existiert.
- Jedes Mal, wenn wir ein neues NP-vollständiges Problem P zur Liste hinzufügen, verstärken wir die (unbewiesene) Vermutung, dass *alle* NP-vollständigen Probleme einen exponentiellen Zeitaufwand erfordern. Die Bemühungen bei der Suche nach einem Algorithmus mit polynomialem Zeitaufwand für Problem P waren eigentlich, wenn auch unbewusst, Bemühungen um den Beweis, dass $\mathcal{P} = \mathcal{NP}$ ist. Es ist der gesamte Aufwand der erfolglosen Versuche vieler hervorragender Mathematiker und Informatiker zu beweisen, dass $\mathcal{P} = \mathcal{NP}$ ist, der uns letztendlich überzeugt, dass $\mathcal{P} = \mathcal{NP}$ sehr unwahrscheinlich ist und dass vielmehr alle NP-vollständigen Probleme einen exponentiellen Zeitaufwand erfordern.

In diesem Abschnitt lernen Sie verschiedene NP-vollständige Probleme im Zusammenhang mit Graphen kennen. Diese Probleme gehören zu denen, die häufig bei der Lösung von Fragen von praktischer Bedeutung auftreten. Wir behandeln das Problem des Handlungsreisenden (TSP), das Sie schon in Abschnitt 10.1.4 kennen gelernt haben. Wir werden außerdem zeigen, dass eine einfachere und ebenso wichtige Version, das Problem des Hamiltonschen Kreises (Hamilton-Circuit Problem, HC), NP-vollständig ist, und damit das allgemeinere TSP-Problem ebenfalls als NP-vollständig nachweisen. Wir stellen einige weitere Probleme vor, die sich mit dem »Überdecken« in Graphen beschäftigen, beispielsweise das Problem der Knotenüberdeckung, das nach der kleinsten Menge von Knoten sucht, die alle Kanten derart überdecken, dass sich wenigstens ein Ende jeder Kante in der gewählten Menge befindet.

10.4.1 NP-vollständige Probleme beschreiben

Bei der Einführung neuer NP-vollständiger Probleme verwenden wir eine stilisierte Form der Definition:

1. Der *Name* des Problems und gewöhnlich eine Abkürzung wie 3SAT oder TSP.
2. Die *Eingabe* des Problems: Was wird wie repräsentiert?

3. Die gewünschte *Ausgabe*: Unter welchen Umständen sollte die Ausgabe »ja« lauten?
4. Das Problem, von dem aus eine Reduktion erfolgt, um das neue Problem als NP-vollständig nachzuweisen.

Beispiel 10.16 So könnte die Beschreibung des Problems 3SAT in Verbindung mit der NP-Vollständigkeit aussehen:

Problem: Erfüllbarkeit von 3-KNF-Ausdrücken (3SAT).

Eingabe: Ein Boolescher Ausdruck in 3-KNF.

Ausgabe: »Ja«, genau dann, wenn der Ausdruck erfüllbar ist.

Reduktion von: CSAT.

10.4.2 Das Problem unabhängiger Mengen

Sei G ein ungerichteter Graph. Eine Teilmenge I der Knoten von G ist eine *unabhängige Menge* (independent Set, IS), wenn keine zwei Knoten aus I durch eine Kante von G verbunden sind. Eine unabhängige Menge ist *maximal*, wenn es keine unabhängige Menge desselben Graphen mit mehr Knoten gibt.

Beispiel 10.16 Im Graphen in Abbildung 10.1 (siehe Abschnitt 10.1.2) ist {1, 4} eine unabhängige Menge. Dies ist die einzige Menge der Größe (Anzahl der Elemente) zwei, die unabhängig ist, da alle anderen Knotenpaare durch eine Kante verbunden sind. Daher ist keine Menge der Größe drei oder mehr unabhängig; beispielsweise ist {1, 2, 4} nicht unabhängig, da eine Kante zwischen 1 und 2 existiert. Daher ist {1, 4} eine maximal unabhängige Menge. Tatsächlich handelt es sich um die einzige maximal unabhängige Menge dieses Graphen, obwohl ein Graph im Allgemeinen viele maximal unabhängige Mengen haben kann. Auch {1} ist eine unabhängige Menge dieses Graphen, aber sie ist nicht maximal. ■

In der kombinatorischen Optimierung wird das Problem maximal unabhängiger Mengen üblicherweise wie folgt ausgedrückt: Gegeben sei ein Graph; finde eine maximal unabhängige Menge. Wie bei allen Problemen in der Theorie der nicht behandelbaren Probleme müssen wir allerdings auch hier unser Problem so formulieren, dass es sich mit Ja/Nein-Antworten lösen lässt. Wir müssen daher einen unteren Grenzwert in die Problemformulierung aufnehmen und fragen, ob wir für den Graphen eine unabhängige Menge finden können, deren Größe größer oder gleich dem unteren Grenzwert ist. Die formale Definition des Problems maximal unabhängiger Mengen lautet wie folgt:

Problem: Unabhängige Menge (IS).

Eingabe: Ein Graph G sowie ein unterer Grenzwert k, der zwischen 1 und der Anzahl der Knoten von G liegen muss.

Ausgabe: »Ja«, genau dann, wenn G eine unabhängige Menge von k Knoten besitzt.

Reduktion von: 3SAT

Wir müssen IS mit einer polinomialen Reduktion von 3SAT als NP-vollständig nachweisen. Diese Reduktion beschreibt der nächste Satz.

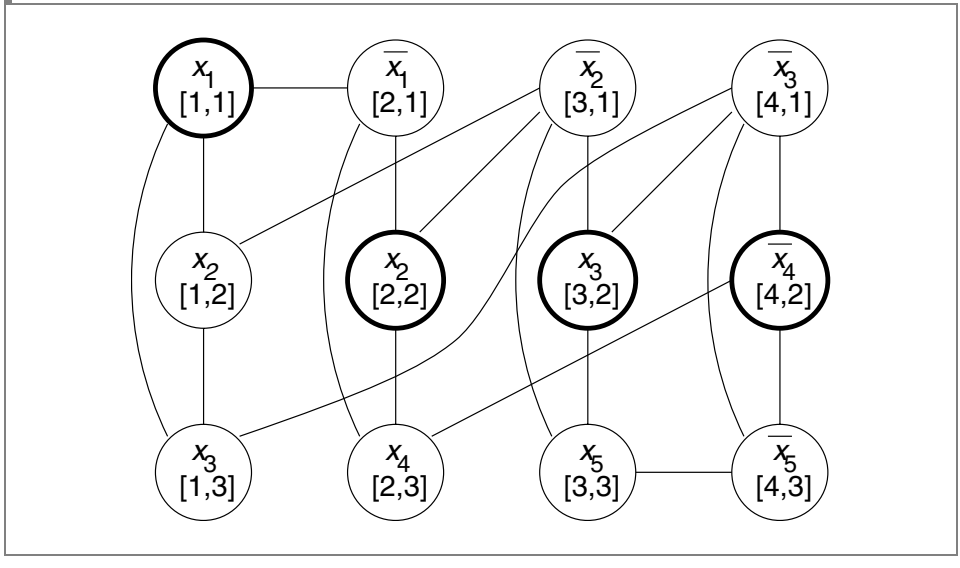
Satz 10.18 Das Problem unabhängiger Mengen ist NP-vollständig.

BEWEIS: Zunächst ist einfach zu sehen, dass IS in $\mathcal{NP}$ enthalten ist. Zu einem gegebenen Graphen G und einem Grenzwert k raten wir k Knoten und prüfen sie auf Unabhängigkeit.

Wir zeigen nun die Reduktion von 3SAT auf IS. Sei $E = (e_1)(e_2)\ldots(e_m)$ ein 3-KNF-Ausdruck. Wir konstruieren aus E einen Graphen G mit $3m$ Knoten, die wir mit $[i, j]$ benennen, wobei $1 \le i \le m$ und $j = 1, 2$ oder 3 ist. Der Knoten $[i, j]$ repräsentiert das j-te Literal in der Klausel e_i. Abbildung 10.6 zeigt ein Beispiel für einen auf dem folgenden 3-KNF-Ausdruck basierenden Graphen:

$$(x_1 + x_2 + x_3)(\overline{x}_1 + x_2 + x_4)(\overline{x}_2 + x_3 + x_5)(\overline{x}_3 + \overline{x}_4 + \overline{x}_5)$$

Abbildung 10.6: Konstruktion einer unabhängigen Menge aus einem erfüllbaren Booleschen Ausdruck in 3-KNF



Die Spalten repräsentieren die Klauseln. Wir erläutern im Folgenden, wie die Kanten zustande kommen.

Das Raffinierte an der Konstruktion von G besteht in der Verwendung von Kanten derart, dass jede unabhängige Menge aus m Knoten eine Möglichkeit darstellt, den Ausdruck E zu erfüllen. Es gibt zwei grundlegende Ideen:

1. Wir wollen sicherstellen, dass zu jeder Klausel jeweils nur ein Knoten gewählt werden kann. Dazu verbinden wir alle Knotenpaare innerhalb einer Spalte mit einer Kante; wir zeichnen also die Kanten ($[i, 1]$, $[i, 2]$), ($[i, 1]$, $[i, 3]$) und ([i, 2], $[i, 3]$) für alle i, wie Abbildung 10.8 zeigt.

2. Wir müssen verhindern, dass Knoten für die unabhängige Menge ausgewählt werden, die komplementäre Literale repräsentieren. Existieren also zwei Knoten $[i_1, j_1]$ und $[i_2, j_2]$, sodass einer eine Variable x und der andere $\overline{x}$ repräsentiert, dann verbinden wir diese beiden Knoten mit einer Kante, sodass nicht beide Knoten in die unabhängige Menge aufgenommen werden können.

Der Grenzwert k für den Graphen, der mit diesen beiden Regeln konstruiert wird, ist m.

Es ist leicht zu sehen, warum der Graph G und der Grenzwert k aus dem Ausdruck E mit polynomialem Zeitaufwand erstellt werden können, der proportional zum Quadrat der Länge von E ist und somit die Konvertierung von E in G eine polynomiale Reduktion ist. Wir müssen zeigen, dass sie 3SAT korrekt auf IS reduziert:

- E ist dann und nur dann erfüllbar, wenn für G eine unabhängige Menge der Größe m gefunden werden kann.

(Wenn-Teil) Zuerst beobachten wir, dass eine unabhängige Menge keine zwei Knoten $[i, j_1]$ und $[i, j_2]$ für $j_1 \neq j_2$ aus der gleichen Klausel enthalten kann. Dies liegt daran, dass solche Knotenpaare durch Kanten miteinander verbunden sind, wie aus den Spalten in Abbildung 10.11 ersichtlich ist. Existiert also eine unabhängige Menge der Größe m, dann muss diese Menge exakt einen Knoten aus jeder Klausel enthalten.

Außerdem kann die unabhängige Menge keine Knoten enthalten, die sowohl zu einer Variablen x als auch zu deren Negation $\overline{x}$ korrespondieren, weil auch solche Knotenpaare durch eine Kante verbunden sind. Finden wir auf diese Weise eine unabhängige Menge I der Größe m, dann definieren wir eine Belegung T für E. Ist ein zu x korrespondierender Knoten in I enthalten, dann soll $T(x) = 1$ sein; ist ein zu einer negierten Variablen $\overline{x}$ korrespondierender Knoten in I enthalten, wählen wir $T(x) = 0$. Enthält I keinen Knoten, der entweder zu x oder zu $\overline{x}$ korrespondiert, dann wählen wir einen beliebigen Wert für $T(x)$. Beachten Sie, dass die obige zweite Konstruktionsidee sicherstellt, dass die Definition von T widerspruchsfrei ist, d. h. $T(x)$ ist entweder 0 oder 1.

Wir behaupten, dass T E erfüllt. Der Grund dafür liegt darin, dass jede Klausel von E einen Knoten in I besitzt, der zu einem ihrer Literale korrespondiert, und T so definiert wurde, dass dieses Literal durch T wahr wird. Wenn also eine unabhängige Menge der Größe m gefunden wurde, ist E erfüllbar.

(Nur-wenn-Teil) Wir nehmen an, E wird von einer Belegung T erfüllt. Da jede Klausel von E durch T wahr wird, können wir ein Literal in jeder Klausel identifizieren, das durch T wahr wird. Bei einigen Klauseln werden wir aus zwei oder drei Literalen auswählen können; in diesem Fall wählen wir aus diesen ein beliebiges. Wir konstruieren eine Menge I aus m Knoten, indem wir aus jeder Spalte den Knoten nehmen, der zu dem gewählten Literal der der Spalte entsprechenden Klausel korrespondiert.

Wir behaupten, dass I eine unabhängige Menge ist. Von Kanten zwischen Knoten, die aus der gleichen Klausel (den Spalten in Abbildung 10.11) stammen, können nicht beide Enden in I enthalten sein, da wir jeweils nur einen Knoten aus einer Klausel auswählen. Von einer Kante, die eine Variable und deren Negation verbindet, können ebenfalls nicht beide Enden in I enthalten sein, da wir für I nur solche Knoten auswählen, die Literalen entsprechen, die durch die Belegung T wahr werden. Natürlich wird durch T entweder x oder $\overline{x}$ wahr, niemals jedoch beide. Wir folgern, dass für G eine unabhängige Menge der Größe m gefunden werden kann, wenn E erfüllbar ist.

Es existiert also eine polynomiale Reduktion von 3SAT auf IS. Da 3SAT als NP-vollständig bekannt ist, ist nach Satz 10.5 auch IS NP-vollständig. ■

Sind Ja/Nein-Probleme einfacher?

Es stellt sich die Frage, ob die Ja/Nein-Version eines Problems einfacher als die Optimierungsversion ist. Beispielsweise könnte es schwierig sein, eine größte unabhängige Menge zu finden, während es bei einem gegebenen kleinen Grenzwert *k* einfach sein könnte, eine unabhängige Menge der Größe *k* zu finden. Dies ist zwar wahr, aber unter Umständen *könnten* wir mit einer Konstante *k* arbeiten, die exakt dem größten Wert entspricht, für den eine unabhängige Menge existiert. In diesem Fall erfordert die Lösung der Ja/Nein-Version, eine maximale unabhängige Menge zu finden.

Tatsächlich sind bei allen allgemeinen NP-vollständigen Problemen die Ja/Nein-Versionen und die Optimierungsversionen in der Komplexität äquivalent, zumindest innerhalb polynomialen Aufwands. Am Beispiel von IS sehen wir: Wenn wir einen polynomialen Algorithmus zur *Suche* maximal unabhängiger Mengen hätten, dann könnten wir das Ja/Nein-Problem lösen, indem wir eine maximal unabhängige Menge finden und sie daraufhin prüfen, ob sie zumindest so groß wie der Grenzwert *k* ist. Da wir gezeigt haben, dass die Ja/Nein-Version NP-vollständig ist, muss die Optimierungsversion ebenfalls nicht behandelbar sein.

Der Vergleich kann ebenso umgekehrt erfolgen. Angenommen, wir haben einen Algorithmus mit polynomialem Zeitaufwand für das Ja/Nein-Problem IS. Besitzt der Graph *n* Knoten, dann liegt die Größe einer maximal unabhängigen Menge zwischen 1 und *n*. Führen wir IS mit allen Grenzwerten zwischen 1 und *n* aus, dann finden wir sicherlich die Größe einer maximal unabhängigen Menge (jedoch nicht unbedingt die Menge selbst) mit einem Zeitaufwand, der dem *n*-fachen der Dauer entspricht, die die einmalige Lösung von IS erfordert. Wenn wir eine Binärsuche verwenden, benötigen wir tatsächlich lediglich das $\log_2(n)$-fache der Ausführungszeit.

Beispiel 10.19 Wir zeigen, wie die Konstruktion aus Satz 10.18 im folgenden Fall funktioniert:

$$E = (x_1 + x_2 + x_3)(\overline{x_1} + x_2 + x_4)(\overline{x_2} + x_3 + x_5)(\overline{x_3} + \overline{x_4} + \overline{x_5})$$

Abbildung 10.11 zeigt den Graphen für diesen Ausdruck. Die Knoten korrespondieren zu den in vier Spalten angeordneten Klauseln. Jeder Knoten ist nicht nur mit seinem Namen (ein Paar aus ganzen Zahlen), sondern auch mit dem Literal bezeichnet, zu dem er korrespondiert. Beachten Sie, dass alle Knotenpaare einer Spalte durch eine Kante verbunden sind. Auch die Knoten, die mit einer Variablen und ihrem Komplement korrespondieren, sind durch eine Kante verbunden. Beispielsweise besitzt der zu $\overline{x}_2$ korrespondierende Knoten [3, 1] Kanten zu den beiden Knoten [1, 2] und [2, 2], die jeweils mit einem Auftreten von x_2 korrespondieren.

Wir haben eine Menge I aus vier Knoten (fett umrandet) ausgewählt, einen aus jeder Spalte. Diese Knoten bilden offensichtlich eine unabhängige Menge. Da deren Literale x_1, x_2, x_3 und $\overline{x}_4$ lauten, können wir daraus eine Belegung T konstruieren, die aus $T(x_1) = 1$, $T(x_2) = 1$, $T(x_3) = 1$ und $T(x_4) = 0$ besteht. Auch für x_5 ist eine Belegung nötig, doch diese kann beliebig gewählt werden, etwa $T(x_5) = 0$. Nun wird E durch T erfüllt, und die Knotenmenge I indiziert aus jeder Klausel ein Literal, das durch T wahr wird. ■

Wozu dienen unabhängige Mengen?

In diesem Buch sollen keine Anwendungen der als NP-vollständig bewiesenen Probleme behandelt werden. Jedoch stammt die Auswahl der Probleme in Abschnitt 10.4 aus einer grundlegenden Veröffentlichung über NP-Vollständigkeit von R. Karp, in der er die wichtigsten Probleme aus dem Bereich des Operations Research untersucht und viele davon als NP-vollständig nachgewiesen hat. Es ist also evident, dass es viele »reale« Probleme gibt, deren allgemeine Lösung ein nicht handhabbares Problem darstellt.

Zum Beispiel könnten wir einen guten Algorithmus gebrauchen, der umfangreiche unabhängige Mengen für die Planung von Abschlussprüfungen findet. Die Knoten des Graphen seien die Fächer. Eine Kante verbindet zwei Fächer, wenn ein oder mehrere Studenten beide Fächer belegt haben und daher die Prüfungen in diesen Fächern nicht zum gleichen Zeitpunkt anberaumt werden können. Wenn wir eine maximal unabhängige Menge finden, dann können wir die Prüfungen in den entsprechenden Fächern für den gleichen Zeitpunkt festlegen und sind sicher, damit Konflikte bei den Studenten zu vermeiden.

10.4.3 Das Problem der Knotenüberdeckung

Eine weitere wichtige Klasse der Probleme kombinatorischer Optimierung betrifft das »Überdecken« eines Graphen. Beispielsweise besteht eine *Kantenüberdeckung* aus einer Menge von Kanten, sodass sich jeder Knoten im Graphen am Ende mindestens einer Kante aus dieser Menge befindet. Eine Kantenüberdeckung ist *minimal*, wenn sie die wenigsten Kanten aller Kantenüberdeckungen des gegebenen Graphen enthält. Das Entscheidungsproblem, ob man für einen Graphen eine Kantenüberdeckung mit k Kanten finden kann, ist NP-vollständig, doch wir werden dies hier nicht beweisen.

Wir werden jedoch die NP-Vollständigkeit des Problems der Knotenüberdeckung beweisen. Eine *Knotenüberdeckung* (engl. *node covering*) eines Graphen ist eine Menge von Knoten, sodass jede Kante mindestens mit einem Ende an einem Knoten aus der Menge endet. Eine Knotenüberdeckung ist *minimal*, wenn sie die wenigsten Knoten aller Knotenüberdeckungen des gegebenen Graphen enthält.

Knotenüberdeckungen und unabhängige Mengen stehen in einem engen Zusammenhang. Tatsächlich ist eine Knotenüberdeckung das Komplement einer unabhängigen Menge und umgekehrt. Wenn wir daher die Ja/Nein-Version des Problems der Knotenüberdeckung (Node-Cover Problem, NC) entsprechend ausdrücken, ist die Reduktion von IS sehr einfach.

Problem: Knotenüberdeckung (NC).

Eingabe: Ein Graph G sowie ein oberer Grenzwert k, der zwischen 1 und der Anzahl der Knoten von G minus 1 liegen muss.

Ausgabe: »Ja«, genau dann, wenn G eine Knotenüberdeckung von k oder weniger Knoten besitzt.

Reduktion von: Unabhängige Menge IS.

Satz 10.20 Das Problem der Knotenüberdeckung NC ist NP-vollständig.

Beweis: Offensichtlich ist NC in $\mathcal{NP}$ enthalten. Wir raten eine Menge von k Knoten und prüfen, ob von jeder Kante von G mindestens ein Ende in der Menge enthalten ist.

Um den Beweis abzuschließen, werden wir IS auf NC reduzieren. Abbildung 10.11 zeigt den Gedankengang: Das Komplement einer unabhängigen Menge ist eine Knotenüberdeckung. Beispielsweise bilden die nicht fett umrandeten Knoten in Abbildung 10.11 eine Knotenüberdeckung. Da es sich bei den fett umrandeten Knoten tatsächlich um eine maximal unabhängige Menge handelt, bilden die anderen Knoten eine minimale Knotenüberdeckung.

Nun geben wir die Reduktion an. Sei G mit unterem Grenzwert k eine Instanz des Problems unabhängiger Mengen. Besitzt G n Knoten, dann sei G mit dem oberen Grenzwert $n - k$ die Instanz des Problems der Knotenüberdeckung, die wir konstruieren. Offensichtlich kann diese Transformation mit einem linearen Zeitaufwand durchgeführt werden. Wir behaupten:

- G besitzt genau dann eine unabhängige Menge der Größe k, wenn G eine Knotenüberdeckung der Größe $n - k$ besitzt.

(Wenn-Teil) Sei N die Menge der Knoten von G und C die Knotenüberdeckung der Größe $n - k$. Wir behaupten, dass $N - C$ eine unabhängige Menge ist. Angenommen, dies sei nicht der Fall; dann existiert ein Paar von Knoten v und w in $N - C$, die in G durch eine Kante verbunden sind. Da weder v noch w in C enthalten sind, hat die Kante (v, w) kein Ende in der Knotenüberdeckung C, und wir haben durch einen Widerspruch bewiesen, dass $N - C$ eine unabhängige Menge ist. Offensichtlich besitzt diese Menge k Knoten, und damit ist diese Richtung des Beweises abgeschlossen.

(Nur-wenn-Teil) Angenommen, I sei eine unabhängige Menge von k Knoten. Wir behaupten, dass $N - I$ eine Knotenüberdeckung mit $n - k$ Knoten ist. Wiederum führen wir einen Widerspruch herbei. Gäbe es eine Kante (v, w), sodass weder v noch w in $N - I$ läge, dann wäre sowohl v als auch w in I enthalten; da sie aber durch eine Kante verbunden sind, steht dies im Widerspruch zur Definition einer unabhängigen Menge. ■

10.4.4 Das Problem des gerichteten Hamiltonschen Kreises

Wir möchten das Problem des Handlungsreisenden (*Traveling Salesman Problem*, TSP) als NP-vollständig nachweisen, da dieses Problem in der Kombinatorik von großem Interesse ist. Der bekannteste Beweis der NP-Vollständigkeit dieses Problems besteht im Wesentlichen in dem Beweis, dass ein einfacheres Problem, das *Problem des Hamiltonschen Kreises* (Hamilton-Circuit Problem, HC) NP-vollständig ist. Dieses Problem kann wie folgt beschrieben werden:

Problem: Hamiltonscher Kreis (HC).

Eingabe: Ein ungerichteter Graph G.

Ausgabe: »Ja«, genau dann, wenn G einen *Hamiltonschen Kreis* besitzt, also einen Zyklus, der jeden Knoten von G exakt einmal passiert.

Beachten Sie, dass das HC-Problem ein Spezialfall des TSP ist, in dem alle Kanten die Gewichtung 1 besitzen. Eine Reduktion mit polynomialem Aufwand von HC auf TSP ist sehr einfach: Wir geben jeder Kante im Graphen das Gewicht 1.

Der Beweis der NP-Vollständigkeit von HC ist sehr schwierig. Unser Ansatz verwendet eine eingeschränkte Version von HC, in der die Kanten mit einer Richtung versehen sind (gerichtete Kanten oder Pfeile) und der Hamiltonsche Kreis den Pfeilen entsprechend ihrer Richtung folgen muss. Wir reduzieren 3SAT auf diese gerichtete Version des HC-Problems, das wir dann auf die ungerichtete Standardversion von HC reduzieren. Formell lautet das Problem wie folgt:

Problem: Gerichteter Hamiltonscher Kreis (Directed Hamilton-Circuit Problem, DHC).

Eingabe: Ein gerichteter Graph G.

Ausgabe: »Ja«, genau dann, wenn G einen gerichteten Zyklus enthält, der jeden Knoten exakt einmal passiert.

Reduktion von: 3SAT.

Satz 10.21 Das Problem des gerichteten Hamiltonschen Kreises (DHC) ist NP-vollständig.

Beweis: Der Beweis, dass DHC in $\mathcal{NP}$ enthalten ist, ist sehr einfach. Wir raten einen Zyklus und prüfen, ob alle erforderlichen Pfeile im Graphen vorhanden sind. Wir müssen 3SAT auf DHC reduzieren, und diese Reduktion erfordert die Konstruktion eines komplizierten Graphen mit spezialisierten Teilgraphen, die jede Variable und jede Klausel der 3SAT-Instanz repräsentieren.

Wir beginnen mit der Konstruktion einer DHC-Instanz aus einem Booleschen Ausdruck in 3-KNF. Sei $E = e_1 \wedge e_2 \wedge \ldots \wedge e_k$ der Ausdruck, wobei jedes e_i eine Klausel aus der Summe von drei Literalen $e_i = (\alpha_{i1} + \alpha_{i2} + \alpha_{i3})$ ist. $x_1, x_2, \ldots, x_n$ seien die Variablen von E. Wie in Abbildung 10.7 dargestellt, konstruieren wir für jede Klausel und jede Variable einen spezialisierten Teilgraphen.

Wir konstruieren für jede Variable x_i einen Teilgraphen H_i mit der in Abbildung 10.7 (a) gezeigten Struktur. m_i ist die größere der Anzahlen des Auftretens von x_i bzw. von $\overline{x_i}$ in E. In den beiden Spalten mit den b- und c-Knoten werden b_{ij} und c_{ij} durch Pfeile in beiden Richtungen verbunden. Außerdem existiert von jedem b ein Pfeil zum darunter stehenden c; b_{ij} besitzt also einen Pfeil nach $c_{i,\,j+1}$, solange $j < m_i$ ist. Entsprechend besitzt c_{ij} einen Pfeil nach $b_{i,\,j+1}$ für $j < m_i$. Der Kopfknoten a_i besitzt Pfeile nach b_{i0} sowie c_{i0} und der Fußknoten d_i Pfeile von b_{im_i} sowie von c_{im_i}.

Abbildung 10.7 (b) beschreibt die Struktur des gesamten Graphen. Jedes Sechseck repräsentiert für eine Variable einen der spezialisierten Teilgraphen mit der in Abbildung 10.7 (a) gezeigten Struktur. Der Fußknoten eines spezialisierten Teilgraphen hat einen Pfeil zum Kopfknoten des nächsten spezialisierten Teilgraphen im Zyklus der H_i.

Wir nehmen an, es existiere ein gerichteter Hamiltonscher Kreis für den in Abbildung 10.7 (b) gezeigten Graphen. Wir können auch annehmen, dieser Zyklus beginne in a_1. Wenn er nach b_{10} führt, dann behaupten wir, dass er danach nach c_{10} führen muss, denn sonst würde c_{10} nicht im Zyklus erscheinen. Zum Beweis achten Sie auf Folgendes: Führt der Zyklus von a_1 nach b_{10} und dann nach c_{11}, dann kann c_{10} nicht im Zyklus enthalten sein, da sich beide Vorgänger von c_{10} (also a_1 und b_{10}) schon im Zyklus befinden.

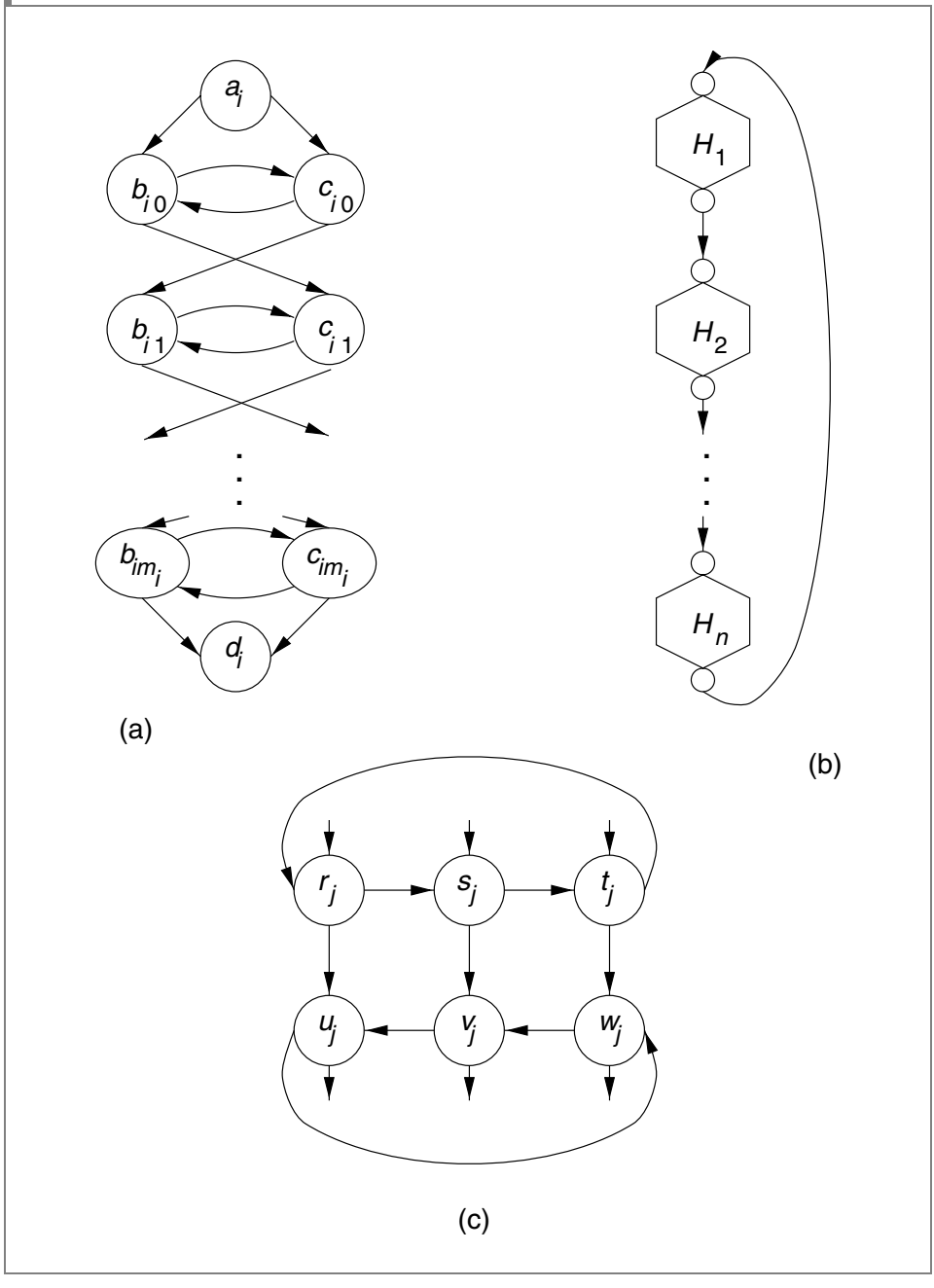
Beginnt der Zyklus daher mit a_1 und b_{10}, dann muss er zwischen den Seiten wechselnd nach unten weitergeführt werden:

$$a_1, b_{10}, c_{10}, b_{11}, c_{11}, \ldots, b_{1m_1}, c_{1m_1}, d_1$$

Beginnt der Pfad jedoch mit a_1 und c_{10}, dann steht das c dem b auf der gleichen Ebene jeweils voran:

$$a_1, c_{10}, b_{10}, c_{11}, b_{11}, \ldots, c_{1m_1}, b_{1m_1}, d_1$$

Abbildung 10.7: Konstruktion für den Beweis, dass das Problem des Hamiltonschen Kreises NP-vollständig ist



Ein entscheidender Punkt des Beweises ist, dass wir die erste Reihenfolge, in der von den c zu den darunter liegenden b gewechselt wird, so behandeln können, als ob die dem spezialisierten Teilgraphen entsprechende Variable wahr ist, während die andere

Reihenfolge, in der von den b zu darunter liegenden c gewechselt wird, dazu korrespondiert, dass diese Variable falsch ist.

Nach Durchqueren des spezialisierten Teilgraphen H_1 muss der Zyklus nach a_2 weitergehen und zwischen b_{20} und c_{20} als nächstem Schritt wählen. Wie bei H_1 angemerkt, ist der Zyklus durch H_2 allerdings festgelegt, sobald wir uns von a_2 aus für die Richtung links oder rechts entscheiden. Im Allgemeinen muss bei jedem H_i gewählt werden, ob der Zyklus nach links oder rechts fortgesetzt wird. Danach ist keine weitere Wahl möglich, da ein Knoten sonst *unzugänglich* wird (der Knoten kann somit auch nicht mehr in einem gerichteten Hamiltonschen Kreis liegen, da alle seine Vorgänger schon passiert wurden).

Im Folgenden ist es vorteilhaft, sich die Wegewahl von a_i nach b_{i0} vorzustellen, als sei die Variable x_i wahr, und die Wahl von a_i nach c_{i0}, als sei diese Variable falsch. Der in Abbildung 10.7 (b) dargestellte Graph besitzt also exakt 2^n gerichtete Hamiltonsche Kreise, die den 2^n Belegungen von n Variablen entsprechen.

Allerdings zeigt Abbildung 10.7 (b) nur das Grundgerüst des Graphen, den wir für den 3-KNF-Ausdruck E generieren. Wir führen für jede Klausel e_j einen weiteren Teilgraphen I_j ein, den Abbildung 10.7 (c) zeigt. Der spezialisierte Teilgraph I_j besitzt die Eigenschaft, dass ein Zyklus den Graphen bei u_j verlassen muss, wenn er ihn bei r_j betritt; betritt er ihn bei s_j, dann muss er ihn bei v_j verlassen, und das Gleiche gilt für t_j und w_j. Dies liegt daran, dass der Zyklus nach Erreichen von I_j den Graphen bei dem Knoten direkt unterhalb des Eintrittsknotens verlassen muss, da sonst ein oder auch mehrere Knoten nicht mehr zugänglich sind und damit nicht im Zyklus auftreten können. Wegen der Symmetrie genügt es, lediglich den Fall zu betrachten, dass r_j der erste Knoten von I_j im Zyklus ist. Es gibt drei Fälle:

1. Die nächsten beiden Knoten im Zyklus sind s_j und t_j. Führt der Zyklus nach w_j und verlässt I_j, dann ist v_j nicht zugänglich. Führt der Zyklus nach w_j sowie v_j und verlässt I_j, dann ist u_j unzugänglich. Daher muss der Zyklus den Graphen bei u_j verlassen, nachdem alle Knoten von I_j passiert wurden.

2. Die nächsten beiden Knoten nach r_j sind s_j und v_j. Führt der Zyklus als Nächstes nicht nach u_j, dann wird u_j unzugänglich. Wenn der Zyklus nach u_j nach w_j führt, kann t_j niemals im Zyklus auftreten. Der Beweis entspricht der »Umkehrung« des Unzugänglichkeitsbeweises. Nun kann t_j zwar von außen erreicht werden, doch wenn der Zyklus t_j später passiert, gibt es keinen nächsten Knoten, da beide Nachfolger von t_j schon im Zyklus enthalten sind. Auch in diesem Fall muss also der Zyklus den Graphen bei u_j verlassen. Beachten Sie allerdings, dass t_j und w_j nicht passiert wurden; diese beiden Knoten müssen also später im Zyklus auftreten, was möglich ist.

3. Der Zyklus führt von r_j direkt nach u_j. Führt der Zyklus ausschließlich nach w_j, dann kann t_j nicht darin erscheinen, da die Nachfolger schon vorher aufgetreten sind, wie in Fall (2) beschrieben. In diesem Fall muss der Zyklus den Graphen also bei u_j verlassen, und die vier anderen Knoten sind später in den Zyklus aufzunehmen.

Um die Konstruktion des Graphen G für den Ausdruck E abzuschließen, verbinden wir die I_j wie folgt mit den H_i. Angenommen, das erste Literal in der Klausel e_j sei x_i, eine nicht negierte Variable. Wir wählen einen Knoten c_{ip}, wobei p im Bereich von 0 bis $m_i - 1$ liegt, der bisher noch nicht genutzt wurde, um eine Verbindung zu einem der spezialisierten Teilgraphen I herzustellen. Wir fügen Pfeile von c_{ip} nach r_j und von u_j

zu $b_{i,\,p+1}$ hinzu. Ist das erste Literal der Klausel e_j die negierte Variable $\overline{x}_i$, dann wählen wir ein noch nicht benutztes b_{ip}. Wir verbinden b_{ip} mit r_j und u_j mit $c_{i,p+1}$.

Für das zweite und dritte Literal von e_j fügen wir mit einer Ausnahme das Gleiche in den Graphen hinzu. Beim zweiten Literal verwenden wir die Knoten s_j und v_j, beim Dritten dagegen t_j und w_j. Jedes I_j besitzt auf diese Weise zu den spezialisierten Teilgraphen H drei Verbindungen, die die Variablen der Klausel e_j repräsentieren. Die Verbindung kommt von einem c-Knoten und führt zum darunter liegenden b-Knoten zurück, wenn das Literal nicht negiert ist. Sie kommt von einem b-Knoten und führt zum darunter liegenden c-Knoten zurück, wenn das Literal negiert ist. Wir behaupten nun Folgendes:

- Für den so konstruierten Graphen G kann genau dann ein gerichteter Hamiltonscher Kreis gefunden werden, wenn der Ausdruck E erfüllbar ist.

(Wenn-Teil) Angenommen, es existiere eine erfüllende Belegung T für E. Wir konstruieren einen gerichteten Hamiltonschen Kreis wie folgt:

1. Wir beginnen mit dem Zyklus, der gemäß der Belegung T zunächst nur durch die H_i führt (also durch den in Abbildung 10.7 (b) dargestellten Graphen). Der Zyklus führt von a_i nach b_{i0}, falls $T(x_i) = 1$, und von a_i nach c_{i0}, falls $T(x_i) = 0$.

2. Wenn der bisher konstruierte Zyklus einem Pfeil von b_{ip} nach $c_{i,p+1}$ folgt und b_{ip} zu einem der I_j einen weiteren Pfeil besitzt, der bisher nicht in den Zyklus aufgenommen wurde, führen wir eine »Umleitung« in den Zyklus ein, die alle sechs Knoten von I_j in den Zyklus einfügt und zu $c_{i,p+1}$ zurückkehrt. Zwar wird der Pfeil $b_{ip} \rightarrow c_{i,p+1}$ nicht mehr zum Zyklus gehören, doch die Knoten b_{ip} und $c_{i,p+1}$ bleiben darin enthalten.

3. Besitzt der Zyklus einen Pfeil von c_{ip} nach $b_{i,p+1}$ und c_{ip} einen weiteren Pfeil, der zu einem I_j führt und bisher nicht im Zyklus ist, dann ändern wir den Zyklus und nehmen eine »Umleitung« durch alle sechs Knoten von I_j darin auf.

Die Tatsache, dass T E erfüllt, stellt sicher, dass der in Schritt (1) konstruierte Originalzyklus mindestens einen Pfeil enthält, der uns erlaubt, in Schritt (2) oder (3) den spezialisierten Teilgraphen I_j für jede Klausel e_j in den Zyklus aufzunehmen. Daher werden alle I_j in den Zyklus aufgenommen, der damit zu einem gerichteten Hamiltonschen Kreis wird.

(Nur-wenn-Teil) Nun nehmen wir an, der Graph G besitze einen gerichteten Hamiltonschen Kreis. Wir müssen zeigen, dass E erfüllbar ist. Zuerst erinnern wir uns an zwei wichtige Punkte aus der bisherigen Analyse:

1. Tritt ein Hamiltonscher Kreis bei einem r_j, s_j oder t_j in ein I_j ein, dann muss er bei einem u_j, v_j bzw. w_j wieder austreten.

2. Sehen wir den Hamiltonschen Kreis so an, als führe er durch den Zyklus der H_i (siehe Abbildung 10.7 (b)), dann können die Umleitungen des Zyklus zu einem I_j angesehen werden, als folge der Zyklus einem Pfeil, der »parallel« zu einem der Pfeile $b_{ip} \rightarrow c_{i,p+1}$ oder $c_{ip} \rightarrow b_{i,p+1}$ verläuft.

Ignorieren wir die Umleitungen zu den I_j, dann muss der Hamiltonsche Kreis einer der 2^n Zyklen sein, die unter ausschließlicher Verwendung der H_i möglich sind – diejenigen, die von a_i entweder nach b_{i0} oder nach c_{i0} führen. Jede dieser Wahlmöglichkei-

ten entspricht einer Belegung für E. Liefert eine dieser Wahlmöglichkeiten einen Hamiltonschen Kreis, der die I_j umfasst, dann muss diese Belegung E erfüllen.

Das lässt sich folgendermaßen begründen: Wenn der Zyklus von a_i nach b_{i0} führt, dann können wir der Umleitung durch I_j nur folgen, wenn die j-te Klausel x_i als eines ihrer Literale enthält. Wenn der Zyklus von a_i nach c_{i0} führt, dann können wir der Umleitung durch I_j nur folgen, wenn die j-te Klausel $\overline{x}_i$ als eines ihrer Literale enthält. Die Tatsache, dass alle Teilgraphen I_j in den Zyklus eingeschlossen werden können, impliziert also, dass durch die Belegung mindestens ein Literal in jeder Klausel wahr ist; E ist also erfüllbar. ■

Beispiel 10.22 Wir zeigen ein sehr einfaches Beispiel für die Konstruktion aus Satz 10.21, die auf dem 3-KNF-Ausdruck $E = (x_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_2 + x_3)$ basiert. Abbildung 10.8 zeigt den konstruierten Graphen. Die Pfeile, die spezialisierte Teilgraphen des Typs H mit solchen des Typs I verbinden, sind zur Verbesserung der Lesbarkeit gepunktet dargestellt, doch sonst unterscheiden sich die Pfeile nicht.

Beispielsweise sehen wir oben links den spezialisierten Teilgraphen für x_1. Da x_1 sowohl negiert als auch nicht negiert auftritt, benötigt die »Leiter« nur einen einzigen Schritt, und es gibt daher zwei Reihen mit b_{1p} und c_{1p}. Unten links sehen wir den spezialisierten Teilgraphen für x_3, das zweimal nicht negiert und niemals negiert auftritt. Wir benötigen also zwei verschiedene Pfeile $c_{3p} \to b_{3,p+1}$, die wir verwenden, um die spezialisierten Teilgraphen für I_1 und I_2 anzuhängen und so das Vorkommen von x_3 in diesen Klauseln zu repräsentieren. Aus diesem Grund benötigt der spezialisierte Teilgraph von x_3 drei b-c-Reihen.

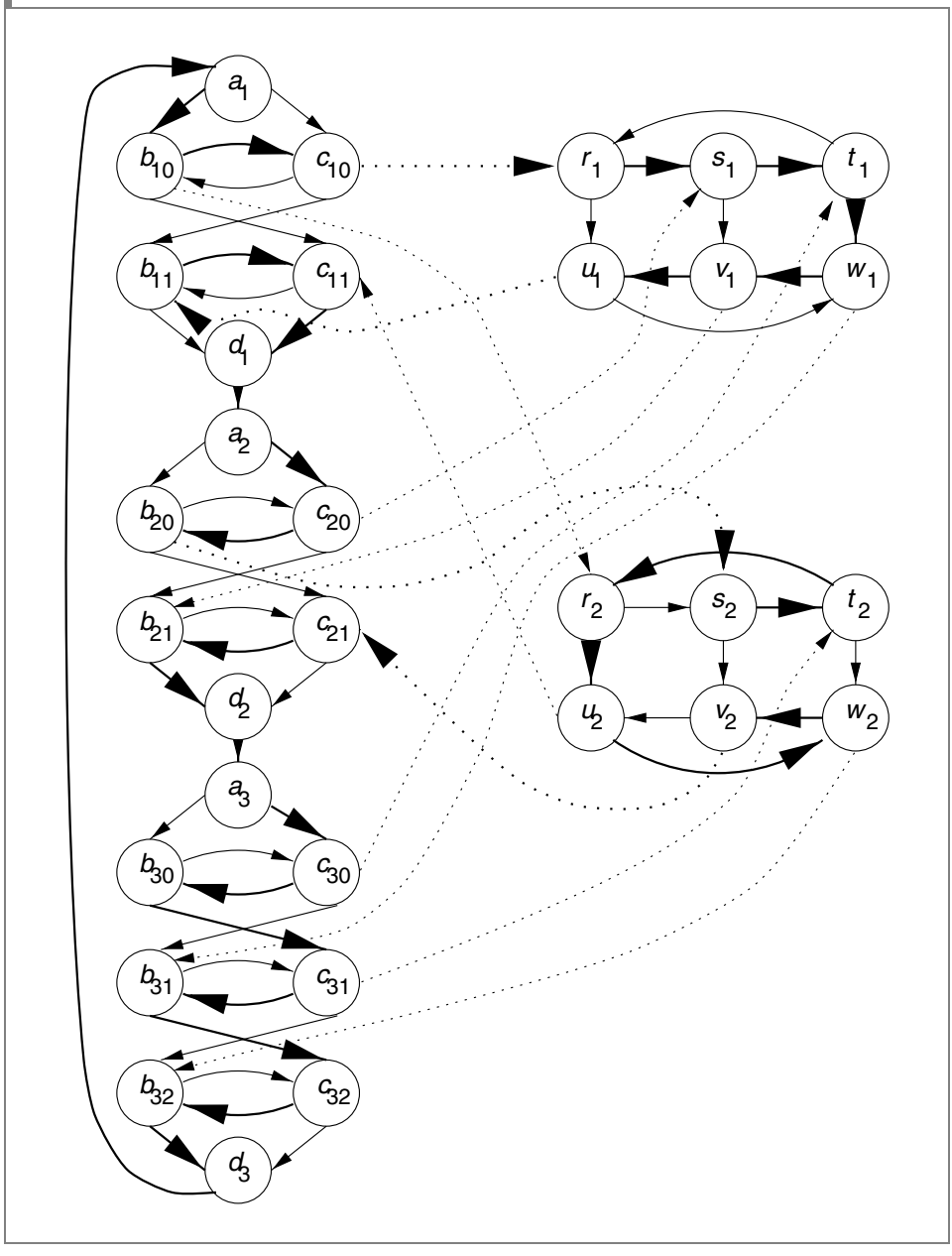
Wir sehen uns den spezialisierten Teilgraphen I_2 an, der mit der Klausel $(\overline{x}_1 + \overline{x}_2 + x_3)$ korrespondiert. Für das erste Literal $\overline{x}_1$ wird b_{10} mit r_2 sowie u_2 mit c_{11} verbunden. Für das zweite Literal führen wir das Gleiche mit b_{20}, s_2, v_2 und c_{21} aus. Für das nicht negierte dritte Literal wird c_{31} mit t_2 sowie w_2 mit b_{32} verbunden.

Eine der erfüllenden Belegungen lautet $x_1 = 1$, $x_2 = 0$ und $x_3 = 0$. Bei dieser Belegung wird die erste Klausel durch ihr erstes Literal x_1 erfüllt, die zweite Klausel dagegen durch ihr zweites Literal $\overline{x}_2$. Auf Basis dieser Belegung können wir einen Hamiltonschen Kreis finden, der die Pfeile $a_1 \to b_{10}$, $a_2 \to c_{20}$ und $a_3 \to c_{30}$ enthält. Der Zyklus deckt die erste Klausel durch eine Umleitung von H_1 nach I_1 ab; es wird also der Pfeil $c_{10} \to r_1$ verwendet, alle Knoten von I_1 werden passiert und dann wird nach b_{11} zurückgekehrt. Die zweite Klausel wird durch eine Umleitung von H_2 nach I_2 abgedeckt, wobei mit dem Pfeil $b_{20} \to s_2$ begonnen wird, alle Knoten von I_2 passiert werden und nach c_{21} zurückgekehrt wird. Der gesamte Hamiltonsche Kreis in Abbildung 10.8 ist durch dickere Linien (durchgezogen oder gepunktet) und größere Pfeile hervorgehoben. ■

10.4.5 Ungerichtete Hamiltonsche Kreise und das Problem des Handlungsreisenden

Die Beweise der NP-Vollständigkeit des Problems des ungerichteten Hamiltonschen Kreises (HC) sowie des Problems des Handlungsreisenden (TSP) sind relativ einfach. Wir haben schon in Abschnitt 10.1.4 gesehen, dass das TSP in $\mathcal{NP}$ enthalten ist. HC ist ein Spezialfall des TSP, und somit ist es ebenfalls in $\mathcal{NP}$ enthalten. Wir müssen DHC auf HC und HC auf TSP reduzieren.

Abbildung 10.8: Beispiel für die Konstruktion des Hamiltonschen Kreises



Problem: Ungerichteter Hamiltonscher Kreis (HC).

Eingabe: Ein ungerichteter Graph G.

Ausgabe: »Ja«, genau dann, wenn G einen Hamiltonschen Kreis enthält.

Reduktion von: Gerichteter Hamiltonscher Kreis (DHC).

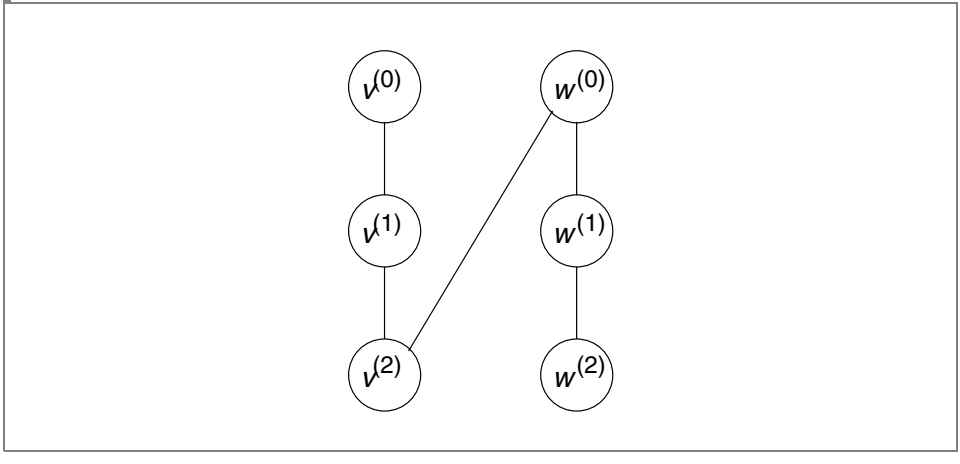
Satz 10.23 HC ist NP-vollständig.

Beweis: Wir reduzieren DHC wie folgt auf HC. Angenommen, es sei ein gerichteter Graph G_d gegeben. Wir werden den ungerichteten Graphen G_u konstruieren. Für jeden Knoten v von G_d hat G_u drei Knoten: $v^{(0)}$, $v^{(1)}$ und $v^{(2)}$. G_u besitzt folgende Kanten:

1. Für alle Knoten v von G_d hat G_u die Kanten $(v^{(1)}, v^{(2)})$ und $(v^{(0)}, v^{(1)})$.
2. Existiert ein Pfeil $v \rightarrow w$ in G_d, dann gibt es eine Kante $(v^{(2)}, w^{(0)})$ in G_u.

Abbildung 10.9 zeigt das Muster der Kanten einschließlich der Kante für einen Pfeil $v \rightarrow w$.

Abbildung 10.9: Die Pfeile in G_d werden in G_u durch Kanten ersetzt, die von Rang 2 zu Rang 0 verlaufen



Es ist leicht ersichtlich, dass G_u mit einem polynomialen Zeitaufwand aus G_d konstruiert werden kann. Wir müssen Folgendes zeigen:

- G_u enthält genau dann einen Hamiltonschen Kreis, wenn G_d einen gerichteten Hamiltonschen Kreis enthält.

(Wenn-Teil) Angenommen, $v_1, v_2, \ldots, v_n, v_1$ sei ein gerichteter Hamiltonscher Kreis. Dann ist

$$v_1^{(0)}, v_1^{(1)}, v_1^{(2)}, v_2^{(0)}, v_2^{(1)}, v_2^{(2)}, v_3^{(0)}, \ldots, v_n^{(0)}, v_n^{(1)}, v_n^{(2)}, v_1^{(0)}$$

mit Sicherheit ein ungerichteter Hamiltonscher Kreis in G_u. Wir werden also in jeder Spalte nach unten gehen und dann zur Spitze der nächsten Spalte springen, um einem Pfeil von G_d zu folgen.

(Nur-wenn-Teil) Achten Sie darauf, dass jeder Knoten $v^{(1)}$ von G_u nur zwei Kanten besitzt und daher $v^{(0)}$ oder $v^{(2)}$ als unmittelbarer Vorgänger sowie der andere der beiden als unmittelbarer Nachfolger in einem Hamiltonschen Kreis von G_u erscheinen muss. Die oberen Indizes der Knoten eines Hamiltonschen Kreises in G_u müssen daher in einer Reihenfolge auftreten, die dem Muster 0, 1, 2, 0, 1, 2, … oder spiegelbildlich 2, 1, 0, 2, 1, 0, … entspricht. Da diese Muster dem Durchlauf eines Zyklus in den zwei verschiedenen Richtungen entsprechen, können wir o.B.d.A. das Muster 0, 1, 2, 0, 1, 2, … wählen. Wenn wir uns die Kanten des Zuklus ansehen, die von einem Knoten mit oberem Index 2 zu einem Knoten mit oberem Index 0 führen, dann wissen wir, dass solche Kanten Pfeile in G_d in dieser Richtung sind. Ein ungerichteter Hamiltonscher Kreis in G_u führt also zu einem gerichteten Hamiltonschen Kreis in G_d. ■

Problem: Problem des Handlungsreisenden (TSP).

Eingabe: Ein ungerichteter Graph G mit ganzzahliger Gewichtung der Kanten und einer Grenze k.

Ausgabe: »Ja«, genau dann, wenn G einen Hamiltonschen Kreis enthält, sodass die Summe der Kantengewichte im Zyklus kleiner oder gleich k ist.

Reduktion von: Hamiltonscher Kreis (HC).

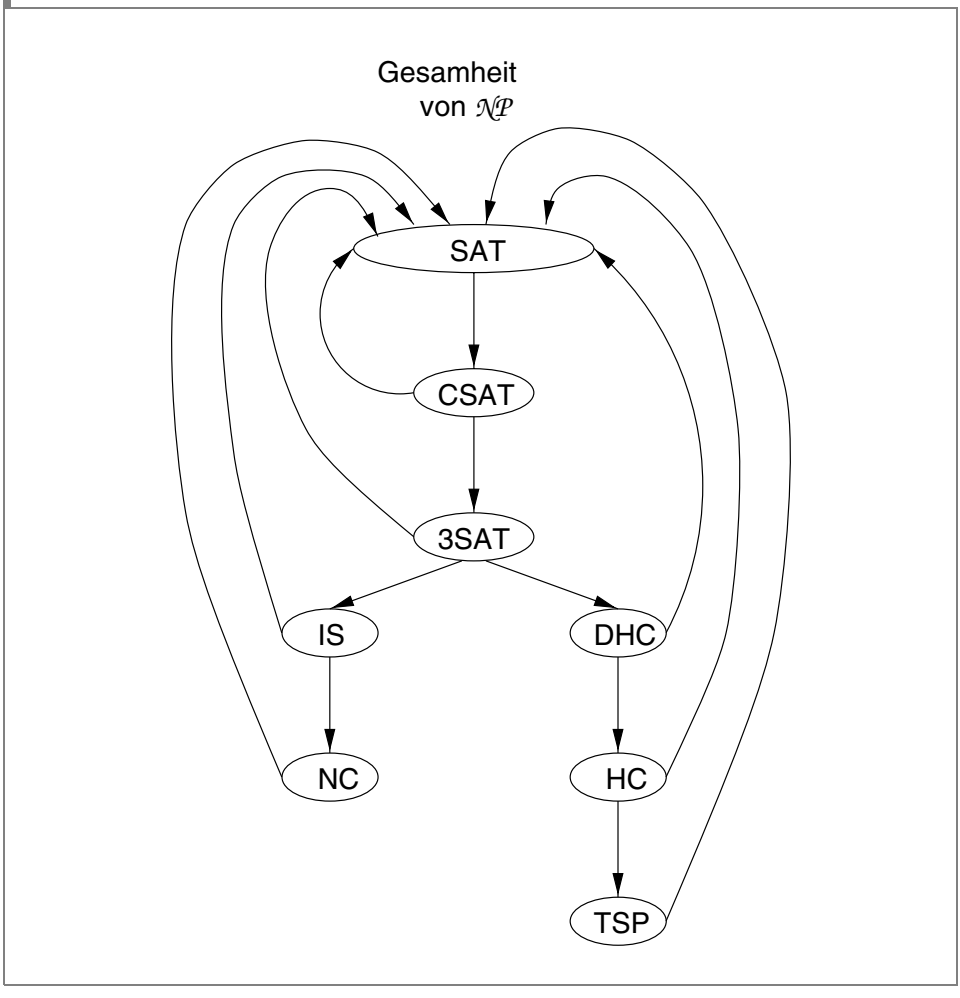
Satz 10.24 Das Problem des Handlungsreisenden (TSP) ist NP-vollständig.

Beweis: Wir führen die Reduktion von HC wie folgt aus. Gegeben sei ein Graph G. Wir konstruieren einen gewichteten Graphen G', dessen Knoten und Kanten mit denen von G übereinstimmen, wobei jede Kante ein Gewicht 1 besitzt. Gegeben sei außerdem ein Grenzwert k, der mit der Anzahl der Knoten n in G übereinstimmt. Dann existiert ein Hamiltonscher Kreis mit Gewichtsumme n in G' nur, wenn G einen Hamiltonschen Kreis enthält. ■

10.4.6 Zusammenfassung der NP-vollständigen Probleme

Abbildung 10.18 zeigt alle Reduktionen aus diesem Kapitel. Beachten Sie, dass wir Reduktionen von allen spezifischen Problemen wie etwa TSP auf SAT eingetragen haben. Gezeigt haben wir in Satz 10.9 die polynomiale Reduktion der Sprachen aller nichtdeterministischen Turing-Maschinen auf SAT. Es wurde zwar nicht explizit erwähnt, doch zu diesen Turing-Maschinen gehört zumindest eine TM zur Lösung von TSP, eine zur Lösung von IS und so weiter. Daher sind alle NP-vollständigen Probleme mit polynomialem Zeitaufwand aufeinander reduzierbar und stellen tatsächlich nur unterschiedliche Ausprägungen desselben Problems dar.

Abbildung 10.10: Reduktionen zwischen NP-vollständigen Problemen



10.4.7 Übungen zum Abschnitt 10.4

* **Übung 10.4.1** In einem Graphen G ist eine *k-Clique* eine Menge von k Knoten von G, sodass zwischen je zwei Knoten der Clique eine Kante existiert. Eine 2-Clique ist daher ein durch eine Kante verbundenes Knotenpaar, und eine 3-Clique ist ein Dreieck. Das Problem CLIQUE lautet wie folgt: Gegeben seien ein Graph G und eine Konstante k. Besitzt G eine k-Clique?

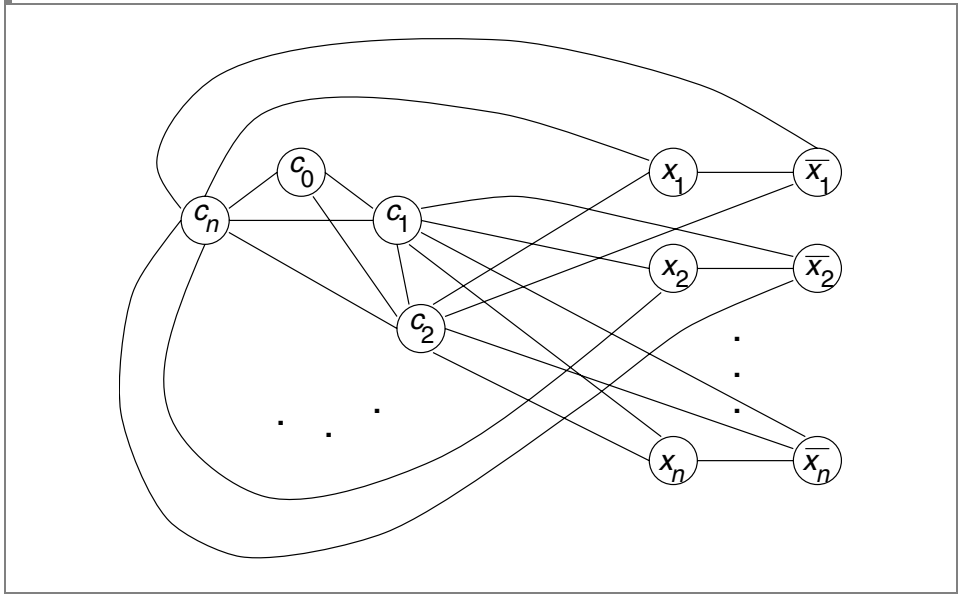
a) Welches ist das größte k, für das der Graph aus Abbildung 10.1 CLIQUE erfüllt?

b) Wie viele Kanten hat eine k-Clique als Funktion von k?

c) Beweisen Sie, dass CLIQUE NP-vollständig ist, indem Sie das Problem der Knotenüberdeckung auf CLIQUE reduzieren.

***! Übung 10.4.2** Das *Färbungsproblem* lautet folgendermaßen: Gegeben seien ein Graph G und eine ganze Zahl k. Ist G »k-färbbar«? Können wir also jedem Knoten von G eine von k Farben zuweisen, sodass die Enden jeder Kante verschieden gefärbt sind? Beispielsweise ist der Graph in Abbildung 10.11 3-färbbar, da wir den Knoten 1 und 4 die Farbe Rot, dem Knoten 2 Grün und dem Knoten 3 Blau zuweisen können. Im Allgemeinen kann ein Graph, der eine k-Clique besitzt, nicht weniger als k-färbbar sein, doch unter Umständen sind wesentlich mehr als k Farben notwendig.

Abbildung 10.11: Ein Teil der Konstruktion, die das Färbungsproblem als NP-vollständig zeigt



In dieser Übung werden wir einen Teil einer Konstruktion beschreiben, die zum Beweis dient, dass das Färbungsproblem NP-vollständig ist; Sie sollen die Konstruktion vervollständigen. Die Reduktion erfolgt von 3SAT. Angenommen, wir haben einen 3-KNF-Ausdruck mit n Variablen. Die Reduktion konvertiert diesen Ausdruck in einen Graphen, der in Abbildung 10.11 teilweise abgebildet ist. Wie links zu sehen ist, gibt es $n + 1$ Knoten c_0, c_1, …, c_n, die eine $(n + 1)$-Clique bilden. Daher muss jeder dieser Knoten in einer anderen Farbe eingefärbt werden. Wir denken uns die c_j zugewiesene Farbe als »die Farbe c_j«.

Außerdem existieren für jede Variable x_i zwei Knoten, die wir uns als x_i und $\overline{x}_i$ vorstellen. Sie sind durch eine Kante verbunden und müssen daher unterschiedlich eingefärbt werden. Zudem ist jeder der Knoten für x_i für alle j ungleich 0 und i mit c_j verbunden. Daraus resultiert, dass einer der beiden Knoten x_i und $\overline{x}_i$ in c_0 und der andere in c_i eingefärbt werden muss. Stellen Sie sich denjenigen in der Farbe c_0 als wahr und den anderen als falsch vor. Die gewählte Färbung entspricht also einer Belegung.

Um die Konstruktion zu vervollständigen, müssen Sie einen Teil des Graphen für jede Klausel des Ausdrucks entwerfen. Es sollte möglich sein, dass der Graph lediglich dann mit den Farben c_0 bis c_n zu färben ist, wenn jede Klausel durch die Belegung, die der Farbwahl entspricht, wahr wird. Der konstruierte Graph ist daher nur dann $(n + 1)$-färbbar, wenn der gegebene Ausdruck erfüllbar ist.

! **Übung 10.4.3** Ein Graph muss nicht besonders groß sein, bevor NP-vollständige Fragen nur schwer von Hand lösbar sind. Sehen Sie sich den Graphen in Abbildung 10.12 an.

* a) Besitzt dieser Graph einen Hamiltonschen Kreis?

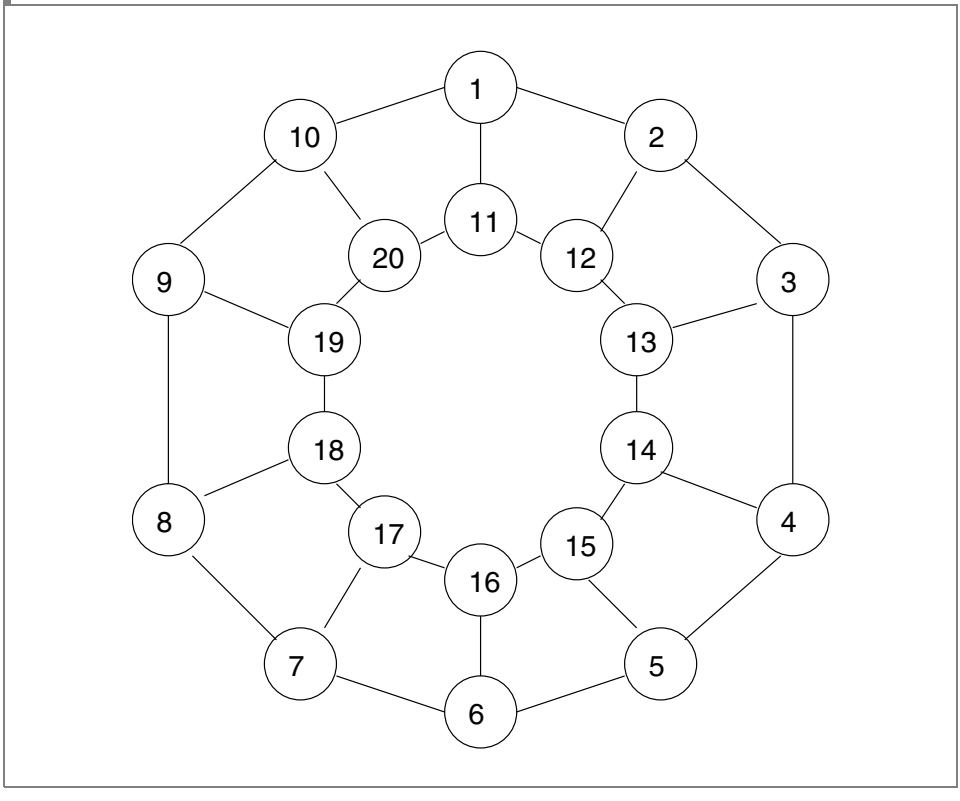
b) Welche ist die größte unabhängige Menge?

c) Welche ist die kleinste Knotenüberdeckung?

d) Welche ist die kleinste Kantenüberdeckung (siehe Übung 10.4.4 (c))?

e) Ist dieser Graph 2-färbbar?

Abbildung 10.12: Ein Graph



Übung 10.4.4 Zeigen Sie, dass die folgenden Probleme NP-vollständig sind:

a) Das *Problem des Teilgraph-Isomorphismus*: Gegeben seien die Graphen G_1 und G_2 Enthält G_1 eine Kopie von G_2 als Teilgraph? Können wir also eine Teilmenge der Knoten von G_1 finden, die zusammen mit ihren Kanten in G_1 eine exakte Kopie von G_2 bilden, wenn wir die Zuordnung zwischen den Knoten von G_2 und den Knoten des Teilgraphen von G_1 entsprechend wählen? *Hinweis:* Ziehen Sie eine Reduktion vom Cliquenproblem aus Übung 10.4.1 in Betracht.

! b) Das *Problem der Kantenüberdeckung* : Gegeben sei ein Graph G sowie eine ganze Zahl k. Besitzt G eine Kantenüberdeckung von k Kanten? Existiert also eine Menge von k Kanten, sodass jeder Knoten aus G ein Ende mindestens einer Kante aus der Kantenüberdeckung ist?

! c) Das *Problem der linearen Integer-Programmierung*: Gegeben sei eine Menge von linearen Ungleichungen der Form $\Sigma_{i=1}^{n} a_i x_i \leq c$ oder $\Sigma_{i=1}^{n} a_i x_i \geq c$, wobei die a und c ganzzahlige Konstanten und x_1, x_2, …, x_n Variablen sind. Existiert eine Zuweisung von ganzen Zahlen zu jeder der Variablen, sodass alle Ungleichungen erfüllt sind?

! d) Das *Problem der dominierenden Menge*: Gegeben seien ein Graph G und eine ganze Zahl k. Existiert eine Teilmenge S mit k Knoten aus G, sodass jeder Knoten entweder in S enthalten oder durch eine Kante mit einem Knoten aus S verbunden ist?

e) Das *Feuerwachenproblem*: Gegeben seien ein Graph G, eine Distanz d und ein Budget f von »Feuerwachen«. Können f Knoten aus G gewählt werden, sodass kein Knoten weiter als die Distanz d (die Anzahl der Kanten, die zu passieren sind) von einer Feuerwache entfernt ist?

*! f) Das *Problem der halben Clique*: Gegeben sei ein Graph G mit einer geraden Anzahl von Knoten. Existiert eine Clique von G (siehe Übung 10.4.1), die aus exakt der Hälfte der Knoten von G besteht? *Hinweis:* Reduzieren Sie CLIQUE auf das Problem der halben Clique. Sie müssen überlegen, wie Knoten hinzuzufügen sind, um die Größe der umfangreichsten Clique anzupassen.

!! g) Das *Problem der Ausführungsplanung in Zeiteinheiten* : Gegeben seien k »Aufgaben«

$$T_1, T_2, \ldots, T_k$$

eine Anzahl von »Prozessoren« p, eine Zeitbegrenzung t sowie Vorrangeinschränkungen der Form $T_i < T_j$ zwischen Aufgabenpaaren. Existiert eine *Zeitplanung* für die Aufgaben, sodass Folgendes gilt:

1. Jede Aufgabe wird einer Zeiteinheit zwischen 1 und t zugewiesen.
2. Höchstens p Aufgaben werden einer Zeiteinheit zugewiesen.
3. Die Vorrangeinschränkungen werden eingehalten; ist $T_i < T_j$ also eine Einschränkung, dann wird T_i einer früheren Zeiteinheit als T_j zugewiesen.

!! h) Das *Problem der exakten Überdeckung* : Gegeben seien eine Menge S und eine Menge von Teilmengen S_1, S_2, …, S_n von S. Existiert eine Menge von Mengen $T \subseteq \{ S_1, S_2, \ldots, S_n\}$, sodass jedes Element x aus S in genau einem Element aus T enthalten ist?

!! i) Das *Rucksackproblem*: Gegeben sei eine Liste von k ganzen Zahlen i_1, i_2, …, i_k. Können wir sie in zwei Mengen partitionieren, deren Summen identisch sind? *Hinweis:* Dieses Problem scheint oberflächlich in $\mathcal{P}$ enthalten zu sein, da wir annehmen können, dass es sich um kleine ganze Zahlen handelt. Sind die Werte der ganzen Zahlen auf ein Polynom in k beschränkt, dann existiert tatsächlich ein polynomialer Algorithmus. Allerdings können in einer Liste aus k

binären ganzen Zahlen mit der Gesamtlänge n auch ganze Zahlen auftreten, deren Werte fast exponentiell in n sind.

Übung 10.4.5 Ein *Hamiltonscher Pfad* in einem Graphen G mit k Knoten ist eine lineare Anordnung $n_1, n_2, \ldots, n_k$ der Knoten, sodass für alle $i = 1, 2, \ldots, k-1$ eine Kante von n_i nach n_{i+1} existiert. Ein *gerichteter Hamiltonscher Pfad* ist das Gleiche für einen gerichteten Graphen; in diesem Fall muss ein Pfeil von n_i nach n_{i+1} existieren. Beachten Sie, dass die Aufforderungen an einen Hamiltonschen Pfad nur wenig schwächer sind als die an einen Hamiltonschen Kreis. Würden wir noch eine Kante oder einen Pfeil von n_k nach n_1 fordern, dann lägen exakt die Bedingungen für einen Hamiltonschen Kreis vor. Das Problem eines (gerichteten) Hamiltonschen Pfades lautet: Gegeben sein ein (gerichteter) Graph. Besitzt er mindestens einen (gerichteten) Hamiltonschen Zyklus?

a) Beweisen Sie, dass das Problem des gerichteten Hamiltonschen Pfades NP-vollständig ist. *Hinweis:* Führen Sie eine Reduktion von DHC aus. Nehmen Sie einen Knoten und teilen Sie ihn in zwei Knoten auf, sodass diese beiden Knoten die Endpunkte eines gerichteten Hamiltonschen Pfades sind. Ein solcher Pfad existiert nur dann, wenn der ursprüngliche Graph einen gerichteten Hamiltonschen Kreis besitzt.

b) Zeigen Sie, dass das Problem des (ungerichteten) Hamiltonschen Pfades NP-vollständig ist. *Hinweis:* Adaptieren Sie die Konstruktion aus Satz 10.23.

c) Zeigen Sie, dass das folgende Problem NP-vollständig ist: Gegeben seien ein Graph G und eine ganze Zahl k. Besitzt G einen aufspannenden Baum mit höchstens k Blättern? *Hinweis:* Führen Sie eine Reduktion vom Problem des Hamiltonschen Pfades aus.

d) Zeigen Sie, dass das folgende Problem NP-vollständig ist: Gegeben seien ein Graph G und eine ganze Zahl d. Besitzt G einen aufspannenden Baum ohne einen Knoten von einem Grad größer als d? (Der *Grad* eines Knotens n im aufspannenden Baum ist die Anzahl der Kanten des Baumes, die n als ein Ende haben.)

10.5 Zusammenfassung von Kapitel 10

- *Die Klassen $\mathcal{P}$ und $\mathcal{NP}$*: $\mathcal{P}$ besteht aus all den Sprachen oder Problemen, die von einer Turing-Maschine akzeptiert werden, die in der Länge der Eingabe einen polynomialen Zeitaufwand benötigt. $\mathcal{NP}$ ist die Klasse der Sprachen oder Probleme, die von einer nichtdeterministischen TM mit einer polynomialen Begrenzung des Zeitaufwands längs jeder Folge nichtdeterministischer Wahlmöglichkeiten akzeptiert werden.
- *Die Frage, ob $\mathcal{P} = \mathcal{NP}$*: Es ist nicht bekannt, ob $\mathcal{P}$ und $\mathcal{NP}$ tatsächlich die gleichen Sprachklassen sind, doch wir haben die starke Vermutung, dass $\mathcal{NP}$ Sprachen enthält, die nicht in $\mathcal{P}$ enthalten sind.
- *Polynomiale Reduktionen*: Wenn wir Instanzen eines Problems mit einem polynomialen Zeitaufwand in Instanzen eines zweiten Problems überführen können, das die gleiche Antwort – ja oder nein – besitzt, dann sagen wir, das erste Problem ist mit polynomialem Zeitaufwand (polynomial) auf das zweite Problem reduzierbar.

- *NP-vollständige Probleme*: Eine Sprache ist NP-vollständig, wenn sie in $\mathcal{NP}$ enthalten ist und von jeder Sprache in $\mathcal{NP}$ eine polynomiale Reduktion auf die fragliche Sprache existiert. Wir sind so gut wie sicher, dass keines der NP-vollständigen Probleme in $\mathcal{P}$ enthalten ist, und die Tatsache, dass niemand bisher einen polynomialen Algorithmus für eines der vielen bekannten NP-vollständigen Probleme gefunden hat, bestärkt uns darin.
- *NP-vollständige Erfüllbarkeitsprobleme*: Der Satz von Cook zeigte das erste NP-vollständige Problem (ob ein Boolescher Ausdruck erfüllbar ist), indem alle in $\mathcal{NP}$ enthaltenen Probleme mit polynomialem Zeitaufwand auf das SAT-Problem reduziert wurden. Zudem bleibt das Problem NP-vollständig, auch wenn der Ausdruck lediglich aus einem Produkt aus Klauseln besteht, die wiederum jeweils aus nur drei Literalen bestehen – das 3SAT-Problem.
- *Weitere NP-vollständige Probleme*: Es gibt außerordentlich viele bekannte NP-vollständige Probleme; jedes ist durch eine polynomiale Reduktion auf ein schon vorher als NP-vollständig bekanntes Problem als NP-vollständig nachgewiesen worden. Wir haben Reduktionen vorgestellt, die die folgenden Probleme als NP-vollständig nachweisen: die unabhängige Menge, die Knotenüberdeckung, gerichtete und ungerichtete Versionen des Problems des Hamiltonschen Kreises sowie das Problem des Handlungsreisenden.

LITERATURANGABEN ZU KAPITEL 10

Das Konzept der NP-Vollständigkeit als Evidenz dafür, dass ein Problem nicht mit polynomialem Zeitaufwand gelöst werden kann, sowie der Beweis der NP-Vollständigkeit von SAT, CSAT und 3SAT stammen von Cook [3]. Eine nachfolgende Veröffentlichung von Karp [7] wird allgemein als ebenso wichtig erachtet, da diese Veröffentlichung gezeigt hat, dass NP-Vollständigkeit nicht nur ein isoliertes Phänomen ist, sondern auf sehr viele der harten kombinatorischen Probleme angewendet werden kann, die von Wissenschaftlern aus dem Bereich Operations Research und weiteren Disziplinen seit Jahren untersucht werden. Jedes der als NP-vollständig nachgewiesenen Probleme aus Abschnitt 10.4 stammt aus dieser Veröffentlichung: die unabhängige Menge, die Knotenüberdeckung, der Hamiltonsche Kreis und das Problem des Handlungsreisenden. Außerdem finden wir hier die Lösungen zu einigen Problemen, die in den Übungen erwähnt werden: die Clique, die Kantenüberdeckung, der Rucksack, das Färbungsproblem sowie die exakte Überdeckung.

Das Buch von Garey und Johnson [4] fasst viel Bekanntes über NP-vollständige Probleme und Spezialfälle mit polynomialem Zeitaufwand zusammen. [6] enthält Artikel über polynomiale Näherungslösungen für NP-vollständige Probleme.

Es sind noch weitere Beiträge zur Theorie der NP-Vollständigkeit zu erwähnen. Das Studium der Klassen von Sprachen, die durch die Ausführungszeit von Turing-Maschinen definiert werden, hat mit Hartmanis und Stearns [5] begonnen. Cobham [2] isolierte als Erster das Konzept der Klasse $\mathcal{P}$ in Gegenüberstellung zu Algorithmen, die eine bestimmte polynomiale Ausführungszeit, etwa $O(n^2)$, besitzen. Levin [8] entwickelte die Idee der NP-Vollständigkeit etwas später als [3], jedoch unabhängig davon.

Die NP-Vollständigkeit der linearen Integer-Programmierung (Übung 10.4.4 (c)) stammt aus [1] sowie aus unveröffentlichten Notizen von J. Gathen und M. Sieveking. Die NP-Vollständigkeit des Problems der Ausführungsplanung in Zeiteinheiten (Übung 10.4.4 (g)) stammt aus [9].

→

1. I. Borosh und L. B. Treybig [1976]. »Bounds on positive integral solutions of linear Diophantine equations«, *Proceedings of the AMS* 55, 299–304.
2. A. Cobham [1964]. »The intrinsic computational difficulty of functions«, *Proc. 1964 Congress for Logic, Mathematics, and the Philosophy of Science*, S. 24–30, North Holland, Amsterdam.
3. S. C. Cook [1971]. »The complexity of theorem-proving procedures«, *Third ACM Symposium on Theory of Computing*, S. 151–158, ACM, New York.
4. M. R. Garey und D. S. Johnson [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, H. Freeman, New York.
5. J. Hartmanis und R. E. Stearns [1965]. »On the computational complexity of algorithms«, *Transactions of the ACM* 117, 285–306.
6. D. S. Hochbaum (Hrsg.) [1996]. *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Co.
7. R. M. Karp [1972]. »Reducibility among combinatorial problems«, *Complexity of Computer Computations*, S. 85–104, Plenum Press, New York.
8. L. A. Levin [1973]. »Universal sorting problems«, *Problemi Peredachi Informatsii* 9:3, 265–266.
9. J. D. Ullman [1975]. »NP-complete scheduling problems«, *J. Computer and System Sciences* 10:3, 384–393.

KAPITEL 11

Weitere Problemklassen

Die Geschichte nicht handhabbarer Probleme beginnt nicht mit $\mathcal{NP}$ und endet auch nicht damit. Es gibt viele weitere Klassen von Problemen, die nicht handhabbar zu sein scheinen oder aus anderen Gründen interessant sind. Verschiedene Fragen zu diesen Klassen wie etwa die Frage $\mathcal{P} = \mathcal{NP}$ sind noch nicht gelöst.

Wir sehen uns zuerst eine Klasse an, die in einem engen Zusammenhang zu $\mathcal{P}$ und $\mathcal{NP}$ steht: die Klasse der Komplemente von $\mathcal{NP}$-Sprachen, oft »Co-$\mathcal{NP}$« genannt. Ist $\mathcal{P} = \mathcal{NP}$, dann stimmt Co-$\mathcal{NP}$ mit beiden überein, da $\mathcal{P}$ unter dem Komplement abgeschlossen ist. Allerdings ist es wahrscheinlich, dass sich Co-$\mathcal{NP}$ von beiden Klassen unterscheidet und kein NP-vollständiges Problem in Co-$\mathcal{NP}$ enthalten ist.

Dann sehen wir uns die Klasse $\mathcal{PS}$ an, die alle Probleme umfasst, die von einer Turing-Maschine gelöst werden können, die ein Band verwendet, dessen Länge polynomial in der Länge der Eingabe ist. Diesen Turing-Maschinen ist ein exponentieller Zeitaufwand erlaubt, solange sie sich innerhalb eines begrenzten Bandabschnitts aufhalten. Im Gegensatz zur Situation bei polynomialem Zeitaufwand können wir beweisen, dass der Nichtdeterminismus die Leistungsfähigkeit der TM nicht verbessert, wenn sich die Einschränkung auf den polynomialen Platz bezieht. Auch wenn $\mathcal{PS}$ offensichtlich ganz $\mathcal{NP}$ umfasst, wissen wir nicht, ob $\mathcal{PS}$ gleich $\mathcal{NP}$ oder etwa gleich $\mathcal{P}$ ist. Wir erwarten jedoch, dass die Gleichheit in keinem der beiden Fälle zutrifft, und stellen ein Problem vor, das für $\mathcal{PS}$ vollständig ist, aber nicht in $\mathcal{NP}$ enthalten zu sein scheint.

Danach wenden wir uns Zufallsalgorithmen sowie zwei Sprachklassen zu, die zwischen $\mathcal{P}$ und $\mathcal{NP}$ liegen. Eine davon ist die Klasse $\mathcal{RP}$ der »zufällig polynomialen« Sprachen. Diese Sprachen besitzen einen Algorithmus, der einen polynomialen Zeitaufwand erfordert und eine Art »Münzen werfen« oder (in der Praxis) einen Zufallsgenerator verwendet. Der Algorithmus bestätigt entweder, dass die Eingabe in der Sprache enthalten ist, oder sagt: »Ich weiß es nicht.« Ist die Eingabe in der Sprache enthalten, dann existiert eine Wahrscheinlichkeit größer 0, dass der Algorithmus Erfolg hat, und damit wird wiederholte Anwendung des Algorithmus mit einer gegen 1 gehenden Wahrscheinlichkeit das Enthaltensein bestätigen.

Auch die zweite Klasse, $\mathcal{ZPP}$ (Zero-Error, Probabilistic Polynomial), arbeitet mit Zufall. Allerdings antworten die Algorithmen für Sprachen aus dieser Klasse entweder mit »Ja, die Eingabe ist in der Sprache enthalten« oder mit »Nein, sie ist es nicht«. Die erwartete Ausführungszeit des Algorithmus ist polynomial. Allerdings sind auch Ausführungszeiten möglich, die jede polynomiale Grenze überschreiten können.

Um diese Konzepte zusammenzuführen, sehen wir uns den wichtigen Bereich der Primzahltests an. Viele kryptographische Systeme beruhen heutzutage auf Folgendem:

1. der Fähigkeit, große Primzahlen schnell zu entdecken (um eine Kommunikation zwischen Maschinen zu ermöglichen, die Außenstehende nicht abfangen können)
2. der Annahme, dass die Faktorisierung ganzer Zahlen einen exponentiellen Zeitaufwand erfordert, wenn die Zeit als Funktion der Länge n der ganzen Zahl in Binärdarstellung gemessen wird

Wir werden sehen, dass der Primzahltest sowohl in $\mathcal{NP}$ als auch in Co-$\mathcal{NP}$ enthalten und es damit sehr unwahrscheinlich ist, dass Primzahltests als NP-vollständig nachgewiesen werden können. Dies ist von Nachteil, da der Nachweis der NP-Vollständigkeit als das überzeugendste Argument angesehen wird, dass ein Problem sehr wahrscheinlich einen exponentiellen Zeitaufwand erfordert. Wir werden außerdem sehen, dass der Primzahltest in der Klasse $\mathcal{RP}$ enthalten ist. Diese Situation hat Vor- und Nachteile. Der Vorteil liegt darin, dass kryptographische Systeme, die Primzahlen benötigen, in der Praxis tatsächlich einen Algorithmus aus der Klasse $\mathcal{RP}$ für die Suche nach diesen Primzahlen verwenden. Nachteilig ist, dass unserer Annahme, Primzahltests seien nicht als NP-vollständig nachweisbar, zusätzliches Gewicht verliehen wird.

11.1 Komplemente von Sprachen, die in $\mathcal{NP}$ enthalten sind

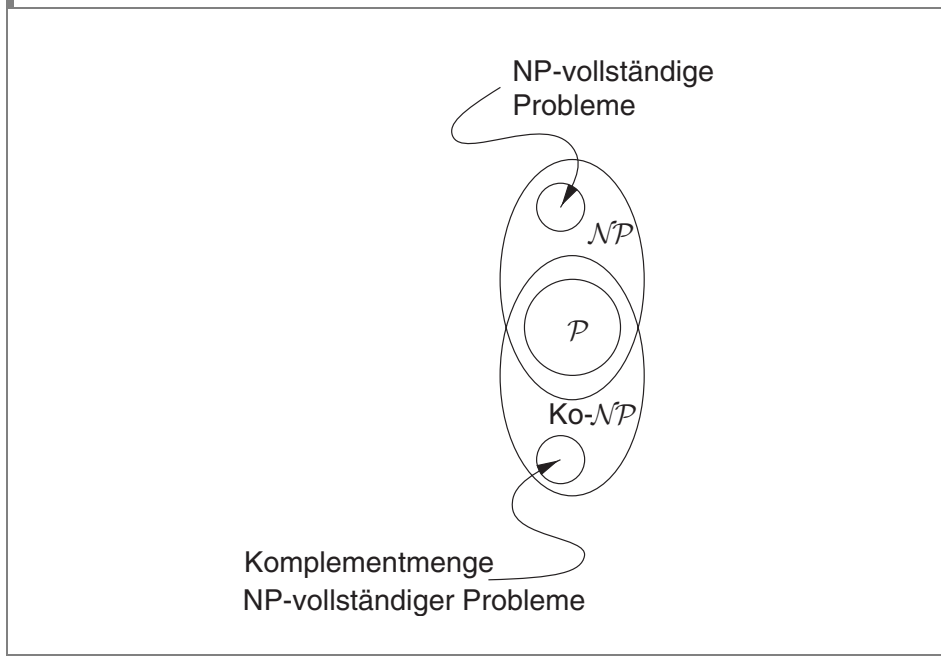
Die Klasse der Sprachen $\mathcal{P}$ ist unter der Komplementbildung abgeschlossen (siehe Übung 10.1.6). Für einen einfachen Beweis, weshalb, sei L in P enthalten und M eine TM für L. Wir ändern M wie folgt, sodass $\overline{L}$ akzeptiert wird. Wir führen einen neuen akzeptierenden Zustand q sowie einen Übergang der neuen TM nach q ein, wann immer M in einem Zustand anhält, der nicht akzeptiert. Die vorherigen akzeptierenden Zustände von M werden in nicht akzeptierende Zustände umgewandelt. Dann akzeptiert die geänderte TM $\overline{L}$ und benötigt den gleichen Zeitaufwand wie M, bis auf eine mögliche zusätzliche Bewegung. $\overline{L}$ ist daher in $\mathcal{P}$ enthalten, wenn auch L darin enthalten ist.

Es ist nicht bekannt, ob $\mathcal{NP}$ unter der Komplementbildung abgeschlossen ist. Dies ist anscheinend nicht der Fall, und wir erwarten insbesondere, dass das Komplement einer NP-vollständigen Sprache L nicht in $\mathcal{NP}$ enthalten ist.

11.1.1 Die Sprachklasse Co-$\mathcal{NP}$

CO-$\mathcal{NP}$ ist die Menge der Sprachen, deren Komplemente in $\mathcal{NP}$ enthalten sind. Wir haben zu Anfang von Abschnitt 11.1 beobachtet, dass das Komplement jeder Sprache in $\mathcal{P}$ ebenfalls in $\mathcal{P}$ und damit in $\mathcal{NP}$ enthalten ist. Auf der anderen Seite glauben wir, dass keines der Komplemente der NP-vollständigen Probleme in $\mathcal{NP}$ und damit kein NP-vollständiges Problem in Co-$\mathcal{NP}$ enthalten ist. Entsprechend glauben wir, dass die Komplemente NP-vollständiger Probleme, die per definitionem in Co-$\mathcal{NP}$ enthalten sind, nicht $\mathcal{NP}$ angehören. Abbildung 11.1 zeigt unsere Vorstellung der Beziehungen zwischen den Klassen $\mathcal{P}$, $\mathcal{NP}$ und Co-$\mathcal{NP}$. Wir sollten allerdings im Auge behalten, dass diese drei Klassen identisch sind, falls sich $\mathcal{P}$ als mit $\mathcal{NP}$ identisch herausstellt.

Abbildung 11.1: Vermutete Beziehungen zwischen Co-$\mathcal{NP}$ und anderen Sprachklassen



Beispiel 11.1 Sehen wir uns das Komplement der Sprache SAT an, die mit Sicherheit in Co-$\mathcal{NP}$ enthalten ist. Wir beziehen uns auf dieses Komplement als *USAT* (UnSATisfiable). Die Zeichenreihen in USAT umfassen alle Zeichenreihen, die nicht erfüllbare Boolesche Ausdrücke codieren. Allerdings sind in USAT auch Zeichenreihen enthalten, die keine gültigen Booleschen Ausdrücke codieren, da mit Sicherheit keine dieser Zeichenreihen in SAT enthalten ist. Wir glauben, dass USAT nicht in $\mathcal{NP}$ enthalten ist, aber dafür existiert kein Beweis.

Ein weiteres Beispiel für ein Problem, von dem wir vermuten, dass es zwar in Co-$\mathcal{NP}$, nicht aber in $\mathcal{NP}$ enthalten ist, heißt TAUT, die Menge aller (codierten) Booleschen Ausdrücke, die *Tautologien* sind, d. h. die bei jeder Belegung wahr sind. Beachten Sie, dass ein Ausdruck E ausschließlich dann eine Tautologie ist, wenn $\neg E$ nicht erfüllbar ist. Daher stehen TAUT und USAT in der Beziehung, dass für jeden Ausdruck E, der in TAUT enthalten ist, $\neg E$ in USAT enthalten ist und umgekehrt. Allerdings enthält USAT auch Zeichenreihen, die keinen gültigen Ausdruck repräsentieren, während es sich bei allen Zeichenreihen in TAUT um gültige Ausdrücke handelt. ■

11.1.2 NP-vollständige Probleme und Co-$\mathcal{NP}$

Wir nehmen an, dass $\mathcal{P} \neq \mathcal{NP}$ sei. Es ist noch immer möglich, dass die Situation von Co-$\mathcal{NP}$ nicht exakt mit der in Abbildung 11.1 beschriebenen übereinstimmt, da $\mathcal{NP}$ und Co-$\mathcal{NP}$ gleich, aber größer als P sein könnten. Wir könnten also entdecken, dass Probleme wie USAT und TAUT nichtdeterministisch polynomial lösbar (also in $\mathcal{NP}$ enthalten) sind, nicht aber deterministisch polynomial. Jedoch ist die Tatsache, dass wir nicht ein einziges NP-vollständiges Problem, dessen Komplement in $\mathcal{NP}$ enthal-

ten ist, gefunden haben, ein starkes Indiz, dass $\mathcal{NP} \neq$ Co-$\mathcal{NP}$ ist, wie wir im nächsten Satz beweisen werden.

Satz 11.2 $\mathcal{NP}$ = Co-$\mathcal{NP}$ dann und nur dann, wenn es ein NP-vollständiges Problem gibt, dessen Komplement in $\mathcal{NP}$ enthalten ist.

Beweis: (Nur wenn) Sollten $\mathcal{NP}$ und Co-$\mathcal{NP}$ übereinstimmen, dann ist mit Sicherheit jedes NP-vollständige Problem L, das ja in $\mathcal{NP}$ enthalten ist, ebenso in Co-$\mathcal{NP}$ enthalten. Da das Komplement eines Problems in Co-$\mathcal{NP}$ in $\mathcal{NP}$ enthalten ist, ist also das Komplement von L in $\mathcal{NP}$ enthalten.

(Wenn) Angenommen, P sei ein NP-vollständiges Problem, dessen Komplement $\overline{P}$ in $\mathcal{NP}$ enthalten ist; dann existiert für jede Sprache L in $\mathcal{NP}$ eine polynomiale Reduktion von L auf P. Diese Reduktion ist genauso eine polynomiale Reduktion von $\overline{L}$ auf $\overline{P}$. Wir beweisen, dass $\mathcal{NP}$ = Co-$\mathcal{NP}$, indem wir das Enthaltensein in beiden Richtungen beweisen.

$\mathcal{NP} \subseteq$ Co-$\mathcal{NP}$: Angenommen, L sei in $\mathcal{NP}$ enthalten. Dann ist $\overline{L}$ in Co-$\mathcal{NP}$ enthalten. Wir kombinieren die polynomiale Reduktion von $\overline{L}$ auf $\overline{P}$ mit dem angenommenen nichtdeterministisch polynomialen Algorithmus für $\overline{P}$, sodass wir einen nichtdeterministisch polynomialen Algorithmus für $\overline{L}$ erhalten. Daher ist für jedes L in $\mathcal{NP}$ auch $\overline{L}$ in $\mathcal{NP}$ enthalten. Damit ist L als Komplement einer Sprache in $\mathcal{NP}$ in Co-$\mathcal{NP}$ enthalten. Diese Beobachtung zeigt, dass $\mathcal{NP} \subseteq$ Co-$\mathcal{NP}$.

Co-$\mathcal{NP} \subseteq \mathcal{NP}$: Angenommen, L sei in Co-$\mathcal{NP}$ enthalten. Dann gibt es eine polynomiale Reduktion von $\overline{L}$ auf P, da P NP-vollständig und L in $\mathcal{NP}$ enthalten ist. Diese Reduktion ist ebenso eine Reduktion von L auf $\overline{P}$. Da $\overline{P}$ in $\mathcal{NP}$ enthalten ist, kombinieren wir die Reduktion mit dem nichtdeterministisch polynomialen Algorithmus von $\overline{P}$ und zeigen so, dass L in $\mathcal{NP}$ enthalten ist. ■

11.1.3 Übungen zum Abschnitt 11.1

! **Übung 11.1.1** Im Folgenden sind einige Probleme beschrieben. Zeigen Sie für jedes Problem, ob es in $\mathcal{NP}$ und ob es in Co-$\mathcal{NP}$ enthalten ist. Beschreiben Sie das Komplement jedes Problems. Ist entweder das Problem oder sein Komplement NP-vollständig, dann beweisen Sie dies.

* a) Das Problem TRUE-SAT: Gegeben sei ein Boolescher Ausdruck E, der wahr ist, wenn alle Variablen wahr sind. Existiert eine andere Belegung neben »alles wahr«, durch die E wahr wird?

b) Das Problem FALSE-SAT: Gegeben sei ein Boolescher Ausdruck E, der falsch ist, wenn alle Variablen falsch sind. Existiert eine andere Belegung neben »alles falsch«, durch die E falsch wird?

c) Das Problem DOUBLE-SAT: Gegeben sei ein Boolescher Ausdruck E. Gibt es mindestens zwei Belegungen, durch die E wahr wird?

d) Das Problem NEAR-TAUT: Gegeben sei ein Boolescher Ausdruck E. Gibt es höchstens eine Belegung, durch die E falsch wird?

***! Übung 11.1.2** Angenommen, es existiere eine 1-1-Abbildung der n-Bit-Ganzzahlen auf die n-Bit-Ganzzahlen, sodass Folgendes gilt:

1. $f(x)$ kann mit polynomialem Zeitaufwand berechnet werden.
2. $f^{-1}(x)$ kann nicht mit polynomialem Zeitaufwand berechnet werden.

Zeigen Sie, dass dann die Sprache der ganzzahligen Paare (x, y) mit

$$f^{-1}(x) < y$$

in ($\mathcal{NP} \cap$ Co-$\mathcal{NP}$) - $\mathcal{P}$ enthalten wäre.

11.2 Probleme, die mit polynomialem Speicherplatz lösbar sind

Wir sehen uns nun eine Problemklasse an, die $\mathcal{NP}$ und anscheinend noch mehr umfasst, obwohl wir dessen nicht sicher sind. Diese Klasse ist dadurch definiert, dass einer Turing-Maschine erlaubt ist, Speicherplatz zu verwenden, der in Bezug zur Größe der Eingabe polynomial ist; die benötigte Zeit ist dabei nicht von Bedeutung. Wir unterscheiden zunächst zwischen den Sprachen, die von deterministischen und nichtdeterministischen TMs mit polynomialer Platzbegrenzung akzeptiert werden, doch diese beiden Sprachklassen werden sich als identisch erweisen.

Es existieren vollständige Probleme P für polynomialen Platz in dem Sinn, dass alle Probleme dieser Klasse mit polynomialem *Zeitaufwand* auf P reduzierbar sind. Ist P also in $\mathcal{P}$ oder $\mathcal{NP}$ enthalten, dann sind alle Sprachen von TMs mit polynomialer Platzbegrenzung in $\mathcal{P}$ bzw. $\mathcal{NP}$ enthalten. Wir stellen ein Beispiel für ein solches Problem vor: »quantifizierte Boolesche Formeln«.

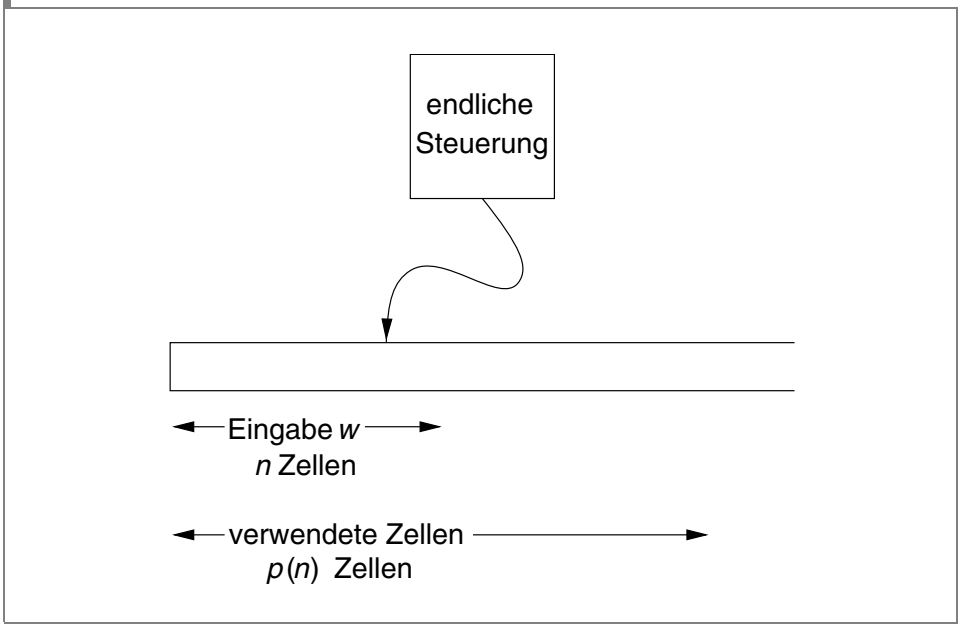
11.2.1 Turing-Maschinen mit polynomialer Platzbegrenzung

Abbildung 11.2 zeigt eine Turing-Maschine mit polynomialer Platzbegrenzung. Es existiert ein polynomiales $p(n)$, sodass die TM bei einer Eingabe w der Länge n niemals mehr als $p(n)$ Zellen auf ihrem Band verwendet. Nach Satz 8.12 können wir annehmen, dass das Band lediglich nach einer Seite hin unendlich ist und sich die TM nie auf eine Position links vom Eingabeanfang bewegt.

Wir definieren die Sprachklasse $\mathcal{PS}$ (*Polynomial Space*), sodass sie genau die Sprachen enthält, die $L(M)$ einer deterministischen Turing-Maschine M mit polynomialer Platzbegrenzung sind. Wir definieren außerdem die Klasse $\mathcal{NPS}$ (*Nondeterministic Polynomial Space*), die all die Sprachen enthält, die $L(M)$ einer nichtdeterministischen TM M mit polynomialer Platzbegrenzung sind. Offensichtlich gilt $\mathcal{PS} \subseteq \mathcal{NPS}$, da jede deterministische TM (aus technischer Sicht) auch nichtdeterministisch ist. Wir werden jedoch das überraschende Ergebnis $\mathcal{PS} = \mathcal{NPS}$[1] beweisen.

1. In anderen Veröffentlichungen zu diesem Thema wird diese Klasse gelegentlich als PSPACE bezeichnet. Wir ziehen jedoch $\mathcal{PS}$ vor, um die Probleme zu benennen, die mit deterministischem (oder nichtdeterministischem) polynomialen Zeitaufwand lösbar sind. Sobald die Äquivalenz $\mathcal{PS} = \mathcal{NPS}$ bewiesen ist, werden wir die Bezeichnung $\mathcal{NPS}$ nicht mehr verwenden.

Abbildung 11.2: Eine TM, die einen polynomialen Platz verwendet



11.2.2 Beziehungen zwischen $\mathcal{PS}$ und $\mathcal{NPS}$ mit früher definierten Klassen

Die Beziehungen $\mathcal{P} \subseteq \mathcal{PS}$ und $\mathcal{NP} \subseteq \mathcal{NPS}$ sollten offensichtlich sein. Dies ist darin begründet, dass eine TM mit einer polynomialen Anzahl von Bewegungen nicht mehr als eine polynomiale Anzahl von Zellen verwendet; insbesondere entspricht die Anzahl höchstens besuchter Zellen der Anzahl der Bewegungen plus 1. Sobald wir also $\mathcal{PS} = \mathcal{NPS}$ bewiesen haben, dass die drei Klassen tatsächlich eine Enthaltenseinskette bilden: $\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PS}$.

TMn mit polynomialer Platzbegrenzung besitzen die wichtige Eigenschaft, dass sie nur eine exponentielle Anzahl von Bewegungen ausführen können, bevor sie eine Konfiguration wiederholen müssen. Wir benötigen diese Tatsache zum Beweis anderer interessanter Tatsachen über $\mathcal{PS}$ und um zu zeigen, dass $\mathcal{PS}$ lediglich rekursive Sprachen enthält, also Sprachen mit Algorithmen. Beachten Sie, dass die Definition von $\mathcal{PS}$ oder $\mathcal{NPS}$ nicht von der TM fordert, anzuhalten. Es ist möglich, dass die TM unendlich lange in Schleifen arbeitet, ohne einen polynomial begrenzten Bereich des Bandes zu verlassen.

Satz 11.3 Ist M eine TM (deterministisch oder nichtdeterministisch) mit polynomialer Platzbegrenzung $p(n)$, dann existiert eine Konstante c, sodass M im Akzeptanzfall die Eingabe w der Länge n innerhalb von $c^{1+p(n)}$ Bewegungen akzeptiert.

Beweis: Die grundlegende Idee liegt darin, dass M eine Konfiguration wiederholen muss, wenn sie mehr als $c^{1+p(n)}$ Bewegungen ausführt. Falls M eine Konfiguration wiederholt und dann akzeptiert, muss eine kürzere Folge von Konfigurationen existieren,

die zur Akzeptanz führt. Gilt also $\alpha \vdash^* \beta \vdash^* \beta \vdash^* \chi$, wobei α die anfängliche Konfiguration, β die wiederholte Konfiguration und χ die akzeptierende Konfiguration ist, dann ist $\alpha \vdash^* \beta \vdash^* \chi$ eine kürzere Folge von Konfigurationen, die zur Akzeptanz führt.

Die Behauptung, c müsse existieren, beruht auf der Tatsache, dass lediglich eine begrenzte Anzahl von Konfigurationen existiert, wenn der von der TM verwendete Platz begrenzt ist. Insbesondere sei t die Anzahl der Bandsymbole und s die Anzahl der Zustände von M. Dann beträgt die Anzahl verschiedener Konfigurationen von M höchstens $sp(n)t^{p(n)}$, wenn nur $p(n)$ Bandzellen verwendet werden. Das heißt wir können einen der s Zustände wählen, den Kopf über einer der $p(n)$ Bandpositionen platzieren und die $p(n)$ Zellen mit jeder der $t^{p(n)}$ Folgen von Bandsymbolen beschreiben.

Wir wählen $c = s + t$. Dann betrachten wir die Binomialentwicklung von $(t + s)^{1+p(n)}$, die wie folgt lautet:

$$t^{1+p(n)} + (1 + p(n))st^{p(n)} + \ldots$$

Beachten Sie, dass der zweite Term mindestens so groß wie $sp(n)t^{p(n)}$ ist, was beweist, dass $c^{1+p(n)}$ mindestens gleich der Anzahl möglicher Konfigurationen von M ist. Wir schließen den Beweis mit der Beobachtung ab, dass M im Fall der Akzeptanz von w mit Länge n eine Folge von Bewegungen ausführt, die keine Konfiguration wiederholt. Daher akzeptiert M mit einer Folge von Bewegungen, die nicht länger als die Anzahl der verschiedenen Konfigurationen ist, also höchstens $c^{1+p(n)}$. ■

Wir können Satz 11.3 verwenden, um jede TM mit polynomialer Platzbegrenzung in eine äquivalente TM zu konvertieren, die nach einer höchstens exponentiellen Anzahl von Bewegungen immer anhält. Der entscheidende Punkt ist, dass wir zählen können, wie viele Bewegungen die TM jeweils ausgeführt hat, und da wir wissen, dass die TM innerhalb einer exponentiellen Anzahl von Bewegungen akzeptiert, können wir die TM auch zum Halten veranlassen, wenn sie genügend Bewegungen ausgeführt hat, ohne zu akzeptieren.

Satz 11.4 Ist die Sprache L in $\mathcal{PS}$ (bzw. $\mathcal{NPS}$) enthalten, dann wird L von einer deterministischen (bzw. nichtdeterministischen) TM mit polynomialer Platzbegrenzung akzeptiert, die nach höchstens $c^{q(n)}$ Bewegungen für ein Polynom $q(n)$ und eine Konstante $c > 1$ anhält.

Beweis: Wir beweisen die Behauptung für deterministische TMn; der Beweis gilt ebenso für NTMs. Wir wissen, dass L von einer TM M_1 akzeptiert wird, die eine polynomiale Platzbegrenzung $p(n)$ besitzt. Nach Satz 11.3 wird M_1 im Akzeptanzfall w nach höchstens $c^{1+p(|w|)}$ Schritten akzeptieren.

Wir entwerfen eine neue TM M_2, die mit zwei Bändern ausgestattet ist. M_2 simuliert auf dem ersten Band M_1 und zählt auf dem zweiten Band in der Basis c bis $c^{1+p(|w|)}$. Erreicht M_2 diesen Wert, dann hält sie an, ohne zu akzeptieren. M_2 verwendet daher $1 + p(|w|)$ Zellen des zweiten Bandes. Wir nahmen außerdem an, dass M_1 nicht mehr als $p(|w|)$ Zellen auf dem eigenen Band verwendet, und damit benutzt auch M_2 höchstens $p(|w|)$ Zellen ihres ersten Bandes.

Konvertieren wir M_2 in eine einbändige TM M_3, dann können wir sicher sein, dass M_3 bei einer beliebigen Eingabe der Länge n höchstens $1 + p(n)$ Bandzellen verwendet.

M_3 könnte zwar das Quadrat der Ausführungszeit von M_2 erfordern, doch der Zeitaufwand beträgt auch dann jedenfalls höchstens $O(c^{2p(n)})$.[2]

Da M_3 also höchstens $dc^{2p(n)}$ Bewegungen für eine Konstante d ausführt, führt M_3 mit $q(n) = 2p(n) + \log_c d$ höchstens $c^{q(n)}$ Schritte aus. Da M_2 immer anhält, hält auch M_3 immer an. Da M_1 L akzeptiert, wird L auch von M_2 und M_3 akzeptiert. Daher erfüllt M_3 die Behauptung des Satzes. ■

11.2.3 Deterministischer und nichtdeterministischer polynomialer Speicherplatz

Da der Vergleich zwischen $\mathcal{P}$ und $\mathcal{NP}$ so schwierig erscheint, ist es überraschend, dass ebendieser Vergleich zwischen $\mathcal{PS}$ und $\mathcal{NPS}$ einfach ist: Es handelt sich bei beiden um die gleiche Sprachklasse. Der Beweis gelingt durch die Simulation einer nichtdeterministischen TM mit polynomialer Platzbegrenzung $p(n)$ durch eine deterministische TM mit polynomialer Platzbegrenzung $O(p^2(n))$.

Im Mittelpunkt des Beweises steht ein deterministischer rekursiver Test, ob eine NTM N mit höchstens m Bewegungen von Konfiguration I in Konfiguration J wechseln kann. Eine DTM D prüft systematisch alle mittleren Konfigurationen K darauf, ob I mit $m/2$ Bewegungen zu K und K mit $m/2$ Bewegungen zu J werden kann. Wir verwenden eine rekursive Funktion *reach*(I, J, m) zur Entscheidung, ob $I \vdash^* J$ in höchstens m Schritten.

Stellen Sie sich das Band von D als Keller vor, in dem die Argumente der rekursiven Aufrufe von *reach* abgelegt werden. Ein *Stackframe* D enthält also $[I, J, m]$. Listing 11.1 zeigt eine Skizze des von *reach* ausgeführten Algorithmus.
Wichtig ist die Beobachtung, dass sich *reach* zwar zweimal selbst aufruft, diese Aufrufe jedoch in Folge ausführt werden und damit immer nur ein Aufruf zu einem Zeitpunkt aktiv ist. Beginnen wir also mit einem Stackframe $[I_1, J_1, m]$, dann existiert zu jedem Zeitpunkt nur ein Aufruf $[I_2, J_2, m/2]$, ein Aufruf $[I_3, J_3, m/4]$, ein Aufruf $[I_4, J_4, m/8]$ und so weiter, bis das dritte Argument irgendwann 1 ist. An diesem Punkt kann *reach* den Basisschritt anwenden und benötigt keine weiteren rekursiven Aufrufe. Die Funktion prüft, ob $I = J$ oder $I \vdash J$ gilt, und gibt `TRUE` zurück, wenn eine der Bedingungen erfüllt ist, sonst jedoch `FALSE`. Abbildung 11.3 zeigt, wie der Stack der DTM D aussieht, wenn alle möglichen Aufrufe von *reach* aktiv sind, wobei m der anfängliche Schrittzähler ist.

Es scheint zwar, dass sehr viele Aufrufe von *reach* möglich sind und das Band in Abbildung 11.3 sehr lang werden kann, doch wir werden zeigen, dass es nicht »zu lang« wird. Wird mit einem Schrittzähler m begonnen, dann können sich lediglich $\log_2 m$ Stackframes zu irgendeinem Zeitpunkt auf dem Band befinden. Da Satz 11.4 zusichert, dass die NTM N höchstens $c^{p(n)}$ Schritte ausführen kann, muss m nicht mit einer Anzahl beginnen, die größer ist. Die Anzahl der Stackframes beträgt also höchstens $\log_2 c^{p(n)}$, d. h. $O(p(n))$. Wir besitzen nun die Grundlagen für den Beweis des folgenden Satzes.

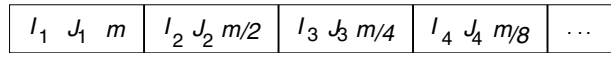
---

2. Tatsächlich ist die allgemeine Regel aus Satz 8.10 nicht die stärkste mögliche Behauptung. Da auf jedem Band lediglich $1 + p(n)$ Zellen verwendet werden, können sich die simulierten Bandköpfe in der Konstruktion der Abbildung vieler Bänder auf ein Band nur um $1 + p(n)$ Zellen voneinander entfernen. Daher können $c^{1+p(n)}$ Bewegungen der mehrbändigen TM M_2 in $O(p(n)c^{p(n)})$ Schritten simuliert werden, und das sind weniger als $O(c^{2p(n)})$ Schritte.

Listing 11.1: Die rekursive Funktion *reach* prüft, ob eine KONFIGURATION mit einer gegebenen Anzahl von Schritten zu einer anderen KONFIGURATION werden kann

```
BOOLEAN FUNCTION reach(I,J,m)
    KONFIGURATION: I,J,; INT: m;
    BEGIN
        IF (m == 1) THEN /* Basis*/ BEGIN
            test, ob I == J oder ob I in einem
            Schritt zu J wird;
            RETURN TRUE in diesem Fall, FALSE sonst;
        END;
        ELSE /* Induktiver Teil */ BEGIN
            FOR jede mögliche KONFIGURATION K DO
                IF (reach(I,K,m/2) AND reach(K,J,m/2)) THEN
                    RETURN TRUE;
            RETURN FALSE;
        END;
    END;
```

Abbildung 11.3: Das Band einer DTM, die mit rekursiven Aufrufen von *reach* eine NTM simuliert



Satz 11.5 (Satz von Savitch) $\mathcal{PS} = \mathcal{NPS}$.

Beweis: Es ist offensichtlich, dass $\mathcal{PS} \subseteq \mathcal{NPS}$, da jede DTM technisch gesehen auch eine NTM ist. Wir müssen daher lediglich zeigen, dass $\mathcal{NPS} \subseteq \mathcal{PS}$ gilt; wird L also von einer NTM N mit Platzbegrenzung $p(n)$ für ein Polynom $p(n)$ akzeptiert, dann wird L auch von einer DTM D mit einer polynomialen Platzbegrenzung $q(n)$ für ein anderes Polynom $q(n)$ akzeptiert. Wir werden zeigen, dass $q(n)$ so gewählt werden kann, dass die Größenordnung dem Quadrat von $p(n)$ entspricht.

Zuerst können wir nach Satz 11.3 annehmen, dass N im Akzeptanzfall höchstens $c^{1+p(n)}$ Schritte für eine Konstante c benötigt. Bei einer Eingabe w der Länge n entdeckt die DTM D, was N mit Eingabe w ausführt, indem sie wiederholt das Tripel $[I_0, J, m]$ auf ihr Band schreibt und *reach* mit diesen Argumenten aufruft, wobei Folgendes gilt:

1. I_0 ist die anfängliche KONFIGURATION von N mit Eingabe w.
2. J ist eine akzeptierende KONFIGURATION, die höchstens $p(n)$ Bandzellen verwendet; die verschiedenen J werden von D unter Verwendung eines Hilfsbands systematisch aufgezählt.
3. $m = \log_2 c^{1+p(n)}$.

Wir haben oben schon festgestellt, dass es nie mehr als $\log_2 m$ rekursive Aufrufe geben wird, die zur gleichen Zeit aktiv sind; also einen mit dem dritten Argument m, einen

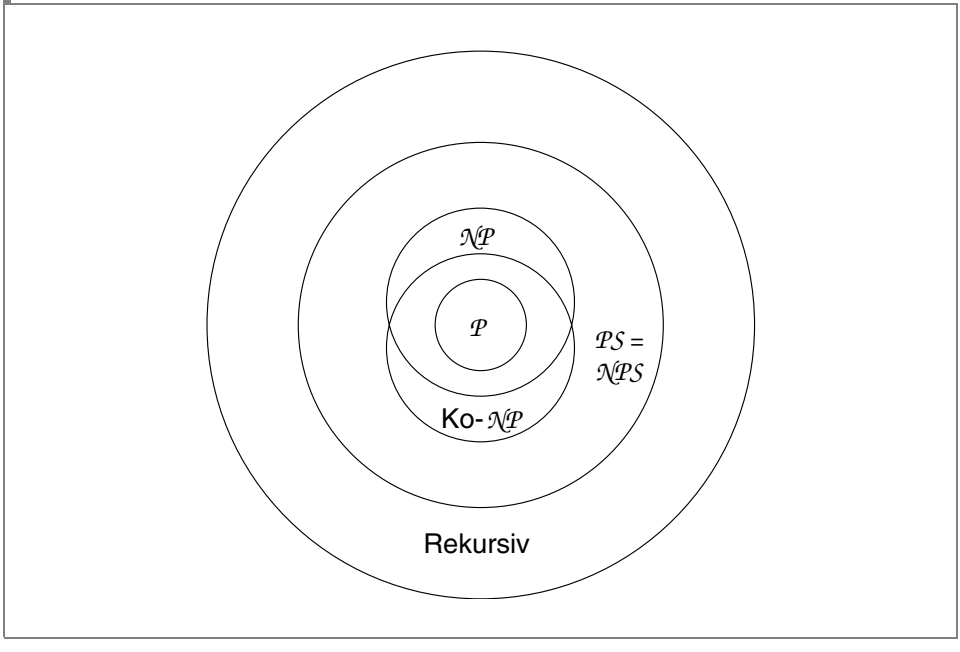
mit $m/2$, einen mit $m/4$ und so weiter, bis 1 erreicht ist. Daher befinden sich auch nicht mehr als $\log_2 m$ Stackframes auf dem Stack, und $\log_2 m$ ist $O(p(n))$.

Weiterhin nehmen die Stackframes selbst einen Platz von $O(p(n))$ ein. Dies liegt daran, dass die beiden Konfigurationen jeweils nur $1 + p(n)$ Zellen zum Schreiben erfordern, und zur Binärdarstellung von m sind $\log_2 c^{1+p(n)}$ Zellen erforderlich, also $O(p(n))$. Daher nimmt der gesamte Stackframe, der aus den beiden Konfigurationen und einer ganzen Zahl besteht, den Platz $O(p(n))$ ein.

Da D höchstens $O(p(n))$ Stackframes benötigt, entspricht der gesamte verwendete Platz $O(p^2(n))$. Dieser Platz ist polynomial, wenn $p(n)$ ein Polynom ist, und so folgern wir, dass es zu L eine DTM gibt, deren Platz polynomial begrenzt ist. ■

Zusammenfassend können wir unser Wissen über Komplexitätsklassen um Klassen mit polynomialem Platz erweitern. Abbildung 11.4 zeigt das vollständige Diagramm.

Abbildung 11.4: Bekannte Beziehungen zwischen Sprachklassen



11.3 Ein für $\mathcal{PS}$ vollständiges Problem

In diesem Abschnitt werden Sie das Problem »quantifizierter Boolescher Formeln« kennen lernen. Wir werden zeigen, dass es in $\mathcal{PS}$ vollständig ist.

11.3.1 PS-Vollständigkeit

Wir definieren ein Problem P als *vollständig für* $\mathcal{PS}$ (PS-vollständig), wenn Folgendes gilt:

1. $\mathcal{P}$ ist in $\mathcal{PS}$ enthalten.
2. Alle Sprachen L in $\mathcal{PS}$ sind mit polynomialem Zeitaufwand auf P reduzierbar.

Beachten Sie, dass die Anforderung an PS-Vollständigkeit derjenigen an NP-Vollständigkeit gleicht, obwohl wir den polynomialen Platz und nicht den polynomialen Zeitaufwand betrachten: Die Reduktion muss mit polynomialem Zeitaufwand ausgeführt werden. Dies liegt daran, dass wir wissen wollen, dass $\mathcal{P} = \mathcal{PS}$ gilt, falls ein PS-vollständiges Problem in $\mathcal{P}$ enthalten sein sollte; entsprechend gilt $\mathcal{NP} = \mathcal{PS}$, wenn ein PS-vollständiges Problem in $\mathcal{NP}$ enthalten ist. Würde die Reduktion lediglich mit polynomialem Platz ausgeführt, könnte die Größe der Ausgabe exponentiell in der Größe der Eingabe sein, und die Folgerungen aus dem nächsten Satz wären nicht möglich. Da wir uns jedoch auf Reduktionen mit polynomialem Zeitaufwand konzentrieren, erhalten wir die gewünschten Beziehungen.

Satz 11.6 Angenommen, P sei ein PS-vollständiges Problem. Dann gilt Folgendes:

a) Wenn P in $\mathcal{P}$ enthalten ist, dann gilt $\mathcal{P} = \mathcal{PS}$.

b) Wenn P in $\mathcal{NP}$ enthalten ist, dann gilt $\mathcal{NP} = \mathcal{PS}$.

Beweis: Wir beweisen (a). Wir wissen für jedes L in $\mathcal{PS}$, dass eine Reduktion mit polynomialem Zeitaufwand von L auf P existiert. Diese Reduktion habe den Zeitaufwand $q(n)$. Außerdem nehmen wir an, P sei in $\mathcal{P}$ enthalten und besitze daher einen Algorithmus mit polynomialem Zeitaufwand $p(n)$.

Um zu prüfen, ob eine gegebene Zeichenreihe w in L enthalten ist, können wir die Reduktion verwenden, um w in eine Zeichenreihe x zu konvertieren, die nur dann in P enthalten ist, wenn w in L enthalten ist. Da die Reduktion den Zeitaufwand $q(|w|)$ erfordert, kann die Zeichenreihe x nicht länger als $q(|w|)$ sein. Wir können mit dem Zeitaufwand $p(|x|)$ prüfen, ob x in P enthalten ist; dies entspricht $p(q(|w|))$, und dieser Wert ist in $|w|$ polynomial. Wir folgern, dass ein Algorithmus mit polynomialem Zeitaufwand für L existiert.

Daher ist jede Sprache L in $\mathcal{PS}$ auch in $\mathcal{P}$ enthalten. Da offensichtlich ist, dass $\mathcal{P}$ in $\mathcal{PS}$ enthalten ist, folgern wir, dass $\mathcal{P} = \mathcal{PS}$ gilt, wenn P in $\mathcal{P}$ enthalten ist. Der Beweis von (b), wo angenommen wird, P sei in $\mathcal{NP}$ enthalten, wird ganz ähnlich durchgeführt; er sei dem Leser überlassen. ■

11.3.2 Quantifizierte Boolesche Formeln

Wir werden ein Problem P vorstellen, das $\mathcal{PS}$-vollständig ist. Zunächst jedoch müssen wir uns mit den Begriffen befassen, mit denen dieses Problem der »quantifizierten Booleschen Formeln« oder QBF (Quantified Boolean Formulas) definiert ist.

Grob gesagt entspricht eine quantifizierte Boolesche Formel einem Booleschen Ausdruck, der zusätzlich die Operatoren $\forall$ (»für alle«) und $\exists$ (»es existiert«) enthält. Der Ausdruck $(\forall x)(E)$ bedeutet, dass E wahr ist, wenn jedes Auftreten von x in E durch 1 (wahr) ersetzt wird, und E ist ebenfalls wahr, wenn jedes Auftreten von x durch 0 (falsch) ersetzt wird. Der Ausdruck $(\exists x)(E)$ bedeutet, dass E wahr ist, wenn alle Auftreten von x entweder durch 1 oder durch 0 ersetzt werden oder wenn beides der Fall ist.

Um unsere Beschreibung zu vereinfachen, nehmen wir an, dass keine QBF mehr als eine *Quantifizierung* ($\forall$ oder $\exists$) der gleichen Variablen x enthält. Diese Einschränkung ist nicht von Bedeutung und etwa dem Verbot gleichzusetzen, dass zwei verschiedene

Funktionen in einem Programm die gleiche lokale Variable verwenden[3]. Formal sind die *quantifizierten Booleschen Formeln* wie folgt definiert:

1. 0 (falsch), 1 (wahr) und jede Variable sind QBFs.
2. Sind E und F QBFs, dann sind auch (E), $\neg(E)$, $(E) \wedge (F)$ sowie $(E) \vee (F)$ QBFs, wobei diese Ausdrücke das E in Klammern, das negierte E sowie das UND bzw. ODER von E und F repräsentieren. Redundante Klammern können weggelassen werden, wobei die üblichen Vorrangregeln gelten: NICHT, dann UND und dann ODER (mit dem geringsten Vorrang). Wir tendieren dazu, UND und ODER »arithmetisch« zu repräsentieren, wobei UND durch die Juxtaposition (kein Operator) und ODER durch + dargestellt wird. Wir verwenden also häufig $(E)(F)$ statt $(E) \wedge (F)$ sowie $(E) + (F)$ statt $(E) \vee (F)$.
3. Ist F eine QBF, die keine Quantifizierung der Variablen x enthält, dann sind $(\forall x)(E)$ und $(\exists x)(E)$ QBFs. Wir sagen, der Ausdruck E ist der Gültigkeitsbereich von x. Intuitiv ist x lediglich innerhalb von E definiert, entsprechend einer Variablen in einem Programm, deren Gültigkeitsbereich die Funktion ist, in der sie deklariert ist. Die Klammern um E (nicht jedoch die der Quantifizierung) können weggelassen werden, wenn keine Mehrdeutigkeiten existieren. Um zu viele verschachtelte Klammern zu vermeiden, können wir eine Kette von Quantifizierungen wie

   $$(\forall x)((\exists y)((\forall z)(E)))$$

   mit lediglich einer Ausdrucksklammer um E schreiben, statt jeden Ausdruck der Kette in Klammern zu setzen, und erhalten $(\forall x)(\exists y)(\forall z)(E)$.

Beispiel 11.7 Es folgt ein Beispiel für eine QBF:

$$(\forall x)((\exists y)(xy) + (\forall z)(\neg x + z)) \tag{11.1}$$

Wir beginnen mit den Variablen x sowie y und verbinden sie mit UND. Dann wenden wir die Quantifizierung $(\exists y)$ an und erhalten den Teilausdruck $(\exists y)(xy)$. Ebenso konstruieren wir den Booleschen Ausdruck $\neg x + z$, und die Anwendung der Quantifizierung $(\forall z)$ führt zum Teilausdruck $(\forall z)(\neg x + z)$. Dann kombinieren wir die beiden Teilausdrücke mit ODER, wenden die Quantifizierung $(\forall x)$ auf diesen Ausdruck an und erhalten so die vollständige QBF. ■

11.3.3 Quantifizierte Boolesche Formeln auswerten

Wir müssen noch formal die Bedeutung einer QBF definieren. Lesen wir allerdings $\forall$ als »für alle« und $\exists$ als »es existiert«, dann erfassen wir die Bedeutung intuitiv. Die QBF (11.1) stellt fest, dass für alle x (also $x = 0$ oder $x = 1$) entweder ein y existiert, sodass sowohl x als auch y wahr sind, oder für alle z $\neg x + z$ wahr ist. Diese Behauptung ist wahr. Um dies zu überprüfen, beachten Sie, dass wir bei $x = 1$ $y = 1$ wählen können und somit xy wahr wird. Ist $x = 0$, dann ist $\neg x + z$ für alle Werte von z wahr.

3. Wird der gleiche Variablenname zweimal in verschiedenen Funktionen verwendet, dann können wir die Variable in einem Fall jederzeit umbenennen, entweder in einem Programm oder in quantifizierten Booleschen Formeln. Bei Programmen spricht nichts dagegen, die erneute Verwendung des gleichen lokalen Namens zu vermeiden, doch in QBFs ist es bequemer, Vermeidung von Wiederverwendungen vorauszusetzen.

Befindet sich eine Variable x im Gültigkeitsbereich einer Quantifizierung von x, dann ist die Verwendung von x *gebunden*. Im anderen Fall ist ein Auftreten von x *frei*.

Beispiel 11.8 Jedes Auftreten einer Variablen in der QBF in Gleichung (11.1) ist gebunden, da sie sich innerhalb des Gültigkeitsbereichs der Quantifizierung dieser Variablen befindet. Beispielsweise entspricht der Gültigkeitsbereich der Variablen y, die in $(\exists y)(xy)$ quantifiziert ist, dem Ausdruck xy. Daher ist das Auftreten von y in diesem Ausdruck gebunden. Die Verwendung von x in xy ist an die Quantifizierung $(\forall x)$ gebunden, deren Gültigkeitsbereich den gesamten Ausdruck umfasst. ■

Der Wert einer QBF, die keine freien Variablen enthält, ist entweder 1 oder 0 (also wahr bzw. falsch). Wir können den Wert einer QBF durch eine Induktion über die Länge n des Ausdrucks berechnen.

Induktionsbeginn: Ist der Ausdruck von der Länge 1, dann kann er nur aus der Konstanten 0 oder 1 bestehen, da jede Variable frei sein würde. Der Wert dieses Ausdrucks ist der Ausdruck selbst.

Induktion: Angenommen, wir haben einen Ausdruck der Länge > 1 ohne freie Variablen und wir können den kürzeren Ausdruck berechnen, solange dieser Ausdruck keine freien Variablen enthält. Eine solche QBF kann eine von sechs Formen besitzen:

1. Der Ausdruck ist von der Form (E). Dann besitzt E die Länge $n - 2$ und kann als 0 oder 1 ausgewertet werden. Der Wert von (E) ist der gleiche.

1. Der Ausdruck ist von der Form $\neg E$. Dann besitzt E die Länge $n - 1$ und kann ausgewertet werden. Ist $E = 1$, dann ist $\neg E = 0$ und umgekehrt.

2. Der Ausdruck ist von der Form EF. Dann sind sowohl E als auch F kürzer als n und können ausgewertet werden. Der Wert von EF ist 1, wenn sowohl E als auch F den Wert 1 besitzen, und er ist 0, wenn E oder F 0 ist.

3. Der Ausdruck ist von der Form $E + F$. Dann sind sowohl E als auch F kürzer als n und können ausgewertet werden. Der Wert von $E + F$ ist 1, wenn E oder F den Wert 1 besitzt, und er ist 0, wenn beide 0 sind.

4. Ist der Ausdruck von der Form $(\forall x)(E)$, dann ersetzen wir zuerst alle Vorkommen von x in E durch 0 und erhalten den Ausdruck E_0. Dann ersetzen wir alle Vorkommen von x in E durch 1 und erhalten den Ausdruck E_1. Beachten Sie, dass sowohl für E_0 als auch für E_1 Folgendes gilt:

   a) Sie besitzen keine freien Variablen, da es sich bei keinem Auftreten einer freien Variablen in E_0 oder E_1 um x handeln könnte; es müsste deswegen eine Variable sein, die auch in E frei wäre.

   b) Sie besitzen die Länge $n - 6$ und sind somit kürzer als n.

   Wir werten E_0 und E_1 aus. Besitzen beide den Wert 1, dann besitzt auch $(\forall x)(E)$ den Wert 1, im anderen Fall dagegen den Wert 0. Beachten Sie, dass diese Regel die Interpretation »für alle x« von $(\forall x)$ widerspiegelt.

5. Lautet der gegebene Ausdruck $(\exists x)(E)$, dann verfahren wir wie in (5), konstruieren E_0 und E_1 und werten die beiden Ausdrücke aus. Wenn E_0 oder E_1 den Wert 1 besitzt, dann besitzt auch $(\exists x)(E)$ den Wert 1, im anderen Fall dagegen

den Wert 0. Beachten Sie, dass diese Regel die Interpretation »Es existiert ein x« von $(\exists x)$ widerspiegelt.

Beispiel 11.9 Wir werten die QBF in Gleichung (11.1) aus. Sie ist von der Form $(\forall x)(E)$, und wir müssen daher zuerst E_0 auswerten:

$$(\exists y)(0y) + (\forall z)(\neg 0 + z) \tag{11.2}$$

Der Wert dieses Ausdrucks hängt von den Werten der beiden durch ODER verbundenen Ausdrücke ab: $(\exists y)(0y)$ und $(\forall z)(\neg 0 + z)$. E_0 hat den Wert 1, wenn einer der Teilausdrücke den Wert 1 besitzt. Um $(\exists y)(0y)$ zu berechnen, müssen wir $y = 0$ und $y = 1$ im Teilausdruck $(0y)$ substituieren und prüfen, ob zumindest einer der beiden den Wert 1 besitzt. Allerdings haben sowohl $0 \wedge 0$ als auch $0 \wedge 1$ und damit auch $(\exists y)(0y)$ den Wert 0.[4]

Der Ausdruck $(\forall z)(\neg 0 + z)$ jedoch besitzt den Wert 1, wie nach der Substitution von $z = 0$ und $z = 1$ ersichtlich ist. Da $\neg 0 = 1$ ist, müssen wir die beiden Ausdrücke $1 \vee 0$ und $1 \vee 1$ auswerten. Da beide den Wert 1 besitzen, wissen wir, dass auch $(\forall z)(\neg 0 + z)$ diesen Wert besitzt. Wir folgern, dass E_0, also die Gleichung (11.2), den Wert 1 besitzt.

Nun prüfen wir, ob E_1, das wir durch die Substitution $x = 1$ in Gleichung (11.1) erhalten, ebenfalls den Wert 1 besitzt:

$$(\exists y)(1y) + (\forall z)(\neg 1 + z) \tag{11.3}$$

Wie wir durch die Substitution $y = 1$ sehen, besitzt der Ausdruck $(\exists y)(1y)$ den Wert 1. Damit besitzt auch E_1, also die Gleichung (11.3), den Wert 1. Wir folgern, dass der gesamte Ausdruck, also Gleichung (11.1), den Wert 1 besitzt. ■

11.3.4 PS-Vollständigkeit des QBF-Problems

Wir können nun das *Problem quantifizierter Boolescher Formeln* definieren: Gegeben sei eine quantifizierte Boolesche Formel (QBF), die keine freien Variablen enthält; besitzt sie den Wert 1? Wir nennen dieses Problem QBF, verwenden den Begriff aber weiterhin auch als Abkürzung für »quantifizierte Boolesche Formel«. Der Kontext sollte Verwechslungen ausschließen.

Wir werden zeigen, dass das QBF-Problem in $\mathcal{PS}$ vollständig ist. Der Beweis verwendet Ideen aus den Sätzen 10.9 sowie 11.5. Aus Satz 10.9 stammt die Idee, eine Berechnung einer TM durch logische Variablen zu repräsentieren, die jeweils darüber informieren, ob eine bestimmte Zelle zu einem Zeitpunkt einen bestimmten Wert enthält. Steht allerdings wie in Satz 10.9 ein polynomialer Zeitaufwand im Mittelpunkt, dann sind nur Variablen einer polynomialen Anzahl von Belang. In diesem Fall könnten wir mit polynomialem Zeitaufwand einen Ausdruck konstruieren, der über die Akzeptanz der Eingabe durch die TM informiert. Steht jedoch die polynomiale Platzbegrenzung im Mittelpunkt, kann die Anzahl der Konfigurationen während der Berechung exponentiell in der Größe der Eingabe sein, und es ist daher nicht möglich, mit polynomialem Zeitaufwand einen Ausdruck zu erstellen, der über die korrekte Berechnung informiert. Wir haben jedoch eine leistungsfähigere Sprache auszudrü-

4. Beachten Sie die alternative Notation für UND und ODER, da wir die Juxtaposition sowie + nicht in Ausdrücken verwenden können, die 0 und 1 enthalten. Der Ausdruck würde sonst einer mehrstelligen Zahl oder einer arithmetischen Addition gleichen. Wir hoffen, die Leser akzeptieren beide Notationen als Repräsentation des gleichen logischen Operators.

cken, was wir sagen wollen, und mithilfe der Quantifizierung können wir eine QBF mit polynomialer Länge erstellen, die aussagt, dass die TM ihre Eingabe mit polynomialer Platzbegrenzung akzeptiert.

Aus Satz 11.5 verwenden wir die »rekursive Verdopplung« und drücken damit die Idee aus, dass eine Konfiguration nach einer großen Anzahl von Bewegungen zu einer anderen Konfiguration werden kann. Um auszudrücken, dass Konfiguration I nach m Bewegungen zu Konfiguration J wird, sagen wir, es existiert eine Konfiguration K, sodass I nach $m/2$ Bewegungen zu K und K nach weiteren $m/2$ Bewegungen zu J wird. Die Sprache quantifizierter Boolescher Formeln ermöglicht, dies in einem Ausdruck polynomialer Länge zusammenzufassen, auch wenn m exponentiell in der Länge der Eingabe ist.

Bevor wir mit dem Beweis beginnen, dass jede Sprache in $\mathcal{PS}$ mit polynomialem Zeitaufwand auf QBF reduzierbar ist, müssen wir zeigen, dass QBF in $\mathcal{PS}$ enthalten ist. Auch dieser Teil des Beweises der PS-Vollständigkeit bedarf einiger Überlegung und wird daher als separater Satz behandelt.

Satz 11.10 QBF ist in $\mathcal{PS}$ enthalten.

Beweis: In Abschnitt 11.3.3 wurde der rekursive Prozess zur Auswertung einer QBF F vorgestellt. Wir können diesen Algorithmus unter Verwendung eines Stacks implementieren, den wir wie im Beweis von Satz 11.5 auf dem Band einer Turing-Maschine speichern. Angenommen, F besitzt die Länge n. Dann erstellen wir einen Datensatz der Länge $O(n)$ für F, der F selbst sowie Platz für eine Information umfasst, mit welchem Teilausdruck von F wir gerade arbeiten. Zwei Beispiele aus den sechs möglichen Formen von F sollen den Auswertungsprozess erläutern.

1. Angenommen, $F = F_1 + F_2$. Dann führen wir Folgendes aus:
   a) Wir legen F_1 in einem eigenen Datensatz rechts vom Datensatz von F ab.
   b) F_1 wird rekursiv ausgewertet.
   c) Ist der Wert von F_1 1, wird für F der Wert 1 zurückgegeben.
   d) Ist der Wert von F_1 dagegen 0, dann ersetzen wir den Datensatz von F_1 durch einen Datensatz von F_2 und werten F_2 rekursiv aus.
   e) Als Wert von F wird der Wert von F_2 zurückgegeben.
2. Angenommen, $F = (\exists x)(E)$. Dann führen wir Folgendes aus:
   a) Wir erstellen den Ausdruck E_0, indem wir jedes Auftreten von x durch 0 ersetzen. E_0 wird in einem eigenen Datensatz rechts vom Datensatz von F abgelegt.
   b) E_0 wird rekursiv ausgewertet.
   c) Ist der Wert von E_0 1, wird für F der Wert 1 zurückgegeben.
   d) Ist der Wert von E_0 dagegen 0, dann erstellen wir E_1, indem wir x in E durch 1 ersetzen.
   e) Wir ersetzen den Datensatz von E_0 durch einen Datensatz von E_1 und werten E_1 rekursiv aus.
   f) Als Wert von F wird der Wert von E_1 zurückgegeben.

Ähnliche Schritte zur Auswertung von F in den vier weiteren möglichen Formen F_1F_2, $\neg E$, (E) und $(\forall x)(E)$ seien Ihnen überlassen. Der Anfangsfall liegt vor, wenn F eine Konstante ist. In diesem Fall wird die Konstante zurückgegeben, und es ist nicht erforderlich, weitere Datensätze auf dem Band zu erstellen.

In jedem Fall befindet sich rechts vom Datensatz für einen Ausdruck der Länge m ein weiterer Datensatz für einen Ausdruck, der kürzer als m ist. Beachten Sie, dass zwar oft zwei verschiedene Teilausdrücke auszuwerten sind, diese jedoch nacheinander ausgewertet werden. In Fall (1) existieren daher nie Datensätze für F_1 und einige seiner Teilausdrücke gleichzeitig mit Datensätzen für F_2 und einige seiner Teilausdrücke. Das Gleiche gilt für E_0 und E_1 in Schritt (2).

Beginnen wir daher mit einem Ausdruck der Länge n, dann befinden sich nie mehr als n Datensätze auf dem Stack. Außerdem besitzt jeder Datensatz die Länge $O(n)$. Das gesamte Band wird also nie länger als $O(n^2)$. Wir haben damit eine Konstruktion für eine TM mit polynomialer Platzbegrenzung, die QBF akzeptiert; ihre Platzbegrenzung ist quadratisch. Beachten Sie, dass dieser Algorithmus im Allgemeinen einen Zeitaufwand benötigt, der exponentiell in n und damit also nicht polynomial ist. ■

Wir wenden uns nun der Reduktion einer beliebigen Sprache L in $\mathcal{PS}$ auf das Problem QBF zu. Wir werden wie in Satz 10.9 aussagenlogische Variablen y_{ijA} verwenden, um zu beschreiben, dass sich A an Position j der Konfiguration i befindet. Da jedoch eine exponentielle Anzahl von Konfigurationen existieren kann, könnten wir nicht einmal eine Eingabe w der Länge n übernehmen und die Variablen in einer Zeit niederschreiben, die polynomial in n ist. Stattdessen nutzen wir die Verfügbarkeit von Quantifizierungen, um mit einer Variablenmenge viele verschiedene Konfigurationen zu repräsentieren. Der folgende Satz basiert auf dieser Idee.

Satz 11.11 Das Problem QBF ist PS-vollständig.

Beweis: Sei L in $\mathcal{PS}$ enthalten. L wird von einer deterministischen TM M akzeptiert, die bei einer Eingabe der Länge n höchstens einen Speicherplatz der Größe $p(n)$ verwendet. Wir wissen aus Satz 11.3, dass eine Konstante c existiert, sodass die TM M höchstens $c^{1+p(n)}$ Bewegungen ausführt, falls sie eine Eingabe der Länge n akzeptiert. Wir werden beschreiben, wie wir mit polynomialem Zeitaufwand eine Eingabe w der Länge n übernehmen und daraus eine QBF E konstruieren, die keine freien Variablen enthält und nur dann den Wert 1 besitzt, wenn w in $L(M)$ enthalten ist.

Beim Schreiben von E müssen wir eine polynomiale Anzahl von *Variablenkonfiguration*en einführen, die aus Mengen von Variablen y_{jA} bestehen, und beschreiben, dass sich A an Position j der repräsentierten Konfiguration befindet. j stammt aus dem Bereich von 0 bis $p(n)$. Das Symbol A ist entweder ein Bandsymbol oder ein Zustand von M. Die Anzahl der aussagenlogischen Variablen in einer Variablenkonfiguration ist also polynomial in n. Wir nehmen an, dass sich alle aussagenlogischen Variablen in den verschiedenen Variablenkonfigurationen unterscheiden; keine aussagenlogische Variable gehört also zu zwei verschiedenen Variablenkonfigurationen. Solange lediglich eine polynomiale Anzahl an Variablenkonfigurationen existiert, ist auch die Gesamtzahl der aussagenlogischen Variablen polynomial.

Es ist bequem, eine Notation $(\exists I)$ zu verwenden, wobei I eine Variablenkonfiguration ist. Diese Quantifizierung repräsentiert $(\exists x_1)$ $(\exists x_2)\ldots(\exists x_m)$, wobei $x_1, x_2, \ldots, x_m$ die aussagenlogischen Variablen der Variablenkonfiguration I sind. Entsprechend repräsentiert $(\forall I)$ den Allquantor, auf alle aussagenlogischen Variablen in I angewendet.

Wir konstruieren für w eine QBF der Form:

$$(\exists I_0)(\exists I_f)(S \wedge N \wedge F)$$

wobei Folgendes gilt:

1. I_0 und I_f sind Konfigurationsvariablen, die die anfängliche bzw. die akzeptierende Konfiguration repräsentieren.
2. S ist ein Ausdruck, der »startet richtig« aussagt; d. h. I_0 ist die anfängliche Konfiguration von M mit Eingabe w.
3. N ist ein Ausdruck, der »bewegt sich richtig« aussagt; M überführt I_0 in I_f.
4. F ist ein Ausdruck, der »endet richtig« aussagt; I_f ist eine akzeptierende Konfiguration.

Beachten Sie, dass zwar der gesamte Ausdruck keine freien Variablen enthält, die Variablen von I_0 jedoch in S, diejenigen von I_f in F und beide Variablengruppen in N frei vorkommen.

Startet richtig

S ist das logische UND von Literalen; jedes Literal ist eine der Variablen von I_0. S besitzt das Literal y_{jA}, wenn sich an der Position j der anfänglichen Konfiguration mit Eingabe w A befindet, und das Literal $\overline{y}_{jA}$ im anderen Fall. Ist also $w = a_1a_2\ldots a_n$, dann erscheinen y_{0q_0}, y_{1a_1}, …, y_{na_n} und alle y_{jB} für $j = n + 1, n + 2, \ldots, p(n)$ ohne Negation, und alle weiteren Variablen von I_0 sind negiert. Hier wird angenommen, dass q_0 der anfängliche Zustand von M und B das Leerzeichen ist.

Endet richtig

Wenn I_f eine akzeptierende Konfiguration ist, muss sie einen akzeptierenden Zustand besitzen. Daher schreiben wir F als die logische ODER-Verknüpfung jener aussagelogischen Variablen y_{jA} von I_f, deren A ein akzeptierender Zustand ist. Die Position j ist dabei unerheblich.

Bewegt sich richtig

Der Ausdruck N wird rekursiv auf eine Weise erstellt, bei der wir die Anzahl der Bewegungen verdoppeln können, ohne mehr als $O(p(n))$ Symbole zum konstruierten Ausdruck hinzuzufügen, und, wichtiger noch, bei der das Schreiben des Ausdrucks lediglich einen Zeitaufwand von $O(p(n))$ erfordert. Es ist nützlich, die logische UND-Verknüpfung von Ausdrücken, die die Gleichheit aller korrespondierenden Variablen von I und J feststellen, durch das Kürzel $I = J$ auszudrücken, wobei I und J Konfigurationsvariablen sind. Besteht also I aus den Variablen y_{jA} und J aus den Variablen z_{jA}, dann ist $I = J$ die Konjunktion der Ausdrücke $(y_{jA}z_{jA} + (\overline{y}_{jA})(\overline{z}_{jA}))$, wobei j aus dem Bereich von 0 bis $p(n)$ stammt und A ein Bandsymbol oder ein Zustand von M ist.

Wir konstruieren nun Ausdrücke $N_i(I, J)$ für $i = 1, 2, 4, 8, \ldots$ mit der Bedeutung, dass $I \overset{*}{\vdash} J$ in i oder weniger Schritten. In diesen Ausdrücken sind nur die aussagenlogischen Variablen der Variablenkonfigurationen I und J frei; alle anderen sind gebunden.

Die folgende Konstruktion von N_{2i} tut es nicht

Unsere erste Idee zur Konstruktion von N_{2i} aus N_i könnte darin bestehen, einen geradlinigen Teile-und-herrsche-Ansatz zu verwenden: Wenn $I \overset{*}{\vdash} J$ aus $2i$ oder weniger Bewegungen folgt, dann muss eine Konfiguration K existieren, sodass sowohl $I \overset{*}{\vdash} K$ als auch $K \overset{*}{\vdash} J$ in i oder weniger Schritten. Wenn wir allerdings die Formel niederschreiben, die diese Idee ausdrückt, etwa $N_{2i}(I, J) = (\exists K)(N_i(I, K) \wedge N_i(K, J))$, dann wird sich die Länge des Ausdrucks verdoppeln, wenn wir i verdoppeln. Da i exponentiell in n sein muss, um alle möglichen Berechnungen von M auszudrücken, würde das Niederschreiben von N zu viel Zeit beanspruchen, und N besäße eine exponentielle Länge.

Induktionsbeginn: Für $i = 1$ ergibt $N_i(I, J)$ entweder $I = J$ oder $I \vdash J$. Wir haben schon diskutiert, wie die Bedingung $I = J$ ausgedrückt werden kann. Für die Bedingung $I \vdash J$ verweisen wir auf den Abschnitt »Bewegt sich richtig« des Beweises von Satz 10.9, der sich mit exakt dem gleichen Problem beschäftigt, nämlich auszudrücken, dass eine Konfiguration aus der vorherigen Konfiguration folgt. Der Ausdruck N_1 ist die logische ODER-Verknüpfung dieser beiden Ausdrücke. Beachten Sie, dass wir N_1 mit einem Zeitaufwand $O(p(n))$ schreiben können.

Induktion: Wir konstruieren $N_{2i}(I, J)$ aus N_i. Der Kasten »Die folgende Konstruktion von N_{2i} tut es nicht« beschreibt, dass der direkte Ansatz, N_{2i} unter Verwendung von zwei Kopien von N_i zu erstellen, nicht die notwendigen Zeit- und Platzbegrenzungen liefert. N_{2i} wird stattdessen geschrieben, indem eine Kopie von N_i im Ausdruck verwendet wird, wobei beide Argumente (I, K) und (K, J) an den gleichen Ausdruck übergeben werden. $N_{2i}(I, J)$ verwendet also einen Teilausdruck $N_i(P, Q)$. Wir schreiben $N_{2i}(I, J)$, um auszudrücken, dass eine Konfiguration K existiert, sodass für alle Konfigurationen P und Q

1. $(P, Q) \neq (I, K)$ und $(P, Q) \neq (K, J)$ oder
2. $N_i(P, Q)$

wahr ist. Äquivalent formuliert heißt das, dass $N_i(I, K)$ und $N_i(K, J)$ wahr sind, und ob $N_i(P, Q)$ für andere (P, Q) wahr ist, ist nicht von Bedeutung. Es folgt eine QBF für $N_{2i}(I, J)$:

$$N_{2i}(I, J) = (\exists K)(\forall P)(\forall Q)(N_i(P, Q) \vee (\neg(I = P \wedge K = Q) \wedge \neg(K = P \wedge J = Q)))$$

Beachten Sie, dass wir N_{2i} mit einem Zeitaufwand schreiben können, der dem Schreiben von N_i zuzüglich $O(p(n))$ Zeichen entspricht.

Um die Konstruktion von N abzuschließen, müssen wir N_m für das kleinste m konstruieren, das eine Potenz von 2 und zudem mindestens $c^{1+p(n)}$ ist, die maximal mögliche Anzahl von Bewegungen, die TM M vor dem Akzeptieren der Eingabe w der Länge n ausführen kann. Der oben beschriebene induktive Schritt muss $\log_2(c^{1+p(n)})$ oder $O(p(n))$-mal angewendet werden. Da jeder Induktionsschritt einen Zeitaufwand von $O(p(n))$ erfordert, folgern wir, dass N mit Zeitaufwand $O(p^2(n))$ erstellt werden kann.

Abschluss des Beweises des Satzes 11.11

Wir haben nun gezeigt, wie eine Eingabe w mit einem in $|w|$ polynomialen Zeitaufwand in eine QBF

$$(\exists I_0)(\exists I_f)(S \wedge N \wedge F)$$

transformiert wird. Wir haben außerdem bewiesen, dass jeder der Ausdrücke S, N und F nur dann wahr ist, wenn deren freie Variablen die Konfigurationen I_0 und I_f repräsentieren, also die anfängliche bzw. akzeptierende Konfiguration einer Berechnung von M mit Eingabe w, und wenn $I_0 \vdash^* I_f$ gilt. Diese QBF besitzt also nur dann den Wert 1, wenn w von M akzeptiert wird. ■

11.3.5 Übungen zum Abschnitt 11.3

Übung 11.3.1 Schließen Sie den Beweis von Satz 11.10 ab, indem Sie folgende Fälle behandeln:

a) $F = F_1F_2$

b) $F = (\forall x)(E)$

c) $F = \neg(E)$

d) $F = (E)$

***!! Übung 11.3.2** Zeigen Sie, dass das folgende Problem PS-vollständig ist. Gegeben sei ein regulärer Ausdruck E; ist E zu Σ^* äquivalent, wobei Σ die Menge der Symbole ist, die in E enthalten sind? *Hinweis:* Statt zu versuchen, QBF auf dieses Problem zu reduzieren, könnte es einfacher sein, zu zeigen, dass jede in $\mathcal{PS}$ enthaltene Sprache darauf reduzierbar ist. Zeigen Sie für jede TM M mit polynomialer Platzbegrenzung, wie aus einer Eingabe w für M mit polynomialem Zeitaufwand ein regulärer Ausdruck konstruiert wird, der alle Zeichenreihen generiert, die keine Folgen von Konfigurationen von M sind, die zur Akzeptanz von w führen.

!! Übung 11.3.3 Das *Shannon Switching Game* wird wie folgt gespielt: Gegeben sei ein Graph G mit zwei Terminalknoten s und t. Es gibt zwei Spieler, die wir SHORT und CUT nennen. SHORT zieht zuerst , dann wählt jeder Spieler abwechselnd einen Knotenpunkt von G, außer s und t, der dem Spieler dann unwiderruflich gehört. SHORT gewinnt durch die Wahl einer Knotenmenge, die zusammen mit s und t einen Pfad in G von s nach t bildet. CUT gewinnt, wenn alle Knoten ausgewählt wurden und SHORT keinen Pfad von s nach t wählen konnte. Zeigen Sie, dass das folgende Problem PS-vollständig ist: Gegeben sei G. Kann SHORT unabhängig von den Zügen von CUT immer gewinnen?

11.4 Zufallsabhängige Sprachklassen

Wir wenden uns nun zwei Klassen von Sprachen zu, die von Turing-Maschinen mit der Fähigkeit, Zufallszahlen bei der Berechung zu verwenden, definiert werden. Sie kennen wahrscheinlich Algorithmen, die in einer der üblichen Programmiersprachen erstellt sind und einen Zufallszahlengenerator verwenden. Technisch gesehen führt die Funktion `rand()` oder eine ähnlich benannte Funktion, die eine »zufällig« erschei-

nende oder nicht vorhersagbare Zahl zurückgibt, einen spezifischen Algorithmus aus, der simuliert werden kann; es ist jedoch sehr schwierig, in der Folge produzierter Zahlen ein »Muster« zu erkennen. Ein einfaches Beispiel für eine solche Funktion (die nicht in der Praxis eingesetzt wird) wäre ein Prozess, der die vorherige ganze Zahl der Folge quadriert und die mittleren Bits des Produkts zurückgibt. Zahlen, die ein komplexer mechanischer Prozess wie dieser produziert, werden *Pseudozufallszahlen* genannt.

Wir werden in diesem Abschnitt einen Typ einer Turing-Maschine definieren, der das Generieren von Zufallszahlen sowie die Verwendung dieser Zahlen in Algorithmen modelliert. Dann definieren wir zwei Sprachklassen, $\mathcal{RP}$ und $\mathcal{ZPP}$, die diese Zufallszahlen in Verbindung mit polynomialer Zeitbegrenzung auf verschiedene Arten verwenden. Es ist interessant, dass diese Klassen nur wenig zu umfassen scheinen, das nicht in $\mathcal{P}$ enthalten ist, doch die Unterschiede sind wichtig. Sie werden insbesondere in Abschnitt 11.5 sehen, dass einige der wichtigsten Grundlagen der Computersicherheit tatsächlich Fragen über die Beziehungen dieser beiden Klassen zu $\mathcal{P}$ und $\mathcal{NP}$ sind.

11.4.1 Quicksort: Ein Beispiel für einen zufallsabhängigen Algorithmus

Sie kennen wahrscheinlich den Sortieralgorithmus »Quicksort«. Dieser Algorithmus führt im Wesentlichen Folgendes aus: Gegeben sei eine Liste von Elementen a_1, a_2, …, a_n, die zu sortieren sind. Wir wählen eines dieser Elemente, etwa a_1, und teilen die Liste in die Elemente auf, die kleiner oder gleich a_1, und solche, die größer als a_1 sind. Das gewählte Element wird *Pivotelement*[5] genannt. Achten wir sorgfältig auf die Repräsentation der Daten, dann können wir die Liste der Länge n mit Zeitaufwand $O(n)$ in zwei Listen teilen, die zusammen die Länge n besitzen. Außerdem können wir dann die Liste der niedrigeren Elemente (kleiner oder gleich als das Pivotelement) rekursiv sortieren und ebenso die Liste der höheren Elemente (größer als das Pivotelement), und zwar unabhängig voneinander. Das Ergebnis ist eine sortierte Liste aller n Elemente.

Haben wir Glück, dann ist das Pivotelement eine Zahl aus dem mittleren Bereich der sortierten Liste, und beide Teillisten besitzen etwa die Länge $n/2$. Haben wir in jedem rekursiven Stadium Glück, dann liegen nach etwa $\log_2 n$ Rekursionsebenen Listen der Länge 1 vor, die schon sortiert sind. Der gesamte Aufwand umfasst daher $O(\log n)$ Ebenen, die jeweils einen Aufwand von $O(n)$ erfordern, d. h. wir haben einen Gesamtaufwand von $O(n \log n)$.

Die Situation kann jedoch auch wesentlich unvorteilhafter sein. Falls die Liste beispielsweise schon sortiert ist, dann führt die Wahl des ersten Elements zu einer Aufteilung in die niedrigere Liste mit einem Element und die höhere Liste, die alle anderen Elemente umfasst. In diesem Fall verhält sich Quicksort ähnlich wie Selection-Sort, und es ist ein Zeitaufwand proportional zu n^2 notwendig, um n Elemente zu sortieren.

Gute Implementierungen von Quicksort wählen daher keine bestimmte Position der Liste als Pivotelement. Vielmehr wird dieses Element zufällig aus allen Elementen der Liste gewählt. Es existiert also eine Wahrscheinlichkeit von $1/n$ für jedes der n Elemente, dass es als Pivotelement gewählt wird. Wir werden dies zwar hier nicht zei-

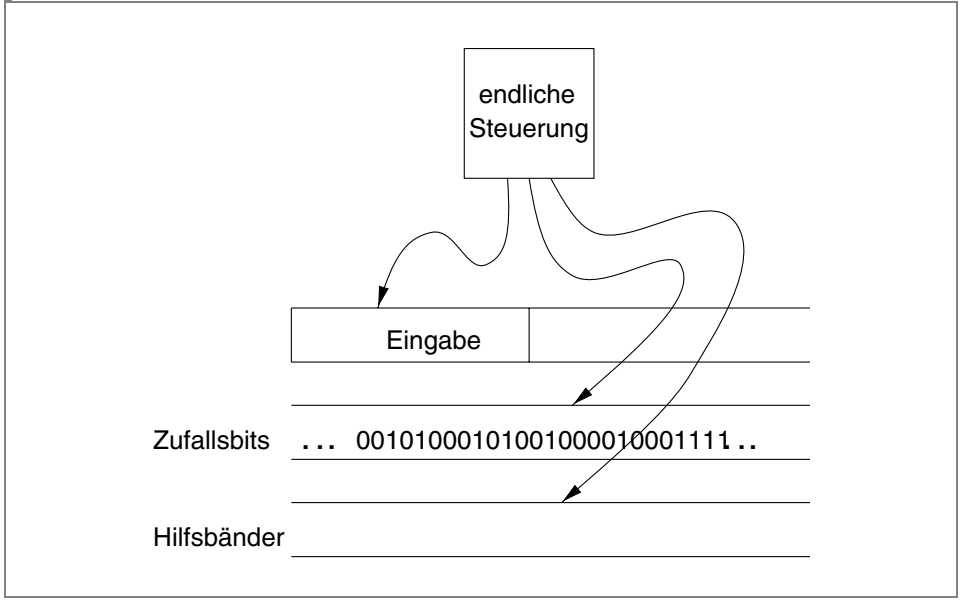
5. in der deutschen Literatur auch *Schnittzahl*

gen[6], doch die erwartete Ausführungszeit von Quicksort mit dieser Zufallsabhängigkeit ist $O(n \log n)$. Da allerdings noch immer eine geringe Chance besteht, dass jedes Mal das kleinste oder größte Element als Pivotelement gewählt wird, beträgt die Ausführungszeit von Quicksort im ungünstigsten Fall nach wie vor $O(n^2)$. Trotzdem ist Quicksort in vielen Anwendungen noch immer die bevorzugte Methode (beispielsweise im Unix-Sortierbefehl), da die erwartete Ausführungszeit verglichen mit anderen Ansätzen wirklich sehr gut ist, auch wenn andere Methoden selbst im ungünstigsten Fall lediglich einen Aufwand von $O(n \log n)$ benötigen.

11.4.2 Ein auf Zufallsabhängigkeit basierendes Modell einer Turing-Maschine

Wir verwenden die Variante einer mehrbändigen TM, um abstrakt die Fähigkeit einer Turing-Maschine zu zeigen, eine zufällige Auswahl zu treffen, entsprechend einem Programm, das einen Zufallszahlengenerator einmal oder mehrfach aufruft. Abbildung 11.5 zeigt diese TM-Variante. Das erste Band enthält die Eingabe, wie es bei einer mehrbändigen TM üblich ist. Das zweite Band beginnt ebenfalls ohne Leerzeichen in den Zellen. Tatsächlich ist das gesamte Band prinzipiell mit den Symbolen 0 und 1 beschrieben, die zufällig und unabhängig mit der Wahrscheinlichkeit von 1/2 für eine 0 bzw. eine 1 gewählt wurden. Wir nennen das zweite Band das *Zufallsband*. Das dritte und die folgenden Bänder sind anfänglich leer und werden von der TM nach Bedarf als »Hilfsbänder« eingesetzt. Wir nennen dieses TM-Modell eine zufallsabhängige *Turing-Maschine*.

Abbildung 11.5: Eine Turing-Maschine, die zufällig »generierte« Zahlen verwenden kann



6. Einen Beweis und eine Analyse der erwarteten Ausführungszeit von Quicksort finden Sie in D. E. Knuth [1973]. *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison Wesley.

Da es unrealistisch wäre, sich vorzustellen, die zufallsabhängige TM würde initialisiert, indem ein unendliches Band mit zufälligen Symbolen 0 und 1 beschrieben wird, kann diese TM äquivalent als eine TM mit einem anfänglich leeren zweiten Band angesehen werden. Liest der zweite Kopf allerdings ein Leerzeichen, dann wird intern »eine Münze geworfen«, und die zufallsabhängige TM schreibt entsprechend entweder eine 0 oder eine 1 in die betreffende Zelle, die danach nicht mehr geändert wird. Auf diese Weise ist keine Arbeit – und bestimmt keine, die unendlich lange andauern würde – vor dem Start der zufallsabhängigen TM erforderlich. Das zweite Band scheint auf diese Weise von vornherein mit den zufällig gewählten Symbolen 0 und 1 beschrieben zu sein, da diese Zufallsbits immer dann da sind, wenn der zweite Kopf der zufallsabhängigen TM eine Zelle liest.

Beispiel 11.12 Wir können die zufallsabhängige Version von Quicksort auf einer zufallsabhängigen TM implementieren. Der wichtigste Schritt ist der rekursive Prozess, eine Teilliste zu nehmen, von der wir annehmen, dass sie aufeinander folgend auf dem Eingabeband gespeichert und durch Markierungen an beiden Enden abgegrenzt ist, zufällig ein Pivotelement zu wählen und die Teilliste in eine niedrige und eine höhere Teilliste aufzuteilen. Die zufallsabhängige TM führt Folgendes aus:

1. Angenommen, die aufzuteilende Teilliste besitze die Länge m. Wir verwenden etwa $O(\log m)$ neue zufällige Bits des zweiten Bands, um eine Zufallszahl zwischen 1 und m zu wählen; das Element m der Teilliste wird zum Pivotelement. Beachten Sie, dass wir womöglich nicht jede ganze Zahl zwischen 1 und m mit gleicher Wahrscheinlichkeit wählen können, da m im Allgemeinen keine Potenz von 2 ist. Wenn wir allerdings etwa $[2 \log_2 m]$ Bits von Band 2 nehmen, als eine Zahl aus dem Bereich 0 bis m^2 betrachten und nach der Division durch m 1 zum Rest addieren, dann erhalten wir alle Zahlen zwischen 1 und m mit einer Wahrscheinlichkeit, die nahe genug an $1/m$ liegt, dass Quicksort korrekt mit der vorgesehenen Wahrscheinlichkeit arbeitet.
2. Das Pivotelement wird auf Band 3 abgelegt.
3. Wir durchsuchen die auf Band 1 gespeicherte Teilliste und kopieren alle Elemente, die nicht größer als das Pivotelement sind, auf Band 4.
4. Wir durchsuchen die auf Band 1 gespeicherte Teilliste erneut und kopieren alle Elemente, die größer als das Pivotelement sind, auf Band 5.
5. Band 4 und Band 5 werden in den Platz auf Band 1 kopiert, der vorher die ursprüngliche Teilliste enthielt. Eine Markierung trennt die beiden Listen.
6. Enthalten beide Listen oder eine der beiden Listen mehr als ein Element, dann werden sie rekursiv mit dem gleichen Algorithmus sortiert.

Beachten Sie, dass diese Implementierung von Quicksort einen Zeitaufwand von $O(n \log n)$ erfordert, auch wenn die Berechnung von einer mehrbändigen TM und nicht von einem konventionellen Computer ausgeführt wird. Allerdings steht in diesem Beispiel nicht die Ausführungszeit im Vordergrund, sondern die Verwendung der Zufallsbits auf dem zweiten Band, die ein zufälliges Verhalten der Turing-Maschine bewirken. ■

11.4.3 Die Sprache einer zufallsabhängigen Turing-Maschine

Wir sind an Situationen gewöhnt, in denen jede Turing-Maschine (oder FA oder PDA) eine Sprache akzeptiert, auch wenn diese Sprache die leere Menge oder die Menge aller Zeichenreihen über dem Eingabealphabet ist. Im Umgang mit zufallsabhängigen Turing-Maschinen müssen wir vorsichtiger damit umgehen, was für die TM das Akzeptieren einer Eingabe bedeutet, und es besteht sogar die Möglichkeit, dass eine zufallsabhängige TM überhaupt keine Sprache akzeptiert. Das Problem liegt darin, dass wir bei der Analyse der Arbeit einer zufallsabhängigen TM M als Reaktion auf eine Eingabe w betrachten müssen, wie sich M bei allen möglichen Inhalten des Zufallsbandes verhält. Es ist ganz und gar möglich, dass M bei manchen zufälligen Zeichenreihen akzeptiert, bei anderen dagegen nicht; soll die zufallsabhängige TM etwas effizienter als eine deterministische TM ausführen, dann ist es sogar wesentlich, dass verschiedene Inhalte des Zufallsbandes zu unterschiedlichem Verhalten führen[7].

Wir stellen uns eine zufallsabhängige TM so vor, dass sie wie eine konventionelle TM durch den Übergang in einen Endzustand akzeptiert, dann besitzt jede Eingabe w der zufallsabhängigen TM M eine Akzeptanzwahrscheinlichkeit, die dem Anteil der möglichen Inhalte des Zufallsbandes entspricht, die zur Akzeptanz führen. Da eine unendliche Anzahl möglicher Bandinhalte existiert, müssen wir bei der Berechnung dieser Wahrscheinlichkeit sorgfältig vorgehen. Allerdings liest jede Folge von Bewegungen, die zur Akzeptanz führen, nur einen endlichen Abschnitt des Zufallsbandes, und somit tritt alles, was gelesen wird, mit einer endlichen Wahrscheinlichkeit von 2^{-m} ein, wobei m der Anzahl der Zellen auf dem Zufallsband entspricht, die gelesen wurden und zumindest eine Bewegung der TM beeinflusst haben. Anhand eines Beispiels wird die Berechung für einen sehr einfachen Fall gezeigt.

Beispiel 11.13 Unsere zufallsabhängige TM M besitzt die in Tabelle 11.1 dargestellte Übergangsfunktion. M verwendet nur ein Eingabeband sowie das Zufallsband. Sie verhält sich sehr einfach, ändert auf keinem der beiden Bänder ein Symbol und bewegt die Köpfe nur nach rechts (Richtung R) oder belässt sie auf der aktuellen Position (Richtung S). Wir haben zwar keine formale Notation für die Übergänge einer zufallsabhängigen TM eingeführt, doch die Einträge in Tabelle 11.1 sollten verständlich sein; jede Zeile korrespondiert mit einem Zustand und jede Spalte mit einem Symbolpaar XY, wobei X das auf dem Eingabeband gelesene Symbol und Y das auf dem Zufallsband gelesene Symbol ist. Der Eintrag $qUVDE$ bedeutet, dass die TM in den Zustand q wechselt, U auf das Eingabeband und V auf das Zufallsband schreibt und den Eingabekopf in Richtung D sowie den Kopf des Zufallsbandes in Richtung E bewegt.

Die folgende Zusammenfassung zeigt, wie sich M bei einer Eingabezeichenreihe verhält, die aus den Symbolen 0 und 1 besteht. Im Startzustand q_0 liest M das erste Zufallsbit und führt abhängig davon, ob es sich um 0 oder 1 handelt, eine von zwei Prüfungen von w aus.

Ist das Zufallsbit 0, dann prüft M, ob w nur aus Nullen oder nur aus Einsen besteht. In diesem Fall liest M keine weiteren Zufallsbits, und der zweite Kopf bleibt stationär. Ist 0 das erste Bit von w, dann wechselt M in den Zustand q_1. In diesem Zustand

7. Sie sollten beachten, dass die in Beispiel 11.12 beschriebene zufallsabhängige TM nicht zur Spracherkennung dient. Stattdessen führt sie eine Transformation ihrer Eingabe aus, und die Ausführungszeit der Transformation, nicht jedoch das Ergebnis, hängt davon ab, was sich auf dem Zufallsband befindet.

Tabelle 11.1: Die Übergangsfunktion einer zufallsabhängigen Turing-Maschine

	00	01	10	11	$B0$	$B1$
$\rightarrow q_0$	$q_1 00RS$	$q_3 01SR$	$q_2 10RS$	$q_3 11SR$		
q_1	$q_1 00RS$				$q_4 B0SS$	
q_2			$q_2 10RS$		$q_4 B0SS$	
q_3	$q_3 00RR$			$q_3 11RR$	$q_4 B0SS$	$q_4 B1SS$
$*q_4$						

bewegt sich M nach rechts über die Symbole 0 hinweg, gelangt jedoch in einen undefinierten Zustand, d. h. ohne zu akzeptieren, falls eine 1 gelesen wird. Erreicht M im Zustand q_1 das erste Leerzeichen auf dem Eingabeband, dann wechselt sie in den Zustand q_4, den Akzeptanzzustand. Ähnlich wird verfahren, wenn 1 das erste Bit von w und 0 das erste Zufallsbit ist, dann wechselt M in den Zustand q_2. In diesem Zustand prüft M, ob alle weiteren Bits von w 1 sind, und akzeptiert nur in diesem Fall.

Sehen wir uns nun an, wie sich M verhält, wenn das erste Zufallsbit 1 ist. M vergleicht w mit dem zweiten und allen folgenden Zufallsbits und akzeptiert nur dann, wenn sie identisch sind. Daher wechselt M, wenn sie im Zustand q_0 eine 1 auf dem zweiten Band liest, in Zustand q_3. Beachten Sie, dass M dabei den Kopf des zweiten Bandes nach rechts bewegt, um ein neues Zufallsbit zu lesen, während der Kopf des Eingabebands auf seiner Position verbleibt, sodass das gesamte w mit dem zweiten und allen folgenden Zufallsbits verglichen wird. In Zustand q_3 vergleicht M beide Bänder und bewegt beide Bandköpfe nach rechts. Existiert an einer Stelle keine Übereinstimmung, dann gelangt M in einen undefinierten Zustand, ohne zu akzeptieren, während beim Erreichen des ersten Leerzeichens auf dem Eingabeband akzeptiert wird.

Wir berechnen nun die Wahrscheinlichkeit der Akzeptanz verschiedener Eingaben. Zuerst betrachten wir eine homogene Eingabe, z. B. 0^i für $i \geq 1$. Das erste Zufallsbit ist 0 mit Wahrscheinlichkeit 1/2, und in diesem Fall ist die Homogenitätsprüfung erfolgreich und 0^i wird akzeptiert. Allerdings ist das erste Zufallsbit 1 mit der gleichen Wahrscheinlichkeit von 1/2. In diesem Fall wird 0^i nur dann akzeptiert, wenn die Zufallsbits 2 bis $i + 1$ alle 0 sind. Dieser Fall tritt mit Wahrscheinlichkeit 2^{-i} auf. Die gesamte Akzeptanzwahrscheinlichkeit von 0^i ist daher

$$\frac{1}{2} + \frac{1}{2}2^{-i} = \frac{1}{2} + 2^{-(i+1)}$$

Betrachten wir nun den Fall einer heterogenen Eingabe w, also einer Eingabe, die etwa wie 00101 sowohl Nullen als auch Einsen enthält. Diese Eingabe wird nie akzeptiert, falls das erste Zufallsbit 0 ist. Ist das erste Zufallsbit dagegen 1, dann beträgt die Wahrscheinlichkeit der Akzeptanz 2^{-i}, wobei i die Länge der Eingabe ist. Die gesamte Akzeptanzwahrscheinlichkeit einer heterogenen Eingabe der Länge i beträgt also $2^{-(i+1)}$. Beispielsweise hat 00101 eine Akzeptanzwahrscheinlichkeit von 1/64. ■

Wir folgern, dass wir eine Wahrscheinlichkeit der Akzeptanz einer gegebenen Zeichenreihe durch eine gegebene zufallsabhängige TM berechnen können. Ob die Zeichenreihe in der Sprache enthalten ist oder nicht, hängt davon ab, wie das »Enthaltensein« in der Sprache einer zufallsabhängigen TM definiert wird. In den nächsten Abschnitten stellen wir zwei verschiedene Definitionen der Akzeptanz vor, die zu verschiedenen Sprachklassen führen.

11.4.4 Die Klasse $\mathcal{RP}$

Unsere erste Sprachklasse wird $\mathcal{RP}$ genannt (Random Polynomial). Um in $\mathcal{RP}$ enthalten zu sein, muss eine Sprache L von einer zufallsabhängigen TM M im folgenden Sinn akzeptiert werden:

1. Ist w nicht in L enthalten, dann akzeptiert M w mit der Wahrscheinlichkeit 0.
2. Ist w in L enthalten, dann akzeptiert M w mit einer Wahrscheinlichkeit von mindestens $1/2$.
3. Es existiert ein Polynom $T(n)$, sodass bei einer Eingabe w der Länge n alle Läufe von M unabhängig vom Inhalt des Zufallsbandes nach höchstens $T(n)$ Schritten anhalten.

Beachten Sie, dass die Definition von $\mathcal{RP}$ zwei unabhängige Punkte anspricht. (1) und (2) definieren eine zufallsabhängige Turing-Maschine eines speziellen Typs, gelegentlich *Monte-Carlo*-Algorithmus genannt. Unabhängig von der Ausführungszeit können wir sagen, dass eine zufallsabhängige TM vom Typ »Monte Carlo« ist, wenn sie entweder mit der Wahrscheinlichkeit 0 oder mit einer Wahrscheinlichkeit von mindestens $1/2$ akzeptiert, wobei nichts im Bereich dazwischen liegt. (3) adressiert einfach die Ausführungszeit, die davon unabhängig ist, ob es sich um eine TM vom Typ »Monte Carlo« handelt oder nicht.

Nichtdeterminismus und Zufälligkeit

Es bestehen einige oberflächliche Ähnlichkeiten zwischen einer zufallsabhängigen TM und einer nichtdeterministischen TM. Wir könnten uns vorstellen, dass die nichtdeterministische Wahl einer NTM von einem Band mit Zufallsbits gesteuert wird; wann immer die NTM aus mehreren Bewegungen wählen muss, konsultiert sie das Zufallsband und wählt eine der Möglichkeiten mit gleicher Wahrscheinlichkeit. Wenn wir eine NTM allerdings auf diese Weise interpretieren, dann unterscheidet sich die Akzeptanzregel stark von der Regel für $\mathcal{RP}$. Eine Eingabe würde zurückgewiesen, wenn deren Akzeptanzwahrscheinlichkeit 0 ist. Dagegen würde jede Eingabe akzeptiert, falls deren Akzeptanzwahrscheinlichkeit ein beliebiger Wert größer als 0 ist, unabhängig davon, wie klein er tatsächlich ist.

Beispiel 11.14 Wir betrachten die zufallsabhängige TM aus Beispiel 11.13. Sie erfüllt Bedingung (3), da ihre Ausführungszeit unabhängig vom Inhalt des Zufallsbandes $O(n)$ beträgt. Allerdings akzeptiert sie überhaupt keine Sprache in dem Sinn, wie es die Definition von $\mathcal{RP}$ verlangt. Dies liegt daran, dass zwar homogene Eingaben wie etwa 000 mit einer Wahrscheinlichkeit von mindestens $1/2$ akzeptiert werden und daher (2) erfüllen, andere Eingaben wie 001 dagegen mit einer Wahrscheinlich-

keit akzeptiert werden, die weder 0 noch mindestens 1/2 beträgt; beispielsweise wird 001 mit der Wahrscheinlichkeit 1/16 akzeptiert. ■

Beispiel 11.15 Wir beschreiben informell eine zufallsabhängige TM, die sowohl vom Typ »Monte Carlo« ist als auch einen polynomialen Ausführungsaufwand benötigt und daher eine Sprache in $\mathcal{RP}$ akzeptiert. Die Eingabe wird als Graph interpretiert, und die Frage lautet, ob der Graph ein Dreieck enthält, also drei Knoten, die paarweise durch Kanten miteinander verbunden sind. Eingaben mit Dreieck sind in der Sprache enthalten, andere dagegen nicht.

Der Monte-Carlo-Algorithmus wird wiederholt zufällig eine Kante (x, y) sowie einen Knoten z wählen, der weder gleich x noch gleich y ist. Jede Wahl wird durch einige neue Zufallsbits auf dem Zufallsband festgelegt. Die TM prüft für jedes gewählte x, y und z, ob die Eingabe die Kanten (x, z) und (y, z) enthält. Ist dies der Fall, dann enthält der Eingabegraph ein Dreieck.

Insgesamt werden k Kombinationen einer Kante und eines Knotens gewählt; die TM akzeptiert, wenn sich eine Kombination als Dreieck erweist, und gibt im anderen Fall auf, ohne zu akzeptieren. Enthält der Graph kein Dreieck, dann kann sich keine der k Kombinationen als Dreieck erweisen, und damit ist Bedingung (1) der Definition von $\mathcal{RP}$ erfüllt: Ist die Eingabe nicht in der Sprache enthalten, dann beträgt die Wahrscheinlichkeit der Akzeptanz 0.

Angenommen, der Graph besitze n Knoten und e Kanten. Enthält der Graph zumindest ein Dreieck, dann beträgt die Wahrscheinlichkeit, dass dessen drei Knoten in einem Experiment gewählt werden, $\left(\frac{3}{e}\right)\left(\frac{1}{n-2}\right)$. Drei der e Kanten bilden also ein Dreieck, und wenn eine dieser drei Kanten gewählt wird, dann beträgt die Wahrscheinlichkeit $1/(n-2)$, dass auch der dritte Knoten ausgewählt wird. Diese Wahrscheinlichkeit ist gering, doch das Experiment wird k-mal wiederholt. Die Wahrscheinlichkeit, dass wenigstens eines der k Experimente zum Dreieck führt, beträgt

$$1-\left(1-\frac{3}{e(n-2)}\right)^k \tag{11.4}$$

Eine allgemein verwendete Approximation sagt aus, dass $(1 - x)^k$ für kleine x annähernd e^{-kx} ist, wobei $e = 2{,}718\ldots$ die Basis des natürlichen Logarithmus ist. Wählen wir daher k, sodass $kx = 1$, dann ist e^{-kx} deutlich kleiner und $1 - e^{-kx}$ deutlich größer als 1/2, nämlich etwa 0,63, um es genauer zu sagen. Wir können daher $k = e(n - 2)/3$ wählen und damit sicherstellen, dass die Akzeptanzwahrscheinlichkeit eines Graphen mit einem Dreieck, wie sie Gleichung 11.4 beschreibt, mindestens 1/2 beträgt. Der beschriebene Algorithmus ist also vom Typ »Monte Carlo«.

Nun müssen wir uns die Ausführungszeit der TM ansehen. Sowohl e als auch n sind nicht größer als die Eingabelänge, und k ist nicht größer als das Quadrat der Eingabelänge, da es proportional zum Produkt von e und n ist. Jedes Experiment ist linear zur Eingabelänge, da es die Eingabe höchstens viermal durchsucht (um eine Kante und einen Knoten zufällig zu wählen und dann das Vorhandensein zweier weiterer Kanten zu prüfen). Die TM hält also nach einem Zeitraum an, der höchstens kubisch in der Eingabelänge ist; die TM besitzt also eine polynomiale Ausführungszeit und erfüllt damit die dritte und letzte Bedingung dafür, dass ihre Sprache in $\mathcal{RP}$ enthalten ist.

Wir folgern, dass die Sprache der Graphen, die ein Dreieck enthalten, in der Klasse $\mathcal{RP}$ enthalten ist. Beachten Sie, dass diese Sprache auch in $\mathcal{P}$ enthalten ist, da eine sys-

tematische Suche nach allen möglichen Dreiecken durchgeführt werden könnte. Wie jedoch schon zu Beginn von Abschnitt 11.4 angemerkt, ist es sehr schwer, Beispiele zu finden, die anscheinend in $\mathcal{RP} - \mathcal{P}$ enthalten sind. ■

11.4.5 In $\mathcal{RP}$ enthaltene Sprachen erkennen

Angenommen, wir haben eine polynomiale Turing-Maschine M vom Typ »Monte Carlo«, die eine Sprache L erkennt. Gegeben sei eine Zeichenreihe w, von der wir wissen wollen, ob sie in L enthalten ist. Führen wir M mit w aus und werfen eine Münze oder nutzen eine andere Möglichkeit der Generierung von Zufallszahlen, um die Erzeugung von Zufallsbits zu simulieren, dann wissen wir Folgendes:

1. Ist w nicht in L enthalten, dann wird unser Lauf mit Sicherheit nicht zur Akzeptanz von w führen.
2. Ist w in L enthalten, dann existiert eine Wahrscheinlichkeit von wenigstens 50% für die Akzeptanz von w.

Sehen wir das Ergebnis dieses Laufes jedoch einfach als endgültig an, dann wird w gelegentlich zurückgewiesen, wenn es eigentlich akzeptiert werden sollte (ein *falsches negatives* Ergebnis), doch wird es nie akzeptiert, wenn eine Akzeptanz auch nicht angebracht wäre (also kein *falsches positives* Ergebnis). Wir müssen also zwischen der zufallsabhängigen TM selbst und dem verwendeten Algorithmus unterscheiden, der entscheidet, ob w in L enthalten ist. Falsche negative Ergebnisse können wir niemals vermeiden, jedoch durch viele Wiederholungen der Prüfung die Wahrscheinlichkeit eines solchen Ergebnisses auf einen beliebig kleinen Wert reduzieren.

Wünschen wir beispielsweise, dass die Wahrscheinlichkeit eines falschen negativen Ergebnisses eins zu eine Milliarde beträgt, dann könnten wir die Prüfung dreißigmal ausführen. Ist w in L enthalten, dann ist die Wahrscheinlichkeit, dass alle dreißig Prüfungen nicht zur Akzeptanz führen, nicht größer als 2^{-30}, d. h. kleiner als 10^{-9} oder eins zu eine Milliarde. Soll die Wahrscheinlichkeit eines falschen negativen Ergebnisses allgemein geringer als $c > 0$ sein, dann müssen wir die Prüfung $\log_2(1/c)$-mal ausführen. Da dies ein konstanter Wert ist, wenn auch c eine Konstante ist, und da ein Lauf der zufallsabhängigen TM M einen polynomialen Zeitaufwand erfordert, wenn L in $\mathcal{RP}$ enthalten ist, benötigt auch die wiederholte Prüfung einen polynomialen Zeitaufwand. Das Ergebnis dieser Betrachtungen fasst der folgende Satz zusammen.

Satz 11.16 Ist L in $\mathcal{RP}$ enthalten, dann gibt es für jede Konstante $c > 0$, unabhängig davon, wie klein sie ist, einen polinomialen zufallsabhängigen Algorithmus, der eine Entscheidung trifft, ob die gegebene Eingabe w in L enthalten ist, keine falschen positiven Fehler macht und falsche negative Fehler mit einer Wahrscheinlichkeit macht, die nicht größer als c ist. ■

11.4.6 Die Klasse $\mathcal{ZPP}$

Die zweite, Zufallsabhängigkeit involvierende Sprachklasse wird $\mathcal{ZPP}$ (*Zero-Error, Probabilistic, Polynomial*) genannt. Diese Klasse basiert auf einer zufallsabhängigen TM, die immer anhält und für die Zeit bis zum Anhalten einen Erwartungswert hat, der ein Polynom in der Länge der Eingabe ist. Diese TM akzeptiert ihre Eingabe, indem sie in einen akzeptierenden Zustand übergeht (und damit zu diesem Zeitpunkt anhält), und sie weist die Eingabe zurück, indem sie anhält, ohne zu akzeptieren. Die

Definition der Klasse $\mathcal{ZPP}$ stimmt also fast mit der von $\mathcal{P}$ überein, außer dass $\mathcal{ZPP}$ eine Zufallsabhängigkeit im Verhalten der TM erlaubt und außerdem der Erwartungswert der Ausführungszeit statt der Ausführungszeit im schlechtesten Fall gemessen wird.

Eine TM, die immer die korrekte Antwort gibt, deren Ausführungszeit jedoch von den Werten einiger Zufallsbits abhängt, wird gelegentlich Turing-Maschine vom Typ »Las Vegas« oder Las-Vegas-Algorithmus genannt. Wir können uns $\mathcal{ZPP}$ daher als die Sprachen vorstellen, die von Las-Vegas-Turing-Maschinen mit einem polynomialen Erwartungswert für die Ausführungszeit akzeptiert werden.

Ist die Zahl 1/2 in der Definition von $\mathcal{RP}$ etwas Besonderes?

Bei der Definition von $\mathcal{RP}$ haben wir die erforderliche Wahrscheinlichkeit für die Akzeptanz einer Zeichenreihe w in L als mindestens 1/2 festgelegt. Allerdings könnten wir $\mathcal{RP}$ mit jeder Konstanten definieren, die zwischen 0 und 1 liegt. Satz 11.16 sagt aus, dass die Wahrscheinlichkeit so hoch festgelegt werden kann, wie wir wünschen (bis zu jedem Wert kleiner 1), wenn wir das von M ausgeführte Experiment nur oft genug wiederholen. Außerdem ermöglicht uns die in Abschnitt 11.4.5 verwendete Technik der Verringerung der Wahrscheinlichkeit des Nichtakzeptierens einer Zeichenreihe in L, eine zufallsabhängige TM mit einer beliebigen Akzeptanzwahrscheinlichkeit größer als 0 zu verwenden und diese Wahrscheinlichkeit bis auf 1/2 zu steigern, indem wir das Experiment wiederholen. Die Anzahl der Wiederholungen ist hierbei eine Konstante.

Wir werden in der Definition von $\mathcal{RP}$ weiterhin 1/2 als Akzeptanzwahrscheinlichkeit verwenden, doch wir sollten uns dessen bewusst sein, dass jede Wahrscheinlichkeit ungleich 0 für die Definition der Klasse $\mathcal{RP}$ ausreichend wäre. Auf der anderen Seite würde eine andere Konstante als 1/2 die von einer bestimmten zufallsabhängigen TM definierte Sprache ändern. Wir haben etwa in Beispiel 11.14 gesehen, wie die Verringerung der erforderlichen Wahrscheinlichkeit auf 1/16 dazu führen würde, dass die Zeichenreihe 001 in der Sprache der dort diskutierten zufallsabhängigen TM enthalten wäre.

11.4.7 Eine Beziehung zwischen $\mathcal{RP}$ und $\mathcal{ZPP}$

Zwischen den beiden definierten zufallsabhängigen Klassen existiert eine einfache Beziehung. Um diese Beziehung in einem Satz auszudrücken, müssen wir uns zunächst die Komplemente dieser Klassen ansehen. Es sollte klar sein, dass mit L auch $\overline{L}$ in $\mathcal{ZPP}$ enthalten ist. Wird nämlich L von einer Las-Vegas-TM M mit polynomialem Erwartungswert für den Zeitaufwand akzeptiert, dann wird $\overline{L}$ von einer modifizierten Las-Vegas-TM M' akzeptiert, bei der die Akzeptanz in M durch Anhalten ohne zu akzeptieren, und Anhalten ohne Akzeptanz in M durch Akzeptanz ersetzt wird.

Es ist allerdings nicht offensichtlich, dass $\mathcal{RP}$ unter Komplementbildung abgeschlossen ist, da die Definition von Monte-Carlo-Turing-Maschinen die Akzeptanz und das Zurückweisen asymmetrisch behandelt. Wir definieren daher die Klasse Co-$\mathcal{RP}$ als die Menge der Sprachen L, sodass $\overline{L}$ in $\mathcal{RP}$ enthalten ist; Co-$\mathcal{RP}$ enthält also die Komplemente der Sprachen in $\mathcal{RP}$.

Satz 11.17 $\mathcal{ZPP} = \mathcal{RP} \cap \text{Co-}\mathcal{RP}$.

Beweis: Zuerst zeigen wir $\mathcal{RP} \cap \text{Co-}\mathcal{RP} \subseteq \mathcal{ZPP}$. Angenommen, L ist in $\mathcal{RP} \cap \text{Co-}\mathcal{RP}$ enthalten. Dann besitzen sowohl L als auch $\overline{L}$ Monte-Carlo-TMs mit polynomialen Ausführungszeiten. $p(n)$ sei ein genügend großes Polynom, um die Ausführungszeiten beider Maschinen zu begrenzen. Wir entwerfen für L eine Las-Vegas-TM M wie folgt:

1. Wir lassen die Monte-Carlo-TM für L laufen; akzeptiert sie, dann akzeptiert M und hält an.
2. Andernfalls lassen wir die Monte-Carlo-TM für $\overline{L}$ laufen. Wenn diese TM akzeptiert, dann hält M an, ohne zu akzeptieren. Andernfalls fährt M mit Schritt (1) fort.

Es ist klar, dass M eine Eingabe w nur dann akzeptiert, wenn w in L enthalten ist, und sie nur dann zurückweist, wenn w nicht in L enthalten ist. Der Erwartungswert der Ausführungszeit einer Runde (eine Ausführung der Schritte (1) und (2)) beträgt $2p(n)$. Außerdem beträgt die Wahrscheinlichkeit, dass eine Runde zu einem Ergebnis führt, mindestens $1/2$. Ist w in L enthalten, dann hat Schritt (1) eine Chance von 50%, zur Akzeptanz durch M zu führen. Ist w dagegen nicht in L enthalten, dann führt Schritt (2) mit einer Chance von 50% zu einer Zurückweisung durch M. Der Erwartungswert der Ausführungszeit von M ist also nicht größer als

$$2p(n) + \frac{1}{2}2p(n) + \frac{1}{4}2p(n) + \frac{1}{8}2p(n) + \ldots = 4p(n)$$

Sehen wir uns nun die Umkehrung an. Wir nehmen an, L sei in $\mathcal{ZPP}$ enthalten, und zeigen, dass L sowohl in $\mathcal{RP}$ als auch in Co-$\mathcal{RP}$ enthalten ist. Wir wissen, dass L von einer Las-Vegas-TM M_1 akzeptiert wird, deren Erwartungswert der Ausführungszeit ein Polynom $p(n)$ ist. Wir konstruieren wie folgt eine Monte-Carlo-TM M_2 für L. M_2 simuliert M_1 für $2p(n)$ Schritte. Akzeptiert M_1 in dieser Zeit, dann akzeptiert auch M_2; im anderen Fall weist M_2 zurück.

Angenommen, eine Eingabe w der Länge n sei nicht in L enthalten. Dann wird w von M_1 und damit auch von M_2 nicht akzeptiert. Nehmen wir nun an, w sei in L enthalten. M_1 akzeptiert w schließlich mit Sicherheit, unter Umständen jedoch nicht innerhalb von $2p(n)$ Schritten.

Allerdings behaupten wir, dass die Wahrscheinlichkeit des Akzeptierens von w durch M_1 innerhalb von $2p(n)$ Schritten mindestens $1/2$ beträgt. Angenommen, die Wahrscheinlichkeit der Akzeptanz von w durch M_1 innerhalb des Zeitraums $2p(n)$ sei eine Konstante $c < 1/2$. Dann beträgt der Erwartungswert der Ausführungszeit von M_1 bei Eingabe w mindestens $(1 - c)2p(n)$, da $1 - c$ die Wahrscheinlichkeit ist, mit der M_1 einen *längeren* Zeitraum als $2p(n)$ benötigt. Ist $c < 1/2$, dann ist $2(1 - c) > 1$, und der Erwartungswert der Ausführungszeit von M_1 mit Eingabe w ist größer als $p(n)$. Wir haben damit einen Widerspruch zur Annahme, der Erwartungswert der Ausführungszeit von M_1 betrage höchstens $p(n)$, und folgern daraus, dass M_2 mit einer Wahrscheinlichkeit von mindestens $1/2$ akzeptiert. M_2 ist also eine Monte-Carlo-TM mit polynomialer Zeitbegrenzung, und d. h. L ist in $\mathcal{RP}$ enthalten.

Wir verwenden im Wesentlichen die gleiche Konstruktion, um zu beweisen, dass L auch in Co-$\mathcal{RP}$ enthalten ist, doch nehmen wir dazu das Komplement $\overline{M}_2$ von M_2. Das heißt, um $\overline{L}$ zu akzeptieren, muss $\overline{M}_2$ akzeptieren, wenn M_1 innerhalb des Zeitraums $2p(n)$ zurückweist; anderenfalls muss $\overline{M}_2$ zurückweisen. $\overline{M}_2$ ist also eine polynomiale Monte-Carlo-TM für $\overline{L}$. ■

11.4.8 Beziehungen zu den Klassen $\mathcal{P}$ und $\mathcal{NP}$

Satz 11.17 sagt aus, dass $\mathcal{ZPP} \subseteq \mathcal{RP}$. Mit den folgenden einfachen Sätzen ordnen wir diese Klassen zwischen $\mathcal{P}$ und $\mathcal{NP}$ an.

Satz 11.18 $\mathcal{P} \subseteq \mathcal{ZPP}$.

Beweis: Jede polynomiale deterministische TM ist ebenso eine polynomiale Las-Vegas-TM, die lediglich ihre Fähigkeit nicht ausnutzt, eine zufällige Auswahl zu treffen. ■

Satz 11.19 $\mathcal{RP} \subseteq \mathcal{NP}$.

Beweis: Angenommen, wir haben eine polynomiale Monte-Carlo-TM M_1 für eine Sprache L. Wir können eine nichtdeterministische TM M_2 für L mit der gleichen Zeitbegrenzung konstruieren. Wann immer M_1 ein Zufallsbit zum ersten Mal untersucht, wählt M_2 nichtdeterministisch aus beiden möglichen Werten dieses Bits und schreibt es auf ein eigenes Band, welches das Zufallsband von M_1 simuliert. M_2 akzeptiert immer dann, wenn M_1 akzeptiert, im anderen Fall dagegen nicht.

Angenommen, w ist in L enthalten. Da M_1 eine Wahrscheinlichkeit von mindestens 50% besitzt, w zu akzeptieren, muss es eine Bitfolge auf dem Zufallsband geben, die zur Akzeptanz von w führt. M_2 wird diese Bitfolge u. a. wählen und akzeptiert daher bei dieser Auswahl ebenfalls. Daher ist w in $L(M_2)$ enthalten. Ist w allerdings nicht in L enthalten, dann existiert keine Folge von Zufallsbits, die zur Akzeptanz von M_1 führt, und damit existiert auch keine Folge von Wahlmöglichkeiten, die dazu führt, dass M_2 akzeptiert. w ist also nicht in $L(M_2)$ enthalten. ■

Abb. 11.6 zeigt die Beziehungen zwischen den beschriebenen und den anderen »nahe gelegenen« Klassen.

11.5 Die Komplexität des Primzahltests

Dieser Abschnitt stellt ein besonderes Problem vor, die Prüfung, ob es sich bei einer ganzen Zahl um eine Primzahl handelt. Wir stellen zunächst die Rolle von Primzahlen und Primzahltests in Sicherheitssystemen für Computer vor. Dann zeigen wir, dass die Primzahlen sowohl in $\mathcal{NP}$ als auch in Co-$\mathcal{NP}$ enthalten sind. Zuletzt lernen Sie einen zufallsabhängigen Algorithmus kennen, der zeigt, dass die Primzahlen auch in $\mathcal{RP}$ enthalten sind.

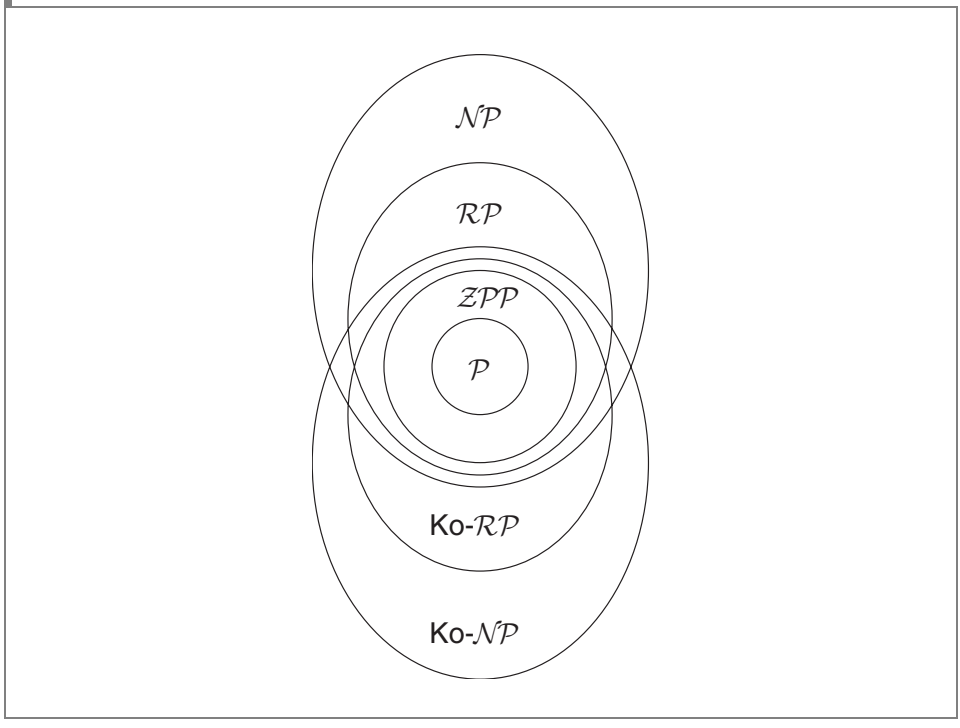
11.5.1 Die Bedeutung des Primzahltests

Eine ganze Zahl p ist eine *Primzahl*, wenn sie größer als 1 und nur durch 1 und sich selbst teilbar ist. Ist eine ganze Zahl keine Primzahl, dann wird sie *zusammengesetzt* genannt. Jede zusammengesetzte Zahl kann bis auf die Reihenfolge der Faktoren als Produkt von Primzahlen eindeutig dargestellt werden.

Beispiel 11.20 Die ersten Primzahlen lauten 2, 3, 5, 7, 11, 13 und 17. Die ganze Zahl 504 ist zusammengesetzt, und ihre Primfaktorzerlegung lautet $2^3 \times 3^2 \times 7$. ■

Es gibt einige Techniken, die die Computersicherheit verbessern, wobei die heute verbreiteten Methoden auf der Annahme beruhen, dass die Zerlegung von Zahlen, also

Abbildung 11.6: Die Beziehungen zwischen $\mathcal{ZPP}$ sowie $\mathcal{RP}$ und anderen Klassen



die Suche nach den Primfaktoren einer gegebenen zusammengesetzten Zahl, schwierig ist. Insbesondere Schemata, die auf den so genannten RSA-Codes (nach den Erfindern der Technik, R. Rivest, A. Shamir und L. Adelman benannt) basieren, verwenden ganze Zahlen von beispielsweise 128 Bit, die aus dem Produkt zweier Primzahlen bestehen, die jeweils etwa 64 Bit lang sind. Es folgen zwei Szenarien, in denen Primzahlen eine wichtige Rolle spielen.

Kryptographie mit öffentlichen Schlüsseln

Sie möchten bei einem Online-Buchhändler ein Buch kaufen. Der Buchhändler fragt Sie nach Ihrer Kreditkartennummer, aber es ist zu riskant, die Nummer einfach in ein Formular einzugeben und über Telefonleitungen oder das Internet zu übermitteln. Jemand könnte Ihre Leitung überwachen oder Datenpakete während der Übertragung über das Internet abfangen.

Damit keine unberechtigte Person Ihre Kreditkartennummer lesen kann, sendet der Buchhändler einen Schlüssel k an Ihren Browser, vielleicht das 128-Bit-Produkt zweier Primzahlen, die der Buchhandelscomputer zu diesem Zweck erstellt hat. Ihr Browser verwendet eine Funktion $y = f_k(x)$, die sowohl den Schlüssel k als auch die Daten x übernimmt, die Sie verschlüsseln müssen. Die Funktion f, die Teil des RSA-Schemas ist, kann allgemein bekannt sein, auch potenziellen Lauschern, doch es wird angenommen, dass ohne Kenntnis der Faktorisierung von k die inverse Funktion f_k^{-1}, sodass $x = f_k^{-1}(y)$ ist, nicht in einem Zeitraum berechnet werden kann, der kürzer als exponentiell zur Länge von k ist.

Auch wenn eine unberechtigte Person y liest und die Arbeitsweise von f kennt, kann sie x, also Ihre Kreditkartennummer, nicht wiederherstellen, ohne k zu kennen und in Faktoren zu zerlegen. Auf der anderen Seite kann der Buchhändler, der die Faktorisierung des Schlüssels k kennt, weil er ihn selbst erzeugt hat, einfach f_k^{-1} anwenden und x aus y errechnen.

Signaturen mit öffentlichen Schlüsseln

RSA-Codes wurden ursprünglich für das folgende Szenario entwickelt. Sie können E-Mail »signieren«, damit andere Personen auf einfache Weise feststellen können, dass die E-Mail von Ihnen stammt, es aber nicht möglich ist, dass jemand anders eine E-Mail mit Ihrem Namen unterzeichnet. Beispielsweise möchten Sie vielleicht die Nachricht x = »Ich verspreche, Sally Lee 10 € zu zahlen« unterzeichnen, aber Sally daran hindern, die signierte Nachricht selbst zu erstellen, und auch sonst niemandem zu ermöglichen, eine solche unterzeichnete Nachricht ohne Ihr Wissen zu erstellen.

Dazu wählen Sie einen Schlüssel k, dessen Primfaktoren nur Ihnen bekannt sind. Sie veröffentlichen k beispielsweise auf Ihrer Website, sodass jeder die Funktion f_k auf Nachrichten anwenden kann. Wollen Sie die obige Nachricht x signieren und an Sally senden, dann berechnen Sie $y = f_k^{-1}(x)$ und senden y an Sally. Sally erhält f_k, Ihren *öffentlichen Schlüssel* , von Ihrer Website und berechnet damit $x = f_k(y)$. Sie weiß also, Sie haben tatsächlich versprochen, die 10 € zu zahlen.

Wenn Sie bestreiten, die Nachricht y gesendet zu haben, kann Sally vor einem Gericht argumentieren, dass nur Sie allein die Funktion f_k^{-1} kennen und es ihr oder anderen »unmöglich« sei, diese Funktion anzuwenden. Daher kommen nur Sie als Urheber der Nachricht y infrage. Dieses System basiert auf der wahrscheinlichen, jedoch nicht bewiesenen Annahme, dass es zu schwierig ist, Zahlen in Faktoren zu zerlegen, die das Produkt zweier großer Primzahlen sind.

Anforderungen an die Komplexität von Primzahltests

Von beiden beschriebenen Szenarien wird angenommen, dass sie sicher sind in dem Sinne, dass die Zerlegung des Produkts zweier großer Primzahlen wirklich einen exponentiellen Zeitaufwand erfordert. Die hier und in Kapitel 10 beschriebene Komplexitätstheorie wirkt sich zweifach auf das Studium der Sicherheit und der Kryptographie aus:

1. Die Konstruktion öffentlicher Schlüssel erfordert, dass große Primzahlen schnell zu finden sind. Es ist eine grundlegende Tatsache der Zahlentheorie, dass die Wahrscheinlichkeit, dass es sich bei einer n-Bit-Zahl um eine Primzahl handelt, etwa $1/n$ beträgt. Hätten wir daher eine Möglichkeit, mit polynomialem Zeitaufwand (polynomial in n, nicht in der Primzahl selbst) zu testen, ob eine n-Bit-Zahl prim ist, dann könnten wir zufällig Zahlen wählen, diese prüfen und aufhören, wenn wir auf eine Primzahl treffen. Damit hätten wir einen Las-Vegas-Algorithmus mit polynomialem Zeitaufwand für die Suche nach Primzahlen, da die erwartete Anzahl zu testender Zahlen etwa n beträgt, bevor eine Primzahl mit n Bit gefunden wird. Suchen wir etwa nach 64-Bit-Primzahlen, dann müssten wir durchschnittlich 64 ganze Zahlen prüfen, im ungünstigsten Fall jedoch wesentlich mehr. Leider gibt es anscheinend keinen garantiert erfolgreichen Primzahltest mit polynomialem Zeitaufwand, doch wir werden in Abschnitt 11.5.4 sehen, dass es einen Monte-Carlo-Algorithmus mit polynomialem Zeitaufwand gibt.

2. Die Sicherheit RSA-basierter Kryptographie hängt davon ab, dass im Allgemeinen keine polynomiale (in der Anzahl der Bits des Schlüssels) Möglichkeit der Faktorisierung existiert, insbesondere keine Möglichkeit, eine als Produkt von exakt zwei großen Primzahlen bekannte Zahl zu zerlegen. Wir wären glücklich, wenn wir zeigen könnten, dass die Menge der Primzahlen eine NP-vollständige Sprache oder die Menge der zusammengesetzten Zahlen NP-vollständig ist. Dann würde ein polynomialer Zerlegungsalgorithmus beweisen, dass $\mathcal{P} = \mathcal{NP}$ gilt, da er Tests mit polynomialem Zeitaufwand für beide Sprachen bieten würde. Leider wird in Abschnitt 11.5.5 gezeigt, dass sowohl die Primzahlen als auch die zusammengesetzten Zahlen in $\mathcal{NP}$ enthalten sind. Da sie jeweils das Komplement der anderen sind, würde im Fall der NP-Vollständigkeit einer der Mengen $\mathcal{NP}$ = Co-$\mathcal{NP}$ folgen, und das bezweifeln wir. Außerdem würde die Tatsache, dass die Menge der Primzahlen in $\mathcal{RP}$ enthalten ist, bedeuten, dass im Fall ihrer NP-Vollständigkeit auf $\mathcal{RP} = \mathcal{NP}$ geschlossen werden könnte, doch auch dies ist sehr unwahrscheinlich.

11.5.2 Einführung in Modulararithmetik

Bevor wir uns Algorithmen zur Erkennung der Menge der Primzahlen ansehen, werden wir einige grundlegende Konzepte der *Modulararithmetik* vorstellen, also der üblichen Arithmetik modulo einer ganzen Zahl, oft einer Primzahl. Sei p eine ganze Zahl. Die *ganzen Zahlen modulo p* lauten $0, 1, \ldots, p - 1$.

Wir können definieren, dass Addition und Multiplikation modulo p lediglich auf diese Menge von p ganzen Zahlen anzuwenden sind, indem wir die übliche Berechnung ausführen, durch p dividieren und den Rest als Ergebnis nehmen. Die Addition ist geradlinig, da die Summe entweder niedriger als p ist, wobei in diesem Fall nichts weiter auszuführen ist, oder zwischen p und $2p - 2$ liegt, wobei wir hier p subtrahieren müssen, um eine ganze Zahl im Bereich $0, 1, \ldots, p - 1$ zu erhalten. Die modulare Addition gehorcht den üblichen algebraischen Gesetzen; sie ist kommutativ, assoziativ und besitzt 0 als Identität. Die Subtraktion ist noch immer invers zur Addition; wir berechnen die modulare Differenz $x - y$ mit einer gewöhnlichen Subtraktion und addieren dann p, falls das Ergebnis kleiner als 0 ist. Die Negation von x, also $-x$, entspricht wie in der üblichen Arithmetik $0 - x$. Daher gilt $-0 = 0$, und falls $x \neq 0$, dann ist $-x = -x + p$.

Beispiel 11.21 Angenommen, $p = 13$. Dann ist $3 + 5 = 8$ und $7 + 10 = 4$. Zur zweiten Addition ist anzumerken, dass $7 + 10$ mit der üblichen Arithmetik zu dem Ergebnis 17 führt, das nicht kleiner als 13 ist. Daher subtrahieren wir 13 und erhalten das korrekte Ergebnis 4. Der Wert von -5 modulo 13 ist $-5 + 13$ gleich 8. Die Differenz $11 - 4$ modulo 13 ist 7, die Differenz $4 - 11$ dagegen 6. Die letzte Berechnung führt mit der üblichen Arithmetik zu -7, und daher addieren wir 13 und erhalten 6. ■

Die Multiplikation modulo p wird wie eine Multiplikation gewöhnlicher Zahlen durchgeführt, und dann ist der Rest der Division durch p das Ergebnis. Auch die Multiplikation gehorcht den üblichen algebraischen Gesetzen; sie ist kommutativ und assoziativ, 1 ist die Identität, 0 der Annulator, und die Multiplikation ist distributiv über der Addition. Allerdings ist die Division modulo p durch Werte ungleich 0 schwieriger. Im Allgemeinen gilt Folgendes: Ist x eine der ganzen Zahlen modulo p, also $0 \leq x < p$, dann entspricht x^{-1} oder $1/x$ der Zahl y, falls sie existiert, sodass $xy = 1$ modulo p ist.

Beispiel 11.22 Tabelle 11.2 zeigt die Multiplikationstabelle für ganze Zahlen ungleich 0 modulo 7 einer Primzahl. Der Eintrag in Zeile i und Spalte j ist das Produkt ij modulo 7. Beachten Sie, dass jede der ganzen Zahlen ungleich 0 eine inverse Zahl besitzt; 2 und 4 sind jeweils invers zueinander, ebenso 3 und 4, während 1 und 6 zu sich selbst invers sind. Daher sind 2×4, 3×5, 1×1 und 6×6 jeweils 1. Wir können daher x durch jede Zahl y ungleich 0 dividieren, indem wir y^{-1} berechnen und dann x mit y^{-1} multiplizieren. Ein Beispiel: $3/4 = 3 \times 4^{-1} = 3 \times 2 = 6$.

Tabelle 11.2: Multiplikation modulo 7

1	2	3	4	5	6
2	4	6	1	3	5
3	6	2	5	1	4
4	1	5	2	6	3
5	3	1	6	4	2
6	5	4	3	2	1

Vergleichen wir diese Situation mit der in Tabelle 11.3 dargestellten Tabelle der Multiplikation modulo 6. Wir beobachten zunächst, dass *nur* 1 und 5 eine inverse Zahl besitzen; sie sind jeweils zu sich selbst invers. Andere Zahlen dagegen haben keine inversen Zahlen. Zusätzlich gibt es Zahlen, die zwar ungleich 0 sind, deren Produkt jedoch 0 ist, etwa 2 und 3. Diese Situation tritt in der üblichen Ganzzahlarithmetik nie ein, ebenso wenig bei der Arithmetik modulo einer Primzahl. ■

Tabelle 11.3: Multiplikation modulo 6

1	2	3	4	5
2	4	0	2	4
3	0	3	0	3
4	2	0	4	2
5	4	3	2	1

Es gibt einen weiteren Unterschied zwischen der Multiplikation modulo einer Primzahl und modulo einer zusammengesetzten Zahl, der für Primzahlprüfungen sehr wichtig ist. Der *Grad* einer Zahl a modulo p ist der Exponent der kleinsten positiven Potenz von a, die gleich 1 modulo p ist. Es folgen einige nützliche Tatsachen zu diesem Thema, die wir hier nicht beweisen werden:

- Ist p eine Primzahl, dann gilt $a^{p-1} = 1$ modulo p für jede nicht durch p teilbare ganze Zahl a. Diese Aussage wird *kleiner Fermatscher Satz*[8] genannt.
- Der Grad von a modulo einer Primzahl p ist immer ein Teiler von $p - 1$.
- Ist p eine Primzahl, dann existiert immer ein a mit dem Grad $p - 1$ modulo p.

Beispiel 11.23 Sehen wir uns erneut die Tabelle der Multiplikation modulo 7 in Tabelle 11.3 an. Der Grad von 2 ist 3, da $2^2 = 4$ und $2^3 = 1$. Der Grad von 3 ist 6, da $3^2 = 2$, $3^3 = 6$, $3^4 = 4$, $3^5 = 5$ und $3^6 = 1$. Durch ähnliche Berechungen erhalten wir für 4 den Grad 3, für 5 den Grad 6, für 6 den Grad 2 und für 1 den Grad 1. ■

11.5.3 Die Komplexität modulararithmetischer Berechnungen

Bevor wir mit den Anwendungen der Modulararithmetik für Primzahltests fortfahren, müssen wir einige grundlegende Tatsachen über die Ausführungszeit der wichtigen Operationen vorstellen. Angenommen, wir wollen eine Berechung modulo einer Primzahl p ausführen und die Binärrepräsentation von p besitzt die Länge n Bit; p selbst ist also etwa 2^n. Wie immer wird die Ausführungszeit einer Berechung in Bezug auf die Länge n der Eingabe ausgedrückt statt in Bezug zu p, dem »Wert« der Eingabe. Beispielsweise erfordert das Hochzählen bis p einen Zeitaufwand von $O(2^n)$, und damit ist jede Berechnung, die p Schritte erfordert, als Funktion von n *nicht* polynomial.

Allerdings können wir zwei Zahlen modulo p mit einem Zeitaufwand von $O(n)$ mit einem typischen Computer oder einer mehrbändigen TM addieren. Sie wissen, dass wir einfach die Binärzahlen addieren und p vom Ergebnis subtrahieren, falls es größer oder gleich p ist. Entsprechend können wir zwei Zahlen mit dem Zeitaufwand $O(n^2)$ mit einem Computer oder mit einer Turing-Maschine multiplizieren. Mit der üblichen Multiplikation erhalten wir ein Ergebnis, das höchstens $2n$ Bit umfasst; dann dividieren wir durch p und nehmen den Rest.

Das Potenzieren einer Zahl x ist schwieriger, da der Exponent selbst exponentiell in n sein kann. Wie wir sehen werden, ist die Erhebung von x in die Potenz $p - 1$ ein wichtiger Schritt. Da $p - 1$ etwa 2^n entspricht, benötigten wir $O(2^n)$ Multiplikationen, um x $(p - 2)$-mal mit sich selbst zu multiplizieren. Selbst da nun jede Multiplikation nur n-Bit-Zahlen involviert und mit einem Zeitaufwand von $O(n^2)$ ausgeführt werden kann, würde der gesamte Zeitaufwand $O(n^2 2^n)$ entsprechen, und dieser Wert ist nicht polynomial in n.

Glücklicherweise gibt es den Trick der »rekursiven Verdopplung«, der die Berechnung von x^{p-1} (oder jeder anderen Potenz von x bis zu x^p) mit einem in n polynomialen Zeitaufwand ermöglicht:

1. Wir berechnen höchstens n Potenzen x, x^2, x^4, x^8, …, bis der Exponent p-1 überschreitet. Jeder Wert ist eine n-Bit-Zahl, die mit einem Zeitaufwand von $O(n^2)$ durch das Quadrieren des vorherigen Wertes in der Folge berechnet wird. Der gesamte Aufwand beträgt also $O(n^3)$.
2. Wir berechnen die Binärrepräsentation von $p - 1$, etwa $p - 1 = a_{n-1}\ldots a_1 a_0$, d. h.

$$p - 1 = a_0 + 2a_1 + 4a_2 + \ldots + 2^{n-1}a_{n-1},$$

8. Verwechseln Sie den kleinen Fermatschen Satz nicht mit der »Fermatschen Vermutung«, die die Nichtexistenz von ganzzahligen Lösungen von $x^n + y^n = z^n$ für $n \geq 3$ feststellt.

wobei jedes a_j entweder 0 oder 1 ist. Wir erhalten so

$$x^{p-1} = x^{a_0+2a_1+4a_2+\ldots+2^{n-1}a_{n-1}}$$

das dem Produkt jener Werte x^{2^j} entspricht, für die $a_j = 1$ ist. Da wir jedes dieser x^{2^j} in Schritt (1) berechnet haben und es sich jeweils um eine n-Bit-Zahl handelt, können wir das Produkt dieser n bzw. weniger Zahlen mit einem Zeitaufwand von $O(n^3)$ berechnen.

Die gesamte Berechnung von x^{p-1} erfordert also den Zeitaufwand $O(n^3)$.

11.5.4 Zufallsabhängige polynomiale Primzahltests

Wir stellen nun die Verwendung zufallsabhängiger Berechnungen großer Primzahlen vor. Präziser, wir werden zeigen, dass die Sprache zusammengesetzter Zahlen in $\mathcal{RP}$ enthalten ist. Die zur Generierung von n-Bit-Primzahlen verwendete Methode arbeitet wie folgt: Es wird zufällig eine n-Bit-Zahl gewählt und der Monte-Carlo-Algorithmus zur Erkennung zusammengesetzter Zahlen mehrfach angewendet; die Anzahl sollte einer größeren Zahl wie etwa 50 entsprechen. Ist die Prüfung, ob es sich um eine zusammengesetzte Zahl handelt, erfolgreich, dann wissen wir, dass keine Primzahl vorliegt. Scheitern alle 50 Versuche, die Zahl als zusammengesetzt zu erkennen, dann ist die Wahrscheinlichkeit, dass es sich tatsächlich um eine zusammengesetzte Zahl handelt, nicht größer als 2^{-50}. Wir können daher mit ziemlicher Sicherheit sagen, dass die Zahl eine Primzahl ist, und unsere Sicherheitsoperation auf dieser Tatsache aufbauen.

Hier wird nicht der vollständige Algorithmus vorgestellt, sondern eher eine Idee, die bis auf sehr wenige Fälle angewendet werden kann. Wir wissen aus dem kleinen Fermatschen Satz, dass x^{p-1} modulo p immer 1 ist, wenn p eine Primzahl ist. Es ist ebenso eine Tatsache, dass für mindestens die Hälfte der Werte von x im Bereich 1 bis $p - 1$ $x^{p-1} \neq 1$ modulo p gilt, falls p eine zusammengesetzte Zahl ist und ein x existiert, für das x^{p-1} modulo p nicht 1 ist.

Wir verwenden daher den folgenden Monte-Carlo-Algorithmus für zusammengesetzte Zahlen:

1. Wir wählen eine Zufallszahl x aus dem Bereich 1 bis $p - 1$.
2. Wir berechnen x^{p-1} modulo p. Beachten Sie, dass diese Berechnung gemäß den Ausführungen am Ende des Abschnitts 11.5.3 einen Zeitaufwand $O(n^3)$ erfordert, falls p eine n-Bit-Zahl ist.
3. Ist $x^{p-1} \neq 1$ modulo p, dann wird akzeptiert; x ist zusammengesetzt. Im anderen Fall wird angehalten, ohne zu akzeptieren.

Ist p eine Primzahl, dann wird immer angehalten, ohne zu akzeptieren. Dies ist eine der Anforderungen an Monte-Carlo-Algorithmen; ist die Eingabe nicht in der Sprache enthalten, dann wird nie akzeptiert. Bei fast allen zusammengesetzten Zahlen gilt für mindestens die Hälfte der Werte von x, dass $x^{p-1} \neq 1$ modulo p ist, und somit besteht eine Chance von mindestens 50%, dass ein beliebiger Ablauf des Algorithmus akzeptiert; dies ist die zweite Anforderung an einen Monte-Carlo-Algorithmus.

Das bisher Beschriebene wäre eine Demonstration, dass die zusammengesetzten Zahlen in $\mathcal{RP}$ enthalten sind, wenn es nicht einige wenige zusammengesetzte Zahlen c gäbe, für die $x^{c-1} = 1$ modulo c für die Mehrzahl der x im Bereich 1 bis $c - 1$ gilt, ins-

besondere für die relativ zu c primen x. Diese Zahlen, *Carmichael-Zahlen* genannt, erfordern eine weitere komplexere Prüfung (die wir hier nicht vorstellen), um sie als zusammengesetzt zu identifizieren. Die kleinste Carmichael-Zahl ist 561. Man kann nämlich zeigen, dass für alle nicht durch 3, 11 oder 17 teilbaren Zahlen im Bereich 1 – 560 $x^{560} = 1$ modulo 561 gilt, obwohl 561 = 3*11*17 offensichtlich zusammengesetzt ist. Wir behaupten also, jedoch ohne vollständigen Beweis:

Satz 11.24 Die Menge der zusammengesetzten Zahlen ist in $\mathcal{RP}$ enthalten. ■

Ist eine Faktorzerlegung mit zufallsabhängigem polynomialen Zeitaufwand möglich?

Beachten Sie, dass uns der Algorithmus in Abschnitt 11.5.4 darüber informieren kann, dass eine Zahl zusammengesetzt ist, nicht jedoch darüber, wie diese zusammengesetzte Zahl in Faktoren zerlegt werden kann. Es wird angenommen, dass es keine Möglichkeit gibt, Zahlen polynomial in Faktoren zu zerlegen, auch wenn Zufallsabhängigkeit zugelassen wird, selbst wenn man sich dabei auf einen polynomialen Erwartungswert beschränkt. Wäre diese Annahme falsch, dann wären die in Abschnitt 11.5.1 diskutierten Anwendungen unsicher und nicht brauchbar.

11.5.5 Nichtdeterministische Primzahltests

Betrachten wir nun ein weiteres interessantes und wichtiges Ergebnis über Primzahltests: Die Sprache der Primzahlen ist in $\mathcal{NP} \cap$ Co-$\mathcal{NP}$ enthalten. Daher ist die Sprache der zusammengesetzten Zahlen, also des Komplements der Primzahlen, ebenfalls in $\mathcal{NP} \cap$ Co-$\mathcal{NP}$ enthalten. Diese Tatsache ist deswegen so wichtig, weil es sehr unwahrscheinlich ist, dass die Primzahlen oder die zusammengesetzten Zahlen NP-vollständig sind, denn wenn einer dieser Fälle gelten würde, hätten wir das unerwartete Ergebnis $\mathcal{NP}$ = Co-$\mathcal{NP}$.

Ein Teil ist einfach: Offensichtlich sind die zusammengesetzten Zahlen in $\mathcal{NP}$ und die Primzahlen in Co-$\mathcal{NP}$ enthalten. Wir beweisen zunächst die erste Tatsache.

Satz 11.25 Die Menge der zusammengesetzten Zahlen ist in $\mathcal{NP}$ enthalten.

Beweis: Der polynomiale nichtdeterministische Algorithmus für zusammengesetzte Zahlen arbeitet wie folgt:

1. Gegeben sei eine n-Bit-Zahl p; wir wählen einen Faktor f von höchstens n Bit. Allerdings wählen wir nicht $f = 1$ oder $f = p$. Dieser Teil ist nichtdeterministisch, wobei alle möglichen Werte von f in einer Folge von Auswahlmöglichkeiten auftreten. Jede dieser Folgen erfordert jedoch den Zeitaufwand $O(n)$.
2. Wir dividieren p durch f und prüfen, ob der Rest 0 ist. In diesem Fall wird akzeptiert. Dieser Teil ist deterministisch und kann auf einer mehrbändigen TM mit dem Zeitaufwand $O(n^2)$ ausgeführt werden.

Ist p zusammengesetzt, dann muss mindestens ein Faktor f neben 1 und p existieren. Die NTM wählt alle möglichen Zahlen bis zu n Bit und trifft in einem Zweig auf f. Dieser Zweig führt zur Akzeptanz. Umgekehrt impliziert die Akzeptanz durch die NTM, dass ein Faktor von p ungleich 1 oder p selbst gefunden wurde. Daher akzeptiert die

beschriebene NTM die Sprache, die genau aus den zusammengesetzten Zahlen besteht. ■

Die Identifikation der Primzahlen mit einer NTM ist schwieriger. Wir sind zwar in der Lage, einen Grund (einen Faktor) zu erraten, weswegen eine Zahl keine Primzahl sein kann, und können dies dann verifizieren, doch wie »erraten« wir einen Grund dafür, dass eine Zahl eine Primzahl *ist*? Der nichtdeterministische Algorithmus mit polynomialem Zeitaufwand basiert auf der Tatsache (die wir jedoch nicht bewiesen haben), dass eine Zahl x zwischen 1 und $p-1$ vom Grad $p-1$ existiert, falls p eine Primzahl ist. Wir haben beispielsweise in Beispiel 11.23 beobachtet, dass für die Primzahl $p = 7$ die beiden Zahlen 3 und 5 vom Grad 6 sind.

Eine Zahl x können wir unter Verwendung der nichtdeterministischen Fähigkeiten einer NTM leicht erraten, doch es ist nicht sofort ersichtlich, wie geprüft wird, ob dieses x den Grad $p-1$ besitzt. Würden wir die Definition von »Grad« direkt anwenden, dann müssten wir prüfen, ob keines der x^2, x^3, …, x^{p-2} gleich 1 modulo p ist. Dazu wären $p-3$ Multiplikationen auszuführen, die einen Zeitaufwand von mindestens 2^n erfordern, wenn p eine n-Bit-Zahl ist.
Eine bessere Strategie basiert auf der Tatsache, die wir feststellen, aber nicht beweisen: Der Grad von x modulo einer Primzahl p ist ein Teiler von $p-1$. Wenn wir die Primfaktoren von $p-1$ kennten[9], dann wäre die Prüfung ausreichend, dass $x^{(p-1)/q} \neq 1$ für jeden Primfaktor q von $p-1$ ist. Ist keine dieser Potenzen von x gleich 1, dann muss x den Grad $p-1$ besitzen. Die Anzahl dieser Prüfungen beträgt $O(n)$, und damit sind sie alle mit einem polynomialen Algorithmus ausführbar. Natürlich können wir $p-1$ nicht einfach in Primzahlen zerlegen. Jedoch können wir die Primfaktoren von $p-1$ nichtdeterministisch raten und Folgendes ausführen:

1. Die Prüfung, ob deren Produkt tatsächlich $p-1$ ist.
2. Die Prüfung, ob es sich jeweils um eine Primzahl handelt. Dazu verwenden wir rekursiv den polynomialen nichtdeterministischen Algorithmus, den wir eben erstellt haben.

Details des Algorithmus sowie den Beweis, dass er nichtdeterministisch ist und polynomialen Zeitaufwand erfordert, zeigt der Beweis des folgenden Satzes.

Satz 11.26 Die Menge der Primzahlen ist in $\mathcal{NP}$ enthalten.

Beweis: Gegeben sei eine n-Bit-Zahl p. Zuerst prüfen wir, ob n nicht größer als 2 ist (p wäre daher 1, 2 oder 3); 2 und 3 sind Primzahlen, 1 dagegen nicht. Im anderen Fall führen wir Folgendes aus:

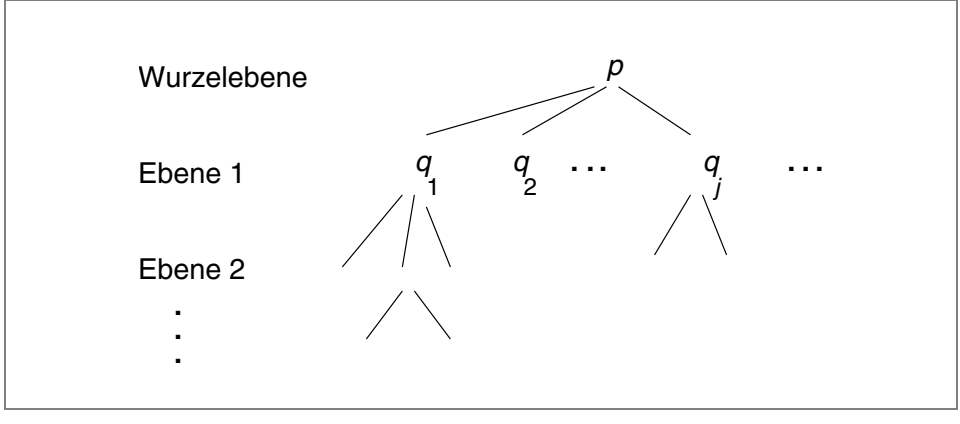
1. Wir raten eine Liste von Faktoren $(q_1, q_2, \ldots, q_k)$, deren Binärrepräsentationen insgesamt höchstens $2n$ Bit umfassen, jeder einzelne dagegen höchstens $n-1$ Bit. Die gleiche Primzahl darf mehrmals auftreten, da $p-1$ einen Faktor besitzen kann, der einer in eine Potenz größer als 1 erhobenen Primzahl entspricht; falls z. B. $p = 13$, dann enthält die Liste (2, 2, 3) die Primfaktoren von $p-1 = 12$. Dieser Teil ist nichtdeterministisch, doch jeder Zweig erfordert den Zeitaufwand $O(n)$.

9. Beachten Sie, dass $p-1$ nie eine Primzahl sein kann, wenn p eine Primzahl ist, ausgenommen der uninteressante Fall $p = 3$. Dies liegt darin begründet, dass alle Primzahlen außer 2 ungerade sind.

2. Wir multiplizieren die q miteinander und stellen fest, ob deren Produkt $p - 1$ ist. Dieser Teil ist deterministisch und erfordert einen Zeitaufwand von höchstens $O(n^2)$.
3. Ist das Produkt $p - 1$, dann stellen wir mit dem hier beschriebenen Algorithmus rekursiv fest, ob jedes q prim ist.
4. Sind alle q Primzahlen, dann raten wir einen Wert von x und prüfen, ob $x^{(p-1)/q_j} \neq 1$ für jedes der q_j gilt. Diese Prüfung stellt sicher, dass x vom Grad $p - 1$ modulo p ist, denn im anderen Fall wäre der Grad ein Teiler von mindestens einem $(p - 1)/q_j$, und dass dies nicht der Fall ist, haben wir gerade verifiziert. Beachten Sie zur Begründung, dass X^{g^h} für den Grad g von x und eine beliebige positive ganze Zahl h immer 1 sein muss. Die Potenzierungen können mit der in Abschnitt 11.5.3 beschriebenen effizienten Methode ausgeführt werden. Es gibt höchstens k Potenzierungen, also mit Sicherheit nicht mehr als n Potenzierungen, wobei jede mit dem Zeitaufwand $O(n^3)$ durchgeführt werden kann. Der gesamte Aufwand für diesen Schritt beträgt also $O(n^4)$.

Zuletzt müssen wir verifizieren, dass dieser nichtdeterministische Algorithmus nur einen polynomialen Zeitaufwand erfordert. Jeder der Schritte, mit Ausnahme des rekursiven Schrittes (3), benötigt entlang eines nichtdeterministischen Zweiges höchstens den Aufwand $O(n^4)$. Die Rekursion ist zwar kompliziert, doch wir können uns die rekursiven Aufrufe als einen Baum vorstellen, wie ihn Abbildung 11.7 zeigt. Die Wurzel bildet die zu verifizierende Primzahl p mit n Bit. Untergeordnet sind die q_j, also die geratenen Faktoren von $p - 1$, die wir ebenfalls als prim verifizieren müssen. Jedem q_j untergeordnet sind die geratenen Faktoren von $q_j - 1$, die wir wieder als prim verifizieren müssen, und so weiter, bis wir bei Zahlen mit höchstens 2 Bit angelangt sind, die den Blättern des Baumes entsprechen.

Abbildung 11.7: Die rekursiven Aufrufe des Algorithmus aus Satz 11.26 bilden einen Baum mit einer Höhe und Breite von höchstens n



Da das Produkt der untergeordneten Knoten eines jeden Knotens geringer als der Wert des Knotens selbst ist, ist das Produkt der Werte in den Knoten jeder Ebene höchstens p. Für einen Knoten mit dem Wert i ist, abgesehen von den rekursiven Aufrufen, ein Aufwand von höchstens $a(\log_2 i)^4$ für eine Konstante a erforderlich; dies

ergibt sich daraus, dass wir für diese Arbeit oben unter Punkt 4 festgestellt haben, dass sie proportional zur vierten Potenz der Anzahl von Bits ist, die zur Binärdarstellung von i nötig sind.

Wir erhalten eine Obergrenze für die Arbeit, die auf jeder Ebene erforderlich ist, indem wir die Summe $\Sigma_j\, a(\log_2 (i_j))^4$ mit der Einschränkung maximieren, dass das Produkt $i_1 i_2 \ldots$ höchstens p beträgt. Da die vierte Potenz konvex ist, tritt das Maximum dann auf, wenn der gesamte Wert in einem der i_j komprimiert ist. Ist $i_1 = p$ und existieren keine weiteren i_j in einer Ebene, dann beträgt die Summe $a(\log_2 p)^4$. Dies ist höchstens an^4, da n die Bitanzahl in der Binärrepräsentation von p ist und $\log_2 p$ höchstens n beträgt.

Daraus ergibt sich, dass auf jeder Ebene ein Aufwand von höchstens $O(n^4)$ erforderlich ist. Da höchstens n Ebenen existieren, ist ein Aufwand von $O(n^5)$ für jeden Zweig des nichtdeterministischen Tests, ob p eine Primzahl ist, ausreichend. ■

Wir wissen nun, dass sowohl die Primzahlen als auch ihr Komplement in $\mathcal{NP}$ enthalten sind. Wäre eine der Mengen NP-vollständig, dann hätten wir nach Satz 11.2 einen Beweis, dass $\mathcal{NP}$ = Co-$\mathcal{NP}$.

11.5.6 Übungen zum Abschnitt 11.5

Übung 11.5.1 Berechnen Sie folgende Ausdrücke modulo 13:

a) $11 + 9$

* b) $9 - 11$

c) 5×8

* d) $5/8$

e) 5^8

Übung 11.5.2 Wir haben in Abschnitt 11.5.4 behauptet, dass für die meisten Werte von x zwischen 1 und 560 $x^{560} = 1$ modulo 561 gilt. Wählen Sie einige Werte für x, und verifizieren Sie diese Gleichung. Drücken Sie 560 zunächst binär aus, und berechnen Sie dann (im Computer) x^{2^j} modulo 561 für verschiedene Werte von j, um gemäß den Ausführungen in Abschnitt 11.5.3 zu vermeiden, 559 Multiplikationen auszuführen.

Übung 11.5.3 Eine ganze Zahl x zwischen 1 und $p - 1$ wird *quadratischer Rest* modulo p genannt, wenn es eine ganze Zahl y zwischen 1 und $p - 1$ gibt, sodass $y^2 = x$.

* a) Wie lauten die quadratischen Reste modulo 7? Sie können Tabelle 11.2 verwenden, um die Frage zu beantworten.

b) Wie lauten die quadratischen Reste modulo 13?

! c) Zeigen Sie: Ist p eine Primzahl, dann beträgt die Anzahl der quadratischen Reste modulo p $(p - 1)/2$; exakt die Hälfte der ganzen Zahlen ungleich 0 modulo p sind also quadratische Reste. *Hinweis:* Untersuchen Sie Ihre Daten aus a) und b). Können Sie ein Muster erkennen, das erklärt, warum jeder quadratische Rest das Quadrat zwei verschiedener Zahlen ist? Könnte eine ganze Zahl das Quadrat von drei verschiedenen Zahlen sein, wenn p eine Primzahl ist?

11.6 Zusammenfassung von Kapitel 11

- *Die Klasse Co-NP*: Eine Sprache ist in Co-$\mathcal{NP}$ enthalten, wenn ihr Komplement in $\mathcal{NP}$ enthalten ist. Alle Sprachen in $\mathcal{P}$ sind mit Sicherheit in Co-$\mathcal{NP}$ enthalten, doch wahrscheinlich gibt es Sprachen in $\mathcal{NP}$, die nicht in Co-$\mathcal{NP}$ enthalten sind und umgekehrt. Insbesondere scheinen NP-vollständige Probleme nicht in Co-$\mathcal{NP}$ enthalten zu sein.
- *Die Klasse PS*: Eine Sprache ist in $\mathcal{PS}$ (Polynomial Space) enthalten, wenn sie von einer deterministischen TM akzeptiert wird, für die ein polynomiales $p(n)$ existiert, sodass die TM bei einer Eingabe der Länge n nie mehr als $p(n)$ Zellen auf ihrem Band verwendet.
- *Die Klasse NPS*: Wir können auch die Akzeptanz durch eine nichtdeterministische TM definieren, deren Bandbenutzung durch eine Funktion begrenzt ist, die polynomial in der Länge der Eingabe ist. Die Klasse dieser Sprachen wird $\mathcal{NPS}$ genannt. Allerdings wissen wir durch den Satz von Savitch, dass $\mathcal{PS} = \mathcal{NPS}$ gilt. Insbesondere kann eine NTM mit Platzbegrenzung $p(n)$ durch eine DTM mit Platzbegrenzung $p^2(n)$ simuliert werden.
- *Zufallsabhängige Algorithmen und Turing-Maschinen*: Viele Algorithmen nutzen Zufälligkeit produktiv. Auf realen Computern werden Zufallszahlengeneratoren eingesetzt, um »das Werfen einer Münze« zu simulieren. Eine zufallsabhängige Turing-Maschine hat das gleiche zufällige Verhalten, wenn sie mit einem zusätzlichen Band ausgestattet wird, das eine Folge von Zufallsbits enthält.
- *Die Klasse RP*: Eine Sprache wird mit einem zufallsabhängigen polynomialen Zeitaufwand akzeptiert, wenn eine zufallsabhängige Turing-Maschine mit polynomialem Zeitaufwand existiert, die eine Eingabe mit einer Chance von mindestens 50% akzeptiert, falls die Eingabe in der Sprache enthalten ist. Andernfalls akzeptiert diese TM nie. Eine solche TM oder ein solcher Algorithmus wird »Monte Carlo« genannt.
- *Die Klasse ZPP*: Eine Sprache ist in der Klasse $\mathcal{ZPP}$ (Zero-Error, Probabilistic Polynomial Time) enthalten, wenn sie von einer zufallsabhängigen Turing-Maschine akzeptiert wird, die immer die korrekte Antwort über das Enthaltensein in der Sprache liefert; diese TM muss mit einem polynomialen Erwartungswert für den Zeitaufwand laufen, obwohl im ungünstigsten Fall die polynomiale Grenze überschritten werden kann. Eine solche TM oder ein solcher Algorithmus wird »Las Vegas« genannt.
- *Beziehungen zwischen Sprachklassen*: Die Klasse Co-$\mathcal{RP}$ ist die Menge der Komplemente der Sprachen in $\mathcal{RP}$. Die folgenden Beziehungen sind bekannt: $\mathcal{P} \subseteq \mathcal{ZPP} \subseteq (\mathcal{RP} \cap \text{Co-}\mathcal{RP})$. Außerdem gilt $\mathcal{RP} \subseteq \mathcal{NP}$ und damit auch Co-$\mathcal{RP} \subseteq$ Co-$\mathcal{NP}$.
- *Primzahlen und NP*: Sowohl die Primzahlen als auch das Komplement der Sprache der Primzahlen, die zusammengesetzten Zahlen, sind in $\mathcal{NP}$ enthalten. Damit ist es wahrscheinlich und wird deshalb vermutet, dass die Primzahlen und die zusammengesetzten Zahlen nicht NP-vollständig sind. Da wichtige kryptographische Methoden auf Primzahlen basieren, würde ein Beweis der obigen Vermutung die Sicherheit dieser Methode weitgehend bestätigen.

- *Primzahlen und RP*: Die zusammengesetzten Zahlen sind in $\mathcal{RP}$ enthalten. Der zufallsabhängige polynomiale Algorithmus zur Prüfung dieser Zahlen wird beim Generieren großer Primzahlen oder zumindest großer Zahlen eingesetzt, die nur mit einer beliebig kleinen Wahrscheinlichkeit zusammengesetzt sind.

LITERATURANGABEN ZU KAPITEL 11

Die Veröffentlichung [2] initiierte das Studium der Klassen von Sprachen, die durch Begrenzungen des von einer Turing-Maschine verwendeten Platzes definiert werden. Die ersten PS-vollständigen Probleme stellte Karp [4] in seiner Veröffentlichung vor, in der er die Bedeutung der NP-Vollständigkeit untersuchte. Die PS-Vollständigkeit des Problems aus Übung 11.3.2 – ob ein regulärer Ausdruck zu Σ^* äquivalent ist – stammt aus dieser Veröffentlichung.

Die PS-Vollständigkeit quantifizierter Boolescher Formeln ist unveröffentlichten Arbeiten von L. J. Stockmeyer entnommen. Die PS-Vollständigkeit des *Shannon Switching Game* (Übung 11.3.3) stammt aus [1].

Die Tatsache, dass die Primzahlen in $\mathcal{NP}$ enthalten sind, beschreibt Pratt [9]. Rabin [10] zeigte als Erster, dass die zusammengesetzten Zahlen in $\mathcal{RP}$ enthalten sind. Es ist interessant, dass zur gleichen Zeit ein Beweis veröffentlicht wurde, dass die Primzahlen in $\mathcal{P}$ enthalten sind, vorausgesetzt dass eine nicht bewiesene, doch allgemein als wahr angesehene Annahme, die so genannte erweiterte Riemannsche Hypothese, zutrifft [6].

Ihnen stehen verschiedene Bücher zur Verfügung, um Ihr Wissen über die in diesem Kapitel behandelten Themen zu vertiefen. [7] behandelt zufallsabhängige Algorithmen, einschließlich der vollständigen Algorithmen für Primzahltests. [5] ist eine Quelle für die Algorithmen der Modulararithmetik. [3] und [8] behandeln einige andere Komplexitätsklassen, die in diesem Kapitel keine Erwähnung fanden.

1. S. Even und R. E. Tarjan [1976]. »A combinatorial problem which is complete for polynomial space«, *J. ACM* **23**:4, S. 710–719.
2. J. O. Hartmanis, M. Lewis II und R. E. Stearns [1965]. »Hierarchies of memory limited computations«, *Proc. Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, S. 179–190.
3. J. E. Hopcroft und J. D. Ullman [1979]. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA.
4. R. M. Karp. »Reducibility among combinatorial problems«, *Complexity of Computer Computations* (R. E. Miller, Hrsg.), S. 85–104 Plenum Press, New York.
5. D. E. Knuth [1997]. *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*, Addison-Wesley, Reading, MA.
6. G. L. Miller [1976]. »Riemann's hypothesis and tests for primalty«, *J. Computer and System Sciences* **13**, S. 300–317.
7. R. Motwani und P. Raghavan [1995]. *Randomized Algorithms*, Cambridge Univ. Press.
8. C. H. Papadimitriou [1994]. *Computational Complexity*, Addison-Wesley, Reading, MA.
9. V. R. Pratt [1975]. »Every prime has a succinct certificate«, *SIAM J. Computing* **4**:3, S. 214–220.

→

10. M. O. Rabin [1976]. »Probabilistic algorithms«, *Algorithms and Complexity: Recent Results and New Directions* (J. F. Traub, Hrsg.), S. 21–39, Academic Press, New York.

11. R. L. Rivest, A. Shamir und L. Adleman [1978]. »A method for obtaining digital signatures and public-key cryptosystems«, *Communications of the ACM* **21**, S. 120–126.

12. W. J. Savitch [1970]. »Relationships between deterministic and nondeterministic tape complexities«, *J. Computer and System Science* **4**:2, S. 177–192.

Register

A

B

C

D

E

F

G

H

I

K

P

Q

R

S

T

U

W

X

Y

Z

Copyright

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de)

herunterladen