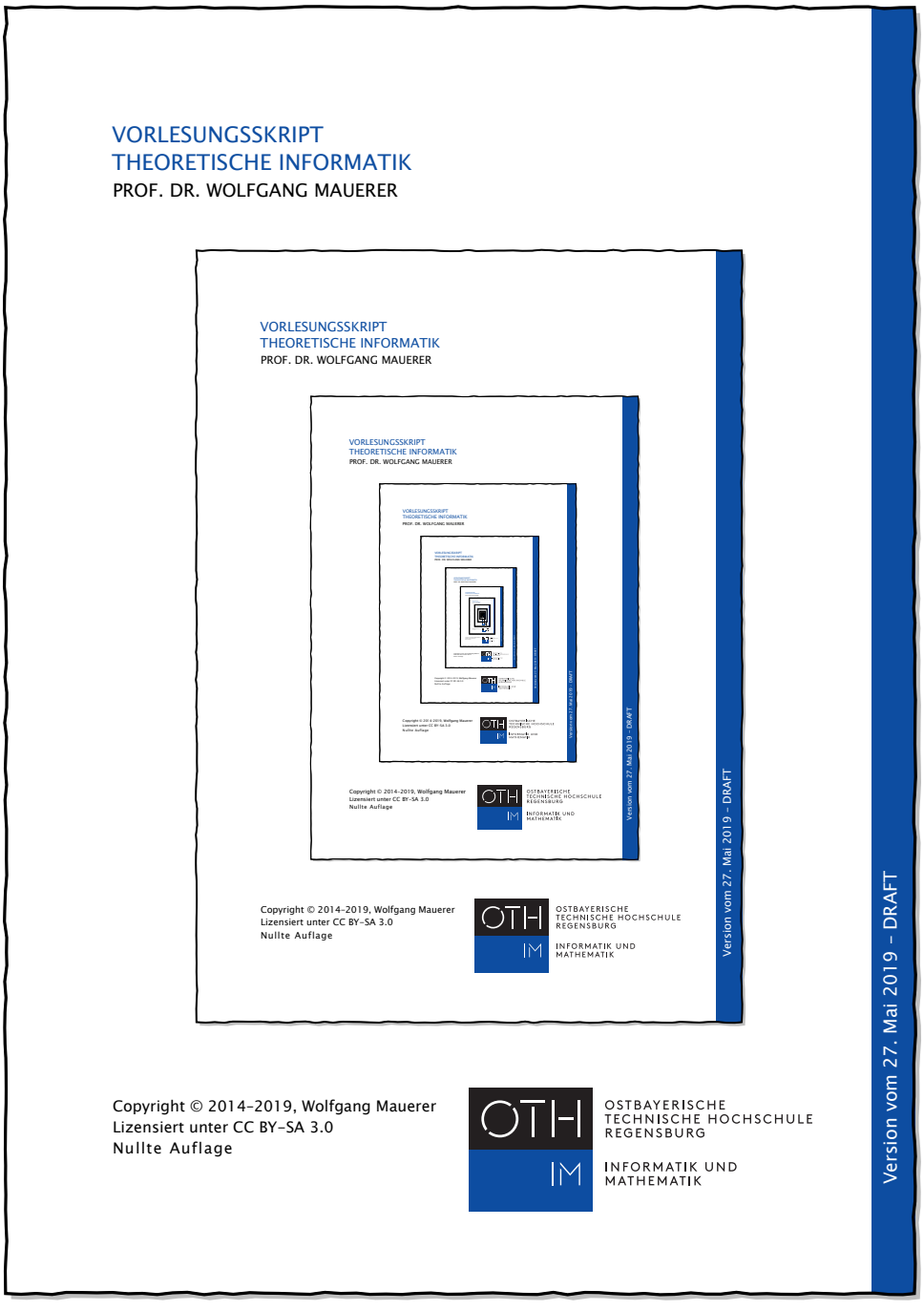
VORLESUNGSSKRIPT
THEORETISCHE INFORMATIK

PROF. DR. WOLFGANG MAUERER



OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

INFORMATIK UND MATHEMATIK

Version vom 27. Mai 2019 – DRAFT

Copyright © 2014–2019, Wolfgang Mauerer
Lizensiert unter CC BY-SA 3.0
Nullte Auflage

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

INFORMATIK UND MATHEMATIK

Version vom 27. Mai 2019 – DRAFT

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1
Einführung

Willkommen in der theoretischen Informatik! Dieses kurze Vorlesungsskript soll Ihnen helfen, den in der Vorlesung behandelten Stoff zu rekapitulieren und anhand von durchgerechneten, vollständigen Beispielen zu verstehen.

1.1 Was ist theoretische Informatik?

Es ist eine verbreitete Annahme vieler Studenten in den frühen Semestern, dass die theoretische Informatik (TI) komplex, anspruchsvoll und sehr formal ist. In der Tat zählt TI zu den arbeitsintensivsten Fächern, mit denen Sie zu Beginn Ihres Studiums konfrontiert werden. Das hat gute Gründe: Das Fach vermittelt nicht nur durchaus praktische Resultate, sondern zeigt insbesondere Denkmuster und Strukturen, die als solide Basis für das gesamte Informatik in ihren zahlreichen Facetten dienen. Je allgemeiner und abstrakter anzuwenden ein Denkmuster ist, desto anspruchsvoller ist es üblicherweise zu verstehen. Aber: Mit der Tiefe steigt die Breitenwirkung; je abstrakter ein Ergebnis ist, desto universeller wird es anwendbar, und desto nützlicher ist es. Damit die TI Ihre langfristige Wirkung entfalten kann, muss sie abstrakt genug sein. Damit sie verständlich bleibt, muss sie konkret genug sein. In diesem Spannungsfeld wird sich die Vorlesung bewegen.

Was ist theoretische Informatik? Dies sieht man besonders gut, wenn man sich von der entgegengesetzten Seite nähert. Eine schöne Charakterisierung dessen, was die Informatik *nicht* ist, wurde von Dijkstra gegeben:

E. W. Dijkstra über Informatik

»In der Informatik geht es genauso wenig um Computer, wie in der Astronomie um Teleskope.«

Computer sind ein wichtiges Hilfsmittel der Informatik, aber keinesfalls der eigentliche Zweck! Genauso wichtig wie die Frage, was mit einer konkreten Maschine, den aktuellsten Smartphones oder dem gerade modernen Tablet-Modell der Stunde erledigt werden kann, ist die Frage, welche Aufgaben *prinzipiell* von Maschinen gelöst werden können, und welche nicht. Und, nebenbei, welche Maschinen überhaupt Computer sind! Betrachtet man die rasende Entwicklung der letzten Jahrzehnte, ist die Frage ohne weitere Abstraktion und eine genaue Modellvorstellung nicht zu beantworten. Genau diese Antworten wird die theoretische Informatik allerdings liefern; das allwissende *Taschenbuch der Informatik* (DTV) stellt die Theorie aufgrund

Edsger W. Dijkstra (1930–2002) Dijkstra war einer der Gründerväter der Informatik, dessen Beiträge von strukturierter Programmierung über Compiler und Betriebssysteme bis hin zu formaler Spezifikation und Verifikation reichten. 1972 wurde ihm mit dem Turing-Preis die höchste Auszeichnung für Informatiker verliehen.

Fotos von Wissenschaftlern sind aus Wikipedia übernommen und unter CC-BY-SA lizenziert, soweit nicht explizit anders angegeben.

ihrer fundamentalen Aufgaben daher an die prominente erste Stelle des Versuchs, Inhalt und Bedeutung des Feldes zu definieren:

Taschenbuch der Informatik

> »Wissenschaft, die sich mit den theoretischen Grundlagen, den Mitteln und Methoden sowie mit der Anwendung der elektronischen Datenverarbeitung (EDV) beschäftigt, d.h. mit allen Aspekten der Informationsverarbeitung unter Einsatz von Computern einschließlich ihres Einflusses auf die Gesellschaft.«

Informatik ist ein Sammelbegriff, der sich aus *Infor*mation und Mathe*matik* zusammensetzt — im Vergleich zu vielen anderen Ingenieursdisziplinen nehmen Mathematik, formale Grundlagen und fundamentale Fragestellungen einen hohen Stellenwert ein, der vergleichbar mit naturwissenschaftlichen Disziplinen ist. Informatiker sollen nicht nur *Techniken anwenden* und *konkrete Probleme* lösen, sondern auch die Bedeutung des automatisierten Rechnens im Hinblick auf seine Möglichkeiten und Grenzen begreifen.

1.2 Was ist Theorie?

Die genaue Definition des Theoriebegriffs ist ein grundlegendes philosophisches Problem, auf das hier nicht weiter eingegangen werden soll. Nichtsdestotrotz kann man sagen, dass jede Theorie ein (notwendigerweise vereinfachtes) *Modell der Realität* darstellt. Mit Hilfe des Modells soll eine möglichst großen Klasse von Phänomenen und Beobachtungen, die in der realen Welt auftreten, mit einer möglichst geringen Anzahl von Annahmen erklärt werden. Zusätzlich soll es möglich sein, basierend auf dem Modell nicht nur bislang bekannte Phänomene zu erklären, sondern auch quantitative, verifizierbare Prognosen zu liefern. Ein besonderes Augenmerk der theoretischen Informatik (im Vergleich zu anderen Naturwissenschaften) liegt auch in der Klassifikation abstrakter Rechenmodelle, der Analyse Ihrer Möglichkeiten und vor allem Ihrer Beschränkungen.

Unglücklicherweise wird der Begriff »Theorie« im populären Sprachgebrauch sehr verwässert bis völlig unzureichend verwendet; »Theorien« zum Verschwinden von Flugzeugen, Heirat und Scheidung von Personen des öffentlichen Lebens, oder den Beweggründen hinter Personalwechseln im Trainerstab von Fußballvereinen fallen typischerweise *nicht* zu den eben eingeführen Begrifflichkeiten, sondern sind (allerhöchstens) Hypothesen, wenn nicht nur bloße Vermutungen oder gar subjektive Meinungen.

Selbst eine Abhandlung von Immanuel Kant von 1793 mit dem Titel Über den Gemeinspruch: Das mag in der Theorie richtig sein, taugt aber nicht für die Praxis konnte den Ausspruch nicht aus der Welt schaffen.

Die häufig angebrachte Vermutung »Das funktioniert nur in der Theorie, aber nicht in der Praxis« ist typischerweise irreführend: Theorie und Praxis sind keineswegs das Gegenteil voneinander, sondern zwei untrennbare Seiten einer Medaille, die sich gegenseitig ergänzen und bedingen. Gerade die Theorie stellt der Informatik sehr konkrete und praktikable Methoden zur Verfügung, mit deren Hilfe man bereits *vor* der praktischen Durchführung einer Aufgabe entscheiden kann, ob es sich überhaupt lohnt, damit anzufangen – oder ob die Aufgabe prinzipiell unlösbar ist, oft sogar unabhängig vom aktuellen Stand der Technik.

Die Grundstudiumsvorlesung theoretische Informatik befasst sich deshalb unter anderem mit folgenden Fragestellungen:

- ❏ Welche Aufgaben kann ein Computer lösen, welche nicht?
- ❏ Kann jeder Computer die gleichen Probleme lösen?
- ❏ Wie kann man einen Computer mathematisch definieren?
- ❏ *Was ist ein Computer überhaupt?* Und: Was ist *kein* Computer?
- ❏ Welche Funktionen sind berechenbar, welche nicht?
- ❏ Mit wie viel Rechenzeit kann ein Problem gelöst werden? Wie viel Speicherplatz (oder welche Menge an anderen Ressourcen, beispielsweise Energie oder Chipfläche) ist dafür notwendig?
- ❏ Offenbar gibt es leichtere und schwerere Probleme – wie können diese sinnvoll klassifiziert werden?

Sollte man die grundlegende Fragestellung der theoretischen Informatik auf einen Satz komprimieren, könnte man dies so tun: *Welche Bestandteile der Informatik sind »zeitlos« gültig?* Fast ist man versucht, noch hinzuzufügen: *Und nicht nach zwei Handygenerationen veraltet?* Die Themen der theoretischen Informatik mögen anspruchsvoll scheinen (oft sind sie es auch – wenn Sie etwas nicht auf Anhieb verstehen, können Sie sicher sein, dass Sie nicht der erste Student sind, dem dies passiert); zum Ausgleich werden Sie aber viele davon in den unterschiedlichsten Aspekten der Informatik von Systemsoftware über Compilerbau und Computer-Algebra bis hin zum angewandten Softwareengineering wiederfinden – und dies unabhängig von den gerade aktuellen technischen Strömungen und Modeerscheinungen. Der Aufwand, um die manchmal steile Lernkurve der TI zu meistern, ist deshalb eine gute Investition in Ihre Zukunft.

Die Bezeichnung »Student« steht in diesen Aufzeichnungen gleichberechtigt für männliche, weibliche, und geschlechtsneutrale Personen-/innen und vollautomatisierte elektronische Wesen. Frauen, Männer und sämtliche anderen Geschlechter sind in unserer Zeit und vor dem Gesetz gleichberechtigt, und das ist gut – aber kein Freibrief, um die deutsche Sprache mit Konstruktionen wie Student_innen zu beleidigen oder völlige Ignoranz gegenüber Bedeutung und Verwendung eines Substantivs im Vergleich zum Partizip Präsens (Studierende) zu demonstrieren. Zumal sich das Substantiv »Student« ursprünglich aus den lateinischen Partizip Präsens »studens« entwickelt hat… Um mit Max Gold die autoritative Kapazität in Fragen sprachlicher Korrektheit zu zitieren: Man kann nicht sagen: »In der Kneipe sitzen biertrinkende Studierende.« Oder nach einem Massaker an einer Universität: »Die Bevölkerung beweint die sterbenden Studierenden.« Niemand kann gleichzeitig sterben und studieren.
Damit schalten wir zurück zur theoretischen Informatik.

1.3 Intention des Skripts

Das Skript orientiert sich absichtlich stark an Struktur und Inhalt der Folien, die in der Vorlesung gezeigt werden; einige Punkte werden nur in der Vorlesung behandelt, während manche andere Themen im Skript etwas genauer ausgeführt sind, als sie in der »Tonspur« der Vorlesung behandelt werden. Die Darstellung ist in beiden Fällen weniger ausführlich als in den zahlreichen hervorragenden Lehrbüchern, die zur theoretischen Informatik existieren und sowohl in deutscher wie auch englischer Sprache verfügbar sind (einige davon stellen wir im folgenden Abschnitt vor). Wenn ein Thema nicht in der Vorlesung behandelt wurde, wird es auch nicht im Skript angesprochen, selbst wenn es eine interessante Erweiterung des Standardstoffs wäre. Die Kürze ist beabsichtigt; das Skript soll kein Lehrbuch ersetzen, sondern die vorhandene Literatur in zwei Aspekten ergänzen:

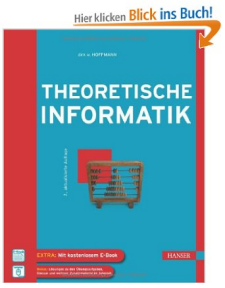
- ❏ Der exakte Stoff, der in der Vorlesung behandelt wird, soll systematisch zusammengefasst werden. Sie halten einen Leitfaden durch den Stoff in Händen, der Ihnen bei der Prüfungsvorbereitung ermöglicht, sich auf die wichtigen Themen und klausurrelevanten Inhalte zu konzentrieren. Auch wenn dies nicht bedeutet, dass die Prüfung der eigentliche Sinn der Vorlesung ist!

❏ Das Skript enthält zahlreiche Aufgaben in einem Schwierigkeitsgrad, der typisch für Übungsblätter und Klausuraufgaben ist. Insbesondere sind die Aufgaben vollständig und ausführlich durchgerechnet; dies soll Ihnen zum einen helfen, den Stoff weiter zu verstehen und zu vertiefen, als dies mit einer rein abstrakten Darstellung möglich ist, soll sie zum anderen aber auch auf den Stil vorbereiten, der zur Lösung von Übungsaufgaben (und natürlich auch wieder in Klausuren) sinnvoll und erwünscht ist. Dies bedeutet *nicht*, dass Lösungen für Übungsaufgaben ähnlich lang sein müssen; die Erläuterungen sind bewusst ausführlich gehalten, damit sie sich zum Selbststudium eignen.

Weißer Spion (1961–)
Dagger (Deckname: Weißer Spion) ist durch seine superpolynomial häufigen Versuche bekannt, den schwarzen Spion über einen effizient berechenbaren surjektiven Morphismus in randomisierte Kohlenstoffverbindungen zu transformieren. Ein Großteil seiner Arbeiten wurden im MAD-Magazin publiziert (Bildquelle: hero.wikia.com).

Graham, Knuth und Patashnik machen im Informatik-Klassiker Concrete Mathematics ebenfalls von studentischen Randkommentaren – Graffitis – Gebrauch. Die Kommentare in den Ränder decken das komplette Spektrum von hilfreichen Nebenrechnungen bis zu Kalauern von weisen Hörern aus den hinteren Reihen ab, bereichern das Buch aber ungemein.

Die Abbildungen der Buchcover entstammen der Webseite `amazon.de`. Wie man zweifelsfrei anhand der lästigen »Blick ins Buch«-Bestandteile feststellt.



Da theoretische Informatik nicht von Maschinen, sondern von Menschen gemacht wird (und dies unter vernünftigen komplexitätstheoretischen Annahmen auch so bleiben wird, wie Sie nach der Lektüre von Kapitel 4 einsehen werden), geben wir Bilder und einige biographische Details zu Personen an, die für wichtige und zentrale Resultate verantwortlich zeichnen. Entsprechende Artikel aus der Online-Enzyklopädie Wikipedia sind verlinkt.

Das Layout des Skripts orientiert sich an Vorschlägen von Edward Tufte, einem bekannten zeitgenössischen Designer, und zeichnet sich vor allem durch die einladenden Ränder aus, die Erläuterungen, Kommentare und illustrierende Bildchen aufnehmen können. Momentan stammen noch alle Bemerkungen vom Autor – dies soll sich aber ändern: Wenn sie aktiv mit dem Skript arbeiten, werden Sie sicher den einen oder anderen Kommentar, eine Hilfsrechnung, einen erläuternden Satz oder andere hilfreiche Dinge im Skript notieren. Diese Erkenntnisse helfen möglicherweise auch Ihren Kommilitonen, da sie ein Problem aus einer anderen Sichtweise beleuchten oder einen Punkt klarmachen, der dem Autor nicht als problematisch bewusst war. Kommentare, die in die Ränder integriert werden sollen, werden daher explizit erbeten! Für zahlreiche Fehlermeldungen bedanke ich mich bei Monika Inderst und Friedrich Holzner.

1.4 *Literaturhinweise*

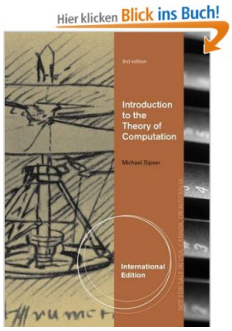
Folgende Bücher wurden vom Autor zur Vorbereitung der Vorlesung verwendet. Aufmerksame Leser werden viele Ähnlichkeiten zum vorliegenden Skriptum entdecken; strukturell orientiert sich die Darstellung stark am Buch von Schöning, das sich als »Standard« an deutschen Universitäten und Hochschulen etabiert hat.

❏ Dirk W. Hoffmann, *Theoretische Informatik*, 2. Auflage, Carl Hanser Verlag, 2011.

Sehr anschauliches Lehrbuch mit zahllosen aufwendigen Abbildungen, das den Stoff anhand vieler gut erklärter Beispiele erläutert. Die meisten Themen werden weniger formal als in der Vorlesung behandelt; der Fokus liegt auf einem intuitiven Verständnis und nicht auf der mathematischen Modellierung von Problemen. Ein Anfangskapitel fasst die Grundlagen der Logik, die in der Vorlesung nicht explizit behandelt, aber dennoch auf Schulniveau vorausgesetzt werden, sehr gut zusammen. Das Buch enthält viele Übungsaufgaben auf meist einfachem Niveau, deren Lösungen zum Download angeboten werden.

❑ Michael Sipser, *Introduction to the Theory of Computation*, 3rd edition, Cengage Learning, 2012.

Standardwerk aus dem angelsächsischen Raum (und entsprechend in Englisch verfasst), das deutlich über den Stoff einer einsemestrigen Vorlesung auf Anfängerniveau hinausgeht, aber neben den liebevoll erläuterten Standardthemen auch aktuellere Forschungsfragen und -ergebnisse behandelt. Viele wichtige Resultate der modernen theoretischen Informatik stammen aus der Feder von Michael Sipser, die Darstellung erfolgt also aus berufenem Munde. Das Buch enthält eine Vielzahl von (teilweise gelösten) Übungsaufgaben und ist umfangreich illustriert.

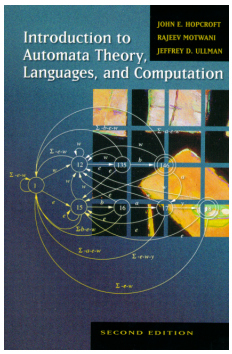


❑ Uwe Schöning, *Theoretische Informatik – kurz gefasst*, 5. Auflage, Spektrum, 2008.

Als erster Text möglicherweise etwas knapp und formal, aber sehr empfehlenswert als kompaktes Repetitorium zur Vorbereitung auf die Klausur oder zum Wiederholen des Stoffes. Es enthält wenige Abbildungen und wenige Beispiele, behandelt dafür aber relativ genau den Stoff, der auch in der Vorlesung besprochen wird (der Abschnitt zur Komplexitätstheorie ist im Buch etwas umfangreicher geraten). Beispiele, die in der Vorlesung behandelt werden, sind typischerweise diesem Buch entnommen, werden aber ausführlicher erläutert.

❑ John E. Hopcroft, Rajeev Motwani und Jeffrey Ullman, *Einführung in die Automatentheorie, formale Sprachen und Berechenbarkeit*, 3. Auflage, Pearson Studium, 20011.

Das Buch hat eine lange Geschichte durchlebt: Die ersten Auflagen, die den Beginn der Informatik an Hochschulen begleitet haben, wurden noch von den Autoren Hopcroft und Ullman alleine betreut und sind deutlich formaler und konziser gehalten, da sie sich an höhere Semester wenden. Die aktuelle Auflage, die sehr ausführlich (mit zeitweiser Tendenz zur Langatmigkeit) eine Übermenge der Themen der Vorlesungen einführen. Der Sprachduktus des englischen Originals tritt in der aktuellen deutschen Übersetzung deutlich hervor, weshalb es je nach Vorliebe des Lesers durchaus empfehlenswert sein kann, einen Blick in die Originalausgaben zu werfen.



Typischerweise ist es insbesondere für Anfänger förderlich, nicht nur mit einem einzigen Buch zu arbeiten, sondern mehrere unterschiedliche Darstellungsweisen durchzuarbeiten, um ein Problem aus verschiedenen Blickwinkeln zu betrachten und daher besser zu verstehen.

Neben den genannten Büchern gibt es zahllose Alternativen, die ähnliche Stoffbereiche in unterschiedlichsten Präsentationsformen abdecken, die wir hier aber nicht explizit aufführen.

2
Formale Sprachen und Automaten

2.1 Grundbegriffe formaler Sprachen

Formale Sprachen gehören zu den zentralen Konzepten der theoretischen Informatik, auch wenn dies auf den ersten Blick möglicherweise erstaunlich scheint – Sprachen bringt man intuitiv eher mit Sprachwissenschaften als mit Informatik in Verbindung. Dennoch sind Sprachen auf den zweiten Blick in allen Facetten der Informatik omnipräsent, wenn auch nicht in den gewohnten Bedeutungen. Beispielsweise kommen Sprachen in folgenden Anwendungen zum Einsatz:

Alle angewandten Aspekte der Informatik sind viel mit Sprachverarbeitung beschäftigt, das Thema ist also keineswegs nur von theoretischem Interesse. Zusätzlich gibt es aber auch tiefgreifende Verbindungen zwischen Sprachverarbeitung und den Grundlagen des Rechnens, die jeder Informatiker kennen sollte und die im Laufe der Vorlesung herausgearbeitet werden. Wirkliches Verständnis der Informatik erfordert nicht nur solider Kenntnisse der aktuellen technischen Entwicklungen, sondern benötigt auch solide, zeitlose Grundlagen, unter anderem wirkliche Neuerungen erkennen zu können.

- Compiler, die Quelltexte in C++, Java, C#- oder anderen Sprachen übersetzen sollen, müssen zunächst überprüfen, ob das Programm in einer syntaktisch korrekten Form vorliegt. Dies wird anhand einer Grammatik entschieden, die die Regeln der jeweiligen Programmiersprache beschreibt. Ist ein Programm syntaktisch korrekt, wird es in eine andere Sprache umgewandelt: Maschinen- bzw. Assembler-Sprache, die nicht mehr von Menschen, sondern von einem Rechner interpretiert wird.
- Netzwerkpakete, die beispielsweise beim Surfen im Internet zwischen Ihrem Computer und einem Server ausgetauscht werden, müssen Informationen und Daten in bestimmten Strukturen bereitstellen, die durch eine formale Sprache beschrieben werden.
- Konfigurationsdateien für Programme folgen ähnlichen Strukturen.
- HTML- und XML-Dateien, die von Browsern zur Beschreibung von Webseiten eingesetzt werden, müssen Regeln folgen, wie strukturierte Informationen syntaktisch korrekt geschachtelt werden.
- Selbst Bankleitzahlen sind einfache formale Sprachen: Nur bestimmte Kombinationen von Zahlen sind gültige Bankleitzahlen, andere wiederum nicht.

In jedem der obigen Fälle werden textuelle Eingabedaten verarbeitet, die gewissen Regeln genügen müssen, um »korrekt« zu sein. Entsprechend sind für Sprachen zwei Zutaten für eine formale Sprache von Bedeutung:

- Eine *Grammatik* beschreibt eine Sprache L, indem Regeln aufgestellt werden, nach denen korrekte Sätze (Elemente der Sprache) aufgebaut sind.
- Das *Wortproblem* stellt folgende Frage: Gegeben eine Sprache L, die durch

Weniger formal: Ist ein Satz mit einer gegebenen Grammatik verträglich?

eine Grammatik G beschrieben wird, und einen Satz p: Gilt $p \in L$ oder $p \notin L$? In der Informatik unterscheidet man (im Gegensatz zur Linguistik) typischerweise nicht zwischen Wörtern und Sätzen, daher der Name *Wortproblem*.

Definition: Alphabet Ein *Alphabet* Σ ist eine endliche Menge unterscheidbarer Symbole. ∎

Alphabete werden typischerweise mit Hilfe der mathematischen Mengenschreibweise dargestellt, wie folgende Beispiele zeigen:

- $\Sigma_1 = \{0, 1\}$: Ziffern des Binärsystems.
- $\Sigma_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$: Dezimalziffern.
- $\Sigma_3 = \{0, \ldots, 9, \text{A}, \text{B}, \text{C}, \text{D}, \text{E}, \text{F}\}$: Hexadezimalziffern.
- $\Sigma_4 = \{\text{a}, \text{b}, \ldots, \text{z}\}$: Kleinbuchstaben.
- $\Sigma_5 = \{\vee, \wedge, \neg\}$: Aussagenlogische Symbole.

Formal unterscheidet man durch folgende Definition zwischen den Begriffen *Zeichen*, *Wort* und *Sprache*:

Definition: Zeichen, Wort, Sprache Die genannten Begriffe (und einige weitere elementare Notationen) sind wie folgt definiert:

- Jedes Element $\sigma \in \Sigma$ ist ein *Zeichen* des Alphabets Σ.
- Jedes Element $\omega \in \Sigma^*$ ist *Wort* über Σ.
- Σ^n ist die Menge aller Wörter mit Länge $n \in \mathbb{N}\backslash\{0\}$. Als Sonderfall definieren wir $\Sigma^0 = \epsilon$.
- Σ^* ist die Menge aller Wörter über Σ, also $\Sigma^* \equiv \cup_{n \in \mathbb{N}_0} \Sigma^n$.
- Das *leere Wort* wird durch ϵ repräsentiert. Es gilt $\epsilon \in \Sigma^*$ und $\epsilon \notin \Sigma^n$ mit $n \in \mathbb{N}$.
- Die Menge aller Wörter beliebiger Länge über *Sigma* mit *mindestens* einem Buchstaben ist durch Σ^+ gegeben.
- Die Länge eines Wortes (Anzahl der Buchstaben) wird durch $|\omega|$ angegeben. Achtung: Es gilt $|\epsilon| = 0$ – das leere Wort hat Länge 0, da es keine Buchstaben enthält.
- Jede Teilmenge $L \subseteq \Sigma^*$ ist eine *formale Sprache*. Naturgemäß ist in dieser sehr abstrakten Definition noch keine Charakterisierung der formalen Sprache durch ein geeignetes Beschreibungsmittel enthalten. ∎

Die natürlichen Zahlen $\mathbb{N}$ stehen in diesem Skript für die Menge $\{1, 2, 3, \ldots\}$. Die Zahl 0 ist nicht darin enthalten. Die Schreibweise $\mathbb{N}\backslash\{0\}$ ist redundant, wir verwenden sie aber gelegentlich, um besonders hervorzuheben, dass 0 von der Betrachtung ausgenommen ist.

2.2 *Endliche Automaten*

Zum Erkennen von Wörtern müssen zwei konkrete Fragen beantwortet werden:

- Welche Möglichkeiten gibt es zur Definition einer formalen Sprache, d.h. wie wird eine Sprache konkret angegeben? Anders betrachtet: Wie wird der Inhalt von $L \subseteq \Sigma^*$ festgelegt? In sehr einfachen Ausnahmefällen

(endliche Sprachen, also Sammlungen endlich vieler Wörter) ist es möglich, den Inhalt der Menge explizit aufzuzählen; bei Sprachen mit unendlich vielen Wörtern scheidet diese Variante aus prinzipiellen Gründen aus – Alternativen wie explizite Grammatiken sind notwendig.

❏ Wie kann man Wörter maschinell erkennen, ohne auf die menschliche Fähigkeiten zur intuitiven Überlegung zurückgreifen zu müssen?

Für eine sehr wichtige, praxisrelevante Klasse von Sprachen, die wir im weiteren Verlauf der Vorlesung umfangreich charakterisieren werden, haben sich *endliche Automaten* als passendes Hilfsmittel herausgestellt. Es handelt sich dabei um sehr einfache Formen von Computern, die beispielsweise in Steuerungsproblemen eingesetzt werden. Wir stellen den Sprachaspekt zunächst in den Hintergrund und betrachten, wie eine Türsteuerung mit Hilfe eines endlichen Automaten umgesetzt werden kann. Angenommen, eine automatische Tür in einem Kaufhaus kann sich nach vorne öffnen und soll aufgehen, wenn ein Besucher das Gebäude verlässt oder betritt. Gleichzeitig soll sichergestellt werden, dass ein Besucher, der sich vor der Tür aufhält, nicht von ihr verletzt wird. Abbildung 2.1 illustriert das Szenario.

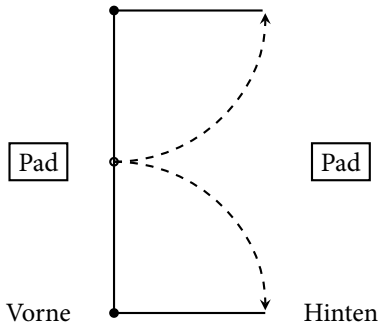


Abbildung 2.1: Automatische Türsteuerung.

	Nirgends	Vorne	Hinten	V+H
Zu	Zu	Offen	Zu	Zu
Offen	Zu	Offen	Offen	Offen

Tabelle 2.1: Zustandsübergänge für eine automatische Tür.

Die notwendigen Übergänge zwischen den beiden Zuständen der Tür – offen und geschlossen – sind in Tabelle 2.1 definiert. Allerdings ist diese Art der Darstellung nicht immer intuitiv leicht verständlich, weshalb man auch graphische Illustrationen wie in Abbildung 2.2 gezeigt einsetzt. Die Zustände der Tür (zu/offen) sind durch Kreise (*Knoten*), die Übergänge zwischen den Zuständen (Personen nähern oder entfernen sich) durch Linien (*Kanten*) dargestellt.

Die Steuerung kann durch sehr einfache Rechenvorschriften umgesetzt werden, ohne dass komplizierte Computer oder Programme verwendet werden müssen! Gleiche oder ähnliche Konzepte werden in vielen anderen Gebieten eingesetzt, beispielsweise:

❏ Industrielle Steuerungsaufgaben (Förderbänder, Greifarme, Ventilklappen, …).

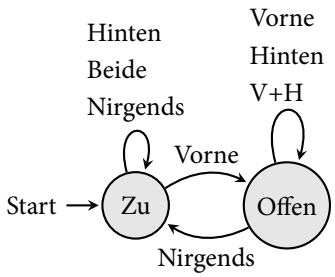


Abbildung 2.2: Steuerung für eine automatische Türe.

❏ Netzwerkprotokolle (TCP, UDP).

❏ Schnelle Mustererkennung in Daten.

❏ Verallgemeinerung: Markov-Ketten.

  ❏ Schriftklassifikation (OCR), Bilderkennung.

  ❏ Sprachverarbeitung.

  ❏ Hochfrequenz-Aktienhandel.

Automaten dieser Art bezeichnet man als *endliche Automaten*, weil die Anzahl ihrer Zustände (Knoten) beschränkt ist. Endliche Automaten können essentiell für drei verschiedene Problemklassen eingesetzt werden:

❏ Reine Zustandsübergänge (wie bei der Türsteuerung).

❏ Erkennen von Sprachen (Fokus dieser Vorlesung).

❏ »Übersetzen« von Eingaben in Ausgaben. In diesem Fall spricht man genau genommen nicht mehr von Automaten, sondern von *Transduktoren*.

2.2.1 *Endlicher Automat zur Sprachverarbeitung*

Betrachten wir den endlichen Automaten in Abbildung 2.3, der zum Erkennen einer Sprache verwendet werden kann.

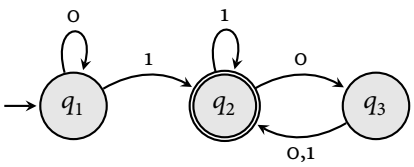


Abbildung 2.3: Deterministischer endlicher Automat zum Erkennen einer Sprache.

Der Automat besteht aus folgenden Komponenten:

❏ Zustände: q_1, q_2, q_3

  ❏ Startzustand q_1 – in diesem Zustand beginnt die Verarbeitung.

  ❏ (Akzeptierender) Endzustand q_2 – dieser Zustand zeigt an, dass ein Wort korrekt verarbeitet bzw. als gültig akzeptiert wurde, wenn ihn der Automat nach Abarbeitung des Wortes erreicht.

❏ Kanten geben *Transitionen* an, also Übergänge zwischen zwei Zuständen. Jede Kante muss mit genau einem Buchstaben beschriftet sein, damit

eindeutig klar ist, welcher Übergang bei welcher Eingabe ausgeführt wird.

Die Aufgabe des Automaten besteht darin, eine Eingabe $w \in \{0,1\}^*$ zu durchlaufen (Buchstabe für Buchstabe), und die entsprechenden Transitionen zwischen den Zuständen durchzuführen. Wenn die Eingabe komplett durchlaufen wurde, also keine neuen Buchstaben mehr vorhanden sind, wird das Eingabewort *akzeptiert*, sofern sich der Automat in einem akzeptierenden Zustand befindet. Anderenfalls wird das Wort *abgelehnt*. Dies entspricht dem weiter oben eingeführten Wortproblem. Man sagt, der Automat *entscheidet* die Sprache.

Der betrachtete Automat akzeptiert beispielsweise die Eingaben »1101« und »01«.

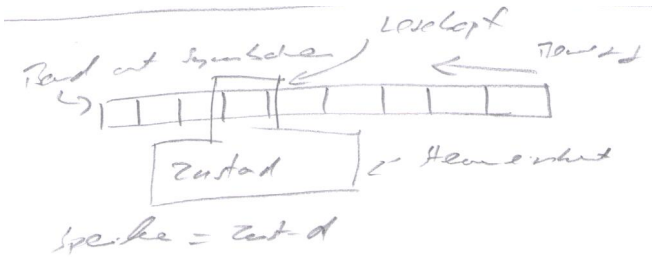


Abbildung 2.4: Implementierungsbeispiel für endliche Automaten

Es ist sehr einfach, endliche Automaten physikalisch zu implementieren, wie Abbildung 2.4 verdeutlicht: Die Eingabe steht auf einem Band, das in Felder unterteilt ist; in jedem Zeitschritt liest ein Kopf den aktuellen Buchstaben und fährt ein Segment weiter nach rechts. Der aktuelle Zustand des Automaten wird in einem (sehr kleinen) Steuerwerk vorgehalten, das auch die Übergänge zwischen den Zuständen regelt, abhängig vom aktuell gelesenen Zeichen. Offensichtlich ist ein DEA deutlich einfacher als ein »richtiger« Computer – es wird dennoch einen guten Teil des Semesters in Anspruch nehmen, die Unterschiede zwischen DEAs und Computern herauszuarbeiten. Dies liegt allerdings nicht an der Komplexität von DEAs, sondern ist vor allem in der Frage begründet, was einen »richtigen« Computer überhaupt ausmacht.

2.2.2 *Formale Definition endlicher Automaten*

Eine formal saubere Definition dient nicht dem Selbstzweck, sondern soll Sachverhalte möglichst genau und ohne die Ambiguitäten natürlicher Sprache beschreiben. Die ist nicht nur für die theoretische Informatik relevant, sondern generell Ihre zukünftige Aufgabe als Informatiker: Sie müssen Algorithmen so beschreiben, dass sie selbst ein Computer versteht, der bekanntlich ohne Fähigkeit zur Interpretation oder gesunden Menschenverstand auskommen muss.

Wir verfeinern die informelle Definition eines endlichen Automaten, stellen aber den mathematisch präzisen Ausdrücken in Abbildung 2.5 auf der rechten Seite eine leichter verständlichere Formulierung auf der rechten Seite gegenüber.

Im Folgenden bezeichnen wir den Automaten aus Abbildung 2.3 als Maschine M_1 und beziehen uns dabei zum einen auf den Automaten an sich, zum anderen auf alle Elemente obiger Liste, die zu einer eindeutigen Definition des Automaten notwendig sind. Es hat sich folgende Terminologie eingebürgert:

- ❏ M_1 *akzeptiert* bestimmte Wörter (beispielsweise 1101, 110000, 1100, 0101010101, 1)
- ❏ M_1 *erkennt* bzw. entscheidet eine Sprache (bei komplizierteren Maschinenmodellen, die später im Skript besprochen werden, ist allerdings eine

Deterministischer endlicher Automat (DEA)

Ein endlicher Automat besteht aus folgenden Komponenten:

- ❏ Zustände, in denen sich der Automat befinden kann.
- ❏ Alphabet, aus dem die Eingabe entnommen wird.
- ❏ Eine Festlegung der Übergänge zwischen den Zuständen in Abhängigkeit davon, welches Zeichen zuletzt gelesen wurde.
- ❏ Einem ausgezeichneten Zustand, in dem der Automat startet
- ❏ Einem oder mehreren (akzeptierenden) Endzuständen, die anzeigen, ob ein Wort nach Ende der Bearbeitung akzeptiert wird.

Ein endlicher Automat ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei:

- ❏ Q ist eine endliche Zustandsmenge
- ❏ Σ ist ein endliches Alphabet,
- ❏ $\delta : Q \times \Sigma \to Q$ ist die Übergangsfunktion,
- ❏ $q_0 \in Q$ ist der Startzustand
- ❏ $F \subseteq Q$ ist die Menge der akzeptierenden Zustände

Abbildung 2.5: Definition des deterministischen endlichen Automaten. Links informell, recht mathematisch-formal.

genauere Differenzierung notwendig).

Eine Maschine akzeptiert (im nicht-pathologischen Fall; man kann natürlich auch Automaten konstruieren, die nur ein einziges Wort erkennen) *mehrere* Wörter, aber nur *eine* Sprache. Formal kennzeichnet man die von einem Automaten M erkannte Sprache mit $\mathcal{L}(M)$; für obiges Beispiel gilt $\mathcal{L}(M_1) = A$ mit

Das geschwungene L »$\mathcal{L}$« steht für language.

$$A = \{w \mid w \text{ enthält mindestens eine Eins, und eine gerade Anzahl von Nullen folgt auf die Eins.}\} \tag{2.1}$$

In Kurzform sagt man: M_1 erkennt (die Sprache) A.

Der Automat M_1 muss anhand des Musters aus Abbildung 2.5 formal definiert werden. Das 5-Tupel $M_1 = (Q, \Sigma, \delta, q_0, F)$ enthält folgende Komponenten:

- ❏ Zustandsmenge Q und Alphabet Σ werden direkt als Menge angegeben:

$$Q = \{q_1, q_2, q_3\}$$
$$\Sigma = \{0, 1\}$$

- ❏ Die Übergangsfunktion wird typischerweise in Form einer Tabelle spezifiziert (man könnte alternativ beispielsweise auch alle Übergänge direkt in der Form $\delta(\cdot, \cdot) = \cdot$ angeben):

Q \ Σ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

❏ Startzustand ist q_1.

❏ Die Menge der akzeptierenden Endzustände ist durch $F = \{q_2\}$ gegeben.

Auch wenn nur ein einziger Zustand als Endzustand ausgezeichnet ist, muss dennoch die Mengenschreibweise berücksichtigt werden!

Zur Illustration betrachten wir die Zustandskette, die M_1 durchläuft, wenn er das Wort »1101« als Eingabe erhält. Ausgehend vom Startzustand q_1 gilt

$$\begin{aligned}\delta(q_1, 1) &= q_2\\ \delta(q_2, 1) &= q_2\\ \delta(q_2, 0) &= q_3\\ \delta(q_3, 1) &= q_2\end{aligned} \tag{2.2}$$

Die durchlaufene Zustandskette ist q_1, q_2, q_2, q_3, q_2 (für jedes Wort mit n Buchstaben werden $n + 1$ Zustände durchlaufen). Das Beispielwort wird akzeptiert, da $q_2 \in F$, der Automat also in einem akzeptierenden Endzustand hält.

Im Gegensatz dazu akzeptiert M_1 das Wort »00« nicht, da wegen

$$\begin{aligned}\delta(q_1, 0) &= q_1\\ \delta(q_1, 0) &= q_1\end{aligned} \tag{2.3}$$

die Zustandskette q_1, q_1, q_1 durchlaufen wird und $q_1 \notin F$ gilt, der Automat also nicht in einem akzeptierenden Endzustand hält.

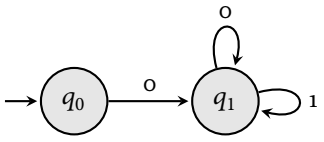
2.2.3 *Konstruktion endlicher Automaten*

Nachdem endliche Automaten bestimmte Sprachen erkennen, liegt es nahe, spezifische Automaten für eine gegebene Sprache zu konstruieren. Beispielsweise sei die Sprache

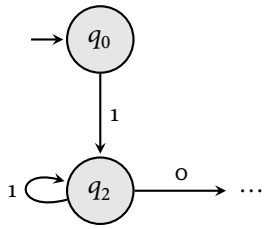
$$\begin{aligned}L &= \{w \in \{0,1\}^* \mid w \text{ beginnt mit 1 und endet mit 0 }\}\\ &= \{w \in \{0,1\}^* \mid |w| \geq 2 \wedge w_0 = 1 \wedge w_{|w|-1} = 0\}\end{aligned} \tag{2.4}$$

vorgegeben (w_i bezeichnet den i-ten Buchstaben des Wortes w). Gesucht ist ein DEA M, der L akzeptiert. Wir gehen bei der Konstruktion systematisch vor:

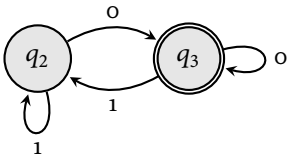
❏ Wenn als erstes Zeichen eine »0« gelesen wird, geht der Automat in einen Fangzustand, also einen nicht-akzeptierenden Zustand, der in einer Selbstschleife alle Zeichen des Alphabets einliest. Da aufgrund des falschen Präfixzeichens kein akzeptierendes Wort mehr entstehen kann, ist es unmöglich, vom Fangzustand aus in einen akzeptierenden Endzustand zu gelangen:



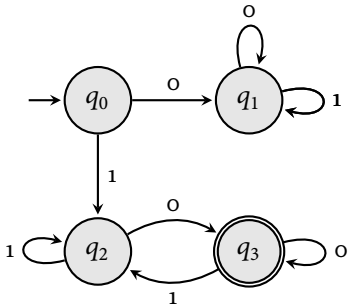
❏ Trifft der Automat auf eine »1« als erstes Zeichen, besteht die Möglichkeit, ein Wort der Sprache zu erkennen. Weitere folgende Einsen können bedenkenlos gelesen werden:



❏ Um das Wort abzuschließen, ist eine »0« notwendig. Diese kann von zusätzlichen Nullen gefolgt werden, das Wort wird weiterhin akzeptiert. Wenn nach einer »0« wieder eine »1« gelesen wird, fällt man auf ein nicht akzeptierbares Wort-Präfix zurück:



Werden alle Komponenten zusammengesetzt, ergibt sich der vollständige Automat

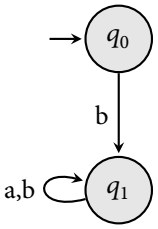


Als weiteres Beispiel betrachten wir die Sprache

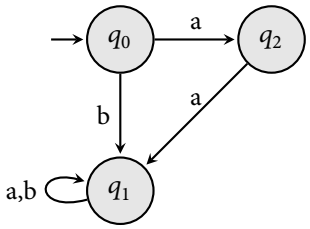
$$L = \{(ab)^n \mid n \in \mathbb{N}\}, \tag{2.5}$$

zu der wiederum ein DEA M gesucht ist, der sie akzeptiert.

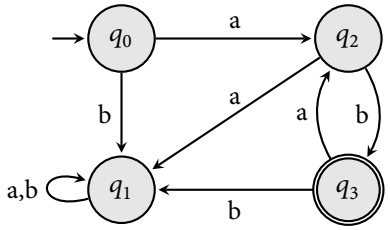
❏ Wird im ersten Schritt ein »b« gelesen, kann offenbar kein gültiges Wort mehr entstehen – der Automat kann in einen Fangzustand übergehen:



❏ Der erste Buchstabe muss ein »a« sein. Wird ein unmittelbar folgendes zweites »a« gelesen, kann ebenfalls kein korrektes Wort mehr entstehen, und der Automat geht in den Trap-Zustand q_1 über:

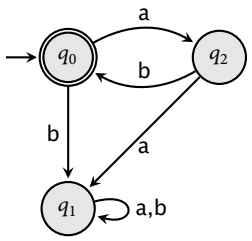


❏ Wird nach dem ersten »a« ein darauffolgendes »b« gelesen, ist ein potentiell akzeptables Wort entstanden (»ab«), der Automat geht also in einen akzeptierenden Endzustand. Folgt ein zweites »b«, kann kein korrektes Wort mehr entstehen – egal, welche Buchstaben im Anschluss folgen. Der Automat geht daher in den Trap-Zustand über:



Wird nach dem String »ab« ein »a« gelesen, besteht weiterhin die Möglichkeit, ein korrektes Wort zu erzeugen. Der Automat geht daher wie gezeigt von q_3 nach q_2 zurück.

Hinweis: Wenn der Fall $n = 0$ möglich ist, in dem das leere Wort als Element der Sprache akzeptiert wird, vereinfacht sich der Automat wie im Rand gezeigt.



2.2.4 *Formalisierung des Rechenvorgangs*

Sei M ein DEA, und sei $w = w_1 w_2 \cdots w_n$ die Eingabe mit $w_i \in \Sigma$ und $|w| = n$. M akzeptiert die Eingabe w, wenn eine Sequenz aus Zuständen $r_0, r_1, \ldots, r_n$ aus Q existiert, die folgende (informellen) Bedingungen erfüllt:

❏ Der Automat beginnt im Startzustand (in den graphischen Darstellungen durch einen Pfeil gekennzeichnet, der »aus dem Nichts« auf einen Knoten zeigt).

❏ Der Automat verhält sich in jedem Schritt gemäß der Übergangsfunktion.

❏ Die Erkennung endet in einem Endzustand.

Formal präziser (und auch knapper!) gibt man die Bedingungen wie folgt an:

❏ $r_0 = q_0$

❏ $\delta(r_i, w_{i+1}) = r_{i+1}$ für $i = 0, \cdots, n-1$

❏ $r_n \in F$

Um die Zustände des Automaten während der Rechnung beschreiben zu können, bedient man sich des Konzeptes der *Konfiguration*. Für einen DEA

Die Variablen r_i sind Platzhalter, die Zustände des Automaten repräsentieren. Wird ein Zustand beim Abarbeiten eines Wortes mehrfach durchlaufen, können auch mehrere Variablen diesen gleichen Zustand repräsentieren. Beispielsweise ist die Kette $r_0 r_1 r_2 = q_0 q_1 q_0$ möglich, wenn ein Automat im zweiten Schritt wieder in den Ausgangszustand zurückkehrt, es gilt dann $r_0 = r_2 = q_0$.

M ist sie durch aktuellen Zustand r und verbleibende Eingabe w eindeutig festgelegt:

❏ Startkonfiguration: $q_0 w_1 w_2 \cdots w_n$

❏ Übergang: $q_0 w_1 w_2 \cdots w_n \to q_1 w_2 \cdots w_n \Leftrightarrow \delta(q_0, w_1) = q_1$

Damit ist es möglich, eine alternative Definition für die Akzeptanz eines Wortes anzugeben, indem man sich der induktiven Fortsetzung der Transitionsfunktion δ bedient, die nicht nur für einen einzigen Buchstaben, sondern für ein ganzes Wort definiert sein soll:

Induktive Fortsetzung von δ auf $\hat{\delta} : Q \times \Sigma^* \to Q$:

Der Unterschied zwischen den Signaturen von δ und $\hat{\delta}$ ist vermeintlich klein, aber dennoch wichtig: Das zweite Argument von δ ist ein einzelner Buchstabe, also ein Element aus Σ. Die induktive Fortsetzung $\hat{\delta}$ erlaubt hier hingegen ein ganzes Wort, also ein Element aus Σ^*.

❏ $\hat{\delta}(q, \epsilon) = q$

❏ Für $n > 0$: $\hat{\delta}(q, w_1 w_2 \ldots w_n) = \delta(\hat{\delta}(q, w_1 \ldots w_{n-1}), w_n)$

Ein Wort wird akzeptiert, wenn die $\hat{\delta}(\cdot)$ auf einen akzeptierenden Endzustand führt:

Definition: Akzeptanz durch DEA DEA akzeptiert $w = w_1 w_2 \cdots w_n \Leftrightarrow \hat{\delta}(q_0, w) \in F$ ■

Die akzeptierte Sprache eines DEA M kann damit sehr kompakt als die Menge aller Wörter definiert werden, die von M akzeptiert werden:

Definition: Akzeptierte Sprache eines DEA

$$\begin{aligned} \mathcal{L}(M) &= \{w_1 w_2 \cdots w_n \mid \hat{\delta}(q_0, w_1 w_2 \ldots w_n) \in F\} \\ &= \{w \mid \hat{\delta}(q_0, w) \in F\} \end{aligned} \tag{2.6}$$

■

Möglicherweise wirkt die Definition etwas formal und unnötig kompliziert im Vergleich zur umgangssprachlich simplen Feststellung, dass ein Automat alle Wörter akzeptiert, für die er vom Startzustand ausgehend in einem akzeptierenden Endzustand landet. Die formale Definition wird uns aber noch wichtige Dienste leisten, wenn es darum geht, äquivalente Ansätze endlicher Automaten systematisch zu untersuchen. Vorerst genügt es, sich die Arbeitsweise der induktiven Definition anhand eines Beispiels zu verdeutlichen. Liest ein Automat das Wort $\vec{w} = w_1 w_2 w_3$, führt die Definition von $\hat{\delta}(\cdot, \cdot)$ zunächst zu folgender Formel:

$$\begin{aligned} \hat{\delta}(q_0, \vec{w}) &= \hat{\delta}(q_0, w_1 w_2 w_3) \\ &= \delta(\hat{\delta}(q_0, w_1 w_2), w_3) \\ &= \delta(\delta(\hat{\delta}(q_0, w_1), w_2), w_3) \\ &= \delta(\delta(\delta(\hat{\delta}(q_0, \epsilon), w_1), w_2), w_3) \end{aligned} \tag{2.7}$$

Der innere Kern der letzten Zeile, $\hat{\delta}(q_0, \epsilon)$ kann anhand der Definition von $\hat{\delta}(\cdot, \cdot)$ direkt aufgelöst werden, es gilt $\hat{\delta}(q_0, \epsilon) = q_0$. Setzt man das Resultat ein, erhält man die Gleichung

$$\delta(\delta(\delta(q_0, w_1), w_2), w_3), \tag{2.8}$$

deren innerer Kern $\hat{\delta}(q_0, w_1) = \delta(q_0, w_1)$ durch Nachschlagen in der Definition der Übergangsfunktion durch einen neuen Zustand ersetzt

werden. Diese Vorgehensweise verfolgt man so lange von unten nach oben, bis ein einziger, nicht mehr weiter auflösbarer Zustand ermittelt wurde, der den Endzustand des Automaten angibt. Angenommen, es gilt $\delta(q_0, w_1) = q_1$, $\delta(q_1, w_2) = q_2$ und $\delta(q_2, w_3) = q_3$. Dann löst sich die Kette wie folgt auf:

$$\begin{aligned}\hat{\delta}(q_0, \vec{w}) &= \ldots = \delta(\delta(\delta(\hat{\delta}(q_0, \epsilon), w_1), w_2), w_3) \\ &= \delta(\delta(\delta(q_0, w_1), w_2), w_3) \\ &= \delta(\delta(q_1, w_2), w_3) \\ &= \delta(q_2, w_3) = q_3 \end{aligned} \tag{2.9}$$

Das Wort wird genau denn akzeptiert, wenn q_3 ein akzeptierender Endzustand des Automaten ist. Dies bedeutet aber nichts anderes, als dass

$$\vec{w} \in \mathcal{L}(M) \Leftrightarrow \hat{\delta}(q_0, w_1 w_2 w_3) \in F. \tag{2.10}$$

2.3 Reguläre Sprachen

DEAs sind ein wichtiger Mechanismus zum Erkennen von Sprachen. Konsequenterweise vergibt man einen eigenen Namen für die daraus resultierende Sprachklasse:

Definition: Reguläre Sprache Eine Sprache, die von einem deterministischen endlichen Automaten erkannt wird, bezeichnet man als *reguläre Sprache* ∎

Reguläre Operationen führen reguläre Sprachen in reguläre Sprachen über, was noch zu beweisen sein wird. Wir fassen die entsprechenden Operationen dennoch zusammen:

Definition: Reguläre Operationen Das Zusammensetzen von Wörtern nach folgenden Regeln führt reguläre Sprachen in reguläre Sprachen über:

- ❏ *Konkatenation*: Für $u, w \in \Sigma^*$ gibt $u \cdot w = uw$ das durch Hintereinanderschreiben zusammengesetzte Wort an.
- ❏ *Vereinigung*: Für $u, w \in \Sigma^*$ gibt $u + w = u|w$ (u oder w) die Vereinigung der Wörter an.
- ❏ *Sternoperation*: $A^* \equiv \{x_1 x_2 x_3 \cdots x_k \mid k \geq 0 \wedge x_i \in A \forall i\}$ ∎

Die bisher diskutieren Möglichkeiten zur Sprachverarbeitung hinterlassen einige offene Fragen; insbesondere sind DEAs mehr auf den analytischen denn auf den generativen Aspekt fokussiert. Außerdem ist es bei komplexeren Sprachen nicht immer einfach, die erzeugte Sprache aus dem dafür zur Verfügung stehenden DEA abzuleiten. Es müssen also zwei Probleme gelöst werden:

Analytisch bedeutet, dass überprüft wird, ob ein gegebenes Wort einer Sprache genügt. Generativ beschreibt den Aspekt, konkrete Wörter aus einer wie auch immer spezifizierten Sprache zu erzeugen.

- ❏ Welche Methoden außer DEAs können verwendet werden, um eine reguläre Sprache zu generieren?
- ❏ Wie kann man sicherstellen, dass eine (wie auch immer spezifizierte Sprache) regulär ist, d.h. mit den Mitteln eines DEAs analysiert oder generiert werden kann?

2.4 *Worterzeugung und Grammatiken*

Ein Mittel, um eine Sprache zu erzeugen, ist die systematische Verwendung der mathematischen Mengenschreibweise:

Beispielsprache I: Gerade Zahlen ∈ ℕ

❏ Symbole (Ziffern): $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

❏ Sprache $L \subseteq \Sigma^*$:

$$L = \{u \mid u \in \Sigma^* \wedge \exists n \in \mathbb{N} : u = 2n\} \tag{2.11}$$

Beispielsprache II: Primzahlen

❏ Symbole wie bei Beispiel I

❏ Sprache $L \subseteq \Sigma^*$:

$$L = \{n \mid n \in \Sigma^* \wedge n \text{ prim }\} \tag{2.12}$$

Die Schreibweise erleichtert zwar die Sprachcharakterisierung gegenüber einer rein informellen Definition, bietet aber noch keine einfach zu testenden Kriterien, ob die Sprache regulär ist oder nicht. Als Ausweg bieten sich explizite Grammatiken an, wie sie auch in den Sprachwissenschaften verwendet werden. Eine Grammatik soll explizite Erzeugungsmöglichkeit für eine (formale) Sprache bieten, die die Ableitung von Wörtern aus Regeln ermöglicht. Die Grammatik in Abbildung 2.6 dient beispielsweise dazu, Wörter aus einer sehr eingeschränkten Teilmenge der deutschen Sprache zu erzeugen.

Abbildung 2.6: Beispiel-Grammatik für eine sehr eingeschränkte Variante der deutschen Sprache.

Beispiel-Grammatik

$$\begin{aligned}
\langle \textit{Satz} \rangle &\rightarrow \langle \textit{Subjekt} \rangle \langle \textit{Prädikat} \rangle \langle \textit{Objekt} \rangle \\
\langle \textit{Subjekt} \rangle &\rightarrow \langle \textit{Artikel} \rangle \langle \textit{Adjektiv} \rangle \langle \textit{Substantiv} \rangle \\
\langle \textit{Artikel} \rangle &\rightarrow \text{Der} \mid \text{Die} \mid \text{Das} \\
\langle \textit{Adjektiv} \rangle &\rightarrow \text{kleine} \mid \text{eloxierte} \mid \text{flinke} \\
\langle \textit{Substantiv} \rangle &\rightarrow \text{Eisbär} \mid \text{Mond} \mid \text{Hypertricher} \\
\langle \textit{Prädikat} \rangle &\rightarrow \text{mag} \mid \text{isst} \mid \text{induziert} \\
\langle \textit{Objekt} \rangle &\rightarrow \text{Kekse} \mid \text{Schokolade} \mid \text{Raum-Zeit-Verschiebungen}
\end{aligned}$$

Als Germanist würde man argumentieren, dass letzteres grammatikalisch falsch ist; Informatiker stellen sich auf den Standpunkt, dass der Satz im Rahmen der gegebenen Grammatik völlig okay ist, geben aber zu, dass die gezeigte Grammatik kein korrektes Modell für die deutsche Sprache ist. Eine vollständige deutsche Grammatik benötigt offenbar weitere Regeln, um beispielsweise falsche Kombinationen von Nomina und definiten Artikeln zu verhindern.

Beispielsweise können Wörter wie (Sätze) »Der kleine Eisbär mag Kekse« oder »Der eloxierte Hypertricher induziert Raum-Zeit-Verschiebungen« erzeugt werden, aber auch Sätze wie »Das flinke Mond induziert Kekse«, die aus Sicht eines Deutschlehrers nicht nur inhaltlich, sondern auch stilistisch und grammatikalisch eher zweifelhaft erscheinen.

Strukturell besteht eine Grammatik aus folgenden Komponenten:

❏ Nicht-Terminal-Symbole: $\langle \textit{Satz} \rangle$, $\langle \textit{Subjekt} \rangle$, …

❏ Terminal-Symbole: Das, Schokolade, Raum-Zeit-Verschiebung, …

❏ Produktionen: lhs → rhs

❏ Startsymbol: ⟨*Satz*⟩

Um Wörter aus einer Grammatik zu erzeugen, wird folgender Algorithmus verwendet:

1. Beginne mit dem Startsymbol S und ersetze es gemäß der Produktionen durch eine Kette aus Terminal- und Nicht-Terminal-Symbolen.
2. Ersetze alle vorhandenen Nicht-Terminal-Symbole gemäß den Produktionen.
3. Solange noch Nicht-Terminal-Symbole vorhanden: Gehe zu Schritt 2.
4. Gib den resultierenden Satz/Wort zurück.

Beispielsweise wird der Satz »Der eloxierte Hypertrichter induziert Raum-Zeit-Verschiebungen« durch folgende Ableitungssequenz erzeugt (um den Unterschied zwischen einem Ableitungsschritt und der Definition einer Produktionsregel zu verdeutlichen, verwendet man in ersterem Fall Doppelpfeile »⇒«):

⟨*S*⟩ ⇒ ⟨*Subjekt*⟩⟨*Prädikat*⟩⟨*Objekt*⟩
⇒ ⟨*Subjekt*⟩ induziert ⟨*Objekt*⟩
⇒ ⟨*Artikel*⟩⟨*Adjektiv*⟩⟨*Substantiv*⟩ induziert ⟨*Objekt*⟩
⇒ Der ⟨*Adjektiv*⟩⟨*Substantiv*⟩ induziert ⟨*Objekt*⟩
⇒ Der eloxierte ⟨*Substantiv*⟩ induziert ⟨*Objekt*⟩
⇒ Der eloxierte Hypertricher induziert ⟨*Objekt*⟩
⇒ Der eloxierte Hypertricher induziert Raum-Zeit-Verschiebungen

Sie haben sicher beobachtet, dass in jedem Schritt genau ein Nicht-Terminal-Symbol durch Auswahl (und Anwendung) einer Regeln aus den Produktionen ersetzt wurde, entweder durch ein Terminal-Wort oder eine Kette anderer Nicht-Terminal-Symbole.

Natürlich kann (und sollte) man eine Grammatik auch formal definieren, um Ihre Eigenschaften möglichst eindeutig darzulegen:

Definition: Grammatik Eine Grammatik G ist ein 4-Tupel (V, Σ, P, S) bestehend aus

❏ der endlichen Variablenmenge V (Nicht-Terminal-Symbole)

❏ dem endlichen Terminalalphabet Σ mit $V \cap \Sigma = \varnothing$

❏ der endlichen Menge $P \subseteq (V \cup \Sigma)^+ \backslash \Sigma^+ \times (V \cup \Sigma)^*$ von Produktionen (Regeln)

❏ dem Startsymbol $S \in V$ ∎

Beachten Sie, dass die Spezifikation für Produktionen eine sehr allgemeine Form darstellt. Nutzt man alle dadurch entstehenden Freiheiten, resultiert

eine Sprachklasse, die deutlich über den Möglichkeiten regulärer Sprachen bzw. deterministischer endlicher Automaten hinausgeht. Wir werden daher einige Einschränkungen einführen, um (unter anderem) explizite Grammatiken für reguläre Sprachen angeben zu können.

Zuvor sollen die Möglichkeiten von Grammatiken allerdings anhand einiger Beispiele demonstriert werden. In Abbildung 2.7 ist eine Grammatik angegeben, die arithmetische Ausdrücke beschreibt (wir geben keine vollständige Liste von Produktionen zur Definition aller möglichen Zahlen 0, 1, 2, …und Variablen x, y, z, …an).

$\langle \mathrm{E} \rangle \to \langle \mathrm{T} \rangle$ $\quad$ $\langle \mathrm{E} \rangle \to \langle \mathrm{E} \rangle + \langle \mathrm{T} \rangle$
$\quad$ $\langle \mathrm{E} \rangle \to \langle \mathrm{E} \rangle - \langle \mathrm{T} \rangle$
$\langle \mathrm{T} \rangle \to \langle \mathrm{F} \rangle$ $\quad$ $\langle \mathrm{F} \rangle \to \langle \mathrm{T} \rangle \times \langle \mathrm{F} \rangle$
$\quad$ $\langle \mathrm{F} \rangle \to \langle \mathrm{T} \rangle / \langle \mathrm{F} \rangle$
$\langle \mathrm{F} \rangle \to \mathtt{0}$ $\quad$ $\mathrm{F} \to \mathtt{1}$
…
$\langle \mathrm{F} \rangle \to \mathtt{x}$ $\quad$ $\langle \mathrm{F} \rangle \to \mathtt{y}$
$\langle \mathrm{F} \rangle \to (\langle \mathrm{F} \rangle)$

Abbildung 2.7: Beispiel-Grammatik für korrekte arithmetische Ausdrücke.

Der (korrekte) arithmetische Ausdruck $x + 2 \times (3 + 2)$ wird beispielsweise durch die Ableitungskette

$$\begin{aligned}
\langle E \rangle &\Rightarrow \langle E \rangle + \langle T \rangle \Rightarrow \langle E \rangle + \langle T \rangle \times \langle F \rangle \\
&\Rightarrow \langle E \rangle + \langle T \rangle \times (\langle E \rangle) \Rightarrow \langle E \rangle + \langle T \rangle \times (\langle E \rangle + \langle T \rangle) \\
&\Rightarrow \langle E \rangle + \langle T \rangle \times (\langle T \rangle + \langle T \rangle) \Rightarrow \langle E \rangle + \langle T \rangle \times (\langle F \rangle + \langle T \rangle) \\
&\Rightarrow \langle E \rangle + \langle T \rangle \times (\langle F \rangle + \langle F \rangle) \Rightarrow \langle E \rangle + \langle F \rangle \times (\langle F \rangle + \langle F \rangle) \\
&\Rightarrow \langle T \rangle + \langle F \rangle \times (\langle F \rangle + \langle F \rangle) \Rightarrow \langle F \rangle + \langle F \rangle \times (\langle F \rangle + \langle F \rangle) \\
&\Rightarrow x + \langle F \rangle \times (\langle F \rangle + \langle F \rangle) \Rightarrow x + 2 \times (\langle F \rangle + \langle F \rangle) \\
&\Rightarrow x + 2 \times (3 + \langle F \rangle) \Rightarrow x + 2 \times (3 + 2) \qquad (2.13)
\end{aligned}$$

aus der Grammatik erzeugt, wobei wieder in jedem Schritt genau ein Nicht-Terminal-Symbol anhand der Produktionen der Grammatik ersetzt wurde.

Die Grammatik erlaubt nicht, inkorrekte arithmetische Ausdrücke zu beschreiben, was dem Leser beispielsweise klar wird, wenn er den Ausdruck)7 × +3) produzieren möchte – es ist nicht möglich, eine passende Ableitungskette zu finden.

Als weiteres Beispiel betrachten wir die Sprache

$$L = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}\}, \qquad (2.14)$$

die durch die etwas kompliziertere Grammatik

$$\begin{aligned}
\langle S \rangle &\to \mathtt{a}\langle S \rangle\langle B \rangle\langle C \rangle & \langle S \rangle &\to \mathtt{a}\langle B \rangle\langle C \rangle \\
\langle C \rangle\langle B \rangle &\to \langle B \rangle\langle C \rangle & \mathtt{a}\langle B \rangle &\to \mathtt{ab} \\
\mathtt{b}\langle B \rangle &\to \mathtt{bb} & \mathtt{b}\langle C \rangle &\to \mathtt{bc} \\
\mathtt{c}\langle C \rangle &\to \mathtt{cc} & & \qquad (2.15)
\end{aligned}$$

generiert wird. Warum die Grammatik tatsächlich funktioniert, veranschaulichen wir anhand eines Beispiels, das das Wort »aaabbbccc« (also $\mathtt{a}^3\mathtt{b}^3\mathtt{c}^3$) aus dem Startsymbol ableitet:

$$\begin{aligned}
\langle S \rangle &\Rightarrow \mathtt{a}\langle S \rangle\langle B \rangle\langle C \rangle \Rightarrow \mathtt{aa}\langle S \rangle\langle B \rangle\langle C \rangle\langle B \rangle\langle C \rangle \\
&\Rightarrow \mathtt{aaa}\langle B \rangle\langle C \rangle\langle B \rangle\langle C \rangle\langle B \rangle\langle C \rangle \\
&\Rightarrow \mathtt{aaa}\langle B \rangle\langle B \rangle\langle C \rangle\langle C \rangle\langle B \rangle\langle C \rangle \\
&\Rightarrow \mathtt{aaa}\langle B \rangle\langle B \rangle\langle C \rangle\langle B \rangle\langle C \rangle\langle C \rangle \\
&\Rightarrow \mathtt{aaa}\langle B \rangle\langle B \rangle\langle B \rangle\langle C \rangle\langle C \rangle\langle C \rangle \\
&\Rightarrow \mathtt{aaab}\langle B \rangle\langle B \rangle\langle C \rangle\langle C \rangle\langle C \rangle \\
&\Rightarrow \mathtt{aaabb}\langle B \rangle\langle C \rangle\langle C \rangle\langle C \rangle
\end{aligned}$$

$$\Rightarrow \texttt{aaabbb}\langle C\rangle\langle C\rangle\langle C\rangle$$
$$\Rightarrow \texttt{aaabbbc}\langle C\rangle\langle C\rangle$$
$$\Rightarrow \texttt{aaabbbcc}\langle C\rangle$$
$$\Rightarrow \texttt{aaabbbccc} \tag{2.16}$$

2.5 Die Chomsky-Hierarchie

2.5.1 Definition

Eine Einteilung aller möglichen Grammatiken in eine Hierarchie mit steigenden Einschränkungen bezüglich der erlaubten Produktionsformen wurde 1956 von Noam Chomsky eingeführt und ist seither als *Chomsky-Hierarchie* bekannt:

Noam Chomsky (1928–)
Als Linguistiker lieferte Chomsky wegweisende Arbeiten zum Konzept einer Universalgrammatik, die biologisch vererbte Strukturen des menschlichen Sprachvermögens beschreiben soll. Die einzelnen Klassen der Chomsky–Hierarchie werden in diesem Umfeld zur Beschreibung verschiedener linguistischer Konzepte wie Morphologie oder Syntax verwendet. Chomsky erlangte durch sein Eintreten für freie Rede und gegen Zensur hohe Bekanntheit als politischer Aktivist.
Achtung: Manche Autoren verwenden den Begriff Phrasenstrukturgrammatik für Typ 1/2-Grammatiken, insbesondere in der germanistischen Linguistik.

Chomsky-Hierarchie

Die Chomsky-Hierarchie ist eine Einteilung der Menge aller Sprachen in verschiedene (absteigend) mächtige Klassen. Folgende Beschränkungen werden allen Regeln der Form $l \to r$ je nach Klasse auferlegt:

- ❏ *Phrasenstrukturgrammatiken (Typ 0)*: Keine Regeleinschränkungen, d.h. rechte und linke Seite dürfen sich aus beliebigen Kombinationen von Terminal- und Nicht-Terminal-Symbolen zusammensetzen.
- ❏ *Kontextsensitive Grammatiken (Typ 1)*: Es gilt $|l| \leq |r|$.
- ❏ *Kontextfreie Grammatiken (Typ 2)*: $l \in V$
- ❏ *Reguläre Grammatik (Typ 3)*: $r \in \Sigma \cup \Sigma V$

Für Typ n-Grammatiken gelten jeweils die Beschränkungen von Typ $n-1$-Grammatiken.

Wenn G eine Typ n-Grammatik ist, bezeichnet man $\mathcal{L}(G)$ als Typ n-Sprache (dito: kontextfreie Sprache etc.)

2.5.2 Anmerkungen zu den Sprachklassen

Die verschiedenen Sprachklassen der Chomsky-Hierarchie besitzen einige interessante Eigenschaften:

- ❏ *Phrasenstrukturgrammatiken (Typ 0)*: Jede Grammatik ist per Definitionem in der Menge enthalten (*Aber*: Es gibt Sprachen, die prinzipiell nicht durch eine Grammatik zu beschreiben sind – die Menge aller Sprachen ist daher größer als die Menge der Typ 0-Sprachen!).
- ❏ *Kontextsensitive Grammatiken (Typ 1)*: Die Anwendung einer Produktion führt niemals zur Verkürzung des abgeleiteten Wortes; die Länge der abgeleiteten Teilwörter ist also monoton steigend.
- ❏ *Reguläre Grammatik*: Die Ableitung eines Wortes verläuft *rechtslinear*, da ein Wachstum der abgeleiteten Teilwörter aufgrund der Form der

Produktionen nur nach rechts hin stattfinden kann.

Regeln, die ϵ-Produktionen enthalten (also ein leeres Wort erzeugen können), bereiten ein Problem mit obiger Definition: Wegen $|l| \leq |r|$ kann in kontextsensitiven (und damit kontextfreien und regulären) Sprachen keine Regel der Form $N \to \epsilon$ auftreten. Wenn $\epsilon \in \mathcal{L}(G)$ erwünscht sein soll, was durchaus der Fall sein kann, führt man die ϵ-Sonderregel ein, die zusätzlich zu den Chomsky-Bedingungen gilt:

Beispielsweise kann eine leere Eingabe korrekt sein, wenn ein Webformular Platz für Freitext-kommentare bietet: Es wird in den meisten Fällen legitim sein, keinen Kommentar anzugeben, weshalb eine Grammatik diesen Fall berücksichtigen können muss.

ϵ-Sonderregel

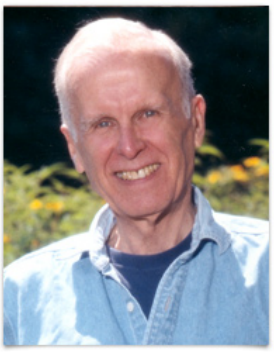
Die Regel $S \to \epsilon$ ist ausschließlich für das Startsymbol (aber kein anderes Nicht-Terminal-Symbol) explizit zugelassen, wenn $\epsilon \in \mathcal{L}(G)$ erwünscht ist, wenn das leere Wort also Teil der Sprache sein soll.

Wie in der Vorlesung besprochen kann jede Grammatik auf eine Form gebracht werden, in der eine ϵ-Regel nur beim Startsymbol auftritt, sofern das leere Wort Mitglied der Sprache sein soll. Trifft letzteres nicht zu, kann die Grammatik auch völlig ohne Verwendung von ϵ-Symbolen angegeben werden. Allerdings ist es häufig einfacher, mit ϵ-Symbolen zu arbeiten, wovon wir an verschiedenen Stellen Gebrauch machen werden – wohl wissend, dass man prinzipiell immer eine äquivalente ϵ-freie Grammatik angeben könnte.

2.5.3 *Grundprobleme formaler Sprachen*

Folgende Probleme sind die grundlegenden Probleme, die im Zusammenhang mit formalen Sprachen betrachtet werden müssen. Wie sich im weiteren Verlauf der Vorlesung herausstellen wird, sind manche davon maschinell leicht handhabbar. Andere sind handhabbar, aber nur sehr ineffizient, und wieder andere sind je nach betrachteter Sprachklasse prinzipiell unlösbar:

Eine andere weit verbreitete Klasse von Problemen sind Optimierungsprobleme, in der nicht nur nach einer Lösung eines Problems verlangt wird, sondern zusätzlich der minimale oder maximale Wert aus der Menge aller möglichen Lösungen ermittelt werden soll.



John P. Backus (1924–2007) Backus hat die Programmiersprache Fortran verbrochen und die ersten Compiler dafür implementiert. Nachdem man ihm dafür 1977 den Turing-Preis verliehen hat, nutzte er die Gelegenheit zum Versuch, die Wissenschaft davon zu überzeugen, dass künftige Sprachen nichts, aber auch gar nichts mit Fortran zu tun haben sollten (Bildquelle: computerhistory.org).

- *Wortproblem*: Gegeben Wort $w \in \Sigma^*$ und Sprache $L \in \Sigma^*$: Gilt $w \in L$ oder $w \notin L$?
- *Leerheitsproblem*: Enthält eine Sprache L kein Wort, d.h. gilt $L = \varnothing$?
- *Endlichkeitsproblem*: Besitzt L nur endlich viele Elemente, d.h. gilt $|L| \neq \infty$?
- *Schnittproblem*: Gilt für zwei Sprachen L_1, L_2, dass $L_1 \cap L_2 = \varnothing$?
- *Äquivalenzproblem*: Sind zwei Sprachen L_1, L_2 gleich, d.h. gilt $L_1 = L_2$?

Jedes der gezeigten Probleme ist ein *Entscheidungsproblem*, das als mögliche Antworten nur »Ja« oder »Nein« besitzt. Praktische Relevanz besitzen vor allem das Wort- und das Äquivalenzproblem: Ersteres ist ohnehin das zentrale Problem der Sprachverarbeitung; letzteres ist beispielsweise wichtig, wenn unterschiedliche Implementierungen einer Sprache durch verschiedene Grammatiken G und G' miteinander verglichen werden sollen, um sicherzustellen, dass beide die gleiche Sprache definieren.

Um das Wortproblem für eine Grammatik G zu lösen, muss festgestellt werden, ob das Wort aus G ableitbar ist oder nicht. Deshalb definieren wir den Begriff der Ableitbarkeit formal:

Definition: Ableitbarkeit Sei $l \in (V \cup \Sigma)^+ \backslash \Sigma^+$ und $r \in (V \cup \Sigma)^*$. Dann ist r aus l

- ❏ *direkt ableitbar*, wenn es eine Regel $(u, v) \in P$ und $\alpha, \beta \in (V \cup \Sigma)^*$ gibt, so dass
$$l = \alpha u \beta, r = \alpha v \beta. \tag{2.17}$$
Formal: $l \Rightarrow r$.
- ❏ *indirekt ableitbar*, wenn es durch endlich viele Ableitungsschritte aus l erzeugbar ist.
Formal: $l \overset{*}{\Rightarrow} r$ (reflexiv-transitive Hülle der Ableitungsrelation $\Rightarrow$, siehe Tafel)

■

Die erzeugte Sprache einer Grammatik besteht aus allen Wörtern, die sich aus der Grammatik ableiten lassen. Dies drückt man formal folgendermaßen aus:

Erzeugte Sprache $\mathcal{L}(G)$

$$\mathcal{L}(G) \equiv \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\} \tag{2.18}$$

Die Ähnlichkeit der Definition zum Fall eines DEAs ist sehr hoch: Die Sprache eines DEAs setzt sich aus allen Wörtern zusammen, die den Automat in endlich vielen Schritten ausgehend vom Startzustand in einen akzeptierenden Endzustand überführen. Im Fall einer Grammatik müssen ausgehend vom Startsymbol endlich viele Ersetzungsschritte durchgeführt werden.

Achtung: Grammatiken besitzen kein Analogon zu akzeptierenden Endzuständen; der Ableitungsprozess ist implizit beendet, wenn keine weitere Produktionsregel mehr angewendet kann und sich das Wort nur aus Terminalsymbolen zusammensetzt.

2.5.4 *Erweiterte Backus-Naur-Form*

Um die Spezifikation von Grammatiken für praktische Zwecke zu erleichtern, verwendet man die erweiterte Backus-Naur-Form (EBNF) die Kurzschreibweisen für häufig gebrauchte Konstrukte definiert:

- ❏ $A \to \beta_1, A \to \beta_2, \ldots, A \to \beta_n$ wird ersetzt durch $A \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$.
- ❏ $A \to \alpha[\beta]\gamma$ steht für die beiden Regeln
$$\begin{aligned} A &\to \alpha\gamma \\ A &\to \alpha\beta\gamma. \end{aligned} \tag{2.19}$$
- ❏ $A \to \alpha\{\beta\}\gamma$ steht für die Regeln
$$\begin{aligned} A &\to \alpha\gamma \\ A &\to \alpha B\gamma \\ B &\to \beta \\ B &\to \beta B. \end{aligned} \tag{2.20}$$

Die EBNF-Form ist häufig bei der Definition von Programmiersprachen anzutreffen, da vor allem komplexe Grammatiken dadurch leichter und übersichtlicher zu spezifizieren sind.

Peter Naur (1928–)
Naur gesellt sich in die lange Reihe der Turing–Preisträger, auf deren Arbeit die Themen der Vorlesung beruhen. Neben seinen Arbeiten zu Algol 60, die praktisch alle modernen Programmiersprachen substantiell beeinflusst hat, ist er für seine Beiträge zur Softwarearchitektur bekannt.
Das Symbol »|« repräsentiert eine Oder–Auswahl zwischen mehreren Regelalternativen, von denen genau eine ausgewählt wird.
Kurz gesprochen stehen eckige Klammern stehen für eine 0– oder 1–fache Wiederholung der eingeklammerten Regel; Geschweifte Klammern repräsentieren eine beliebig häufige Wiederholung (einschließlich 0–fach).

2.5.5 *Ein- und Mehrdeutigkeit*

Grammatiken müssen nicht notwendigerweise die Eigenschaft besitzen, dass für jedes erzeugbare Wort genau ein Ableitungsbaum existiert; es ist durchaus möglich, dass ein- und dasselbe Wort über verschiedene Pfade erzeugt werden kann. Trifft dies allerdings für *kein* Wort der Sprache zu, spricht man von einer *eindeutigen Grammatik*:

Definition: Eindeutige Grammatik Eine Grammatik G heißt eindeutig, wenn alle Ableitungen eines Wortes $w \in \mathcal{L}(G)$ immer zu genau einem Syntaxbaum führen. Eine nicht eindeutige Grammatik heißt mehrdeutig. ∎

Nachdem eine Grammatik entweder eindeutig oder mehrdeutig ist, folgt die Definition der Mehrdeutigkeit trivial als Gegenteil der Definition der Eindeutigkeit. Folgende Punkte sind zu beachten:

Nachdem eindeutige Grammatiken leichter handzuhaben sind, versucht man in technischen Anwendungen der Theorie (beispielsweise Compilerbau) bevorzugt mit solchen Grammatiken zu arbeiten. Compiler-Compiler wie Bison oder ANTLR, die einen Compiler aus einer gegebenen Grammatik erzeugen, geben beispielsweise explizite Warnungen aus, wenn eine mehrdeutige Grammatik spezifiziert wird.

- ❏ Die Mehrdeutigkeit ist eine Eigenschaft der Grammatik, nicht der Sprache!
- ❏ Eine mehrdeutige Grammatik ist häufig in eine eindeutige Grammatik G' mit $\mathcal{L}(G) = \mathcal{L}(G')$ überführbar.
- ❏ Es existieren Sprachen, die ausschließlich durch mehrdeutige Grammatiken erzeugt werden können (*inhärent mehrdeutige* Sprachen).

Zur Illustration betrachten wir die Grammatik

$$\begin{aligned} \langle S\rangle &\to \mathtt{a}\langle B\rangle \\ \langle S\rangle &\to \langle A\rangle \mathtt{c} \\ \langle A\rangle &\to \mathtt{ab} \\ \langle B\rangle &\to \mathtt{bc} \end{aligned} \tag{2.21}$$

Die Sprache ist nicht inhärent mehrdeutig, da es sehr einfach ist, eine eindeutige Grammatik anzugeben: Die einzelne Regel ⟨S⟩ → `abc` reicht dazu aus.

Das Wort »abc« kann durch die beiden in Abbildung 2.8 gezeigten inäquivalenten Ableitungsbäume aus der Grammatik erzeugt werden.

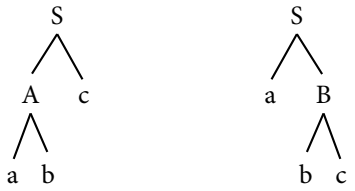


Abbildung 2.8: Beispiele für nichtäquivalente Ableitungsbäume.

2.6 *Reguläre Sprachen und endliche Automaten*

Wir hatten weiter oben bereits angesprochen, dass die Erkennbarkeit durch deterministische endliche Automaten eine mögliche Definition der Klasse der regulären Sprachen ist. Im Folgenden soll eine alternative Charakterisierung regulärer Sprachen über Grammatiken eingeführt und die Äquivalenz zur bisherigen Definition bewiesen werden.

Definition: Reguläre Sprachen Eine Grammatik $G(V, \Sigma, P, S)$ ist regulär, wenn für alle Regeln $l \to r$ gilt:

- ❑ $|l| \leq |r|$, die rechte Regelseite ist also mindestens so lang wie die linke Seite.
- ❑ $l \in V$ und $r \in (\Sigma \cup \Sigma V)$, die linke Seite besteht also aus einem einzigen NichtTerminalsymbol, und die rechten Regelseiten sind eingeschränkt auf ein Terminalzeichen oder ein Terminalzeichen gefolgt von einem Nicht-Terminalsymbol. ∎

Reguläre Sprachen sind in praktischen Anwendungen omnipräsent:

- ❑ In praktisch allen Programmiersprachen »verbaut«: Perl, Grep, Java (`java.util.regex`), Go (`regexp`), D (`std.regex`), C# (`System.Text.RegularExpressions`), PHP (`preg_*`), …
- ❑ Systemadministration, Konfigurationsdateien, Eingabechecker, Datenmanipulation, …
- ❑ Achtung: »Reguläre Ausdrücke« sind in Programmiersprachen häufig über obige Definition hinaus erweitert, können also eine Übermenge der hier behandelten Sprachmenge erkennen. Allerdings hat sich *regulärer Ausdruck* als fester Terminus in der Programmiersprachenwelt etabliert.

D ist einer der Nachfolger von C, die wiederum Nachfolger der Sprache B ist. Deren Vorgänger war allerdings nicht A, sondern BCPL.

Als Beispiel für eine reguläre Sprache betrachten wir die Grammatik $G = (\{\langle S\rangle, \langle B\rangle, \langle C\rangle\}), \{\mathrm{a}, \mathrm{b}\}, P, \langle S\rangle$ mit den Produktionen

$$\begin{aligned} \langle S\rangle &\to \mathtt{a}\langle B\rangle \\ \langle B\rangle &\to \mathtt{b} \\ \langle B\rangle &\to \mathtt{b}\langle C\rangle \\ \langle C\rangle &\to \mathtt{a}\langle B\rangle \end{aligned} \tag{2.22}$$

Nachdem alle Produktionen den beschriebenen Einschränkungen genügen, ist die Sprache per Definition regulär. Die erzeugte Wortmenge ist

$$\mathcal{L}(G) = \{(\mathrm{ab})^n \mid n \in \mathbb{N}\}, \tag{2.23}$$

wie man sich anhand folgender Beispiele klarmacht, die Wörter aus dem Startsymbol ableiten:

1. $\langle S\rangle \Rightarrow \mathrm{a}\langle B\rangle \Rightarrow \mathrm{ab}$
2. $\langle S\rangle \Rightarrow \mathrm{a}\langle B\rangle \Rightarrow \mathrm{ab}\langle C\rangle \Rightarrow \mathrm{aba}\langle B\rangle \Rightarrow \mathrm{abab}\langle C\rangle \Rightarrow \mathrm{ababa}\langle B\rangle \Rightarrow \mathrm{ababab} = (\mathrm{ab})^3$

Der Syntaxbaum für das zweite Beispiel ist in Abbildung 2.9 gezeigt. Reguläre Grammatiken bezeichnet man als *rechtslineare* Grammatiken; dies wird visuell anhand der Struktur des gezeigten Ableitungsbaumes klar: Die Wachstumsrichtung des Baums ist von links nach rechts, wobei Buchstaben bis auf den letzten immer an der linken Seite des Baums hängen.

Um die Behauptung zu beweisen, dass reguläre Sprachen und die erkannte Sprachklasse endlicher Automaten zueinander äquivalent sind, müssen wir eine Konstruktion finden, die beide Beschreibungsmöglichkeiten ineinander überführt. Die zugrundeliegende Konstruktion ist es, Zustände des DEA als Nicht-Terminal-Symbole auffassen und eine Grammatik entsprechend den Übergängen zu gestalten.

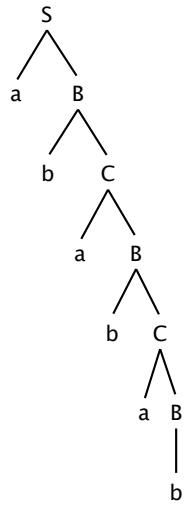
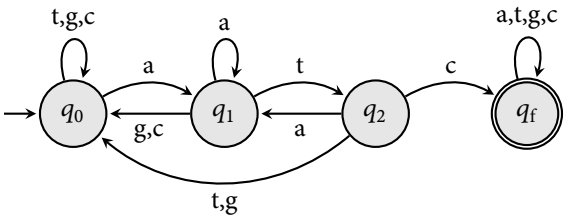


Abbildung 2.9: Ableitungbaum für das Wort $(\mathrm{ab})^3$ aus Grammatik (2.22).

Wir betrachten die Konvertierung anhand eines Beispiels aus der Biologie, wo man oft vor dem Problem steht, Muster in Genomsequenzen zu finden. Angenommen, die Basen der Desoxyribonukleinsäure (Adenin, Thymin, Guanin, Cytosin) sind in einer Sequenz durch die Buchstaben »a«, »t«, »g«, »c« beschrieben. Man möchte das Muster »atc« an einer *beliebigen* Position in der Sequenz finden, da man vermutet, dass diese Kombination Superkräfte beim Lösen von Differenzialgleichungen für Eintagsfliegen verspricht. Man kann das Problem sehr effizient mit folgendem Automaten lösen:

Effizienz ist beim Mining von DNS besonders wichtig, da große Datenmengen verarbeitet werden müssen.



Bekanntlich interessieren sich Informatiker nicht für Eintagsfliegen mit Superkräften, sondern für Grammatiken. Glücklicherweise kann man den Automaten unmittelbar in eine Grammatik verwandeln, indem man

1. jedem Zustand ein Nicht-Terminal-Symbol zuweist. Der Startzustand des Automaten entspricht dann dem Startsymbol der Grammatik.
2. jedem Übergang des Automaten eine Produktion der Grammatik zuweist: Der Übergang $\delta(A, \mathrm{a}) = B$ wird beispielsweise durch die Regel $\langle A \rangle \to \mathtt{a}\langle B \rangle$ repräsentiert.

   Wenn ein Zustand in der Menge der akzeptierenden Endzustände enthalten ist, führt man zusätzlich einen ϵ-Übergang ein, um die Produktion von Wörtern an der passenden Stelle abbrechen zu können.

Alle erzeugten Regeln sind regulär, also ist die erzeugte Sprache eine reguläre Sprache. Die reguläre Grammatik für obiges Beispiel ist

$$\begin{aligned}
\langle Q \rangle_0 &\to \mathtt{a}\langle Q \rangle_1 \mid \mathtt{t}\langle Q \rangle_0 \mid \mathtt{g}\langle Q \rangle_0 \mid \mathtt{c}\langle Q \rangle_0 \\
\langle Q \rangle_1 &\to \mathtt{a}\langle Q \rangle_1 \mid \mathtt{t}\langle Q \rangle_2 \mid \mathtt{g}\langle Q \rangle_0 \mid \mathtt{c}\langle Q \rangle_0 \\
\langle Q \rangle_2 &\to \mathtt{c}\langle Q \rangle_\mathrm{f} \mid \mathtt{a}\langle Q \rangle_1 \mid \mathtt{t}\langle Q \rangle_0 \mid \mathtt{g}\langle Q \rangle_0 \\
\langle Q \rangle_\mathrm{f} &\to \mathtt{a}\langle Q \rangle_\mathrm{f} \mid \mathtt{t}\langle Q \rangle_\mathrm{f} \mid \mathtt{g}\langle Q \rangle_\mathrm{f} \mid \mathtt{c}\langle Q \rangle_\mathrm{f} \mid \epsilon
\end{aligned} \tag{2.24}$$

Allgemein gilt folgendes Theorem:

Theorem: Endliche Automaten und reguläre Grammatiken Zu jedem deterministischen endlichen Automaten M gibt es eine reguläre Grammatik G mit $\mathcal{L}(G) = \mathcal{L}(M)$.

Um das Theorem zu beweisen, müssen wir zunächst die oben beschrieben Transformation formalisieren. Wir gehen dabei von einem deterministischen endlichen Automaten $M = (Q, \Sigma, \delta, q_0, F)$ aus, der die Sprache $A = \mathcal{L}(M)$ erkennt. M soll in eine Grammatik $G = (V, \Sigma, P, S)$ transformiert werden.

Das Alphabet Σ ist offenbar für M und G identisch, da die Konstruktion ansonsten nicht sinnvoll ist. Die Menge aller Zustände wird »uminterpretiert« als Menge von Nicht-Terminal-Symbolen; es gilt also

$$V = Q$$

$$S = q_0, \tag{2.25}$$

wobei wir das Startsymbol der Grammatik mit dem Startzustand des Automaten identifiziert haben.

Jeder Übergang $\delta \in P$ wird gemäß der Transformation

$$\delta(q, a) = q' \Rightarrow \langle q \rangle \rightarrow \mathtt{a}\langle q' \rangle \tag{2.26}$$

in eine Produktion der Grammatik überführt. Falls $q' \in F$, wird zusätzlich die Produktionsregel $\langle q \rangle \rightarrow a$ eingeführt, es gilt also

$$q' \in F \Rightarrow \langle q \rangle \rightarrow a. \tag{2.27}$$

Um die Äquivalenz von M und G zu zeigen, muss jedes Wort, das vom Automaten M erkannt wird, auch von der Grammatik G erkannt werden – und umgekehrt. Es muss also $\mathcal{L}(M) = \mathcal{L}(G)$ gelten. Dazu betrachten wir ein beliebiges Wort $\vec{x} = a_1 a_2 \dots a_n \in \mathcal{L}(M)$. Wir rechnen

$$\begin{aligned} x \in \mathcal{L}(M) &\Leftrightarrow \exists q_0, q_1, \dots, q_n \in Q^n \text{ mit Bedingung 1} && (2.28)\\ &\Leftrightarrow \exists \langle q \rangle_0 \langle q \rangle_1 \dots \langle q \rangle_n \in V^n \text{ mit Bedingung 2} && (2.29)\\ &\Leftrightarrow x \in \mathcal{L}(G) && (2.30) \end{aligned}$$

wobei folgende Nebenbedingungen gelten:

- Bedingung 1 in Gleichung (2.28) fordert, dass q_0 der Startzustand und q_n ein Endzustand des Automaten ist, also $q_n \in F$ gilt. Weiterhin gilt $\delta(q_{i-1}, a_i) = q_i$ für alle $i \in [1, n]$.
- Bedingung 2 aus Gleichung (2.29) fordert, dass $\langle q \rangle_0$ der Startzustand von G ist. Nachdem man damit gemäß der oben definierten Transformation

$$\langle q \rangle_0 \Rightarrow a_1 \langle q \rangle_1 \Rightarrow a_1 a_2 \langle q \rangle_2 \Rightarrow \dots \Rightarrow a_1 a_2 \dots a_n \tag{2.31}$$

  folgern kann, folgt Zeile (2.30).

Effektiv haben wir damit $x \in \mathcal{L}(M) \Leftrightarrow x \in \mathcal{L}(A)$ gezeigt, da die Überlegung sowohl von links nach rechts als auch von rechts nach links gelesen werden kann und in beiden Fällen korrekt ist. Da daraus $\mathcal{L}(M) = \mathcal{L}(G)$ folgt, ist die Äquivalenz zwischen den Konstruktionen bewiesen. □

Achtung: Auch wenn wir die Äquivalenz $\mathcal{L}(G) = \mathcal{L}(M)$ gezeigt haben, also sowohl $\mathcal{L}(G) \subseteq \mathcal{L}(M)$ und $\mathcal{L}(M) \subseteq \mathcal{L}(G)$ gilt, ist das Theorem an sich nur für *eine* Transformationsrichtung korrekt, da wir noch keine Möglichkeit angeben können, aus einer regulären Grammatik G einen endlichen Automaten M zu konstruieren! Die Rückwärtsrichtung verbleibt zu zeigen; dazu wird es allerdings zunächst notwendig sein, neue Konzepte einzuführen, um die Aufgabe zu erleichtern. Um die Übersicht zu bewahren, fassen wir die bislang gewonnenen Erkenntnisse zusammen:

- Reguläre Sprachen sind auf unterschiedliche Arten erzeugbar.
- Die erkannte Sprache jedes endlichen Automaten kann alternativ durch eine reguläre Grammatik beschrieben werden, die ausgehend vom Automaten systematisch konstruiert werden kann.

2.7 Nicht-deterministische endliche Automaten (NEAs)

Wenn es zu jedem deterministischen endlichen Automaten M eine reguläre Grammatik G mit $\mathcal{L}(G) = \mathcal{L}(M)$ gibt, liegt es nahe, dass auch die Umkehrung – L regulär $\Rightarrow$ $\exists$ DEA für L – gilt. Die Aussage ist zwar wahr, kann aber nicht auf dem naiven Weg bewiesen werden, indem man die Regeln der Grammatik 1:1 in Regeln eines DEAs übersetzt. Das Problem: Die Transitionsfunktion eines DEAs gibt den Übergang, der nach einem gelesenen Zeichen ausgeführt werden muss, eindeutig vor – bei einer Grammatik ist es aber möglich, mehrere Ableitungen aus einer einzigen Regel zu erzeugen, die zunächst alle den gleichen Buchstaben produzieren, dann aber unterschiedliche Pfade einschlagen.

Die Eindeutigkeit der Transitionsfunktion ist für den Anteil deterministisch im Namen DEA verantwortlich.

2.7.1 Definition von NEA und ϵ-NEA

Eine mögliche Lösung ist es, DEAs auf Nichtdeterminismus zu erweitern! Damit führt man einen *Nicht-deterministischen endlichen Automaten (NEA)* ein, der leicht zu jeder Grammatik konstruiert werden kann, indem man die Nicht-Terminalsymbole als Zustände identifiziert und für jede Produktion eine Transition anbietet. Lediglich die Definition der Übergangsfunktion muss erweitert werden. Während die Transitionsfunktion $\delta : Q \times \Sigma \to Q$ des DEA eine *eindeutige* Abbildung des Tupels (Q, Σ) auf einen Folgezustand erledigt und damit zu eindeutigen Rechenschritten führt, verwendet ein NEA Transitionsfunktionen folgender Form:

$\mathcal{P}(Q)$ gibt die aus der Mathematik bekannte Potenzmenge der Zustandsmenge Q an; siehe hierzu insbesondere Übung 5.9.

Nicht-Deterministische Übergangsfunktion

$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

Jedes Tupel Q, Σ wird dabei auf *mehrere* potentielle Folgezustände abgebildet. Dies führt zu mehrdeutige Rechenschritte und damit zu einem zwar nützlichen, aber unphysikalischen Berechnungsmodell.

NEAs sind häufig einfacher als ihre deterministischen Äquivalente, weshalb man sie in der Praxis beispielsweise als Grundlage für lexikalische Scanner wie flex (siehe `flex.sourceforge.net`) verwendet.

Wir betrachten zwei NEAs, die sich gut zum Erkennen folgender Sprachen eignen und deutlich einfacher als die entsprechenden DEAs aufgebaut sind:

- Wörter $x \in \{0, 1\}^*$, die mit 00 enden oder gleich 0 sind. Die Sprache wird durch den nicht-deterministischen endlichen Automaten aus Abbildung 2.10 erkannt.

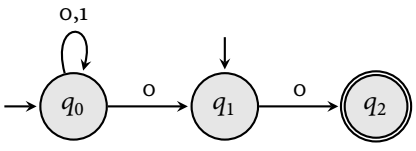


Abbildung 2.10: Nicht-Deterministischer endlicher Automat für die Sprache aller binären Wörter, die mit »00« enden oder gleich 0 sind.

Der Zustand q_0 enthält einen nicht-deterministischen Übergang, da der Automat nach Lesen der Ziffer »0« sowohl in q_0 verbleiben wie auch nach q_1 übergehen kann.

Nachdem wir uns vom klassischen Automatenkonzept verabschiedet haben, können wir an dieser Stelle zwei weitere Vereinfachungen vornehmen:

1. Ein NEA darf mehr als einen Startzustand besitzen; man kann entsprechend wählen, wo man die Verarbeitung eines Wortes beginnt.
2. Der Automat muss nicht vollständig definiert sein, man braucht also nicht für jedes $\sigma \in \Sigma$ einen expliziten Übergang für jeden Zustand zu spezifizieren. Wird ein Zeichen gelesen, für das kein Übergang vorhanden ist, gilt das Wort als abgelehnt.

❏ Wörter $x \in \{0,1\}^*$, deren k-letzte Ziffer eine »0« ist. Die Sprache wird durch den nicht-deterministischen endlichen Automaten aus Abbildung 2.11 erkannt.

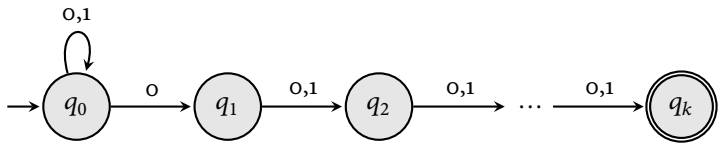


Abbildung 2.11: Nicht-deterministischer endlicher Automat, der binäre Wörter mit einer »0« an k-letzter Stelle erkennt (nach der 0 folgen noch $k-1$ beliebige Ziffern).

Für $k = 3$ sind »0000«, »1011« oder »110001« Beispiele für gültige Wörter; abgelehnt werden beispielsweise »110« und »1111«. Auch in diesem Beispiel ist der nicht-deterministische Übergang im Zustand q_0 enthalten, da der Automat nach dem Lesen der Ziffer »0« sowohl im Zustand q_0 bleiben wie auch in den Zustand q_1 wechseln kann.

Natürlich ist es notwendig, einen NEA genau so wie einen DEA systematisch zu definieren. Wir erledigen dies sowohl informal wie auch mathematisch präzise in Abbildung 2.12.

Nicht-deterministischer endlicher Automat (NEA)

Ein nicht-deterministischer endlicher Automat besteht aus folgenden Komponenten:

- ❏ Zuständen, in denen sich der Automat befinden kann
- ❏ Alphabet, aus dem die Eingabe generiert wird
- ❏ Eine Festlegung der *verschiedenen möglichen* Übergänge
- ❏ Einen *oder mehrere* Startzustände
- ❏ Einen oder mehrere Endzustände

Ein nicht-deterministischer endlicher Automat ist ein 5-Tupel $M = (Q, \Sigma, \delta, E, F)$, wobei:

- ❏ Q eine endliche Zustandsmenge
- ❏ Σ ein endliches Alphabet,
- ❏ $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ die Übergangsfunktion,
- ❏ $E \subseteq Q$ die Menge der Startzustände
- ❏ $F \subseteq Q$ die Menge der akzeptierenden Zustände ist.

Abbildung 2.12: Definition des nicht-deterministischen endlichen Automaten. Links informell, rechts mathematisch-formal.

Wie man aus den Definitionen sieht (vergleiche die Definition eines DEAs in Abbildung 2.5 auf Seite 22), unterscheidet sich ein NEA in zwei Punkten von einem DEA:

❏ Mehrere alternative Übergänge mit gleichem Symbol sind möglich.

❏ Mehrere Startzustände sind möglich.

Die Übergangsfunktion kann weiterhin als Tabelle angegeben werden, allerdings sind die Einträge keine einzelnen Zustände mehr, sondern *Mengen*

von Zuständen. Für den Automaten aus Abbildung 2.10 lautet die Tabelle beispielsweise

Q \ Σ	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\varnothing$
q_2	$\varnothing$	$\varnothing$

Ein NEA ist ein unphysikalisches Berechnungsmodell, das nicht unmittelbar in Form einer Maschine umgesetzt werden kann. Dies macht es notwendig, das Konzept entsprechend zu interpretieren:

- ❏ Ein NEA ist ein massiv paralleler Computer, der bei Bedarf unendlich viele Prozesse (analog des Unix-Modells `fork`/`exec`) gleichzeitig starten kann, wenn eine nicht-deterministische Auswahl vorgenommen wird: Alle Möglichkeiten werden gleichzeitig verfolgt. Auf einem echten Rechner würde man damit natürlich irgendwann den verfügbaren Speicher erschöpfen oder an einer maximale Anzahl von Systemprozessen scheitern.
- ❏ Ein NEA ist eine Rechnung unter Auflistung aller Möglichkeiten (*Achtung*: Es wird nicht ein einziges Rechenergebnis mit einer bestimmten Wahrscheinlichkeit berechnet, sondern wirklich alle Ergebnisse gleichzeitig).
- ❏ Ein NEA enthält ein »Orakel«, das an einer Abzweigung automatisch den (bzw. einen) korrekten Pfad wählt und damit zum richtigen Ergebnis gelangt. Wie diese Wahl getroffen wird, ist nicht weiter spezifiziert. Ein Vorteil der Theorie gegenüber der Praxis ist deshalb offensichtlich, dass man sich Dinge wünschen darf, die nicht notwendigerweise in der Realität funktionieren, aber dennoch nützlich sind.

Nach der bisherigen Definition *muss* ein DEA genau ein Zeichen lesen, um einen Übergang zwischen zwei Zuständen auszuführen. Für manche Probleme ist es vorteilhaft, auch leere Übergänge zu erlauben, d.h. von einem Zustand in einen direkt benachbarten Zustand zu wechseln, ohne ein Zeichen zu lesen. Die Erweiterung bezeichnet man als ϵ-NEA.

Definition: ϵ-NEA Ein ϵ-NEA ist definiert wie ein regulärer NEA, bei dem folgende Änderungen vorgenommen werden:

- ❏ Das Alphabet $\Sigma_* \equiv \Sigma \cup \{\epsilon\}$ wird anstatt Σ verwendet.
- ❏ Die Übergangsfunktion wird erweitert auf

$$\delta : Q \times \Sigma_* \to \mathcal{P}(Q). \tag{2.32}$$

Dadurch sind »leere« Übergänge möglich, also Zustandswechsel, die *ohne* Lesen eines Buchstabens aus der Eingabe vor sich gehen. ∎

2.7.2 *Akzeptierte Sprache eines NEA*

Man kann die Übergangsfunktion $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ eines NEAs (analog zur Überlegung beim DEA) von einem einzelnen Zeichen auf komplette Wörter erweitern. Die Schwierigkeit dabei ist, dass δ einen Zustand auf eine *Menge*

von Zuständen abbildet. Die Verallgemeinerung $\mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ ist wie folgt definiert:

$$\hat{\delta}(Q', \epsilon) = Q' \text{ für alle } Q' \subseteq Q \tag{2.33}$$

$$\hat{\delta}(Q', w_1 w_2 \cdots w_n) = \bigcup_{q \in Q'} \hat{\delta}(\delta(q, w_1), w_2 \cdots w_n) \tag{2.34}$$

Wir illustrieren die Definition wiederum anhand des Automaten aus Abbildung 2.10, der das Wort »10100« bearbeitet.

$$\begin{aligned}
\hat{\delta}(\{q_0\}, 10100) &= \bigcup_{q \in \{q_0\}} \hat{\delta}(\delta(q, 1), 0100) && (2.35)\\
&= \hat{\delta}(\delta(q_0, 1), 0100) = \hat{\delta}(\{q_0\}, 0100) && (2.36)\\
&= \bigcup_{q \in \{q_0\}} \hat{\delta}(\delta(q, 0), 100) && (2.37)\\
&= \hat{\delta}(\delta(q_0, 0), 100) = \hat{\delta}(\{q_0, q_1\}, 100) && (2.38)\\
&= \bigcup_{q \in \{q_0, q_1\}} \hat{\delta}(\delta(q, 1), 00) && (2.39)\\
&= \hat{\delta}(\delta(q_0, 1), 00) \cup \hat{\delta}(\delta(q_1, 1), 00) && (2.40)\\
&= \hat{\delta}(\{q_0\}, 00) \cup \underbrace{\hat{\delta}(\varnothing, 00)}_{=\varnothing} && (2.41)\\
&= \bigcup_{q \in \{q_0\}} \hat{\delta}(\delta(q, 0), 0) && (2.42)\\
&= \hat{\delta}(\delta(q_0, 0), 0) = \hat{\delta}(\{q_0, q_1\}, 0) && (2.43)\\
&= \bigcup_{q \in \{q_0, q_1\}} \hat{\delta}(\delta(q, 0), \epsilon) && (2.44)\\
&= \hat{\delta}(\delta(q_0, 0), \epsilon) \cup \hat{\delta}(\delta(q_1, 0), \epsilon) && (2.45)\\
&= \hat{\delta}(\{q_0, q_1\}, \epsilon) \cup \hat{\delta}(\{q_2\}, \epsilon) && (2.46)\\
&= \{q_0, q_1\} \cup \{q_2\} = \{q_0, q_1, q_2\}. && (2.47)
\end{aligned}$$

Das Wort wird akzeptiert, da $\{q_0, q_1, q_2\} \cap F = \{q_2\} \neq \varnothing$.

Der hauptsächliche Nutzen der erweiterten Übergangsfunktion $\hat{\delta}(\cdot, \cdot)$ liegt allerdings in der Möglichkeit, die erkannte Sprache eines NEAs zu charakterisieren. Dies verläuft praktisch analog zum DEA:

Akzeptierte Sprache eines NEA

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid \hat{\delta}(E, w) \cap F \neq \varnothing\} \tag{2.48}$$

Abbildung 2.13 illustriert die Unterschiede zwischen Berechnungen eines DEAs und NEAs: In beiden Fällen wird pro Übergang genau ein Zeichen gelesen; die gesamte Berechnung ist direkt proportional zur Länge des gelesenen Wortes $\vec{w}$, da in jedem Rechenschritt genau ein Zeichen gelesen wird. Nachdem es in einem deterministischen Automaten genau eine Möglichkeit gibt, zum nächsten Zustand zu gelangen, wird die gesamte Berechnung durch eine lineare Abfolge von Zuständen gegeben. Im nicht-deterministischen Fall kann sich die Rechnung in viele Zweige aufspalten; manche davon führen zu einem akzeptierenden Endzustand, während andere nicht akzeptieren. Ein Wort gilt in diesem Fall als akzeptiert, wenn es im gesamten Baum *mindestens* einen Pfad gibt, der vom Startzustand zu einem akzeptierenden Endzustand führt. Umgekehrt wird ein Wort abgelehnt, wenn

Prinzipiell kann ein NEA mehrere Startzustände besitzen, weshalb es korrekterweise einen Pfad von einem der Startzustände zu einem akzeptierenden Endzustand geben muss. Dieses Szenario ist der Einfachheit halber aber nicht in Abbildung 2.13 berücksichtigt.

es *keinen* Pfad gibt, der in einem akzeptierenden Endzustand endet. Die entsprechende Asymmetrie zwischen Akzeptanz und Ablehnung ist eine zentrale Eigenschaft nicht-deterministischer Berechnungsmodelle.

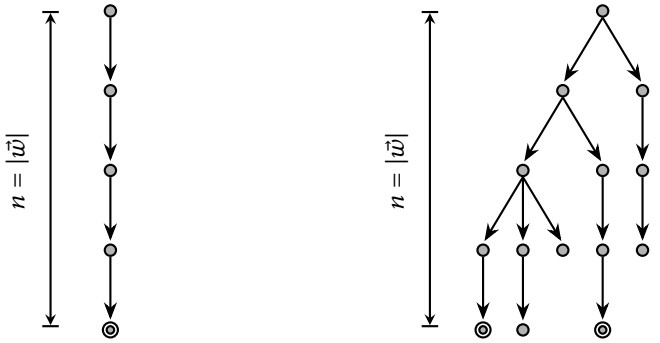
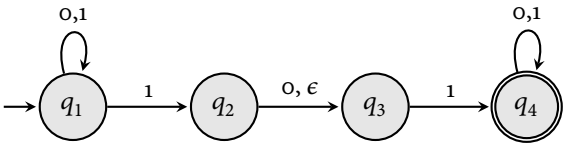


Abbildung 2.13: Unterschiede zwischen deterministischen (links) und nicht-deterministischen (rechts) Berechnungsbäumen.

Um einen konkreten Berechnungsbaum zu diskutieren, betrachten wir folgenden ϵ-NEA:



Die Eingabe »010110« wird vom Automaten erkannt; der zugehörige Berechnungsbaum ist in Abbildung 2.14 gezeigt.

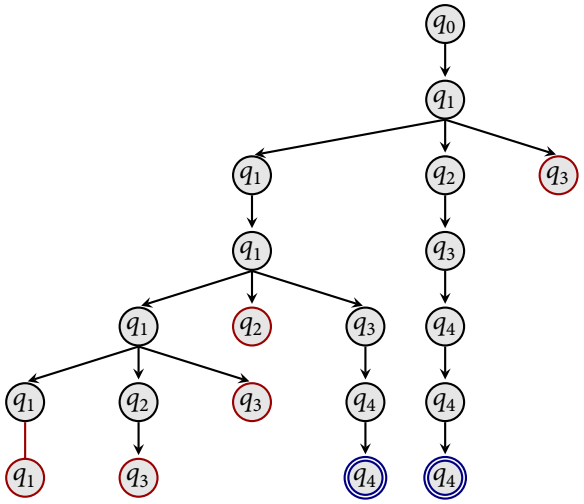


Abbildung 2.14: Berechnungsbaum eines nicht-deterministischen endlichen Automaten. Akzeptierende Pfade sind mit einem blauen, nicht-akzeptierende Pfade mit einem roten Zustand am Ende des Pfades markiert.

Aus der Abbildung wird klar, dass es fünf nicht-akzeptierende Pfade (rot), aber nur zwei akzeptierende Pfade (blau) gibt. Nachdem allerdings ein einziger akzeptierender Pfad ausreicht, um ein Wort zu akzeptieren, ist das Wort dennoch ein Element des erkannten Sprache des Automaten. Der Übergang $q_1 \rightarrow \{q_1, q_2, q_3\}$ im zweiten Automatenschritt soll besonders hervorgehoben werden. Aus dem Zustand q_1 kommt man durch Lesen des Buchstabens »1« zunächst auf den Zustand q_2, oder bleibt aufgrund der Selbstschleife in q_1. Aufgrund des ϵ-Übergangs von q_2 nach q_3 kann der Automat allerdings zusätzlich nach q_3 übergehen, womit sich die gezeigte Dreifachaufspaltung ergibt.

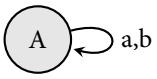
Abschließend weisen wir auf zwei wichtige Unterschiede zwischen DEAs und NEAs hin:

- ❏ Ein NEA muss nicht vollständig sein, d.h. die Übergangsfunktion für einen bestimmten Zustand muss nicht für das gesamte Alphabet Σ definiert sein. Dies bedeutet folgendes:
  - ❏ Es sind Zustände erlaubt, die für ein oder mehrere $\sigma \in \Sigma$ keine ausgehende Kante besitzen
  - ❏ $\delta(q, \sigma) = \perp$ ist erlaubt
  - ❏ Die Übergangsfunktion eines NEAs ist *partiell* definiert.
- ❏ Die Verarbeitung eines NEAs in einem Zweig kann stoppen/abbrechen; das Wort wird in diesem Fall nicht akzeptiert. ($\varnothing \in \mathcal{P}(Q)$).
- ❏ Ein NEA akzeptiert ein Wort, wenn *mindestens* ein Berechnungspfad das Wort akzeptiert, unabhängig davon, in wie vielen Pfaden es abgelehnt wird.
- ❏ Gibt es für ein Zeichen keine möglichen Übergänge, gilt das bearbeitete Wort als nicht akzeptiert.

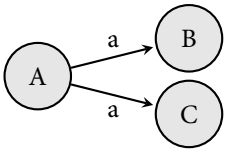
2.8 Grammatiken und NEAs

Die ursprüngliche Motivation für die Einführung von NEAs war es, einen bidirektionalen Zusammenhang zwischen regulären Grammatiken und endlichen Automaten herzustellen. Um eine reguläre Grammatik G in einen NEA M umzuwandeln, verwendet man folgenden Algorithmus:

- ❏ Nicht-Terminalsymbole der Grammatik werden zu Zuständen des Automaten.
- ❏ $\langle A \rangle \rightarrow \mathtt{a}\langle A \rangle \mid \mathtt{b}\langle A \rangle$ wird zur Automatenkomponente



- ❏ $\langle A \rangle \rightarrow \mathtt{a}\langle B \rangle \mid \mathtt{a}\langle C \rangle$ wird zur Automatenkomponente



- ❏ Existiert eine Regel der Form $\langle A \rangle \rightarrow \epsilon$, wird A zu einem Endzustand.
- ❏ Regeln der Form $\langle A \rangle \rightarrow \mathtt{a}$ werden umgeschrieben zu

$$\langle A \rangle \rightarrow \mathtt{a}\langle A' \rangle$$
$$\langle A \rangle' \rightarrow \epsilon.$$

- ❏ Das Startsymbol S der Grammatik wird zum Anfangszustand des Automaten

Bei näherer Betrachtung ist klar, dass der Algorithmus alle denkbaren Fälle regulärer Grammatiken abdeckt; ein formaler Beweis findet sich in der eingangs zitierten Literatur. Verdeutlicht wird der Zusammenhang in Übungsaufgabe 5.14 sowie anhand folgender Grammatik, deren äquivalenter NEA in Abbildung 2.15 gezeigt ist.

Die Sprache der Grammatik (2.49) umfasst alle Wörter $w \in \{a, b, c\}^*$, die »ab« oder »aca« enthalten.

$$\begin{aligned}
\langle S \rangle &\to \mathtt{a}\langle S \rangle \mid \mathtt{b}\langle S \rangle \mid \mathtt{c}\langle S \rangle \mid \mathtt{a}\langle A \rangle \\
\langle A \rangle &\to \mathtt{b}\langle B \rangle \mid \mathtt{c}\langle C \rangle \\
\langle B \rangle &\to \mathtt{a}\langle B \rangle \mid \mathtt{b}\langle B \rangle \mid \mathtt{c}\langle B \rangle \mid \epsilon \\
\langle C \rangle &\to \mathtt{a}\langle B \rangle
\end{aligned} \tag{2.49}$$

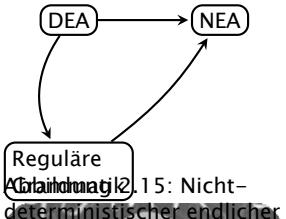
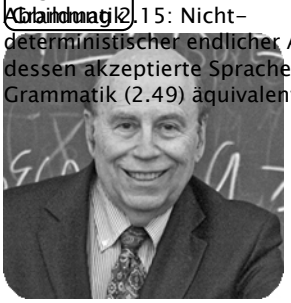
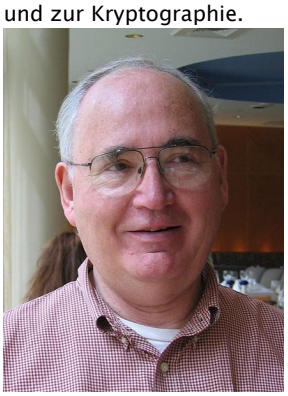


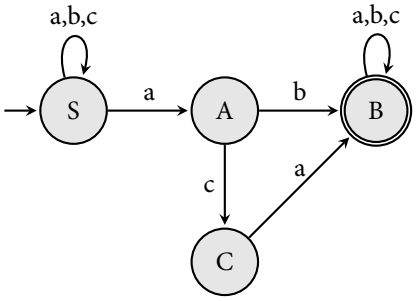
Abbildung 2.15: Nicht-deterministischer endlicher Automat, dessen akzeptierte Sprache zur Grammatik (2.49) äquivalent ist.



Michael O. Rabin (מיכאל רבין) (1931–)
Neben einer 1976 mit dem Turing-Award ausgezeichneten Arbeit (gemeinsam mit Scott), die das Konzept des Nicht-Determinismus in der Informatik etabliert hat, ist Rabin (ein Schüler von Alonzo Church) auch für zahlreiche hochgradig praktische Algorithmen bekannt – beispielsweise zum Testen von Primzahlen, zur Suche in Zeichenketten, und zur Kryptographie.



Dana Scott (1932–)
Mit seinen Arbeiten zur Semantik von Programmiersprachen, unter anderem über denotationale Semantik, hat Scott (ebenfalls ein Schüler von Alonzo Church) wesentlich dazu beigetragen, die Informatik auf eine solide theoretische Grundlage zu stellen. Seine Arbeiten dehnen sich auch auf reine Mathematik und Logik aus.



Der Status Quo unserer Betrachtungen (siehe auch nebenstehende Abbildung) ist wie folgt:

❏ DEA → reguläre Grammatik

❏ DEA → NEA (trivial)

❏ Reguläre Grammatik → NEA

Zum Ringschluss fehlt allerdings noch eine Konvertierungsmöglichkeit zwischen NEAs und DEAs.

2.9 *Äquivalenz von NEAs und DEAs*

2.9.1 *Satz von Rabin und Scott*

Wir haben die vollständige Äquivalenz zwischen NEAs und regulären Grammatiken bewiesen, obwohl wir ursprünglich die Äquivalenz zwischen letzterem und DEAs zeigen wollten. Dies werden wir nun nachholen, indem wir beweisen, dass NEAs und DEAs gleichmächtige Berechnungsmodelle sind, die sich gegenseitig simulieren können. Dies ist ein auf den ersten Blick erstaunliches Ergebnis, da man Intuitiv annehmen würde, dass Nichtdeterminismus eine mächtigere Ressource als Determinismus ist. Der Satz von Rabin und Scott, den wir in diesem Abschnitt betrachten werden, garantiert allerdings die Äquivalenz von DEAs und NEAs. Der Satz ist ein sehr wichtiges, zentrales Resultat der frühen theoretischen Informatik.

Satz von Rabin und Scott

Jede von einem NEA akzeptierbare Sprache L ist auch von einem DEA akzeptierbar.

2.9.2 Illustration

Wir verdeutlichen die Konstruktion von Rabin und Scott zunächst anhand eines Beispiels: Betrachten Sie den NEA aus Abbildung 2.16, der die Sprache $\{x \in \{0,1\}^*, x = 0 \text{ oder } x \text{ endet auf } 00\}$ erkennt.

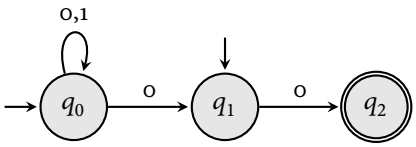


Abbildung 2.16: NEA, der mit der Konstruktion von Rabin und Scottin einen DEA umgewandelt werden soll.

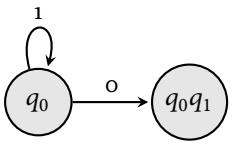
Die Grundidee des Algorithmus von Rabin und Scott ist wie folgt:

- ❏ Alle möglichen Zustandsmengen werden als *ein* Zustand aufgefasst
- ❏ Die Transitionen zwischen den Zustandsmengen werden gemäß der Regeln des NEAs angegeben.
- ❏ Ein neuer Zustand ist ein Endzustand, wenn die Menge mindestens einen alten Endzustand enthält.

Wir betrachten die ersten Schritte der Konstruktion anhand des in Abbildung 2.16 gezeigten Automaten. Vom Zustand q_0 ausgehend gibt es die im Rand gezeigten Übergänge.

	0	1
q_0	q_0, q_1	q_0

Für »0« kann der nicht-deterministische Automat in die Zustände q_0 und q_1 übergehen. Zur Konvertierung betrachten wir die gesamte *Menge* $\{q_0, q_1\}$als einen *einzelnen* Zustand q_0q_1. Er wird vom Zustand q_0 nach Lesen einer »0« erreicht. Ein Übergang von q_0 mit »1« führt zu q_1. Fasst man die Überlegung graphisch zusammen, erhält man das Automatenelement



Wir verzichten zunächst auf die Auszeichnung eines Anfangszustands.

	0	1
q_0	q_0, q_1	q_0
q_1	q_2	$\varnothing$

Um die Übergänge zu betrachten, die von q_0q_1 aus weiterführen, betrachtet man die Tabelle im Rand, die die Übergänge des nicht-deterministischen Automaten für die im zusammengesetzten Zustand enthaltenen Zustände q_0 und q_1 aufführt. Nachdem es von q_1 aus keinen Übergang für den Buchstaben »1« und damit keinen gültigen Zielzustand gibt, ist in der Tabelle an der entsprechenden Stelle die leere Menge $\varnothing$ eingetragen.

Um den »0«-Übergang vom Zustand q_0q_1 zu ermitteln, sammelt man alle Zielzustände auf, die von q_0 *oder* q_1 aus erreicht werden, in diesem Fall $\{q_0, q_1, q_2\}$. Im deterministischen Automaten gibt es daher den Übergang $\delta(q_0q_1, 0) = q_0q_1q_2$. Unter Lesen einer »1« kommt man im NEA von q_0 nach

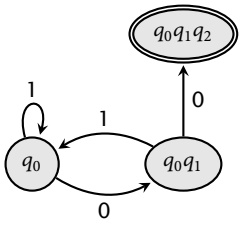
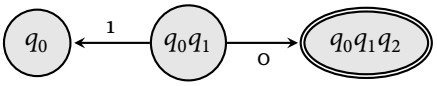


Abbildung 2.17: Zum NEA aus Abbildung 2.16 äquivalenter DEA.

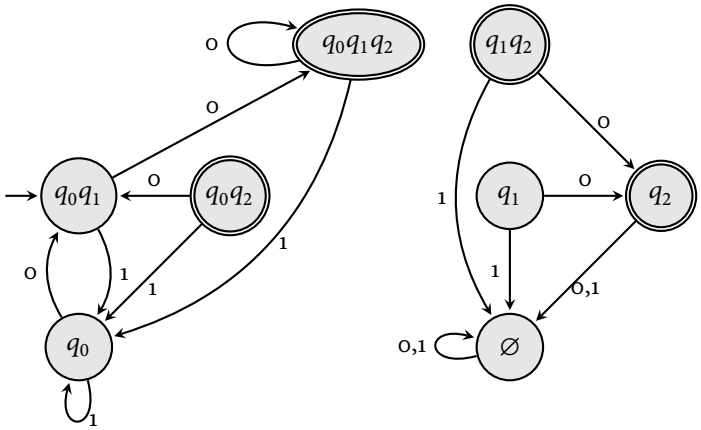
q_0; dies überträgt sich auch auf den deterministischen Automaten. Graphisch zusammengefasst ergibt sich



Der Zustand $q_0q_1q_2$ ist ein akzeptierender Endzustand, da er einen akzeptierenden Zustand q_2 des Ausgangsautomaten enthält.

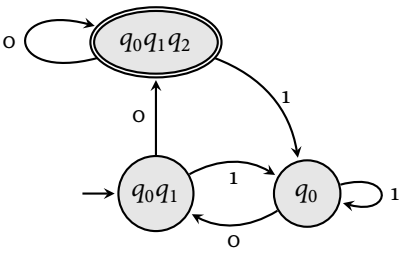
Kombiniert man das Resultat mit dem Ergebnis der vorhergehenden Überlegung erhält man die im Rand gezeigte Automatenkomponente aus drei Zuständen.

Führt man die beschriebenen Schritte für jedes Element von $\mathcal{P}(Q)$ aus, erhält man das in Abbildung 2.17 gezeigte deterministische Äquivalent zum Automaten aus Abbildung 2.16.



Der ursprüngliche Automat besitzt zwei Startzustände, q_0 und q_1. Im transformierten Automaten ist daher q_0q_1 der neue (und einzige) Startzustand. Bei genauer Betrachtung stellt sich heraus, dass das neue Automat offenbar komplizierter als notwendig ist; beispielsweise sind die Zustände auf der rechten Seiten des Graphen nicht vom Startzustand aus erreichbar und daher prinzipiell überflüssig. Mit hinreichend gesundem Menschenverstand kann man die nutzlosen Zustände entfernen und erhält den äquivalenten Minimalautomaten, der in Abbildung 2.18 gezeigt ist (wir werden in Abschnitt 2.12 einen systematischen Algorithmus zur Minimierung endlicher Automaten besprechen).

Abbildung 2.18: Zum DEA aus Abbildung 2.17 äquivalenter Minimalautomat.



Fassen wir den bisherigen Stand der Überlegungen zur Äquivalenz von NEAs und DEAs zusammen:

❏ Konvertierungsalgorithmus NEA→ DEA vorhanden.

❏ Die Rückrichtung ist trivial, da jeder DEA automatisch auch ein NEA ist – der Automat muss dazu lediglich auf nicht-deterministische Übergänge verzichten.

❏ Beim Vergleich von DEAs und NEAs gibt es keine Effizienzverschlechterung in Bezug auf die Laufzeit beim Erkennen von Worten mit n Buchstaben, da $|w| = n$ in beidem Fällen n Bearbeitungsschritte je Pfad nach sich zieht.

Ein NEA kann die Bearbeitung sogar vorzeitig abbrechen, wenn kein passender Übergang zur Verfügung steht; n ist damit eine obere Grenze für die Anzahl von Zeitschritten.

❏ Bei der Konvertierung eines NEAs in einen DEA gibt es einen potentiell exponentiellen Zuwachs der Anzahl von Zuständen.

2.9.3 *Formale Darstellung*

Formal fasst man die Konstruktion von Rabin und Scott wie folgt zusammen:

Äquivalenz DEA/NEA: Formale Darstellung

Sei $M = (Q, \Sigma, \delta, E, F)$ ein NEA. Dann existiert ein DEA $M' = (Q', \Sigma, \delta', q_0', F')$ mit

❏ $Q' \equiv \mathcal{P}(Q)$ Potenzmenge von Q

❏ $\delta'(Z, a) \equiv \bigcup_{q \in Z} \delta(q, a)$ mit $Z \in Q' = \mathcal{P}(Q)$

❏ $q_0' = E$ (Interpretation als kombinierter Einzelzustand!)

❏ $F' = \{Z \subseteq Q \mid Z \cap F \neq \varnothing\}$

der die gleiche Sprache wie M akzeptiert, also

$$\mathcal{L}(M) = \mathcal{L}(M').$$

Natürlich soll die Konstruktion für jeden beliebigen nicht-deterministischen Automaten funktionieren. Um dies sicherzustellen, müssen wir einen allgemeinen Beweis führen. Angenommen, es gibt einen NEA M, der in einen DEA M' überführt wird. Wenn die Transformation erfolgreich war, wird der DEA die exakt gleiche Sprache wie der NEA erkennen, also die selben Wörter wie der NEA akzeptieren, aber auch die selben Wörter ablehnen. Formal ausgedrückt bedeutet dies, dass $\mathcal{L}(M) = \mathcal{L}(M')$ gilt. Folglich müssen wir zeigen, dass für alle Wörter $\vec{w} = a_1 a_2 \dots a_n \in \Sigma^*$ gilt, dass $\vec{w} \in \mathcal{L}(M)$ genau dann gilt, wenn $\vec{w} \in \mathcal{L}(M')$ gilt. Wir berechnen folgende Kette von Äquivalenztransformationen:

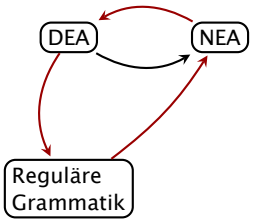
Gestrichene Größen beziehen sich im folgenden auf den deterministischen, nicht-gestrichene Größen auf den nicht-deterministischen Automaten.

$$\begin{aligned}
\vec{w} \in \mathcal{L}(M) &\Leftrightarrow \hat{\delta}(E, \vec{w}) \cap F \neq \varnothing && (2.50)\\
&\Leftrightarrow \exists (Q_1, Q_2, Q_3, \dots, Q_n) \text{ mit } Q_i \subseteq Q \\
&\quad \wedge \delta(E, a_1) = Q_1 \\
&\quad \wedge \delta(Q_{i-1}, a_i) = Q_{i+1}\ \forall i \in [2, n] \\
&\quad \wedge Q_n \cap F \neq \varnothing && (2.51)\\
&\Leftrightarrow \hat{\delta}'(q_0', \vec{w}) \in F' && (2.52)\\
&\Leftrightarrow \vec{w} \in \mathcal{L}(M') && (2.53)
\end{aligned}$$

In Zeile (2.50) haben wir die Akzeptanzdefinition von NEAs aus Gleichung (2.48) verwendet – die erweiterte Übergangsfunktion muss als Resultat einen der akzeptierenden Endzustände aus der Menge F liefern, damit ein Wort vom Automaten akzeptiert wird. Wenn ein NEA ein Wort bearbeitet, wird in jedem Schritt eine Menge potentieller Zustände unter Lesen des jeweiligen Buchstabens in eine neue Zustandsmenge überführt; dafür ist natürlich die Transitionsfunktion $\delta(\cdot,\cdot)$ verantwortlich. Die buchstabenweise Abarbeitung eines Wortes kann man also auch als Sequenz dieser Zustandsmengen auffassen; die erste Menge in der Kette entspricht notwendigerweise den Anfangszuständen E. Nachdem das Wort akzeptiert wird, muss die letzte Zustandsmenge wiederum (mindestens) einen Endzustand enthalten. Die genannten Bedingungen werden formal kompakt durch Gleichung (2.51) wiedergegeben. In Gleichung (2.52) interpretieren wir die Elemente der Sequenz $E, Q_1, \ldots, Q_n$ nicht mehr als Mengen, sondern als kombinierte Einzelzustände des deterministischen Automaten M'; wir nutzen die Verbindung zwischen $\delta(Q_n, a_n) = Q_{n+1}$ und $\delta'(Q_n, a_n) = Q_{n+1}$, um zur nächsten Zeile zu gelangen. Aufgrund der weiter in der Gleichungskette definierten Bedingungen gibt es einen Pfad vom Startzustand zu einem *akzeptierenden* Endzustand; das betrachtete Wort wird also auch von M' erkannt und ist daher Mitglied von $\mathcal{L}(M')$, wie Gleichung (2.53) abschließend feststellt.

Dies bedeutet notwendigerweise auch, dass die gleichen Wörter abgelehnt werden: $\vec{w} \notin \mathcal{L}(M) \Leftrightarrow \vec{w} \notin \mathcal{L}(M')$.

Nachdem sämtliche Implikationen der Rechnung in beide Richtungen korrekt sind, kann man den Beweis sowohl von links nach rechts als auch von rechts nach links lesen; wir haben damit gezeigt, dass die erkannten Sprachen von M und M' exakt übereinstimmen, da $\vec{w} \in \mathcal{L}(M) \Leftrightarrow \vec{w} \in \mathcal{L}(M')$ gilt. □



Mit dem Beweis des Satzes von Rabin und Scott ist ein Ringschluss vollendet, der die Äquivalenz aller bislang gezeigten Mechanismen (Reguläre Grammatik, DEA und NEA) garantiert:

- ❏ DEA → reguläre Grammatik
- ❏ DEA → NEA (trivial)
- ❏ Reguläre Grammatik → NEA
- ❏ NEA → DEA

Der Ringschluss ist ein erstaunliches Resultat, da er sehr unterschiedliche Berechnungsmodelle und vor allem zwei völlig unterschiedliche Klassen von Konzepten – Sprachen und Automaten – als identisch aufdeckt.

2.9.4 *Exponentieller Zustandszuwachs*

In diesem Abschnitt werden wir zeigen, dass tatsächlich NEAs existieren, die bei der Konvertierung in einen äquivalenten deterministischen endlichen Automaten einen exponentiellen Zuwachs an Zuständen erfahren, also (ungefähr) $2^{|Q|}$ Zustände benötigen im Vergleich zu den $|Q|$ Zuständen des NEAs benötigen. Als Beispiel betrachten wir die Sprache

$$L = \{w \in \{0,1\}^n \mid k\text{-letztes Zeichen von } w \text{ ist »0«}\}, \tag{2.54}$$

die durch den auf Seite 39 gezeigten NEA erkannt wird.

Um zu zeigen, dass für einen deterministischen endlichen Automaten tatsächlich exponentiell mehr Zustände als im nicht-deterministischen Fall

notwendig sind, führen wir einen Widerspruchsbeweis: Wir nehmen an, dass die Sprache L aus Gleichung (2.54) mit einem DEA $M = (Q, \Sigma, \delta, q_0, F)$ mit $|Q| < 2^k$ Zuständen akzeptiert werden kann.

Wir betrachten zwei unterschiedliche binäre Strings y_1 und y_2, die der Automat bearbeitet. Angenommen, der Automat durchläuft für die ersten Zeichen die gleiche Zustandssequenz. Ab welchem Buchstaben ist garantiert, auf den gleichen Zustand zu treffen? Man kann natürlich keine untere Grenze angeben (außer dem Startzustand, der trivialerweise gleich sein muss und den wir daher ignorieren), ohne genaueres über die Struktur des Automaten zu wissen dennoch kann man eine Aussage darüber machen, bei welchem Buchstaben dies spätestens der Fall sein muss: Nachdem es 2^k unterschiedliche Binärstrings mit k Stellen gibt, müssen die Zustände *spätestens* beim k-ten Zeichen identisch sein. Wir können von folgendem ausgehen:

$$\exists y_1, y_2 \in \{0,1\}^k \text{ mit } y_1 \neq y_2 : \hat{\delta}(q_0, y_1) = \hat{\delta}(q_0, y_2). \tag{2.55}$$

Es ist nicht genau bekannt, an welcher Position sich die Zeichenkette y_1 und y_2 sich zum ersten Mal unterscheiden; nachdem $y_1 \neq y_2$ gilt, muss es aber eine Position i mit $1 \leq i \leq k$ geben, an der das erste unterschiedliche Zeichen auftritt. Man kann die Zeichenkette deshalb aufspalten in

$$y_1 = u0v \tag{2.56}$$
$$y_2 = u1v', \tag{2.57}$$

wobei das Präfix u bei beiden Zeichenketten identisch ist und die Länge $|u| = i - 1$ besitzt, da der Unterschied an der i-ten Position im Wort auftritt. Ohne Beschränkung der Allgemeinheit dürfen wir annehmen, dass an der i-ten Position in y_1 eine Null und in y_2 eine Eins auftritt, da die Indizes 1 und 2 beliebige Labels sind und auch vertauscht werden könnten. Nachdem $|y_i| = k$, müssen die Suffixe $k - i$ Buchstaben lang sein, also $|v| = |v'| = k - i$.

Wir verlängern die Wörter y_1 mit eine weiteren Suffix w der Länge $|w| = i - 1$ und erhalten die beiden Zeichenketten

$$y_1 w = u0vw \tag{2.58}$$
$$y_2 w = u1v'w, \tag{2.59}$$

für die gilt, dass sie sich an der k-letzten Stelle unterscheiden, da

$$|v| + |w| = k - i + i - 1 = k - 1. \tag{2.60}$$

»0« tritt in $y_1 w$, »1« in $y_2 w$ an k-letzter Stelle auf. Folglich muss nach Definition der Sprache gelten, dass $y_1 w \in L$ und $y_2 w \notin L$.

TODO: *Beweis fertig erläutern*

Daraus folgt, dass die Potenzmengenkonstruktion NEA→ DEA optimal sein *kann*, aber nicht sein *muss* – NEAs können Sprachen im Allgemeinen also *exponentiell kompakter* als DEAs darstellen. Allerdings sollte auch bemerkt werden, dass der *Worst Case* des exponentiellen Zustandszuwachses in der Praxis eher selten auftritt. NEAs tendieren aber zu einer geringeren Komplexität im Auge menschlicher Betrachter, und führen damit häufig zu einer besseren Verständichkeit.

Der NEA besitzt $k + 1$ Zustände, also würde man bei exakt exponentiellem Zuwachs eine Obergrenze von 2^{k+1} erwarten. Nachdem $2^{k+1} = 2 \cdot 2^k$, unterscheiden sich beide Grenzen nur um den (inessentiellen) Faktor 2; das exponentielle Verhalten in k ist in beiden Fällen vorhanden.

Als illustratives Beispiel betrachten wir zwei Strings y_1 und y_2 mit $|y| = 10$, die sich an Stelle $i = 7$ unterscheiden. Wir wählen

$$y_1 = \underbrace{101101}_{u}\,0\,\underbrace{100}_{v}$$
$$y_2 = \underbrace{101101}_{u}\,1\,\underbrace{001}_{v'}$$

um zu sehen, dass $|v| = k - i = 10 - 7 = 3$ Buchstaben lang ist, dito v'. Verlängert man die Wörter mit einem Suffix der Länge $i - 1 = 7 - 1 = 6$, erhält man

$$y_1 w = \underbrace{101101}_{u}\,0\,\underbrace{100}_{v}\,\underbrace{110011}_{w}$$
$$y_2 w = \underbrace{101101}_{u}\,1\,\underbrace{001}_{v'}\,\underbrace{110011}_{w}$$

woraus sofort ersichtlich ist, dass sich die beiden Wörter per Konstruktion an der $k = 10$–letzten Stelle unterscheiden.

2.10 Reguläre Ausdrücke

Wir haben bereits bemerkt, dass reguläre Ausdrücke ein bequemer Mechanismus sind, der in vielen Programmiersprachen verwendet wird, um reguläre Sprachen zu spezifizieren. Wir werden sie in diesem Kapitel genauer analysieren.

Beispiele für reguläre Ausdrücke

- ❏ Postleitzahl & Ort: (0-9)+␣(A-Z|a-z)+
  - ❏ (): Gruppierung
  - ❏ A-Z: Zeichengruppe (entspricht A | B | C | … | Z)
  - ❏ $\xi\{m,n\}$: Zwischen m und n Wiederholungen des regulären Ausdrucks ξ
- ❏ Beispielsprache aller Wörter $x \in \{0,1\}^*$, die mit 00 enden oder gleich 0 sind: 0|((0|1) ∗ 00)
- ❏ eMail-Adressen matchen (*vereinfachter* Ansatz): (A-Z | a-z | 0-9 | . | _ | -) + @(A-Z | a-z | 0-9 | . | -) + .(A-Z | a-z){2, 6}

2.10.1 Syntax regulärer Ausdrücke

Die Syntax regulärer Ausdrücke definieren wir in Abbildung 2.19 wie üblich in informaler und mathematisch präziser Darstellung

Syntax regulärer Ausdrücke

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ gegeben durch:

1. Die leere Menge und das leere Wort.
2. Ein beliebiges Symbol aus Σ.
3. Die Konkatenation zweier regulärer Ausdrücke.
4. Die Auswahl zwischen zwei regulären Ausdrücken.
5. Die Sternoperation angewendet auf einen regulären Ausdruck.
6. Gruppierung regulärer Ausdrücke.

Sei Σ ein beliebiges Alphabet. Dann wird die Menge aller *regulären Ausdrücke* R_Σ induktiv gebildet durch:

1. $\varnothing, \epsilon \in R_\Sigma$.
2. $\Sigma \in R_\Sigma$.
3. $r_1, r_2 \in R_\Sigma \Rightarrow r_1 r_2 \in R_\Sigma$.
4. $r_1, r_2 \in R_\Sigma \Rightarrow (r_1 \mid r_2) \in R_\Sigma$.
5. $r \in R_\Sigma \Rightarrow r^* \in R_\Sigma$.
6. $r \in R_\Sigma \Rightarrow (r) \in R_\Sigma$.

Abbildung 2.19: Syntax regulärer Ausdrücke: Links informal, rechts formal dargestellt.

zwischen den beiden Symbolen ∅ und ϵ zu unterscheiden: Während ersteres eine komplett leere Sprache repräsentiert, die kein Wort (insbesondere auch das leere Wort nicht!) akzeptiert und ein entsprechender DEA jede Eingabe ablehnt, enthält letztere Klasse das leere Wort (eine leere Zeichenkette) als gültiges Element.

Die Definition regulärer Ausdrücke ist bislang eine rein *syntaktische* Zusammenstellung von Zeichenketten. Wir benötigen zusätzlich eine *Semantik*, die angibt, welche Sprache (d.h. Menge von Wörtern) durch einen reguläre Ausdruck (eine Zeichenkette) repräsentiert wird. Strukturell entspricht dieser einer Zuordnung zwischen Zeichenkette (regulärer

Ausdruck) und Menge (Sprache des regulären Ausdrucks), die man induktiv über den oben definierten Regeln aufbaut:

Semantik regulärer Ausdrücke

- ❑ $\mathcal{L}(\varnothing) = \varnothing, \mathcal{L}(\epsilon) = \{\epsilon\}$
- ❑ $\mathcal{L}(a \in \Sigma) = \{a\}$
- ❑ $\mathcal{L}(rs) = \mathcal{L}(r)\mathcal{L}(s)$
- ❑ $\mathcal{L}(r|s) = \mathcal{L}(r) \cup \mathcal{L}(s)$
- ❑ $\mathcal{L}(r^*) = \mathcal{L}(r)^*$
- ❑ $\mathcal{L}((r)) = \mathcal{L}(r)$

2.10.2 *Satz von Kleene*

Wenn eine Sprache endlich ist, d.h. aus einer endlichen Anzahl von Wörtern besteht, ist es in jedem Fall möglich, sie durch einen regulären Ausdruck zu beschreiben. Dazu verwendet man folgende triviale Konstruktion:

Endliche Sprachen und reguläre Ausdrücke

Alle endlichen Sprachen sind durch reguläre Ausdrücke beschreibbar. Sei $A = \{\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_n\}$ eine *endliche* Menge der *endlichen* Wörter $\vec{w}_i$. Dann ist $(\vec{w}_1|\vec{w}_2|\vec{w}_3|\cdots|\vec{w}_n)$ ein regulärer Ausdruck für die Sprache A.

Die (kompliziertere) Äquivalenz regulärer Ausdrücke und regulärer Sprachen wird im Satz von Kleene ausgedrückt:

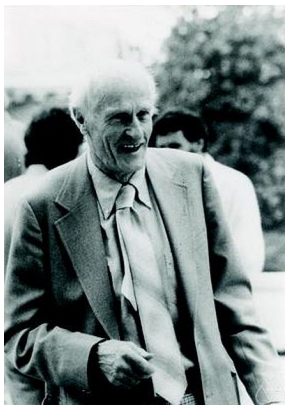
Satz von Kleene

Die Menge der durch reguläre Ausdrücke beschreibbaren Sprachen ist genau die Menge der regulären Sprachen.

Obige Aussage besteht bei genauer Betrachtung aus zwei Teilen:

1. Jeder reguläre Ausdruck beschreibt eine reguläre Sprache (die daher durch einen endlichen Automaten erkennbar ist).
2. Die Sprache jedes regulären Automaten kann durch einen regulären Ausdruck beschrieben werden.

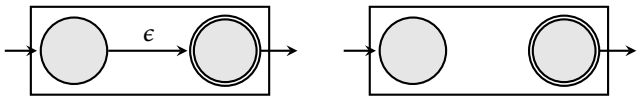
Wir illustrieren die Vorwärtsrichtung anhand einer informellen Argumentation, die jedoch alle wichtigen Überlegungen des formalen Beweises enthält. Jeder reguläre Ausdruck muss durch einen regulären Automaten zu erkennen sein – dieser muss zu jedem möglichen regulären Ausdruck konstruierbar sein. Dabei machen wir uns zunutze, dass sich reguläre Ausdrücke nach den weiter oben besprochenen Regeln aus elementaren Einheiten zusammensetzen, die zu größeren Ausdrücken kombiniert werden. Wenn man zu jedem Element ein passendes Element eines endlichen Automaten konstruiert (wir werden mit ϵ-NEAs arbeiten, die bekanntlich in DEAs umgewandelt werden können), das mit anderen Komponenten zu einem vollständigen Automaten »verschaltet« werden kann, ist die Übereinstimmung gezeigt.



Stephen C. Kleene (1909–1994)
Wie viele andere in diesem Skript vorgestellten Wissenschaftler war Kleene ein Schüler von Church. Neben mathematischen Arbeiten, von denen sein Fixpunktsatz die höchste Bekanntheit erreichte, hat er sich insbesondere um die Fundamente der Informatik verdient gemacht. Beispielsweise hat er gemeinsam mit Post verschiedene Grade der Unlösbarkeit von Problemen aufgedeckt.

Achtung: Beim leeren Wort ist der Eingangszustand mit dem akzeptierenden Endzustand verbunden, bei der leeren Menge wird dieser nie erreicht!

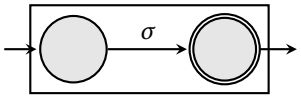
Wir beginnen mit den beiden einfachsten Elementarausdrücken, den RegExps für die leere Menge $\varnothing$ und das leere Wort ϵ. Beide sind durch einfache Automatenelemente zu erkennen (links das leere Wort, rechts die leere Menge):



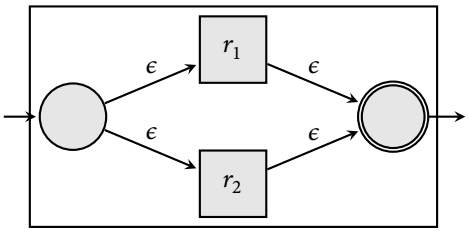
Man könnte das leere Wort (und die leere Menge) auch mit einem Automaten mit nur einem einzigen Startzustand entscheiden. Dies führt allerdings zu Problemen, wenn die Komponentenautomaten als Teil größerer Automaten verwendet werden, weshalb wir explizite (Nicht-)Übergänge und explizite Eingangs- und Ausgangszustände verwendet haben.

Jeweils ein Pfeil weist links in die Kästen hinein und aus den Kästen hinaus; an diesen Stellen können die Elemente miteinander verschaltet werden (die Ausgabe eines Elements wird als Eingabe für das nächste Element verwendet, analog Bauteilen in einem elektronischen Schaltplan).

Der nächstkomplexere reguläre Ausdruck ist ein einzelnes Zeichen σ, das von folgendem Element erkannt wird:

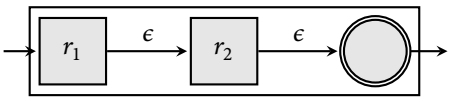


Zwei reguläre Ausdrücke r_1 und r_2 können durch eine »Oder«-Verknüpfung zu einem neuen Ausdruck $r_1|r_2$ kombiniert werden. Folgendes Automatenelement erkennt die daraus resultierende Sprache:

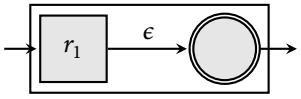


Da nichts über die regulären Unterausdrücke r_1 und r_2 bekannt ist, werden Sie duch Kästen repräsentiert – in die man beispielsweise den Teilautomaten für ein einzelnes Zeichen oder eine weiter »Oder«-Verknüpfung einsetzen kann, je nach Gesamtausdruck. Weiterhin ist a priori unklar ist, welcher Pfad (oben oder unten) durchlaufen werden muss, was aber kein Problem ist – ein ϵ-NEA kann nichtdeterministisch den passenden wählen.

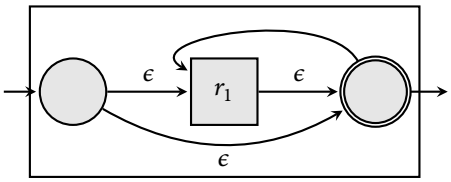
Die Konkatenation $r_1 r_2$ wird umgesetzt, indem die Teilelemente für r_1 und r_2 unmittelbar hintereinander positioniert werden:



Der Ausdruck (r) ist einfach in ein Automatenelement umzusetzen, da die Klammerung hierfür keine Rolle spielt:



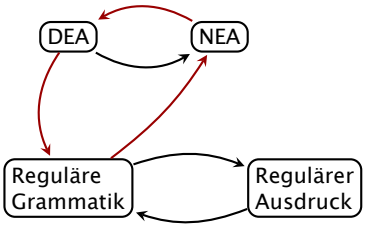
Etwas komplizierter gestaltet sich der Kleene-Stern r_1^* – das dazugehörige Automatenelement muss sicherstellen, dass der Ausdruck 0, 1, 2, …mal hintereinander auftreten kann. Wir erledigen dies mit folgender Konstruktion:



Damit haben wir ein Automatenelement zu jedem möglichen Element eines regulären Ausdrucks angegeben, die Vorwärtsrichtung ist damit bewiesen.

Da die Rückwärtsrichtung formal etwas aufwendig zu beweisen ist, aber keinen substantiellen Erkenntnisgewinn gebenüber dem gesunden Menschenverstand bringt, verweisen wir auf die eingangs zitierte Literatur (außer dem Buch von Hoffmann, das den Beweis ebenfalls überspringt). □

Durch den Satz von Kleene wird der Status Quo bezüglich der Zusammenhänge zwischen den unterschiedlichen Sprachklassen um die Facette reguläre Ausdrücke ↔ reguläre Grammatik erweitert, aber nicht fundamental verändert, wie nebenstehende Abbildung zeigt.



Reguläre Ausdrücke bringen keine strukturellen Änderungen in die Zusammenhänge zwischen DEAs/NEAs und regulären Grammatiken; die Äquivalenz zwischen den assoziierten Sprachklassen gilt unabhängig von der Existenz regulärer Ausdrücke. Sind diese damit überflüssig? Keineswegs. Auch wenn durch RegExps keine neuen theoretischen Facetten zum Thema hinzugefügt werden, wäre es extrem unpraktisch, einfache Sprachen in einer Programmiersprache durch endliche Automaten oder eine nicht weniger unhandliche reguläre Grammatik auszudrücken. Hätte es dann aber nicht genügt, unmittelbar mit den praxistauglichen regulären Ausdrücken zu starten und die restlichen Aspekte beiseitezuschieben? Ebenfalls keineswegs: DEAs (und NEAs) sind extrem effizient zu implementieren, und die in der Theorie ermittelten Zusammenhänge erlauben unmittelbar, für jeden RegExp einen endlichen Automaten zu implementieren, der die Sprachverarbeitung durchführt und damit den regulären Ausdruck in ein reales Hilfsmittel verwandelt. Die Praxis hätte ohne Theorie schwerer und wäre ärmer, aber auch die umgekehrte Richtung gilt: Ohne praktische Anwendung in Form von RegExps wären reguläre Grammatiken eine Spielwiese, die je nach Standpunkt zwar interessant, aber für die Informatik sicher nicht allzu relevant wären. Wie immer sind Theorie und Praxis keine Opponenten, sondern sind aufs engste miteinander verwunden!

2.11 *Das Pumping-Lemma*

2.11.1 *Reguläre Sprachen und geklammerte Ausdrücke*

Reguläre Ausdrücke (und damit reguläre Sprachen) eignen sich häufig zur Beschreibung komplexer Sprachstrukturen, scheitern aber gelegentlich an vermeintlich einfachen Problemen, wie man sich anhand des folgenden Beispiels klarmacht. Nehmen wir an, Sie möchten eine Vorlage, die (wie dieses Skript) in LaTeX geschrieben ist, in ein XML-ähnliches Format konvertieren, wie in Abbildung 2.20 dargestellt ist.

Machen Sie sich mit LaTeX vertraut! Studien haben ergeben, dass Abschlussarbeiten, die mit Word oder ähnlichen Programmen verfasst werden, in 100% aller Fälle zu Formatierungsfehlern, Panikattacken und Verzweiflungsanfällen 3 Tage vor Abgabe führen.

Wir greifen hier auf die Sprache Perl (siehe `www.perl.org`) zurück.

Nachdem reguläre Ausdrücke von praktisch allen Programmiersprachen unterstützt werden, liegt es nahe, diese zur Konvertierung der einfachen

Vorlagenkonvertierung

Vorlage in LaTeX:

```
\section{Einleitung}
In dieser \textbf{Einleitung},
die \textit{sehr \textbf{schnell}
    neu} ...
```

Nach XML konvertierter Text:

```
<abschnitt>Einleitung</abschnitt>
In dieser <fett>Einleitung</fett>,
die <kursiv>sehr
    <fett>schnell</fett> neu</kursiv>
    ...
```

Abbildung 2.20: Konvertierung zwischen LaTeX und XML.

Textstruktur zu verwenden.

Der offensichtlich einfachste Versuch, den RegExp `{(.*)}` zur Erkennung der Argumente zu verwenden, scheitert, wenn zwei Befehle ineinander verschachtelt sind: In der Phrase `\textit{sehr \textbf{schnell} neu}}` wird dadurch lediglich die Komponente `\textit{sehr \textbf{schnell}` erkannt. Nach einigen Versuchen wird man sich sehr schnell überzeugen, dass es nicht möglich ist, beliebig verschachtelt geklammerte Ausdrücke mit einem allgemeinen regulären Ausdruck zuverlässig zu parsen. Allerdings stellt sich die Frage, ob dies ein prinzipielles Problem regulärer Sprachen oder ein Mangel an Kreativität des Experimentierenden ist. Etwas genereller formuliert stellen sich die Probleme

Zu Ihrer Beruhigung, falls Sie schon länger herumprobiert haben: Wir werden gleich zeigen, dass eine Erkennung mit Standard-RegExps prinzipiell unmöglich ist.

- ❏ Ist eine gegebene Grammatik G regulär?
- ❏ Kann eine Sprache L mit regulären Ausdrücken beschrieben werden?

2.11.2 *Illustration des Lemmas*

Um beweisen zu können, dass eine Sprache sicher *nicht* regulär ist, existiert das *Pumping-Lemma*, das wir zunächst anhand eines konkreten Beispiels illustrieren. Betrachten Sie dazu den DEA mit sieben Zuständen in Abbildung 2.21.

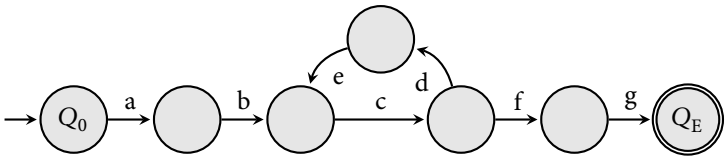


Abbildung 2.21: Beispiel-DEA zur Illustration des Pumping-Lemmas.

Obwohl der Automat konventionsgemäß nur über endlich viele Zustände verfügt, können aufgrund der enthaltenen Schleife beliebig lange Wörter erzeugt werden:

- ❏ abcfg ⇒ Kein Zyklus ($|w| = 5$)
- ❏ abcdecfg ⇒ Einfacher Zyklus ($|w| = 8$)
- ❏ abcdecdecfg ⇒ Zweifacher Zyklus ($|w| = 11$)

Wörter, die länger als fünf Buchstaben sind, durchlaufen den Zyklus dec allerdings mindestens einmal, längere Wörter entsprechend häufiger. Allgemein kann man fuer DEAs festhalten, dass mit Zustandsanzahl $|Q| = n$ für Wörter w mit $|w| \geq n$ ein Zyklus durchlaufen werden muss. Die Kernidee des Pumping-Lemma ist, dass dies für jede reguläre Sprache gelten muss, die bekanntlich immer durch einen endlichen Automaten repräsentiert werden kann.

Um das Pumping-Lemma formal zu formulieren, halten wir die beschriebene Situation etwas abstrakter fest:

1. L sei eine reguläre Sprache; entsprechend existiert ein DEA M mit $L = \mathcal{L}(M)$.
2. M habe $|Q| = n$ Zustände. Die Anzahl der Zustände ist nicht notwendigerweise bekannt, aber es gilt sicher $n < \infty$.
3. Sei $w \in L$ ein hinreichend langes Wort der Sprache mit $|w| \geq n$, das durch $w = a_1 a_2 \ldots a_m$ beschrieben ist, wobei $a_i \in \Sigma$ einzelne Buchstaben des Alphabet sind (entsprechend gilt $|w| = m \geq n = |Q|$).
4. Sei Q_i der Zustand des DEA A nach Lesen der ersten i Symbole von w.
5. Dann gibt es Zahlen $i, j : 0 \leq i < j \leq n : Q_i = Q_j$, d.h. mindestens ein Zustand in der Ableitungssequenz tritt mehrfach auf, der Automat hat also einen Zyklus durchlaufen.

Allgemein hat jeder Automat mit n Zuständen, der Wörter mit $m > n$ Buchstaben akzeptiert, daher die in Abbildung 2.22 dargestellte Struktur.

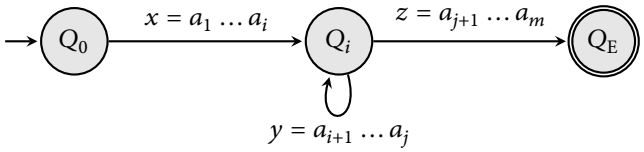


Abbildung 2.22: Allgemeine Struktur eines DEA mit n Zuständen, der Wörter der Länge $m > n$ akzeptiert.

Anhand der bisherigen Überlegungen fassen wir das Schleifenlemma wie folgt zusammen:

Das Lemma ist in der Literatur alternativ auch als Schleifenlemma, *uvw*–Theorem oder Lemma von Bar–Hillel bekannt.

Pumping-Lemma

Sei L eine reguläre Sprache. Dann gibt es eine Zahl n, so dass sich alle Wörter $\vec{w} \in L, |\vec{w}| \geq n$ zerlegen lassen in $\vec{w} = xyz$, so dass

1. $|y| \geq 1$
2. $|xy| \leq n$
3. Für alle $i \in \mathbb{N}_0 : xy^i z \in L$

Wir zeigen, dass das Pumping-Lemma für jede reguläre Sprache gilt, können also davon ausgehen, dass ein endlicher Automat $M = (Q, \Sigma, \delta, q_0, E)$ existiert, der die Sprache erkennt. Gemäß der Voraussetzung des Lemma wählen wir ein Wort $\vec{w} \in L$ mit $|\vec{w}| \geq n$. Der Automat durchläuft beim Erkennen des Wortes $|\vec{w}+1|$ Zustände. Nachdem $|\vec{w}| \geq n$, folgt $|\vec{w}|+1 \geq n+1$;

nach dem Schubfachprinzip muss also mindestens einer der insgesamt n Zustände des Automaten mehrfach durchlaufen worden sein. Anders ausgedrückt: Der DEA hat notwendigerweise eine Schleife durchlaufen.

Wir wählen nun eine Zerlegung von $\vec{w}$ nach $\vec{w} = xyz$, wobei x den Teil des Wortes beschreibt, der *vor* dem Schleifendurchlauf gelesen wurde, und y den Anteil, der nach einem *einmaligen* Schleifendurchlauf gelesen wurde. Daher gilt, dass der Zustand nach Lesen von x und nach Lesen von xy der gleiche sein muss – y durchläuft die Schleife, und eine Schleife beginnt per Definition dort, wo sie aufhört. Über die Struktur beider Bestandteile können wir folgendes aussagen:

- ❏ Es muss $|y| \geq 1$ gelten, da die kleinstmögliche Schleife (eine Selbst-Schleife) mindestens einen Buchstaben produziert.
- ❏ Die Schleife ist spätestens nach n gelesenen Buchstaben einmal durchlaufen, da der Automat dann $n + 1$ Zustände durchlaufen hat und die Schleife aufgrund obiger Überlegung traversiert haben muss. Also kann man die Länge des Präfix xy begrenzen auf $|xy| \leq n$. *Achtung*: Die Schleife könnte natürlich auch früher auftreten; nachdem wir aber keine Information über die genaue Struktur des Automaten haben, müssen wir den Worst Case annehmen. Aber selbst in diesem Fall ist die Schleife nach einer maximalen Anzahl gelesener Buchstaben einmal durchlaufen.

Wenn der Zustand nach Lesen von x und xy gleich ist, ist auch der Zustand nach Lesen von xz und xyz identisch – in beiden Fällen wird von einem gemeinsamen Ausgangszustand ausgehend das Suffix z gelesen. Nachdem $xyz = \vec{w}$ akzeptiert wird, wird auch das Wort $xz = xy^0z$ vom Automaten akzeptiert. Kurz gesprochen: $xyz \in L \Rightarrow xz = xy^0z \in L$.

Wenn die Zustand nach Lesen von x und xy gleich ist, kann man an beide Präfixe das Teilwort y anhängen, um die neuen Präfixe xy und $xyy = xy^2$ zu erhalten. Der Automat befindet sich nach Abarbeitung der beiden Teilstrings wiederum im gleichen Zustand. Da aber $xyz \in L$ gilt, muss auch $xyyz \in L$ gelten, das Wort xy^2z ist also Teil der Sprache. Diese Überlegung kann man mit beliebig vielen angehängten Komponenten y wiederholen; $xyyz$, $xyyyz$, …sind daher gültige Elemente der Sprache. Allgemein gilt $xy^nz \in L$ für alle $n \in \mathbb{N}$. □

Achtung: Das Pumping-Lemma macht nur eine *einseitige* Aussage: Wenn eine Sprache regulär ist, müssen die genannten Bedingungen erfüllt sein. Es gilt allerdings *nicht* die Umkehrung, dass eine Sprache regulär ist, *wenn* die Bedingungen erfüllt sind. Dies ist in Abbildung 2.23 verdeutlicht.

Wortbeispiele für die Sprache 2.61: ab, aabb, aaabbb, aaaabbbb, … Es ist nicht relevant, ob wir explizite Klammern (), Buchstaben wie »a« und »b« oder andere Symbole verwenden, da die Struktur des Problems in jeden Fall gleich ist.

Der Index n in der Dyck-Sprache D_n steht für die Anzahl der unterschiedlichen Klammerpaare (je zwei Symbole, die in korrekter Reihenfolge und korrekt geschachtelt vorkommen müssen), die in der Sprache vorkommen. Die Sprachen selbst sind nach dem Münchner Mathematiker Walther von Dyck benannt.

2.11.3 *Anwendungsbeispiele*

Ein traditionelles Anwendungsbeispiel für die Verwendung des Pumping-Lemmas ist die *Dyck-Sprache* D_1, die alle korrekt geklammerten Ausdrücke enthält. Wir werden zeigen, dass die am Anfang des Kapitels diskutierte Sprache

$$L \equiv \{a^n b^n \mid n \in \mathbb{N}^+\} \tag{2.61}$$

keine reguläre Sprache ist.

Zum Beweis nehmen wir an, die Sprache L wäre regulär; daher muss

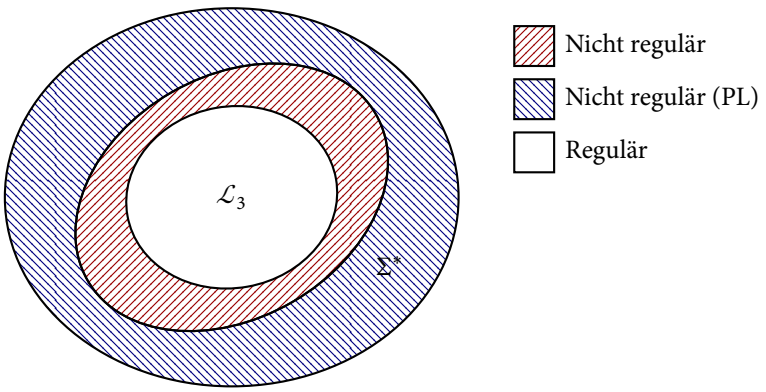


Abbildung 2.23: Illustration des Anwendungsgebiets des Pumping-Lemmas. Der weiße Kreis repräsentiert die Menge aller regulären Sprachen. Die Gesamtmenge kennzeichnet die Menge aller Sprachen; blau markiert sind alle Sprachen, deren nicht-Regularität mit Hilfe des Pumping-Lemmas gezeigt werden kann. Die rot schraffierte Menge enthält Sprachen, die nicht regulär sind, für das Pumping-Lemma aber dennoch gilt.

es eine Pumping-Zahl n geben, die allerdings unbekannt ist. Als Wort betrachten wir $\vec{w} = \mathrm{a}^n\mathrm{b}^n$, das Mitglied der Sprache ist und die Länge $|\vec{x}| = 2n > n$ besitzt. Die Bedingungen des Pumping-Lemmas sind damit erfüllt, was eine Zerlegung $\vec{x} = uvw$ garantiert. Dabei ist v nicht leer, und $|uv| \leq n$. Nachdem uv ein Präfix des Wortes $\vec{w}$ ist, also direkt am Anfang steht, muss uv zunächst »a«s enthalten. Das betrachtete Wort $\vec{w}$ enthält aufgrund seiner Struktur n mal den Buchstaben »a« am Anfang; da uv *maximal* n Buchstaben lang ist, kann es also *nur* aus »a«s bestehen und keine »b«s enthalten. Dies bedeutet insbesondere, dass der aufgepumpte Teilstring v ebenfalls nur aus »a«s besteht, auch wenn unklar ist, wie viele Buchstaben v enthält (sicher ist lediglich, dass es auch mindestens einem und maximal n »a«s besteht).

Aufgrund des Pumping-Lemmas muss nicht nur $uvw \in L$, sondern auch $uv^0w = uw \in L$ gelten. Vergleicht man uvw und uw, fehlen bei letzterem alle »a«s, die in v enthalten sind. Das Wort hat daher die Struktur

$$uw = \mathrm{a}^{n-|v|}\mathrm{b}^n. \tag{2.62}$$

Nachdem das Pumping-Lemma garantiert, dass $|v| \geq 1$ gilt, besitzt uw *echt weniger* »a«s als »b«s – und kann damit kein Mitglied der Sprache sein. Es gilt folglich $uw \notin L$. Dies ist aber ein Widerspruch zur vorhergehenden Aussage, dass $uw \in L$. Die einzige Annahme, die wir im Verlauf des Beweises gemacht haben, muss daher falsch sein. Nachdem die Annahme war, dass L regulär ist, folgt also, dass L *nicht* regulär ist. □

Als zweites Beispiel betrachten wir die Sprache aller uniformer Strings, deren Länge als Quadrat einer ganzen Zahl ausgedrückt werden kann:

Quadratzahlen

$L \equiv \{\mathrm{a}^n \mid \exists m \in \mathbb{N} : n = m^2\}$. Beispiele: a, aaaa, aaaaaaaaa, …

Um zu beweisen, dass die Sprache *nicht* regulär ist, machen wir die gegenteilige Annahme: L ist regulär. Wir werden zeigen, dass diese Annahme zu einem logischen Widerspruch führt, weshalb sie nicht wahr sein kann und daher das Gegenteil gelten muss.

Sei n die Konstante des Pumping-Lemmas. Wie wählen einen String $z = a^{n^2} \in L$, dessen Länge die Pumping-Zahl übersteigt: $|z| = n^2 > n$. Nach dem Pumping-Lemma gibt es daher eine Zerlegung: in $z = uvw$, die den Bedingungen $\leq |v| \leq |uv| \leq n$ genügt. Weiterhin können wir anhand des

Es gilt $|v| < |v^2|$ mit der Echt-Kleiner-Relation, da v mindestens einen Buchstaben lang ist und die Quadrat-Operation nicht numerische Exponentiation, sondern zweifache String-Konkatenation angibt! Selbst im Fall $|v| = 1$ gilt also $|v^2| = 2$, also $|v| < |v^2|$. Der Fall $|v| = 0$ ist nach Bedingung 1 des Pumping-Lemmas ausgeschlossen.

Lemmas folgern, dass $uv^2w \in L$ gilt. Wir rechnen nun rein algebraisch

$$n^2 = |z| = |uvw| < |uv^2w| \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2,$$

worin die Ungleichung

$$n^2 < |uv^2w| < (n+1)^2$$

enthalten ist. Dies bedeutet, dass $|uv^2w|$ eine Quadratzahl zwischen n^2 und $(n+1)^2$ ist – was aber offenbar nicht möglich sein kann, da keine weitere Quadratzahl im Intervall $[n^2, (n+1)^2]$ existieren kann, schließlich werden zwei direkt aufeinanderfolgende Zahlen quadriert. Wir haben den gewünschten Widerspruch konstruiert, die ursprüngliche Behauptung kann also nicht richtig sein. Konsequenterweise ist L nicht regulär. □

2.12 *Automatenminimierung*

In Kapitel 2.9.1 sind wir dem Problem eines Automaten mit überflüssigen Zuständen begegnet, der mit Hilfe des gesunden Menschenverstandes auf eine äquivalente kleinere Form gebracht werden konnte. In diesem Abschnitt werden wir das Problem systematischer angehen und eine allgemeine Konstruktionsmethode zur Minimierung endlicher Automaten angeben, die von John Myhill und Anil Nerode gefunden wurde.

2.12.1 *Satz von Myhill und Nerode*

Die Basis der Argumentation bildet eine Äquivalenzrelation, die auf den Wörtern einer Sprache gebildet werden kann:

Äquivalenzrelation für Sprache L

Es gilt $xR_L y$ genau dann, wenn für alle Wörter $z \in \Sigma^*$ gilt

$$xz \in L \Leftrightarrow yz \in L.$$

Wenn aus dem Kontext eindeutig klar ist, auf welche Sprache sie die Relation bezieht, verzichtet man in der Literatur auf den Index L beim Operator R_L. Folgende Punkte sind wichtig:

- ❏ Das Anfügen beliebiger Teilstrings zieht keine Änderung der Mitgliedschaft in einer Äquivalenzklasse nach sich.
- ❏ Es ist möglich, dass z leer ist: $z = \epsilon : x \in L \Leftrightarrow y \in L$. Entsprechend ist es nicht möglich, dass ein Wort der Sprache sich in einer Äquivalenzklasse mit einem String befindet, der kein Element der Sprache ist. Allerdings können die Wörter der Sprache durchaus in mehrere Äquivalenzklassen aufgeteilt sein.
- ❏ Die Anzahl der durch die Relation erzeugten Äquivalenzklassen wird als *Index* von R_L bezeichnet.

Satz von Myhill und Nerode

Eine Sprache L ist genau dann regulär, wenn der Index von R_L endlich ist.

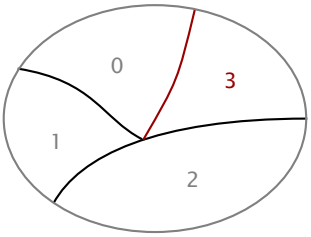
Beweis: Zunächst betrachten wir die Richtung »⇒«; es ist also eine reguläre Sprache $L \in \mathcal{L}_3$ gegeben. Nachdem reguläre Sprachen von endlichen Automaten erkannt werden, gibt es einen DEA $M = (Q, \Sigma, \delta, q_0, F)$ mit $\mathcal{L}(M) = L$. Ausgehend von M definieren wir eine neue Äquivalenzrelation R_M durch

$$xR_M y, \text{ wenn } \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y). \tag{2.63}$$

Zwei Wörter x und y sind damit in der gleichen Äquivalenzklasse, wenn bei der Abarbeitung der gleiche Endzustand erreicht wird. Wir werden zeigen, dass $R_M \subseteq R_L$ gilt, R_M also eine Verfeinerung von R_L ist. Dies bedeutet, dass Wörter x und y, die über R_M verbunden sind, notwendigerweise auch durch R_L verbunden sind, also $xR_M y \Rightarrow xR_L y$ (wenn $xR_L y$, muss allerdings nicht notwendigerweise folgen, dass auch $xR_M y$ gilt).

Nehmen wir nun an, dass x und y durch R_M verbunden sind, also $xR_M y$ gilt. $z \in \Sigma^*$ sei ein beliebiger String, der kein Mitglied der Sprache sein muss (aber sein kann). Mit diesen Zutaten können wir folgende Rechnung aufstellen, die die Eigenschaften der $\hat{\delta}$-Funktion ausnutzt:

$$\begin{aligned}
xz \in L &\Leftrightarrow \hat{\delta}(q_0, xz) \in F && (2.64)\\
&\Leftrightarrow \hat{\delta}(\hat{\delta}(q_0, x), z) \in F && (2.65)\\
&\Leftrightarrow \hat{\delta}(\hat{\delta}(q_0, y), z) \in F && (2.66)\\
&\Leftrightarrow \hat{\delta}(q_0, yz) \in F && (2.67)\\
&\Leftrightarrow yz \in L && (2.68)
\end{aligned}$$



TODO: Verfeinerung von Äquivalenzklassen erläutern

In Gleichung (2.66) haben wir ausgenutzt, dass $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$ gilt, da die beiden Wörter über R_M verbunden sind. Aus der Rechnung folgt, dass $xz \in L \Leftrightarrow yz \in L$ gilt, womit x und z auch über R_L miteinander verbunden sind, wenn Sie über R_M verbunden sind. Oder, anders ausgedrückt: $xR_M y \Rightarrow xR_L y$. Damit ist R_M in der Tat eine Verfeinerung von R_L, weshalb $\text{Index}(R_L) \leq \text{Index}(R_M)$ gilt. Der Index von R_M entspricht der Anzahl unterschiedlicher Zustände, die von q_0 aus erreicht werden können, und ist daher sicher echt kleiner als unendlich. Formal betrachtet: $\text{Index}(R_L) \leq |Q| < \infty$. Wenn L regulär ist, ist $R_L < \infty$, wie vom Satz gefordert.

Betrachten wir die nun die Richtung »⇐«: Wir können davon ausgehen, dass der Index von R_L endlich ist, und müssen zeigen, dass daraus die Regularität von L folgt. Ein endlicher Index impliziert, dass es nur endlich viele Äquivalenzklassen gibt, in die die Wörter über dem Alphabet zerfallen. Jede Äquivalenzklasse wird durch einen *Repräsentanten* eindeutig definiert, weshalb es eine endliche Menge an Wörtern $x_1, x_2, \ldots, x_n$ gibt, die die Äquivalenzklassen $[x_1], [x_2], \ldots, [x_n]$ aufspannen. Da die Äquivalenzklassen die Sprache aller Wörter über dem Alphabet (und nicht nur die betrachtete Sprache L!) in disjunkte Anteile zerlegen, gilt

$$\Sigma^* = [x_1] \cup [x_2] \cup \cdots \cup [x_n]. \tag{2.69}$$

Auf diese Zerlegung aufbauend definieren wir einen DEA $M = (Q, \Sigma, \delta, q_0, F)$ mit der Zustandsmenge

$$Q = \{[x_1], [x_2], \ldots, [x_n]\}, \tag{2.70}$$

wobei wir die Symbole $[x_i]$ nicht mehr als Mengen, sondern als (etwas ungewöhnliche, aber dennoch vollkommen gültige) Beschriftungen der

Zustände interpretieren. Die Übergangsfunktion ist gegeben durch

$$\delta([x], a) = [xa] \tag{2.71}$$

weiterhin gilt $q_0 = [\epsilon]$ (der Startzustand korrespondiert zu der Klasse, die das leere Wort enthält) und $F = \{[x] \mid x \in L\}$ (die akzeptierenden Endzustände entsprechen den Klassen, die ein Wort aus der Sprache enthalten – wenn ein einziges Wort x einer Äquivalenzklasse Mitglied der Sprache L ist, sind es automatisch auch alle anderen Wörter, da $x\epsilon \in L \Leftrightarrow y\epsilon \in L$ wegen $xR_L y$ gilt).

Wird ein Wort von diesem M akzeptiert, können wir folgende Rechnung ausführen:

$$\begin{aligned} x \in \mathcal{L}(M) &\Leftrightarrow \hat{\delta}(q_0, x) \in F && (2.72)\\ &\Leftrightarrow \hat{\delta}([\epsilon], x) \in F && (2.73)\\ &\Leftrightarrow [x] \in F && (2.74)\\ &\Leftrightarrow x \in L, && (2.75) \end{aligned}$$

weshalb die vom Automaten akzeptierte Sprache $\mathcal{L}(M)$ identisch mit L ist. In Gleichung (2.74) haben wir dabei ausgenutzt, dass nach Definition der Übergangsfunktion $\hat{\delta}([\epsilon], x) = [x]$ gilt. □

Im Gegensatz zum Pumping-Lemma kann der Satz von Myhill und Nerode nicht nur verwendet werden, um sicherzustellen, dass eine Sprache *nicht* regulär ist, sondern kann auch eingesetzt werden, um das Gegenteil – die Regularität einer Sprache – abzusichern.

2.12.2 *Anwendungsbeispiel*

Wir betrachten folgende Sprache, um den Satz von Myhill und Nerode zur Anwendung zu bringen:

$$L \equiv \{x \in \{0, 1\}^* \mid x \text{ endet mit } 00\} \tag{2.76}$$

Als Äquivalenzklassen identifizieren wir:

- ❑ $[\epsilon] = \{x \mid x \text{ endet nicht mit } 0\}$
- ❑ $[0] = \{x \mid x \text{ endet mit } 0, \text{ aber nicht mit } 00\}$
- ❑ $[00] = \{x \mid x \text{ endet mit } 00\}$

Nachdem die Anzahl der Äquivalenzklassen endlich ist, folgern wir, dass L regulär ist. □

Man kann die identifizierten Äquivalenzklassen nutzen, um einen zum Originalautomaten äquivalenten Automaten zu zeichnen, der mit der kleinstmöglichen Anzahl von Zuständen auskommt. Um dies zu illustrieren, betrachten wir, wie sich die Mitgliedschaft in den Äquivalenzklassen verändert, wenn Zeichen des Alphabets Σ an Wörter in den einzelnen Klassen angehängt werden:

$$\begin{aligned} \delta([\epsilon], 0) &= [\epsilon 0] = [0] && (2.77)\\ \delta([\epsilon], 1) &= [\epsilon 1] = [\epsilon] && (2.78)\\ \delta([0], 0) &= [00] && (2.79) \end{aligned}$$

$$\delta([0], 1) = [01] = [\epsilon] \tag{2.80}$$

$$\delta([00], 0) = [000] = [00] \tag{2.81}$$

$$\delta([00], 1) = [001] = [\epsilon] \tag{2.82}$$

Gleichung (2.80) gilt, da $01 \in [\epsilon]$ – beide Präfixe werden durch die gleichen Suffixe zu Wörtern der Sprache vervollständigt. Gleiches gilt für Zeile (2.82).

Berücksichtigt man zusätzlich $q_0 = [\epsilon]$ und $F = \{[x] \mid x \in L\} = \{[00]\}$, ist ein vollständiger endlicher Automat entstanden, dessen Graph in Abbildung 2.24 angegeben ist.

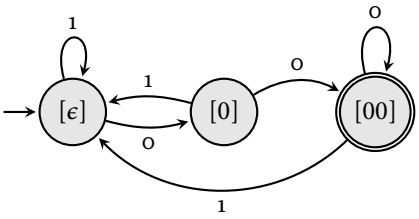


Abbildung 2.24: Endlicher Automat für Sprache (2.76), definiert auf Basis der Äquivalenzklassenzerlegung der Sprache.

In der Tat ist der auf diese Weise konstruierte Automat der kleinstmögliche Automat, der eine gegebene Sprache erkennt:

Egal, wie sehr man sich anstrengt – es wird nicht gelingen, einen Automaten mit weniger Zuständen anzugeben, der die exakt gleiche Sprache erkennt.

Satz: Minimalautomat

Der Äquivalenzklassenautomat M_0 ist minimal, d.h. er besitzt die kleinstmögliche Anzahl von Zuständen.

Von der Korrektheit des Satzes kann man sich überzeugen, indem man einige Folgerungen aus den vorhergehendem Beweis ableitet: Sei M ein Automat mit $\mathcal{L}(M) = L$, und sei M_0 der Automat, der auf Basis der eben betrachteten Äquivalenzklassenzerlegung definiert wird. Offensichtlich gilt $\mathcal{L}(M_0) = L$; beide Automaten erkennen die gleiche Sprache. Wir haben gezeigt, dass $R_M \subseteq R_L$ gilt; R_M ist eine Verfeinerung von R_L. Nachdem $R_{M_0} = R_L$, muss die Anzahl der Zustände von M entsprechend größer oder gleich der Zustandsanzahl von M_0 sein. Wenn die Zustandszahl von M und L gleich ist, sind die Automaten identisch bis auf *Isomorphie*, d.h. Umbenennung der Zustände. □

2.12.3 *Konstruktion des Minimalautomaten*

Um den Minimalautomaten systematisch ohne eine explizite Berechnung der Äquivalenzklassen zu konstruieren, überlegen wir uns, dass ein Automat *nicht* minimal ist, wenn es zwei Zustände q, q' gibt, für die gilt:

$$\hat{\delta}(q, x) \in F \Leftrightarrow \hat{\delta}(q', x) \in F. \tag{2.83}$$

Die beiden Zustände (q, q') können zu einem einzigen Zustand verschmolzen werden, da die Existenz der beiden Zustände keinen Unterschied auf die erkannte Sprachklasse hat: Gleichung (2.83) besagt nichts anderes, als dass jede Zeichenkette x, die

- ausgehend vom Zustand q akzeptiert wird, auch ausgehend vom Zustand q' akzeptiert wird.

❏ ausgehend vom Zustand q abgelehnt wird, auch ausgehend vom Zustand q' abgelehnt wird wird.

Achtung bei Schritt 2: Paare, bei denen beide Zustände in F oder beide Zustände nicht in F liegen, werden nicht markiert!

Die Erkenntnis setzt man in folgenden Algorithmus um, der in der Literatur unter der Bezeichnung *Table-Filling-Algorithmus* bekannt ist:

Table-Filling-Algorithmus

1. Tabelle aller unterschiedlichen Zustandspaare (q, q') mit $q \neq q'$ aufstellen.
2. Alle Paare (q, q') markieren, für die $q \in F$ und $q' \notin F$ oder umgekehrt gilt.
3. Für alle unmarkierten Paare (q, q') und alle $a \in \Sigma$: Testen, ob
$$(\delta(q, a), \delta(q', a))$$
bereits markiert ist. Wenn ja, *Ausgangspaar* (q, q') markieren.
4. Schritt 3 solange wiederholen, bis die Tabelle invariant bleibt, d.h. bis keine neuen Einträge mehr hinzukommen.
5. Alle nicht-markierten Paare werden zu je einem Zustand verschmolzen.

TODO: *Beispiel: Siehe Tafel*

Nachdem anschaulich klar ist, dass die Konstruktion für das betrachtete konkrete Beispiel funktioniert und den kleinstmöglichen Automaten liefert, bleibt noch formal zu zeigen, dass die Vorgehensweise für *alle* Automaten zum Ziel führt. Dies ist der Fall, wenn es keine äquivalenten Zustände mehr gibt – der Algorithmus darf also nur unterscheidbare Zustände hinterlassen.

Theorem: Korrektheit des TF-Algorithmus Zwei Zustände, die durch den Table-Filling-Algorithmus nicht unterschieden werden, sind äquivalent.

Beweis: Wir führen einen Widerspruchsbeweis. Gegeben sei ein DEA $M = (Q, \Sigma, \delta, q_0, F)$. Wir nehmen an, dass ein unterscheidbares Zustandspaar p, q existiert, für dass es eine Zeichenkette $w \in \Sigma^*$ gibt, für die *entweder* $\hat{\delta}(p, w)$ *oder* $\hat{\delta}(q, w)$ gilt, aber nicht beides zugleich.

Der Table-Filling-Algorithmus soll im Widerspruch zur Aussage des Satzes das Paar (p, q) *nicht* unterscheiden. Im folgenden bezeichnen wir (p, q) als »schlechtes Paar«.

Wenn ein schlechtes Paar existiert, gibt es auch eine kürzeste Zeichenreihe $w = a_1 a_2 \cdots a_n$ mit $|w| = n$, für die entweder $\hat{\delta}(p, w)$ oder $\hat{\delta}(q, w)$ gilt (wenn es mehrere Zeichenketten mit dieser Eigenschaft gibt, wählt man die kürzeste davon aus). Der Fall $w = \epsilon$ kann nicht auftreten, da ununterscheidbare Anfangszustände sicher im ersten Schritt des TF-Algorithmus markiert werden. Wir können also von $|w| \geq 1$ ausgehen.

Wir betrachten die Zustände

$$r = \delta(p, a_1), \qquad s = \delta(q, a_1), \tag{2.84}$$

die der Automat einnimmt, wenn das erste Zeichen ausgehend von den beiden schlechten Zuständen gelesen wurde. Die Zustände r und s sind dann

sicher durch die Zeichenkette $a_2 \dots a_n$ unterscheidbar, da die Zeichenkette w die Zustände p und q unterscheidet. Nachdem $|a_2 \cdots a_n| < n$ gilt, ist die unterscheidende Zeichenkette kürzer als die Zeichenkette, bei der die Unterscheidung des schlechten Paares fehlschlägt. Der TF-Algorithmus funktioniert daher für das Paar (r, s), das entsprechend korrekt unterschieden und markiert wird.

Das Ausgangspaar für (r, s) ist das Paar (p, q) wegen $r = \delta(p, a_1)$ und $s = \delta(q, a_1)$. Nach Konstruktion des Algorithmus wird aber (p, q) markiert, wenn (r, s) markiert ist – das Paar wird damit im Widerspruch zu obiger Annahme doch korrekt markiert. Der Algorithmus funktioniert daher korrekt. □

2.13 *Abschlusseigenschaften regulärer Sprachen*

Reguläre Sprachen sind vergleichsweise robust unter Operationen, die Sprachen weiterverarbeiten oder zwei Sprachen miteinander kombinieren. Die Sprachklasse ist abgeschlossen gegenüber folgenden wichtigen Operationen:

Eine etwas andere Formulierung für Abgeschlossenenheit unter einer Operation ist, dass man zwei Sprachen auf eine spezifische Art miteinander verknüpfen kann, ohne dabei die Sprachklasse zu verlassen und deren Eigenschaften zu verlieren. Dies ist beispielsweise praktisch, wenn ein großer, komplizierter regulärer Ausdruck aus der Komposition vieler kleinerer Teilsprachen entsteht.

Abschlusseigenschaften regulärer Sprachen

Wir betrachten zwei beliebige Sprachen $L_1, L_2 \in \mathcal{L}_3$. Die regulären Sprachen sind abgeschlossen unter

- ❏ Vereinigung, also $L_1 \cup L_2 \in \mathcal{L}_3$.
- ❏ Schnitt, also $L_1 \cap L_2 \in \mathcal{L}_3$.
- ❏ Komplement, also $\bar{L}_1 \in \mathcal{L}_3$.
- ❏ Produkt, also $L_1 L_2 \in \mathcal{L}_3$.
- ❏ Stern, also $L_1^* \in \mathcal{L}_3$.

Beweis: Die Regularität von Vereinigungs-, Produkt- und Stern-Operation folgt unmittelbar aus den Eigenschaften regulärer Ausdrücke.

Um zu zeigen, dass Komplementbildung eine reguläre Operation ist, betrachten wir einen Automaten $M = (Q, \Sigma, \delta, q_0, F)$, der die Sprache $L = \mathcal{L}(M)$ erkennt. Die Komplementsprache $\bar{L}$ besteht aus allen Wörtern, die *nicht* in L enthalten sind – also aus allen Wörtern, die der Automat M ablehnt. Vertauscht man die akzeptierenden und ablehnenden Zustände, erhält man den Automaten $M' = (Q, \Sigma, \delta, q_0, Q - F)$, der $\bar{L} = \mathcal{L}(M')$ erkennt.

Nach den Gesetzen von de Morgan gilt für zwei Mengen L_1 und L_2 die Identität

$$L_1 \cap L_2 = \overline{\bar{L}_1 \cup \bar{L}_2}. \tag{2.85}$$

Interpretiert man L_i als Sprachen, sieht man, dass sich die rechte Seite aus regulären Operationen (Komplementbildung und Vereinigung) zusammensetzt. Wenn die rechte Seite regulär ist, muss dies aber auch die linke Seite sein, was die Regularität der Schnittoperation sicherstellt. □

Die Abschlusseigenschaften führen zu einer neuen Beweistechnik, die verwendet werden kann, um die Nicht-Regularität einer Sprache nachzuweisen: Angenommen, man betrachtet eine Sprache L_2, von der unbekannt ist, ob sie regulär ist, und eine reguläre Sprache L_1. Berechnet

man $L_3 = L_1 \circ L_2$ (für eine geeignete Operation $\circ$ aus obiger Liste) und kann nachweisen, dass L_3 nicht regulär ist – dies ist je nach entstehender Sprache möglicherweise leichter als ein direkter Beweis für L_2 –, folgt automatisch, dass L_2 nicht regulär gewesen sein kann, da anderenfalls die Abschlusseigenschaften verletzt wären.

2.13.1 Entscheidbarkeit

Reguläre Sprachen sind eine sehr gutmütige, leicht zu berechnende Sprachklasse: Alle Probleme, die in Abschnitt 2.5.3 angesprochen wurden, können für beliebige reguläre Sprachen gelöst werden (wir geben die teils offensichtlichen Beweise nicht im Detail an, sondern verweisen auf die Diskussion in der Vorlesung):

- *Wortproblem*: Gegeben ein Wort $w \in \Sigma^*$ und eine Sprache $L \in \Sigma^*$: Gilt $w \in L$ oder $w \notin L$?

  Beweis: DEA verwenden.

- *Leerheitsproblem*: Enthält eine Sprache L kein Wort, d.h. gilt $L = \varnothing$?

  Beweis: Prüfen, ob DEA keinen Pfad von Start- zu (einem) Endzustand besitzt.

- *Endlichkeitsproblem*: Besitzt L nur endlich viele Elemente, d.h. gilt $|L| \neq \infty$?

  Beweis: $|\mathcal{L}(M)| = \infty$ gilt genau dann, wenn ein Zyklus existiert, der vom Startzustand q_0 aus erreichbar ist und in einem akzeptierenden Endzustand aus F endet. Durch Analyse des Automaten ist entscheidbar, ob ein entsprechender Zyklus existiert oder nicht.

- *Schnittproblem*: Gilt für zwei Sprachen L_1, L_2, dass $L_1 \cap L_2 = \varnothing$?

  Beweis: Seien

  $$M_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1) \tag{2.86}$$
  $$M_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2) \tag{2.87}$$

  Maschinen mit $L_i = \mathcal{L}(M)$ (oBdA verwenden beide Automaten das gleiche Alphabet). Ein Automat für $L_1 \cap L_2$ ist gegeben durch

  $$M = (Q_1 \times Q_2, \Sigma, \delta, (q_0^1, q_0^2), F_1 \times F_2) \tag{2.88}$$

  mit der Übergangsfunktion

  $$\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a)). \tag{2.89}$$

  Die Maschine vollzieht beide Automatenbewegungen »parallel« nach und akzeptiert ein Wort nur, wenn die beiden Ausgangsautomaten das Wort akzeptieren.

- *Äquivalenzproblem*: Sind zwei Sprachen L_1, L_2 gleich, d.h. gilt $L_1 = L_2$?

  Beweis: Prüfen, ob die Minimalautomaten beider Sprachen (bis auf Isomorphie) identisch sind.

2.14 Kontextfreie Sprachen

2.14.1 Definition und Beispiele

Die regulären Sprachen sind mittlerweile sehr ausführlich charakterisiert; wir wenden uns daher der nächsthöheren Sprachklasse der in Abschnitt 2.5 diskutierten Hierarchie zu: Den kontextfreien Sprachen. Zur Erinnerung an Ihre Definition sei festgestellt, dass eine kontextfreie Sprache wie immer durch Grammatik $G = (V, \Sigma, P, S)$ gegeben ist; ihre Regeln der Form $l \rightarrow r \in P$ unterliegen zwei Beschränkungen:

- ❑ $|l| \leq |r|$, d.h. die rechte Seite einer Regel ist immer mindestens so lang wie die linke Seite. Dies bedeutet, dass die Länge der Kette aus Terminal- und Nicht-Terminal-Symbolen während der Ableitung eines Wortes über einer kontextfreie Sprache niemals kürzer werden kann.
- ❑ $l \in V$, d.h. eine Regel ersetzt immer nur ein einziges Nicht-Terminal-Symbol – unabhängig davon, in welchem Kontext dieses Symbol auftritt (welche Zeichen also links und rechts davon stehen). Diese Einschränkung ist die offensichtliche Grundlage der Bezeichnung »*kontextfreie* Sprache«.

Ebenfalls erinnern wir daran, dass eine kontextfreie Grammatik G ϵ-frei gemacht werden kann. Obwohl wir im Folgenden nicht jede Grammatik in der dafür notwendigen Form angeben, können wir in Beweisen davon ausgehen, dass eine Grammatik die entsprechende Eigenschaft besitzt.

Als erstes Beispiel betrachten wir die Sprache aller *Palindrome*, also Wörter, die von rechts nach links gelesen gleich sind wie von links nach rechts gelesen. Sie werden durch die Menge

$$L_\mathrm{p} \equiv \{w \in \Sigma^* | w = v0v^\mathrm{R} \vee w = v1v^\mathrm{R}\} \tag{2.90}$$

gegeben.

Die Sprachdefinition funktioniert genau betrachtet nur für Palindrome ungerader Länge, da $|v0v^\mathrm{R}| = 2|v| + 1$. Palindrome mit einer geraden Anzahl von Buchstaben könnten leicht durch vv^R generiert werden, was für dieses Beispiel allerdings nur zu formalen Schwierigkeiten ohne strukturellen Erkenntnisgewinn führt, weshalb wir uns auf ungerade Wortlängen beschränken.

Eine kontextfreie Grammatik fur Palindrome über $\Sigma = \{0, 1\}$ ist einfach anzugeben:

$$\langle S \rangle \rightarrow \mathtt{0} \mid \mathtt{1} \mid \epsilon \mid \mathtt{0}\langle S \rangle \mathtt{0} \mid \mathtt{1}\langle S \rangle \mathtt{1}$$

Die Sprache eignet sich unmittelbar, um zu zeigen, dass $\mathcal{L}_3 \subset \mathcal{L}_2$ gilt; reguläre Sprachen sind eine *echte* Teilmenge der kontextfreien Sprachen.

Beweis: Nach Konstruktion der Sprachklassen gilt, dass jede reguläre Sprache automatisch kontextfrei ist, also $\forall L \in \mathcal{L}_3 : L \in \mathcal{L}_2$ gilt. Wie man unmittelbar durch Anpassung vorhergehender Pumping-Lemma-Beweise erkennt, sind binäre Palindrome *nicht regulär*, d.h. $L_\mathrm{p} \notin \mathcal{L}_3$. Wie die eben vorgestellte Grammatik zeigt, sind binäre Palindrome allerdings *kontextfrei*, d.h. $L_\mathrm{p} \in \mathcal{L}_2$. Ergo gilt $\mathcal{L}_3 \subset \mathcal{L}_2$ □

In Kapitel 2.11.3 haben wir gezeigt, dass die Dyck-Sprache D_1 nicht regulär ist. Man kann aber leicht eine Grammatik dafür angeben:

$$\langle S \rangle \rightarrow (\langle S \rangle) \mid \epsilon$$

Nachdem alle Bedingungen an eine kontextfreie Grammatik erfüllt sind, ist die Dyck-Sprache offenbar kontextfrei.

2.14.2 *Die Chomsky-Normalform*

Die größeren Freiheiten, die bei der Spezifikation kontextfreier Sprachen zur Verfügung stehen, scheinen es auf den ersten Blick schwerer zu machen, Aussagen über die Sprachklasse zu beweisen, da in jedem Beweis eine hohe Vielfalt an potentiellen Regelstrukturen berücksichtigt werden muss. Man kann die Vielfalt allerdings einschränken, indem man eine Grammatik in die *Chomsky-Normalform* transformiert: Den Regeln werden zusätzliche Einschränkungen gegenüber den Bedingungen der allgemeinen Definition kontextfreier Sprachen auferlegt, um sie in eine einfachere Struktur zu bringen:

Definition: Chomsky-Normalform Eine Grammatik G mit $\epsilon \notin \mathcal{L}(G)$ ist in *Chomsky-Normalform* gegeben, wenn alle Regeln eine der beiden Formen

$$A \to BC$$
$$A \to \mathtt{a}$$

haben, wobei $A, B, C \in V$ und $\mathtt{a} \in \Sigma$. ∎

Eine Normalform ist nur dann sinnvoll, wenn sie die Ausdrucksstärke der Sprachklasse nicht verändert, also für alle möglichen kontextfreien Sprachen existiert. Dass dies möglich ist, garantiert folgender Satz:

Satz

Zu jeder kontextfreien Grammatik G, die das leere Wort nicht enthält ($\epsilon \notin \mathcal{L}(G)$), gibt es eine äquivalente Grammatik G' in Chomsky-Normalform, die die gleiche Sprache generiert, für die also $\mathcal{L}(G) = \mathcal{L}(G')$ gilt.

Zum Beweis betrachten wir einen expliziten Algorithmus zur Transformation einer beliebigen ϵ-freien kontextfreien Grammatik (kfG) $G = (V, \Sigma, P, S)$ in die Chomksy-Normalform (CNF). Die Umwandlung erfolgt in einem zweistufigen Verfahren.

Schritt 1: Zunächst verwandeln wir G in eine neue Grammatik G', die nur die beiden Regelformen

1. $A \to a$ mit $A \in V', a \in \Sigma$
2. $A \to x$ mit $A \in V', x \in (V' \cup \Sigma)^+, |x| \geq 2$

Dazu verwenden wir folgenden Konvertierungsalgorithmus:

- ❏ *Zyklen eliminieren*: Gibt es $B_1, B_2, \ldots, B_k$ mit $B_1 \to B_2 \to B_3 \to \cdots \to B_k \to B_1$, dann ersetze alle B_i durch B
- ❏ *Variablen sortieren und transformieren*: $V' = \{A_1, \ldots, A_n\}$ wird so numeriert, dass $A_i \to A_j \Rightarrow i < j$ gilt. Dann Elimination dieser Regeln:
  - ❏ Iteriere von $k = n - 1$ bis $k = 1$
  - ❏ Für alle $A_k \to A_{k'}$ mit $k' > k$ und $A_{k'} \to x_1|x_2|x_3|\cdots|x_m$: Neue Regel einführen: $A_k \to x_1|x_2|\cdots|x_m$

Schritt 2: Wir verwandeln die eingeschränkte Form aus Schritt 1 vollständig in die CNF:

1. Eine neue Regel $B \to a$ wird für jedes Terminalzeichen `a` hinzugefügt: $\forall a \in \Sigma\colon V_{\text{neu}} = V \cup B, P_{\text{neu}} = P \cup (B \to a)$.
2. Alle a auf der rechten Regelseite werden durch B ersetzt (außer eine Regel ist bereits in Form $A \to a$). Dies ergibt Regeln der Form:

$$A \to a \text{ oder } A \to B_1 B_2 \dots B_k, k \geq 2 \tag{2.91}$$

3. Für alle Regeln $A \to B_1 B_2 \dots B_k$ mit $k \geq 3$: Neue Variablen $C_1, C_2, \dots C_{k-2}$ einführen mit

$$A \to B_1 C_1,\ C_1 \to B_2 C_2,$$
$$C_2 \to B_3 C_3,\ \dots,\ C_{k-2} \to B_{k-1} B_k.$$

Wir verzichten auf einen formalen Beweis, dass der beschriebene Algorithmus für jede kontextfreie Grammatik funktioniert.

2.14.3 *Beispiel zur Chomsky-Normalform*

Wir illustrieren die Transformation in die CNF anhand folgender Beispielgrammatik:

Die Grammatik ist aus Sipser, Beispiel 2.10 entnommen.

$$\begin{aligned}
\langle S\rangle &\to \langle A\rangle\langle B\rangle \\
\langle S\rangle &\to \langle A\rangle\langle B\rangle\langle A\rangle \\
\langle A\rangle &\to \mathtt{a}\langle A\rangle \\
\langle A\rangle &\to \mathtt{a} \\
\langle B\rangle &\to \langle B\rangle\mathtt{b} \\
\langle B\rangle &\to \mathtt{b} \mid \epsilon
\end{aligned} \tag{2.92}$$

Nachdem $\epsilon \notin \mathcal{L}(G)$ gilt, kann die Grammatik komplett ϵ-frei gemacht werden; dazu wird die Produktion $\langle B\rangle \to \epsilon$ an allen möglichen Stellen eingesetzt und das Resultat als neue Regel in die Grammatik aufgenommen:

$$\begin{aligned}
\langle S\rangle &\to \langle A\rangle\langle B\rangle \mid \langle A\rangle \mid \langle A\rangle\langle B\rangle\langle A\rangle \mid \langle A\rangle\langle A\rangle \\
\langle A\rangle &\to \mathtt{a}\langle A\rangle \mid \mathtt{a} \\
\langle B\rangle &\to \langle B\rangle\mathtt{b} \mid \mathtt{b}
\end{aligned} \tag{2.93}$$

Auf der ϵ-freien Form führen wir die eben besprochenen Konvertierungsschritte aus:

1. Numerieren und transformieren: Die Variablenmenge $V = \{S, A, B\}$ wird in $V = \{A_0, A_1, A_2\}$ umbenannt. Die einzige Produktionsregel, die ein Nichtterminal in ein einziges Nichtterminal überführt, ist $\langle S\rangle \to \langle A\rangle$; mit den neuen Symbolen erhält sie die Form $\langle A\rangle_0 \to \langle A\rangle_1$, was mit der Anforderung $A_i \to A_j \Rightarrow i < j$ kompatibel ist. Die Regeln haben nun die Form

$$\begin{aligned}
\langle A\rangle_0 &\to \langle A\rangle_1\langle A\rangle_2 & \langle A\rangle_0 &\to \langle A\rangle_1 \\
\langle A\rangle_0 &\to \langle A\rangle_1\langle A\rangle_2\langle A\rangle_1 & \langle A\rangle_0 &\to \langle A\rangle_1\langle A\rangle_1 \\
\langle A\rangle_1 &\to \mathtt{a}\langle A\rangle_1 & \langle A\rangle_1 &\to \mathtt{a} \\
\langle A\rangle_2 &\to \langle A\rangle_2\mathtt{b} & \langle A\rangle_2 &\to \mathtt{b}
\end{aligned} \tag{2.94}$$

2. Da keine Zyklen vorhanden sind, entfällt der Eliminationsschritt.

3. Produktionsregeln für Terminalzeichen einführen: Wir erweitern die Regelmenge um die zusätzlichen Einträge

$$\langle A\rangle \to \mathtt{a} \qquad\qquad \langle B\rangle \to \mathtt{b}. \tag{2.95}$$

Die neuen Regeln werden in allen bestehenden Regeln angewendet, in denen Terminalzeichen auf der rechten Seite auftreten:

$$\begin{aligned}
&\langle A\rangle_0 \to \langle A\rangle_1\langle A\rangle_2 && \\
&\langle A\rangle_0 \to \langle A\rangle\langle A\rangle_1 && \langle A\rangle_0 \to \mathtt{a} \\
&\langle A\rangle_0 \to \langle A\rangle_1\langle A\rangle_2\langle A\rangle_1 && \langle A\rangle_0 \to \langle A\rangle_1\langle A\rangle_1 \\
&\langle A\rangle_1 \to \langle A\rangle\langle A\rangle_1 && \langle A\rangle_1 \to \mathtt{a} \\
&\langle A\rangle_2 \to \langle A\rangle_2\langle B\rangle && \langle A\rangle_2 \to \mathtt{b} \\
&\langle A\rangle \to \mathtt{a} && \langle B\rangle \to \mathtt{b}.
\end{aligned} \tag{2.96}$$

4. Lange Ketten aufbrechen: Alle Regeln, die mehr als zwei Nicht-Terminal-Symbole auf der rechten Seite aufweisen, müssen in Zweierketten aufgebrochen werden. Dies betrifft lediglich $\langle A\rangle_0 \to \langle A\rangle_1\langle A\rangle_2\langle A\rangle_1$, die durch die beiden Regeln

$$\langle A\rangle_0 \to \langle A\rangle_1\langle C\rangle \qquad\qquad \langle C\rangle \to \langle A\rangle_2\langle A\rangle_1 \tag{2.97}$$

ersetzt wird. Damit ist die Grammatik in Chomsky-Normlform.

Als zweites Beispiel betrachten wie die Umwandlung der Dyck-Sprache in CNF:

$$\langle S\rangle \to \mathtt{ab} \mid \mathtt{a}\langle S\rangle\mathtt{b} \tag{2.98}$$

Mit Hilfe des gesunden Menschenverstandes sieht man, dass

$$\begin{aligned}
&\langle S\rangle \to \langle A\rangle\langle B\rangle && \langle S\rangle \to \langle A\rangle\langle C\rangle \\
&\langle C\rangle \to \langle S\rangle\langle B\rangle && \langle A\rangle \to \mathtt{a} \\
&\langle B\rangle \to \mathtt{b}
\end{aligned} \tag{2.99}$$

eine äquivalente Grammatik in CNF ist. Abbildung 2.25 zeigt den binären Ableitungsbaum, der für das Wort »aaabbb« entsteht.

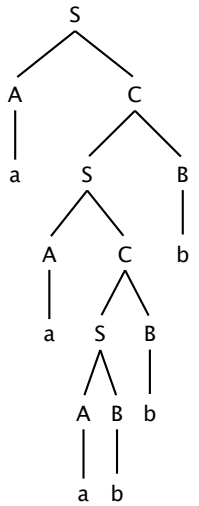


Abbildung 2.25: Ableitungsbaum für »aaabbb« mit Grammatik (2.99).

2.14.4 *Die Greibach-Normalform*

Die Chomsky-Normalform ist nicht die einzige mögliche Normalform für kontextfreie Grammatiken. In der Literatur findet beispielsweise die Greibach-Normalform häufige Verwendung:

Greibach-Normalform

Eine ϵ-freie kontextfreie Grammatik G ist in Greibach-Normalform, wenn alle Regeln die Form

$$A \to \mathtt{a}B_1B_2B_3 \ldots B_k, k \geq 0 \tag{2.100}$$

haben ($B_i \in V, a \in \Sigma$)

Ohne Beweis (siehe Literatur) bemerken wir, dass es zu jeder ϵ-freien kontextfreien Grammatik G eine Grammatik G' in Greibach-Normalform mit $\mathcal{L}(G') = \mathcal{L}(G)$ gibt, die systematisch konstruiert werden kann. Die GNF bildet ein interessantes Bindeglied zwischen regulären und kontextfreien Sprachen, da reguläre Grammatiken automatisch immer in Greibach-Normalform (mit $k = 0$ oder $k = 1$) angegeben sind.

2.14.5 Pumping-Lemma für kontextfreie Sprachen

Nachdem kontextfreie Sprachen nicht die allgemeinste Sprachklasse sind, gibt es offensichtlich Sprachen, die nicht durch eine entsprechende Grammatik ausgedrückt werden können, sondern beispielsweise eine kontextsensitive Grammatik benötigen. Um beweisen zu können, dass eine Sprache sicher nicht kontextfrei repräsentiert werden kann, steht eine Variante des Pumping-Lemmas zur Verfügung, die allerdings etwas komplizierter strukturiert ist als im regulären Fall. Bevor wir den Satz formal formulieren, betrachten wir die Funktionsweise des Lemmas anhand eines Beispiels, das sich auf die Tatsache stützt, dass kontextfreie Ableitungen eines Wortes immer in einer Baumstruktur dargestellt werden können. Abbildung 2.26 zeigt dies anhand einer Grammatik für arithmetische Formeln.

$$
\begin{array}{lcl}
\mathrm{E} & \rightarrow & \mathrm{E} + \mathrm{T} \mid \mathrm{E} - \mathrm{T} \mid \mathrm{T} \\
\mathrm{T} & \rightarrow & \mathrm{T} \times \mathrm{F} \mid \mathrm{T}/\mathrm{F} \mid \mathrm{F} \\
\mathrm{F} & \rightarrow & (\mathrm{E}) \mid x \mid y \mid z
\end{array}
$$

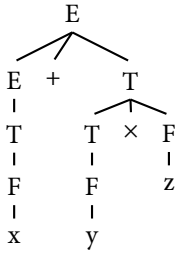


Abbildung 2.26: Grammatik für arithmetische Formeln (links), die sich aus Ausdrücken (Expression, E), Termen (Term, T) und Faktoren (Factor, F) zusammensetzen und über den Variablen x, y und z aufgebaut sind. Die rechte Seite zeigt einen Ableitungsbaum für das Wort »$x + y \times z$«.

Strukturelle Aussagen über Ableitungsbäume sind allerdings nur schwer möglich, wenn alle Möglichkeiten kontextfreier Sprachen genutzt werden – beispielsweise haben manche Knoten in Abbildung einen Nachfolger, andere dafür drei. Die Situation wird allerdings sehr uniform, wenn man Grammatiken in CNF betrachtet, wie in Abbildung 2.27 gezeigt: Jeder Ableitungsbaum besitzt in diesem Fall die Struktur eines Binärbaums, in dem jeder Knoten exakt zwei Nachfolger besitzt (mit Ausnahme der Blättern, die notwendigerweise keine Nachfolger haben).

Anhand der Struktur von Binärbäumen werden wir bestimmte Eigenschaften herleiten, die jede kontextfreie Sprache erfüllen muss. Treffen sie auf eine Sprache *nicht* zu, kann sie analog der Argumentation beim regulären Pumping-Lemma auch nicht kontextfrei sein. Wir betrachten dazu die bereits bekannte Sprache aller korrekt geklammerten Ausdrücke, die durch die einfachen Regeln

$$\langle S \rangle \rightarrow \langle S \rangle \langle S \rangle \mid (\langle S \rangle) \mid ()$$

definiert ist. Diese sind noch nicht in CNF, können aber trivial konvertiert werden, wodurch man das Regelwerk

$$
\begin{aligned}
\langle S \rangle &\rightarrow \langle S \rangle \langle S \rangle \mid \langle L \rangle \langle A \rangle \mid \langle L \rangle \langle R \rangle \\
\langle L \rangle &\rightarrow (\\
\langle R \rangle &\rightarrow)
\end{aligned}
$$

Abbildung 2.27: Beispiel für einen Ableitungsbaum in Chomsky–Normalform.

$$\langle A \rangle \rightarrow \langle S \rangle \langle R \rangle$$

erhält. Abbildung 2.28 zeigt auf der linke Seiten den Ableitungsbaum für das Wort »((()))« zusammen mit einer Reihe an Manipulationen, die man mit dem Baum ausführen kann.

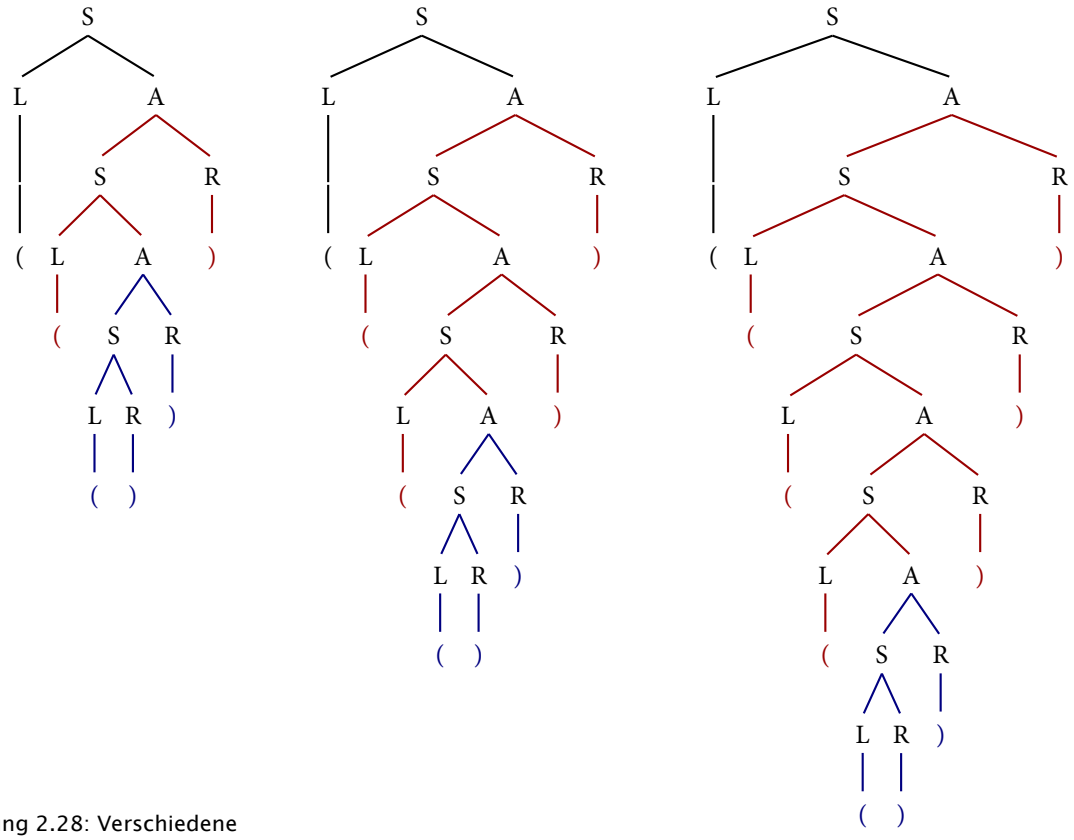


Abbildung 2.28: Verschiedene Manipulationen am Ableitungsbaum des Wortes »((()))«.

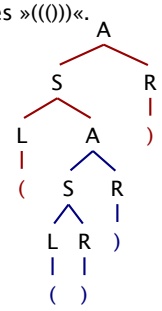


Abbildung 2.29: »Roter« Teilbaum des ersten Ableitungsbaumes aus Abbildung 2.28.

Abbildung 2.30: Ableitungsbaum aus Abbildung 2.28 ohne blauen Teilbaum.

Zunächst fällt auf, dass das Nicht-Terminal-Symbol $\langle A \rangle$ mehrfach im Ableitungsbaum auftritt: Einmal im »roten« Teilbaum, und darin nochmals im »blauen« Teilbaum. Der »rote« Teilbaum (nochmals explizit in Abbildung 2.29 gezeigt) wird vom Nicht-Terminal-Symbol $\langle A \rangle$ aus abgeleitet und darf daher an jeder beliebigen Stelle in der Ableiteitungsbaum eingesetzt werden, an der ein $\langle A \rangle$ auftritt – dies ist eine der elementaren Eigenschaften kontextfreier Grammatiken.

Entfernt man den »blauen« Teilbaum aus dem ursprünglichen Ableitungsbaum (siehe Abbildung 2.30), entsteht ein freies $\langle A \rangle$, in das der Teilbaum aus Abbildung 2.29 eingesetzt werden kann – das Resultat findet sich im mittleren Teil von Abbildung 2.28. Wie man leicht aus dem Baum ermittelt, ist das abgeleitete Wort von »((()))« auf »(((()))))« angewachsen. Wendet man die Manipulation (blauen Teilbaum entfernen, Teilbaum aus Abbildung 2.29 einsetzen) nochmals an, wächst das Wort weiter auf »((((()))))«, und es ist offensichtlich, dass man die Operation beliebig oft wiederholen könnte, um beliebig lange Wörter zu erzeugen. Dies entspricht der Aufpump-Operation, die wie beim regulären Pumping-Lemma besprochen haben. Im kontextfreien Fall hat dies funktioniert, da in einem Pfad von der Wurzel des Baumes bis zu den Blättern zweimal das gleiche Nicht-Terminal-Symbol (in diesem Fall $\langle A \rangle$) aufgetreten ist. Um die Konstruktion zu verallgemeinern und ein

Pumping-Lemma fur kontextfreie Sprachen zu erhalten, müssen wir zwei Fragen beantworten:

❏ Unter welchen quantitativen Bedingungen kann man garantieren, dass ein Nicht-Terminalzeichen sicher mehrfach in einem Ableitungspfad auftritt?

❏ Wie ist die Struktur der aufgepumpten (im Beispiel rot/blau) und fixen (im Beispiel schwarz) Anteile?

Für die erste Frage erweist sich folgender Satz als hilfreich:

Satz: Pfadlänge in Binärbäumen

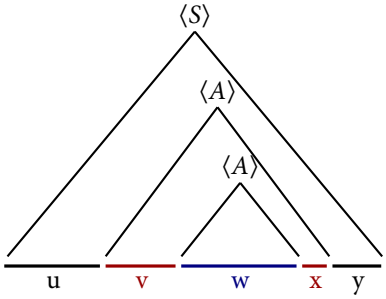
Sei $k \in \mathbb{N}$ und B ein Binärbaum. Mindestens ein Pfad der Länge $\geq k$ existiert in B, wenn die Blattanzahl $|B| \geq 2^k$ ist.

Beweis: Wir verwenden eine strukturelle Induktion über die Länge k des längsten Pfades in einem Baum. Der Induktionsanfang ($k = 0$) ist trivial; ein Baum mit einem einzigen Knoten besitzt offensichtlich einen Pfad der Länge 0.

Um den Induktionsschritt $k \to k+1$ zu zeigen, betrachten wir den Aufbau eines Baumes aus zwei Teilbäume, wie er in Abbildung 2.31 zu sehen ist.

Sei $|B_1| \geq 2^{k_1}$, $|B_2| \geq 2^{k_2}$. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $|B_1| \geq |B_2|$ gilt. Damit gilt also $|B| \leq 2 \cdot 2^{k_1} = 2^{k_1+1}$; die untere Schranke der Bedingung des Satzes ist also in jedem Fall erfüllt. Ein Pfad der Länge k_1 in B_1 existiert nach Induktionsvoraussetzung; ein Pfad der Länge $k_1 + 1$ existiert aufgrund der in Abbildung 2.31 gezeigten Konstruktion, die die beiden Bäume miteinander verbindet: Ein Pfad der Länge k_1 in B_1 wird durch den Schritt vom Wurzelelement von B_1 zu $\langle S \rangle$ zu einem Pfad der Länge $k + 1$ erweitert. □

Betrachten wie nun eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ mit $|V|$ Nicht-Terminalsymbolen (die Anzahl muss nicht explizit numerisch bekannt sein, sie ist aber sicher endlich). Wie Abbildung 2.32 illustriert, macht ein Pfad der Länge $|V|$ die Wiederholung eines Nicht-Terminal-Symbols erforderlich, da $|V| + 1$ Knoten notwendig sind. Dies ist für Wörter ab Länge $\geq 2^{|V|}$ garantiert.



Nun bleibt noch die Frage nach der Struktur aufgepumpter und fixer Anteile zu beantworten, die wir anhand von Abbildung 2.33 beantworten. Aus der Grammatik G wählen wir dazu ein Wort z mit $|z| \geq 2^{|V|}$, das durch einen binärer Syntaxbaum mit $\geq 2^{|V|}$ Blättern repräsentiert wird. Ein Pfad der Länge $\geq |V|$ ist nach obigem Satz vorhanden; dieser enthält $|V| + 1$ Nicht-

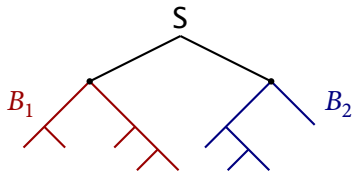


Abbildung 2.31: Aufbau eines Binärbaumes aus zwei Teilbäumen. Die Annahme ist keine Beschränkung, da man einfach die Bezeichner 1 und 2 vertauschen könnte, wenn sie nicht gelten würde.

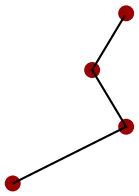


Abbildung 2.32: Pfadlänge und Knotenanzahl: Der gezeigte Pfad mit Länge 3 besteht aus 4 Knoten.

Abbildung 2.33: Struktur fixer und aufgepumter Anteile im Pumping-Lemma für kontextfreie Sprachen.

Terminal-Symbole. Ein mehrfaches Auftreten eines Symbole $\langle A \rangle$ ist daher garantiert. Läuft die Suche von unten nach oben ob, kann man zudem sicher sagen, dass $\langle A \rangle$ maximal $|V|$ Schritte von den Blättern entfernt ist. Weiterhin muss $\langle A \rangle$ in einer Regel der Form $\langle A \rangle \rightarrow \langle B \rangle \langle C \rangle$ existieren, da ansonsten kein weiteres $\langle A \rangle$ erzeugt werden kann (CNF!). Damit gilt auch $|vx| \geq 1$, die roten Anteile enthalten also zusammengenommnen mindestens ein Zeichen. Nachdem das oberes $\langle A \rangle$ maximal $|V|$ Schritte von den Blättern entfernt ist, steht zusätzlich fest, dass $|vwx| \leq 2^{|V|}$ gilt.

Einer der beiden Teilstrings kann leer sein, falls sich das Symbol $\langle A \rangle$ direkt am »Rand« des Ableitungsbaums befindet.

Damit sind alle Zutaten vorhanden, um das Pumping-Lemma für kfS als konsistenten Satz zu formulieren:

Pumping-Lemma für kontextfreie Sprachen

Sei L eine kontextfreie Sprache. Dann gibt es eine Konstante n, so dass sich alle Wörter $z \in L, |z| \geq n$ zerlegen lassen in $z = uvwxy$, so dass

1. $|vwx| \leq n$ (*Der Mittelteil des Wortes ist begrenzt.*)
2. $|vx| \geq 1$ (*Mindestens eine aufgepumpte Zeichenreihe ist nicht leer.*)
3. Für alle $i \geq 0$ git $uv^i wx^i y \in L$.

2.14.6 Anwendungsbeispiel

Als Anwendungsbeispiel für das eben bewiesene Lemma betrachten wir die Frage, ob die Sprache

$$L \equiv \{\mathrm{a}^i \mathrm{b}^i \mathrm{c}^i \mid i \in \mathbb{N}^+\} \tag{2.101}$$

Beispiele für Wörter der Sprache sind »abc«, »aabbcc«, »aaabbbcccc«, …

kontextfrei ist oder nicht. Wir werden zeigen, dass dies nicht der Fall ist. Wie bei den vorhergehenden PL-Beweisen nehmen wir dazu zunächst an, die Sprache sei kontextfrei, und das PL gelte daher. Entsprechend muss es ein $n \in \mathbb{N}$, Wort $z = \mathrm{a}^i \mathrm{b}^i \mathrm{c}^i$ geben, das hinreichend lang ($|z| \geq n$ ist und in der Zerlegung $uvwxy$ darstellbar ist, wobei $|vx| \geq 1$ und $|vwx| \leq n$ gelten. Wir wählen $i = n$, wodurch das Wort die Längenbedingung erfüllt – unabhängig davon, ob der konkrete numerische Wert für n bekannt ist, können wir sicher davon ausgehen, dass ein passendes Wort in der Sprache enthalten ist. Wir argumentieren nun in vier Schritten:

1. $uv^0 wx^0 y = uwy \in L$, ebenso $uvwxy \in L$.
2. vx muss die gleiche Anzahl der Buchstaben a,b,c enthalten – aufgrund der Bedingung $|vx| \geq 1$ ist mindestens ein Buchstabe enthalten.
3. Aufgrund der Bedingung $|vwx| \leq n$ kann vwx aber nicht gleichzeitig a und c enthalten.
4. vx *kann* also nicht gleiche Anzahl a,b,c enthalten.

Betrachtet man als illustratives Beispiel, das keine Beweiskraft hat, den Fall $n = 3$, sieht man, dass nur die Teilstrings »aaa«, »aab«, »abb«, »bbb«, »bbc«, »bcc« und »ccc« möglich sind – die Forderung nach einem gleichmäßigen Auftreten der Buchstaben kann nicht eingehalten werden.

Das aufgepumpte Wort ist damit nicht in der Sprache enthalten, was einen Widerspruch zur Annahme darstellt, dass das PL gilt und das Wort Mitglied der Sprache ist. Konsequenterweise kann L deshalb nicht kontextfrei sein. □

Prinzipiell könnte man Sprachen, die sich als nicht kontextfrei herausgestellt haben, auch mit Grammatiken der höheren Chomsky-Hierarchiebenen

(es verbleiben die kontextsensitiven Sprachen und Phrasenstrukturgrammatiken) beschreiben. In der Praxis beschränkt man sich allerdings auf kontextfreie Sprachen, da diese mit vertretbarem Aufwand verarbeitet werden können, wie wir später noch im Detail analysieren werden. Dennoch gibt es verschiedene Tricks, um nicht-kontextfreie Sprachen mit den einfachen Mitteln kontextfreier Sprachen handzuhaben, indem man beispielsweise eine zweistufige Verarbeitung einführt. Anstatt Wörter der Form $a^n b^n c^n$ direkt zu parsen, kann man zunächst über eine kontextfreie Analyse sicherstellen, dass ein Wort der Form $a^n b^n c^m$ genügt, um dann in einem zweiten Schritt – typischerweise als *semantische Analyse* bezeichnet – sicherzustellen, dass $n = m$ gilt. Ähnliche mehrstufige Verfahren werden beispielsweise in Compiler angewendet, um Typprüfungen von syntaktisch korrekten Programmen durchzuführen oder sicherzustellen, dass Variablen vor ihrer Verwendung deklariert wurden. Beide Aspekte sind auf grammatikalischer Ebene nur sehr schwer zu verwirklichen.

Alternativ könnte man auch zunächst mit regulären Mitteln parsen, dass das Wort die Form $a^n b^m c^k$ besitzt.

2.14.7 *Abschlusseigenschaften kontextfreier Sprachen*

Abschließend betrachten wir die Abschlusseigenschaften kotextfreier Sprachen. Im Vergleich zu regulären Sprachen sind kfS etwas weniger gutmütig, da nicht alle Operationen kontextfreie Sprachen in kontextfreie Sprachen überführen:

Abschlusseigenschaften kontextfreier Sprachen

Die kontextfreien Sprachen sind abgeschlossen unter

- ❏ Vereinigung
- ❏ Produkt
- ❏ Stern

Sie sind *nicht* abgeschlossen unter

- ❏ Schnitt
- ❏ Komplement

Beweis: Wir betrachten die beiden kontextfreien Grammatiken (ohne Beschränkung der Allgemeinheit verwenden sie das gleiche Alphabet Σ)

$$G_1 = (V_1, \Sigma, P_1, \langle S \rangle_1) \tag{2.102}$$

$$G_2 = (V_2, \Sigma, P_2, \langle S \rangle_2). \tag{2.103}$$

Um zu zeigen, dass die Vereinigungsmenge zweier kontextfreier Sprachen wieder kontextfrei ist, konstruieren wir eine kfG G aus G_1 und G_2, für die $\mathcal{L}(G) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$ gilt. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $V_1 \cap V_2 = \varnothing$ gilt. Durch

$$G = (V_1 \cup V_2 \cup \{\langle S \rangle\}, \Sigma, P_1 \cup P_2 \cup \{\langle S \rangle \to \langle S \rangle_1 \mid \langle S \rangle_2\}, \langle S \rangle) \tag{2.104}$$

ist eine Grammatik mit den gewünschte Eigenschaften gegeben, da durch die neue Produktionsregel Wörter aus beiden Sprachen generiert werden können.

Nachdem G den Anforderungen an eine kontextfreie Grammatik genügt, ist die Vereinigung kontextfreier Sprachen offenbar kontextfrei.

Die Konstruktion einer Grammatik für die Produktsprache $\mathcal{L}(G_1)\mathcal{L}(G_2)$ (ein Wort aus Sprache 1 gefolgt von einem Wort aus Sprache 2) verläuft äquivalent; es wird lediglich $\langle S\rangle \to \langle S\rangle_1\langle S\rangle_2$ als neue Regel für das Startsymbol eingefügt.

Um den Kleene-Stern zu »implementieren«, definieren wir die neue Grammatik $G = (V, \Sigma, P, \langle S\rangle)$, die auf G_1 aufbaut:

$$G = (V \cup \langle S\rangle', \Sigma, P \cup \{\langle S\rangle' \to \epsilon, \langle S\rangle' \to \langle S\rangle, \langle S\rangle \to \langle S\rangle\langle S\rangle\} - \{\langle S\rangle \to \epsilon\}, \langle S\rangle'\}$$

Dies ermöglicht die Produktion von $\mathcal{L}(G)^*$, da beliebig lange Ketten des Startsymbols abgeleitet werden können und $\mathcal{L}(G) = \mathcal{L}(S)$ gilt. Die Regel $\langle S\rangle \to \epsilon$ wird explizit aus der Regelmenge *entfernt*, um den Abbau von $\langle S\rangle$-Ketten zu vermeiden. Weiterhin nehmen wir ohne Beschränkung der Allgemeinheit an, dass $\langle S\rangle$ nicht auf der rechten Seite einer Regel aus P_1 enthalten ist, da sich ansonsten über die neue Regel $\langle S\rangle \to \langle S\rangle\langle S\rangle$ neue Wörter in die Sprache einschleichen könnten.

Beispielsweise leitet man die ersten vier Elemente von $\mathcal{L}(G_1)^*$ wie folgt ab:

$$\begin{aligned}
&\langle S\rangle' \Rightarrow \epsilon\\
&\langle S\rangle' \Rightarrow \langle S\rangle\\
&\langle S\rangle' \Rightarrow \langle S\rangle \Rightarrow \langle S\rangle\langle S\rangle\\
&\langle S\rangle' \Rightarrow \langle S\rangle \Rightarrow \langle S\rangle\langle S\rangle \Rightarrow \langle S\rangle\langle S\rangle\langle S\rangle
\end{aligned}$$

Allgemein gilt

$$\begin{aligned}
\mathcal{L}(\langle S\rangle') &= \{\epsilon, \mathcal{L}(\langle S\rangle), \mathcal{L}(\langle S\rangle\langle S\rangle), \mathcal{L}(\langle S\rangle\langle S\rangle\langle S\rangle), \dots\}\\
&= \{\epsilon, \mathcal{L}(\langle S\rangle), \mathcal{L}(\langle S\rangle)^2, \mathcal{L}(\langle S\rangle)^3, \dots\}\\
&= \mathcal{L}(\langle S\rangle)^*.
\end{aligned}$$

Um zu zeigen, dass eine Kombinationsoperation *nicht* kontextfrei ist, reicht es, ein einziges Gegenbeispiel anzugebnen. Für die Schnitt-Operation wählen wir die Sprachen

$$L_1 = \{\mathrm{a}^n\mathrm{b}^m\mathrm{c}^m \mid n, m \in \mathbb{N}\} \tag{2.105}$$

$$L_2 = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^m \mid n, m \in \mathbb{N}\} \tag{2.106}$$

Eine kontextfreie Grammatik für L_1 ist gegeben durch

$$\begin{aligned}
&\langle S\rangle \to \langle A\rangle\langle B\rangle\\
&\langle A\rangle \to \mathtt{a} \mid \mathtt{a}\langle A\rangle\\
&\langle B\rangle \to \mathtt{bc} \mid \mathtt{b}\langle B\rangle\mathtt{c}
\end{aligned} \tag{2.107}$$

Die Sprache L_2 wird durch die kontextfreie Grammatik

$$\begin{aligned}
&\langle S\rangle \to \langle A\rangle\langle B\rangle\\
&\langle A\rangle \to \mathtt{ab} \mid \mathtt{a}\langle A\rangle\mathtt{b}\\
&\langle B\rangle \to \mathtt{c} \mid \mathtt{c}\langle B\rangle
\end{aligned} \tag{2.108}$$

beschrieben. Die Schnittmenge $L_1 \cap L_2$ ist gegeben durch

$$L_1 \cap L_2 = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \in \mathbb{N}\}. \tag{2.109}$$

Wir haben weiter oben bewiesen, dass die Sprache *nicht* kontextfrei ist. Die Schnittmenge kontextfreier Sprachen ist im Allgemeinen also nicht kontextfrei.

Eine der bereits weiter oben verwendeten de Morgan'schen Regeln besagt, dass $\overline{\bar{L}_1 \cup \bar{L}_2} = L_1 \cap L_2$ gelten muss. Angenommen, das Komplement einer kontextfreien Sprache wäre wiederum kontextfrei. Dann folgt

$$\overline{\bar{L}_1 \cup \bar{L}_2} \in \mathcal{L}_2 \Rightarrow L_1 \cap L_2 \in \mathcal{L}_2, \tag{2.110}$$

da die Vereinigung kontextfreier Sprachen kontextfrei ist. Allerdings haben wir gerade gezeigt, dass die Schnittsprache *nicht* notwendigerweise kontextfrei sein muss, was zu einem Widerspruch führt. Daher ist die Annahme falsch, dass kontextfreie Sprachen unter Komplementbildung abgeschlossen sind. □

2.15 Kellerautomaten

Nachdem die bislang betrachteten Maschinenmodelle (DEA und NEA) genau die regulären Sprachen erkennen, wir in Abschnitt 2.14.1 aber bewiesen haben, dass kontextfreie Sprachen eine echte Obermenge davon sind, benötigen wir ein erweitertes Maschinenmodell, das zu dieser Sprachklasse korrespondiert. Warum können kontextfreie Sprachen Form $L_1 \equiv \{a^n b^n\}$ oder $L_2 \equiv \{ww^R\}$ eigentlich nicht von endlichen Automaten erkannt werden? Intuitiver betrachtet fehlt der Maschine ein »Gedächtnis« für bislang gelesene Eingaben, um sich die Eigenschaften zurückliegender Strings merken zu können: Für die Sprache L_1 könnte man mit Hilfe eines Speichers beispielsweise die Anzahl der gelesenen Buchstaben »a« festhalten, um nach Lesen aller »b«s zu prüfen, ob die gleiche Anzahl davon vorhanden ist

2.15.1 Definition und Interpretation

Ein *Kellerautomat* ist ein nach vorhergehender Überlegung erweiterter DEA, dessen Definition in Abbildung 2.34 zu finden ist. Abbildung 2.35 illustriert die Definition anhand eines graphischen Beispiels, das das Zusammenspiel der Maschinenkomponenten veranschaulicht. Wie beim DEA ist ein Band mit den Zeichen der Eingabe beschriftet; in jedem Zeitschritt des Automaten wird ein Zeichen vom Band gelesen. Ein zweiter Kopf zeigt auf den Stapelspeicher; in jedem Zeitschritt wird das oberste Zeichen gelesen und entfernt. Im Gegensatz zum Bandspeicher dürfen in jedem Zeitschritt aber beliebig viele neue Zeichen auf den Kellerspeicher gelegt werden – oder auch keines.

Insbesondere die Transitionsfunktion hat sich gegenüber einem DEA verkompliziert, weshalb man sie sorgfältig betrachten muss. Man interpretiert die Spezifikation

$$\delta(q, a, A) \ni (q', B_1, \ldots, B_k) \tag{2.111}$$

so, dass sich die Maschine M zunächst im Zustand q befindet und das Zeichen »a« liest. Oberstes Zeichen im Kellerspeicher ist der Buchstabe »A«. Die Bedeutung des Übergangs ist wie folgt:

Wir verwenden üblicherweise Kleinbuchstaben für den Band- und Großbuchstaben für den Kellerinhalt, um Definitionen leichter lesbar zu machen.

❑ *M kann* in den Zustand q' übergehen.

Kellerautomat

Ein *Kellerautomat* (*Pushdown Automaton, PDA*) besteht aus

- ❏ Zuständen, in denen sich der Automat befinden kann.
- ❏ einer Menge von Zeichen, die der Automat verarbeitet.
- ❏ einer Menge von Zeichen, die im Kellerspeicher stehen können.
- ❏ einer Regel für Zustandsübergänge.
- ❏ einem Anfangszustand.
- ❏ einem anfänglichen Zeichen im Keller.

Ein *Kellerautomat* (*Pushdown Automaton, PDA*) ist gegeben durch ein 6-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \#)$:

- ❏ Q: endliche Zustandsmenge
- ❏ Σ: endliches Bandalphabet
- ❏ Γ: endliches Kelleralphabet
- ❏ $\delta : Q \times (\Sigma \cup \epsilon) \times \Gamma \to \mathcal{P}_\mathrm{f}(Q \times \Gamma^*)$, wobei $\mathcal{P}_\mathrm{f}(X)$ die Menge aller *endlichen* Teilmengen der Menge X angibt
- ❏ $q_0 \in Q$: Anfangszustand
- ❏ $\# \in \Gamma$: Ursprüngliches Kellersymbol

Abbildung 2.34: Definition von Kellerautomaten. Links informell, rechts mathematisch-formal.

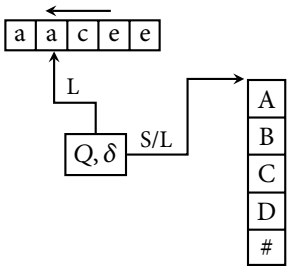


Abbildung 2.35: Illustration eines Kellerautomaten.

❏ Wenn ja, wird das Kellerzeichen A durch $B_1, \ldots, B_k$ ersetzt (B_1 steht oben).

❏ Möglich: $B_1, B_2, \ldots, B_k = A, B_2, \ldots, B_k$, d.h. *push-Operation*.

❏ Ebenfalls möglich: $k = 0$, d.h. *pop-Operation*.

Kellerautomaten sind standardmäßig als *nicht-deterministische* Maschinen definiert! Dadurch sind

❏ mehrere simultane Übergänge möglich.

❏ spontane Übergänge (mit $a = \epsilon$), die kein Zeichen vom Band lesen, aber dennoch den Stapel manipulieren, sind möglich.

Ein weiterer wichtiger Unterschied zu DEAs und NEAs ist, dass keine expliziten akzeptierenden Endzustände verwendet werden. Vielmehr wird anhand folgender Kriterien festgelegt, ob eine Eingabe akzeptiert wird oder nicht:

Akzeptanzkriterien für Kellerautomaten

❏ *Kein* akzeptierender Endzustand!

❏ Akzeptanzkriterien für Wörter $x \in \Sigma^*$:

1. Wort komplett gelesen
2. Keller leer

Ohne Beweis (der sich in der Literatur findet, aber im Wesentlichen formaler Natur ist) geben wir an, dass diese Formulierung äquivalent zur Verwendung expliziter Endzustände ist. Wir verringern durch den Verzicht darauf nicht die Mächtigkeit des Rechenmodells – allerdings kann man einige Probleme ohne explizite Endzustände mit weniger formalem Aufwand behandeln.

Um den Zustand eines Kellerautomaten während der Verarbeitung einer Zeichenkette eindeutig beschreiben zu können, bedient man sich des bereits aus Abschnitt 2.2.4 bekannten Konzepts der *Konfiguration*. Im Vergleich zum DEA muss zusätzlich der Kellerinhalt berücksichtigt werden:

Konfiguration

Konfiguration eines PDA gegeben durch Drei-Tupel (Q, Σ^*, Γ^*)

❏ $q \in Q$: Momentaner Zustand

❏ $w' \in \Sigma^*$: Noch zu lesender Anteil der Eingabe

❏ $\gamma \in \Gamma^*$: Aktueller Kellerinhalt

Die Rechnung eines Kellerautomaten kann eindeutig durch Übergänge zwischen den Konfigurationen charakterisiert werden:

Konfigurationsübergang eines Kellerautomaten

Die Relation $\vdash\colon Q \times \Sigma^* \times \Gamma^* \rightarrow Q \times \Sigma^* \times \Gamma^*$ gilt, wenn die Kellerautomaten-Konfiguration $k' \in (Q, \Sigma^*, \Gamma^*)$ aus einer Konfiguration $k \in (Q, \Sigma^*, \Gamma^*)$ durch einfache Anwendung der δ-Funktion hervorgeht. Man schreibt dann $k \vdash k'$.

Für NEAs und DEAs hat sich das Symbol $\Rightarrow$ für Konfigurationsübergänge etabliert; für Kellerautomaten kommt $\vdash$ zum Einsatz. Letztlich ist die Wahl des Symbols aber nur eine soziale Konvention. Sofern die Wahl konsistent eingesetzt wird, ist es egal, welches Symbol man verwendet.

Für einen »normalen« Übergang, bei dem ein Zeichen gelesen wird und der durch einen Übergang der Form $\delta(q, w_1, A_1) \ni (q', B_1 \dots B_k)$ gegeben ist, ist die Relation definiert durch

$$(q, w_1 w_2 \dots w_n, A_1, \dots A_m) \vdash (q', w_2 \dots w_n, B_1 \dots B_k A_2 \dots A_m). \quad (2.112)$$

Wenn ein ϵ-Übergang durch eine Regel der Form $\delta(q, \epsilon, A_1) \ni (q', B_1 \dots B_k)$ gegeben ist, bei dem also kein Zeichen gelesen wird, ist die Relation durch

$$(q, w_1 w_2 \dots w_n, A_1, \dots A_m) \vdash (q', w_1 w_2 \dots w_n, B_1 \dots B_k A_2 \dots A_m) \quad (2.113)$$

gegeben.

2.15.2 *Beispiele für Kellerautomaten*

Wir verdeutlichen die Funktionsweise eines Kellerautomaten anhand zweier Beispielsprachen:

$$L_1 \equiv \{a_1 a_2 \dots a_n \$ a_n a_{n-1} \dots a_1 \mid a_i \in \Sigma \backslash \{\$\}\} \quad (2.114)$$

$$L_2 \equiv \{a_1 a_2 \dots a_n a_n a_{n-1} \dots a_1 \mid a_i \in \Sigma\} \quad (2.115)$$

Die Transitionen für L_1, die die beschriebene Strategie umsetzen, lauten folgendermaßen:

❏ Stapelaufbau:

$$\begin{aligned} &\delta(q_0, \mathrm{a}, \#) \ni (q_0, \mathrm{A}\#) && \delta(q_0, \mathrm{a}, \mathrm{A}) \ni (q_0, \mathrm{AA}) \\ &\delta(q_0, \mathrm{a}, \mathrm{B}) \ni (q_0, \mathrm{AB}) && \delta(q_0, \mathrm{b}, \#) \ni (q_0, \mathrm{B}\#) \\ &\delta(q_0, \mathrm{b}, \mathrm{A}) \ni (q_0, \mathrm{BA}) && \delta(q_0, \mathrm{b}, \mathrm{B}) \ni (q_0, \mathrm{BB}) \end{aligned} \quad (2.116)$$

❏ Mittenübergang:

$$\begin{aligned} &\delta(q_0, \$, \#) \ni (q_1, \#) && \delta(q_0, \$, \mathrm{A}) \ni (q_1, \mathrm{A}) \\ &\delta(q_0, \$, \mathrm{B}) \ni (q_1, \mathrm{B}) \end{aligned} \quad (2.117)$$

❏ Stapelabbau:

$$\begin{aligned} &\delta(q_1, \mathrm{a}, \mathrm{A}) \ni (q_1, \epsilon) && \delta(q_1, \mathrm{b}, \mathrm{B}) \ni (q_1, \epsilon) \\ &\delta(q_1, \epsilon, \#) \ni (q_1, \epsilon) \end{aligned} \quad (2.118)$$

Die Regel $\delta(q_1, \epsilon, \#) \ni (q_1, \epsilon)$ entfernt das letzte Zeichen vom Stapel und bringt die Maschine damit in eine akzeptierende Konfiguration.

Zur Illustration des Automaten betrachten wir die Ableitungssequenz (d.h. Abfolge von Automatenkonfigurationen) für das Wort »ba$ab«:

$$(q_0, ba\$ab, \#) \vdash (q_0, a\$ab, B\#) \vdash (q_0, \$ab, AB\#)$$

$$\begin{aligned}
&\vdash (q_1, ab, AB\#) \vdash (q_1, b, B\#) \\
&\vdash (q_1, b, B\#) \vdash (q_1, \epsilon, \#) \\
&\vdash (q_1, \epsilon, \epsilon)
\end{aligned} \tag{2.119}$$

Nachdem sich der Automat nach dem letzten Schritt in einer akzeptierenden Endkonfiguration (leeres Band und leerer Kellerspeicher) befindet, wird das Wort korrekterweise akzeptiert.

Wenn keine explizite Mittenmarkierung durch das \$-Zeichen vorhanden ist, wird die Worterkennung erschwert. Mit einem nicht-deterministischem Automaten ist es dennoch möglich, die Sprache L_2 zu erkennen: Anstatt den Übergang vom Stapelauf- zum Stapelabbau an einer definierten Mittenposition zu vollziehen, werden *alle* potentiellen Mittenpositionen durchprobiert bzw. nicht-deterministisch erraten. Nicht jede Stelle im Wort kann eine Mittelposition sein; links und rechts von der Mitte muss sich der gleiche Buchstaben Übergang von Auf- nach Abbau ist also nur dann sinnvoll, wenn sich auf dem Band und im Kellerspeicher das gleiche Zeichen findet.

Anstatt der in Gleichung (2.117) definierten Mittenübergänge verwenden wir nun die Regeln

$$\begin{aligned}
&\delta(q_0, a, A) \ni (q_1, \epsilon) \\
&\delta(q_0, b, B) \ni (q_1, \epsilon) \\
&\delta(q_0, \epsilon, \#) \ni (q_1, \epsilon)
\end{aligned} \tag{2.120}$$

Dies führt zu den nicht-deterministischen Übergängen

$$\begin{aligned}
\delta(q_0, a, A) &= \{(q_0, AA), (q_1, \epsilon)\} \\
\delta(q_0, b, B) &= \{(q_0, BB), (q_1, \epsilon)\},
\end{aligned} \tag{2.121}$$

bei denen der Automat zwischen Auf- und Abbau des Stapels unterscheiden muss. Wenn das Wort symmetrisch ist, gibt es eine gültige Mittelposition, und der Automat wird im entsprechenden Berechnungszweig in eine akzeptierende Endkonfiguration übergehen.

TODO: *Beispiel Berechnungsbäume*

2.15.3 *Kellerautomaten und kontextfreie Sprachen*

Nachdem wir Kellerautomaten eingeführt haben, um die Defizite von DEAs beim Erkennen kontextfreien Sprachen zu beseitigen, ist es nicht verwunderlich, dass sie genau diese Sprachklasse erkennen. Dazu stellt sich allerdings zunächst die Frage, wie die erkannten Sprache eines PDA überhaupt charakterisiert ist. Mit Hilfe des Konfigurationsbegriffes kann man eine Definition angeben, die formal sehr ähnlich zur Definition für Grammatiken aus Abschnitt 2.5.3 ist:

Erkannte Sprache eines PDA

Sei $\overset{*}{\vdash}$ die reflexiv-transitive Hülle von $\vdash$. Die durch einen PDA M akzeptierte Sprache $\mathcal{L}(M)$ ist gegeben durch

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid (q_0, w, \#) \overset{*}{\vdash} (q, \epsilon, \epsilon) \text{ für ein } q \in Q\} \tag{2.122}$$

Die Beispiele aus Abschnitt 2.15.2 haben bereits angedeutet, dass die Wahl zwischen Determinismus und Nicht-Determinismus in der Übergangsfunktion des Kellerautomaten zu unterschiedlichen Sprachklassen führt. In der Tat ist die von auf deterministische Übergänge eingeschränkten Kellerautomaten erkannte Sprachklasse kleiner als die Sprachklasse des allgemeinen Kellerautomaten, auch wenn wir dies hier nicht explizit beweisen werden (die Details finden sich beispielsweise in den Büchern von Sipser und Hopcroft).

Man kann anhand der Übergangsfunktion entscheiden, ob ein Kellerautomat rein deterministisch arbeitet oder nicht, indem man folgende Bedingung überprüft, die zugleich als Definition des deterministischen Kellerautomaten dient:

Die von einem DPDA erkannte Sprache bezeichnet man als deterministisch kontextfrei; sie ist sehr eng verwandt mit der LR(k)–Familie, die in der Vorlesung »Compilerbau« noch erschöpfend charakterisiert werden wird. Wir merken zudem an, dass die Inklusionsbeziehung $\mathcal{L}_3 \subset \mathcal{L}_{\text{def. kF}} \subset \mathcal{L}_2$ den Zusammenhang zwischen den Sprachklassen beschreibt – deterministisch kontextfreie Sprachen sind weniger mächtig als kontextfreie, aber mächtiger als reguläre Sprachen.

Achtung: Ein DPDA benötigt im Allgemeinen einen expliziten Endzustand q_f und kann nicht die weiter oben definierten Akzeptanzkriterien verwenden, worauf wir allerdings nicht im Detail eingehen.

Deterministischer Kellerautomat (DPDA)

Ein *deterministischer Kellerautomat* ist ein Kellerautomat, dessen Transitionsfunktion *ohne* nicht-deterministische Übergänge auskommt:

$$|\delta(q, a, A)| + |\delta(q, \epsilon, A)| \leq 1 \; \forall q \in Q, a \in \Sigma, A \in \Gamma \tag{2.123}$$

Abschließend geben wir noch das zentrale Resultat an, dass ein Kellerautomat genau die kontextfreien Sprachen erkennt – dies zeigt einen weiteren tiefen Zusammenhang zwischen Maschinenmodellen und Sprachklassen:

Satz: Kellerautomaten und kfS

Eine Sprache L ist genau dann kontextfrei, wenn L von einem nichtdeterministischen Kellerautomaten erkannt wird.

Beweis:

Die Grammatik muss nicht in CNF vorliegen!

❑ Für die Vorwärtsrichtung können wir davon ausgehen, dass eine kontextfreie Sprache L mit Grammatik $G = (V, \Sigma, P, S)$ zur Verfügung steht. Wir konstruieren einen PDA $M = (\{q_0\}, \Sigma, \Gamma, q_0, \delta, \langle S \rangle)$ mit $\Gamma = V \cup \Sigma$. Wie aus der Definition ersichtlich ist, verwenden wir als initiales Kellersymbol das Startsymbol $\langle S \rangle$, *nicht* das ansonsten übliche Doppelkreuz. Dies ist ungewöhnlich, aber formal vollkommen in Ordnung.

Für jede Produktion $(\langle A \rangle \to \alpha) \in P$ definieren wir eine Übergangsfunktion

$$\delta(q_0, \epsilon, A) \ni (q_0, \alpha). \tag{2.124}$$

Für jeden Buchstaben $\sigma \in \Sigma$ definieren wir die Regel

$$\delta(q_0, \sigma, \sigma) \ni (q_0, \epsilon). \tag{2.125}$$

Aufgrund dieser Regeln führt der Automat zwei unterschiedliche Typen von Aktionen aus:

1. Wenn das oberste Kellerzeichen ein Nicht-Terminal-Symbol ist, wird eine Produktion der Grammatik angewendet (d.h. das Nicht-Terminal-Symbol auf dem Stapel durch die rechte Seite der Grammatikregel ersetzt), *ohne* ein Zeichen vom Band zu lesen.
2. Wenn das oberste Kellerzeichen ein Terminalsymbol ist, wird es entfernt, sofern das aktuelle Bandsymbol das gleiche Zeichen ist (das Bandzeichen wird implizit ebenfalls entfernt).

Trifft keine der beiden Bedingungen zu, beendet der Automat die Arbeit in einer nicht-akzeptierenden Konfiguration und lehnt das Wort ab. Wenn von einem Nicht-Terminal-Symbol mehrere Grammatikregeln ausgehen, führt die Maschine einen nicht-deterministischen Übergang durch.

Um zu zeigen, dass der konstruierte Automat M jedes Wort der Sprache, also jedes Element von $\mathcal{L}(G)$ akzeptiert und alle anderen Wörter verwirft, führen wir folgende Rechnung durch, die für alle $x \in \Sigma^*$ gilt:

$$x \in \mathcal{L}(G) \Leftrightarrow \exists \text{ Ableitung } \langle S \rangle \overset{*}{\Rightarrow} x \tag{2.126}$$

$$\Leftrightarrow \exists \text{ Sequenz } \{\zeta_i\} \text{ mit } \zeta_i \in (\Sigma \cup V)^+ :$$

$$\langle S \rangle \Rightarrow \zeta_1 \Rightarrow \zeta_2 \cdots \Rightarrow \zeta_m \Rightarrow x \tag{2.127}$$

$$\Leftrightarrow \exists \text{ Konfigurationsfolge von } M :$$

$$(q_0, x, \langle S \rangle) \vdash (q_0, \zeta_1, \ldots) \vdash \cdots \vdash (q_0, \epsilon, \epsilon) \tag{2.128}$$

$$\Leftrightarrow x \in \mathcal{L}(M) \tag{2.129}$$

Damit ist gezeigt, dass die vom Automaten M erkannte Sprache identisch zur Sprache ist, die von der Grammatik G produziert wird.

❏ Die Rückrichtung »⇐« ist formal relativ aufwendig, ohne substantielle Erkenntnisse nach sich zu ziehen. Wie überlassen sie daher der Literatur (man zeigt dort, wie sich jeder Kellerautomat in eine äquivalente kontextfreie Grammatik umwandeln lässt). □

2.15.4 *Kellerautomaten als Parser*

Aus dem Äquivalenzbeweis zwischen Kellerautomaten und kontextfreien Sprachen folgt unmittelbar eine konstruktive Vorschrift, wie aus einer Grammatik G ein Parser konstruiert werden kann, der von einem Kellerautomaten M umgesetzt wird. Um die Arbeitsweise des Automaten zu verdeutlichen, zeigt Abbildung 2.36 den vollständigen Konfigurationsbaum für das Wort »aabb« über der Grammatik (2.115), der nach beschriebener Methode konstruiert wurde.

2.16 *CYK-Algorithmus*

Die Freude über den eben vorgestellten Parser für kontextfreie Sprachen wird durch eine einschneidende Einschränkung gedämpft: Der dazu notwendige Kellerautomat ist nichtdeterministisch und daher ein unphysikalisches Berechnungsmodell. Eine Simulation auf deterministischen Maschinen ist nur mit inakzeptablem Aufwand möglich (siehe Kapitel 4), weshalb die Methode nicht praxistauglich ist. Es gibt allerdings andere Algorithmen, die das Problem deterministisch und effizient lösen. Eine interessante Methode wurde von Cocke, Younger und Kasami vorgeschlagen; die in der Literatur als *CYK-Algorithmus* bekannte Vorgehensweise werden wir in diesem Abschnitt besprechen.

Angenommen, es gibt in jedem Analyseschritt zwei mögliche Regeln, die nicht-deterministisch angewendet werden können: Dann resultieren für ein Wort mit n Buchstaben insgesamt 2^n Berechnungszweige, die simuliert werden müssten. Da Exponentialfunktionen bekanntlich sehr schnell wachsen, steigt der Aufwand für längere Wörter ins unermessliche.
TODO: Personendaten zu Cocke, Younger und Kasami.

2.16.1 *Konstruktionsprinzip*

Wir nehmen ohne Beschränkung der Allgemeinheit an, dass die untersuchte Grammatik G in Chomsky-Normalform vorliegt. Beim Ableiten von Wörtern

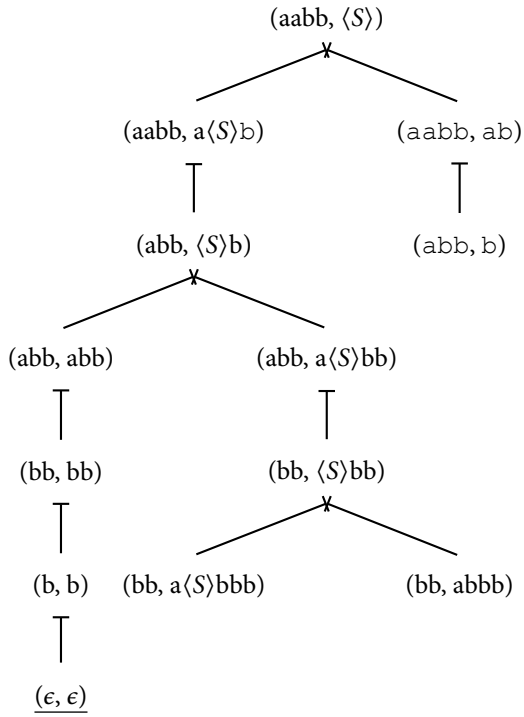


Abbildung 2.36: Ableitungsbaum für das Wort »aabb« über Grammatik 2.115. Die unterstrichene Konfiguration ist ein akzeptierender Endzustand; nicht unterstrichenen Blätter kennzeichnen nichtakzeptierende Berechnungspfade.

aus einem Nicht-Terminalsymbol können zwei Fälle auftreten:

- ❏ Ein einzelner Buchstabe ($x = a$) wird durch eine Regel der Form $\langle A \rangle \rightarrow a$ abgeleitet.
- ❏ Bei mehreren Buchstaben ($x = a_1 a_2 \ldots a_n$) ist notwendigerweise eine Regel der Form $\langle A \rangle \rightarrow \langle B \rangle \langle C \rangle$ die Grundlage. Aus $\langle B \rangle$ erwächst das Anfangsstück $a_1 a_2 \ldots a_k$ des Wortes, aus $\langle C \rangle$ das Endstück $a_{k+1} \ldots a_n$.

Abbildung 2.37 demonstriert, wie sich die beschriebene Struktur auf die Zusammensetzung eines Ableitungsbaumes auswirkt.

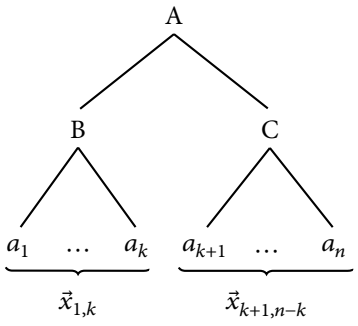


Abbildung 2.37: Illustration des CYK-Algorithmus. Der Baum repräsentiert die Ableitung des Wortes $\vec{x} = \mathsf{a}_1 \ldots \mathsf{a}_n$.

Man kann die Struktur nutzen, um das Gesamtproblem – die Ableitung eines Wortes aus dem Startsymbol der Grammatik und damit auch die Frage $x \in \mathcal{L}(G)$ – auf kleinere, einfachere Teilprobleme zu reduzieren. Um zu überprüfen, ob $x = a_1 a_2 \ldots a_n$ aus $\langle A \rangle$ abgeleitet werden kann, müssen zwei Teilprobleme gelöst werden:

❏ Gilt $a_1 a_2 \ldots a_k \in \mathcal{L}(B)$?

❏ Gilt $a_{k+1} \ldots a_n \in \mathcal{L}(C)$?

Achtung: Sowohl der konkrete Wert von k als auch die benötigte Regel, die als Ausgangsbasis verwendet wird, sind unbekannt! Eine Möglichkeit ist es, alle möglichen Varianten systematisch auszuprobieren, was allerdings ebenfalls nicht sonderlich effizient ist. Kernpunkt des CYK-Algorithmus ist es, das »Durchprobieren« durch geeignetes Speichern von Zwischenergebnissen effizienter zu machen. Nachdem wir in der folgenden Diskussion mit vielen Indizies arbeiten müssen, vereinbaren wir einige Konventionen: $x_{i,j}$ ist Teilwort von x,

In anderen Worten: Man verringert den Zeitaufwand auf Kosten eines höheren Speicherbedarfs; dies ist ein oft angewendetes Konstruktionsprinzip im Algorithmenentwurf.

❏ das an Position i (Eins-basierte Indizierung) beginnt.

❏ das Länge j besitzt, d.h. $|x_{i,j}| = j$.

Beispiele: x = abcdef $\rightarrow x_{2,2}$ = bc, $x_{1,4}$ = abcd, $x_{1,6} = x$

Der CYK-Algorithmus bedient sich einer systematischen Analyse immer längerer Teilwörter eines gegebenen Wortes, wobei die bisherigen Analyseergebnisse für kürzere Wörter in einem Zwischenspeicher verfügbar sind:

❏ Zuerst werden mögliche Ableitungen für alle Teilwörter mit Länge 1 (d.h. einzelne Buchstaben) gefunden. Dazu sucht man alle Produktionen der Form $\langle A \rangle \rightarrow \mathtt{a}$ für $\langle A \rangle \in V$ und $\mathtt{a} \in \Sigma$.

❏ Um mögliche Ableitungen für Teilwörter der Länge $j \leq n = |x|$ zu finden, stellt man fest, dass

  ❏ j zerlegbar in eine Summe zweier Zahlen $((1, j{-}1); (2, j{-}2); \ldots; (j{-}1, 1))$ ist.

  ❏ alle bisherigen Ergebnisse der Analyse für Teilwörter w' mit $w' \leq j - 1$ verwendbar sind!

❏ Ein Teilwort der Länge j kann verschiedene Startpositionen $1, 2, \ldots, n - j + 1$ innerhalb des Gesamtwortes einnehmen, die untersucht werden müssen.

Als konkretes Beispiel für die Zerlegung betrachten wir Teilwörter mit Länge $j = 4$ des Gesamtwortes w = abcdef, mit $|w| = 6$:

❏ $i = 1$: abcd ☞ (a, bcd); (ab, cd); (abc, d)

❏ $i = 2$: bcde ☞ (b, cde); (bc, de); (bcd, e)

❏ $i = 3 = 6 - 4 + 1$: cdef ☞ (c, def); (cd, ef); (cde, f)

2.16.2 Der Algorithmus

Das beschriebene Konstruktionsprinzip wird in folgendem systematischen Algorithmus umgesetzt:

Cocke-Younger-Kasami-Algorithmus

```
1:  procedure CYK(w⃗, G)
2:      n = |w⃗|
3:      for i ← 1, n do
4:          for (A → w_i) ∈ P do
5:              Erweitere T_{i,1} um den Eintrag A
6:          end for
7:      end for
8:      for j ← 2, n do                          ▷ j: Länge Teilwort
9:          for i ← 1, n − j + 1 do              ▷ i: Startposition
10:             for k ← 1, j − 1 do              ▷ k: Aufteilung
11:                 Erweitere T_{i,j} um den Eintrag A, wenn es
12:                 eine Regel (A → BC) ∈ P gibt, für die gilt:
13:                 1.) B ∈ T_{i,k}
14:                 2.) C ∈ T_{i+k,j−k}
15:             end for
16:         end for
17:     end for
18:     if S ∈ T_{1,n} then return »Akzeptiert«
19:     else return »Nicht akzeptiert«
20:     end if
21: end procedure
```

In Übung 5.33 wird die Anwendung des Algorithmus an einem praktischen Beispiel diskutiert.

2.16.3 Laufzeitkosten

Wie viele Rechenschritte braucht der CYK-Algorithmus, um zu entscheiden, ob ein Wort aus einer Grammatik erzeugt werden kann oder nicht? Insbesondere ist interessant, wie die Anzahl der Schritte mit steigender Länge der analysierten Wörter ansteigt. Durch eine elementare Analyse des obigen Algorithmus kann man eine grobe Abschätzung für die Laufzeit gewinnen.

Die erste Schleife zur Iteration über alle Einzelbuchstaben kann ignoriert werden, da ihre Laufzeit im Vergleich zu den drei folgenden verschachtelten(!) Schleifen vernachlässigbar ist. Jede dieser Schleifen durchläuft etwa n Iterationen, der Gesamtaufwand beträgt also in etwa n^3 Iterationen (in Kapitel 4 werden wir eine präzisere Notation für solche Aussagen einführen). Eine kubische Skalierung betrachtet man in der Informatik als effizient: Zwar steigt die Rechenzeit mit längeren Wörtern überproportional an, eine kubische Skalierung ist auf Computern aber ohne größere Probleme zu verkraften.

Warum weist der Algorithmus keine exponentielle Skalierung auf, wie sie bei der Simulation des nichtdeterministischen Parsers aus Abschnitt 5.33 auftritt? Der Schlüssel dazu ist das Zwischenspeichern von Analyseergebnissen kürzerer Wörter, die bei längeren Teilwörtern wiederverwendet werden.

Das Prinzip ist in der Literatur als *Dynamisches Programmieren* bekannt:

Mehr über dieses Entwurfsprinzip werden Sie in der Vorlesung Algorithmen und Datenstrukturen erfahren.

- ❏ Die Gesamtlösung wird aus Teillösungen ermittelt
- ❏ Teillösungen werden in einem Zwischenspeicher vorrätig gehalten und können bei Bedarf verwendet werden, ohne jedesmal neu berechnet werden zu müssen.

2.16.4 Entscheidbarkeit

Abschließend betrachten wir, welche Eigenschaften kontextfreier Sprachen entscheidbar sind:

- ❏ *Wortproblem*: Entscheidbar. Es reicht, den CYK-Algorithmus anzuwenden, um zu überprüfen, ob ein Wort Mitglied einer kontextfreien Sprache ist.
- ❏ *Leerheitsproblem*: Entscheidbar durch folgenden Algorithmus:
  - Zuerst wird die Grammatik in Chomsky-Normalform transformiert.
  - Im ersten Schritt werden alle Regeln markiert, die Terminale erzeugen (beispielsweise $\langle A \rangle \to \mathtt{a}$).
  - Anschließend werden alle Regeln markiert, die markierte Regel enthalten (beispielsweise $\langle B \rangle \to \langle A \rangle \langle C \rangle$). Dieser Schritt wird wiederholt, bis sich die Markierungen nicht mehr ändern.
  - Wenn $\langle S \rangle$ markiert wird, ist die Sprache nicht leer, da es mindestens ein Wort gibt, das aus dem Startsymbol abgeleitet werden kann.
- ❏ *Endlichkeitsproblem*: Entscheidbar. Nachdem explizit bekannt ist, dass die betrachtete Sprache kontextfrei ist, existiert eine kontextfreie Grammatik, mit deren Hilfe die Pumping-Konstante n berechnet werden kann. Man versucht nun, Wörter w der Länge $n \geq |w| < 2n$ zu finden (davon gibt es nur eine endliche Anzahl). Gelingt dies, kann man das gefundene Wort gemäß des Pumping-Lemma beliebig aufpumpen, die Sprache ist daher unendlich (anderenfalls ist die Sprache endlich).
- ❏ *Schnittproblem* und *Äquivalenzproblem* sind nicht entscheidbar. Die Argumentation für die nicht-Entscheidbarkeit können wir allerdings erst in Kapitel 4 liefern, da zuvor das methodische Arsenal erweitert werden muss.

3
Berechenbarkeitstheorie

Durch systematische Erweiterung des endlichen Automaten um eine zusätzliche Speichermöglichkeit (den Kellerspeicher) konnten wir von der Erkennung regulärer Sprachen zu kontextfreien Sprachen, der nächsthöheren Klasse in der Chomsky-Hierarchie, übergehen. Intuitiv betrachtet sind Kellerautomaten aber noch weit von den Möglichkeiten eines modernen Computers entfernt. Die hauptsächlichen Nachteile sind:

- ❏ Es ist kein wahlfreier Zugriff auf den Stapelspeicher möglich, man kann nur auf das oberste Element zugreifen.
- ❏ Ebenfalls gibt es keinen wahlfreien Zugriff auf das Eingabewort; man kann das Band nicht vor- und zurückspulen.
- ❏ Selbst wenn man spulen könnte, ist es nicht möglich, »Markierungen« in der Eingabe anzulegen, die bei der Berechnung helfen.

3.1 Turing-Maschinen

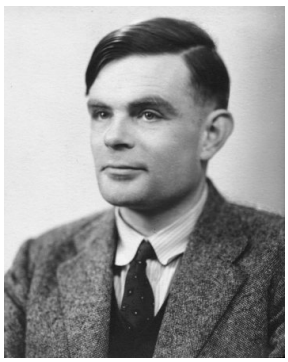
3.1.1 Definition

Um diese Nachteile zu beseitigen, konstruieren wir ein erweitertes Maschinenmodell, das die Trennung zwischen Eingabeband und Kellerspeicher aufhebt, indem Schreibzugriffe auf das Eingabeband erlaubt werden. Ebenfalls ermöglichen wir wahlfreie Zugriffe auf die Eingabe, indem beliebige Bandpositionen gelesen werden dürfen und die Leseoperation auch beliebig oft wiederholt werden darf. Diese Modifikationen führen zu einer Maschine, die in der Literatur als *Turing-Maschine* bekannt ist. Wir fassen ihre Eigenschaften in einer Definition zusammen:

(Nicht-)Deterministische Turing-Maschine

Eine *Turing-Maschine* ist gegeben durch ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$. Dabei ist

- ❏ Q eine endliche Zustandsmenge
- ❏ Σ ein endliches Eingabealphabet
- ❏ $\Gamma \supset \Sigma$ ein endliches Arbeitsalphabet
- ❏ δ die Transitionsfunktion mit Signatur



Alan Turing (1912–1954)
Seine zentrale Arbeit »On computable numbers, with an application to the Entscheidungsproblem«, in der das Konzept der Turing-Maschine eingeführt wird, hat Turing im Alter von 24 Jahren veröffentlicht. Zahlreiche weitere bahnbrechende Beiträge zur Wissenschaft, die neben biologischen Arbeiten auch entscheidende kryptanalytische Erkenntnisse im zweiten Weltkrieg enthalten, haben die prüde britische Nachkriegsjustiz nicht davon abgehalten, Turing 1952 wegen des »Verbrechens« der Homosexualität zur chemischen Kastration zu verurteilen, aufgrund derer er sich zwei Jahre später das Leben nahm. 2014 hat sich die britische Königin nach wenig mehr als 60 Jahren postum bei Turing entschuldigt.

- Deterministisch: $\delta : Q \backslash F \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, N\}$
- Nicht-Deterministisch: $\delta : Q \backslash F \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$

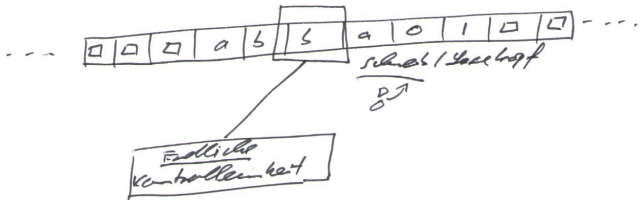
❑ $q_0 \in Q$ der Startzustand

❑ $\square \in \Gamma - \Sigma$ das »Blank«-Symbol

❑ $F \subseteq Q$ die Menge der Endzustände

Abbildung 3.1 zeigt ein mögliche physikalische Umsetzung des Konzepts.

Abbildung 3.1: Turing-Maschine.



Wie bei allen anderen Maschinenmodellen kann man die Turing-Maschine je nach Gestaltung der Übergangsfunktion in einer deterministischen und einer nicht-deterministischen Variante verwenden:

❑ Deterministisch: $\delta(q, a) = (q', b, x)$, d.h. der nächste Rechenschritt ist eindeutig festgelegt.

❑ Nicht-Deterministisch: $\delta(q, a) \ni (q', b, x)$, d.h. von einem Zustand mit gelesenem Eingabesymbol ausgehend gibt es mehrere Möglichkeiten, wie die Berechnung fortschreiten kann.

Die Transitionsfunktion induziert folgende Aktionen der Maschine:

1. M befindet sich im Zustand q und liest Zeichen a.
2. M geht in Zustand q' über, schreibt das Zeichen b *auf den Platz von* a, und führt die Kopfbewegung $x \in \{L, R, N\}$ aus – geht also genau einen Schritt nach links, genau einen Schritt nach rechts, oder verbleibt auf der aktuellen Position.

Das bereits bekannte Konfigurationskonzept, mit dessen Hilfe sich der aktuelle Zustand eines Rechenmodells in Form eines »Schnappschusses« festhalten lässt, kann auch auf Turing-Maschinen angewendet werden. Nachdem die aktuelle Position auf dem Band im Gegensatz zu einfacheren Modellen vom bisherigen Verlauf der Rechnung abhängt, müssen wir neben den üblichen Informationen wie Zustand und Bandinhalt auch berücksichtigen, an welcher Stelle sich der Schreib/Lesekopf befindet. Eine Konfiguration einer TM wird daher per Konvention durch ein Wort $k \in \Gamma^* Q \Gamma^*$ angegeben. Beispielsweise gibt $k = \alpha q \beta$ an, dass

❑ der aktuelle Zustand der TM q ist.

❑ α das Wort links des Schreib/Lese-Kopfes und

❑ $\beta = \beta_0 \beta_1 \cdots \beta_n$ das Wort rechts des S/L-Kopfes ist. Der Kopf steht dabei auf dem Feld mit der Beschriftung β_0 .

Die Startkonfiguration ist entsprechend durch $q_0\vec{w}$ gegeben:

- ❏ Der String $w \in \Sigma^*$ ist der ursprüngliche Bandinhalt; die Maschine M befindet sich im Startzustand q_0.
- ❏ Der S/L-Kopf steht auf dem erstem Buchstaben des Anfangswortes w.
- ❏ Das restliche Band ist in beide Richtungen unendlich lang mit Blanks □ befüllt, die beliebig überschrieben werden können.

Jeder Rechenschritt der TM führt zu einem Übergang zwischen zwei Konfigurationen. Die Spezifikation des Übergangs ist im Vergleich zu den bisherigen Modellen etwas erschwert, da zum einen die Kopfbewegung berücksichtigt werden muss und zum anderen der Kopf auf bislang nicht besuchte Felder treffen kann.

Konfigurationsübergänge einer Turing-Maschine werden durch eine zweistellige Relation $\vdash$ beschrieben. Der einfache (aber häufigste) Fall, in dem sich der S/L-Kopf vor und nach dem Übergang auf einer bereits beschriebenen Bandposition befindet, wird durch folgende Definition abgedeckt (der rot markierte Anteil auf der rechten Seite der Gleichung gibt an, welche Elemente der Konfiguration sich durch den Übergang geändert haben):

$$a_1 \dots a_m q b_1 \dots b_n = \begin{cases} a_1 \dots a_m \color{red}{q' c} b_2 \dots b_n & \delta(q, b_1) = (q', c, N), \\ & m \geq 0, n \geq 1 \\ a_1 \dots a_m \color{red}{c q'} b_2 \dots b_n & \delta(q, b_1) = (q', c, R), \\ & m \geq 0, n \geq 2 \\ a_1 \dots a_{m-1} \color{red}{q'} a_m c b_2 \dots b_n & \delta(q, b_1) = (q', c, L), \\ & m \geq 1, n \geq 1 \end{cases}$$

Sonderfälle, in denen sich der S/L-Kopf über den bereits beschriebenen Bandanteil hinausbewegt, werden durch folgende Definitionen erfasst:

- ❏ Sonderfall 1: $n = 1$, M läuft nach rechts:

$$a_1 \dots a_m q b_1 \vdash a_1 \dots a_m c q' \square \text{ für } \delta(q, b_1) = (q', c, R)$$

- ❏ Sonderfall 2: $m = 0$, M läuft nach links:

$$q b_1 \dots b_n \vdash q' \square c b_2 \dots b_n \text{ für } \delta(q, b_1) = (q', c, L)$$

3.1.2 *Rechnungsbeispiel*

Wir betrachten die Wirkung der Regeln anhand eines konkreten Beispiels, in dem wir eine TM zum Erkennen der Sprache $L \equiv \{0^n 1^n \mid n \in \mathbb{N}\}$ verwenden. Die Grundidee der Maschine ist es, Nullen und Einsen auf dem Band in Zweiergruppen (d.h. jeweils eine Null und eine Eins) in X respektive Y zu ändern. Wenn alle Ziffern 0 und 1 auf dem Band abgearbeitet wurden, ist sichergestellt, dass die gleiche Anzahl an Nullen und Einsen vorhanden waren, da das Umschreiben nur paarweise erfolgt ist. Dazu sind folgende Schritte notwendig:

1. Die erste 0 des Wortes wird in X verändert, indem das Bandfeld überschrieben wird.

2. Der S/L-Kopf wird nach rechts bewegt, bis er auf die erste 1 trifft und diese in Y ändert.

3. Der S/L-Kopf fährt so lange nach links, bis er das erste mit X markierte Feld auf dem Band erreicht.

4. Der S/L-Kopf bewegt sich eine Bandposition nach rechte. Befindet sich dort eine 0, wird die Rechnung in Schritt 1 neu gestartet. Wenn sich keine Null rechts des X befindet, wird überprüft, ob noch Einsen auf dem Band vorhanden sind.

   ❏ Wenn dies nicht zutrifft, kann das Wort akzeptiert werden, da aufgrund des paarweisen Abstreichens und der Abwesenheit von Einsen und Nullen auf dem Band sichergestellt ist, dass das Wort die Form 0^n1^n besitzt und daher Mitglied der Sprache ist.

   ❏ Befinden sich noch Einsen auf dem Band, wird das Wort abgelehnt.

Die Übergangsfunktion δ der Maschine

$$M = (\{q_0, q_1, q_2, q_3, q_f\}, \{0, 1\}, \{0, 1, X, Y, \square\}, \delta, q_0, \{q_f\}),$$

die den beschriebenen Algorithmus umsetzt, ist in Tabelle 3.1 definiert.

Tabelle 3.1: Übergangsfunktion einer Turing-Maschine, die die Sprache 0^n1^n durch paarweises Abstreichen von Nullen und Einsen erkennt.

	0	1	X	Y	□
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)	$(q_f, \square, R)$
q_f	∅	∅	∅	∅	∅

Betrachten wir die Wirkung der Übergangsfunktion anhand eines akzeptierenden Beispiels, in dem das Wort »0011« bearbeitet wird. Folgende Konfigurationsübergänge werden ausgeführt:

$$\begin{aligned} q_00011 &\vdash Xq_1011 \vdash X0q_111 \vdash Xq_20Y1 \vdash q_2X0Y1 \vdash Xq_00Y1 \\ &\vdash XXq_1Y1 \vdash XXYq_11 \vdash XXq_2YY \vdash Xq_2XYY \vdash XXq_0YY \\ &\vdash XXYq_3Y \vdash XXYYq_3\square \vdash XXYY\square q_f\square \end{aligned}$$

Die Maschine beendet ihre Arbeit korrekterweise im Zustand q_f, akzeptiert das Wort also. Beim zweiten Beispielwort »0010« soll dies nicht der Fall sein, da es kein Mitglied der Sprache ist:

$$\begin{aligned} q_00010 &\vdash Xq_1010 \vdash X0q_110 \vdash Xq_20Y0 \vdash q_2X0Y0 \vdash Xq_00Y0 \\ &\vdash XXq_1Y1 \vdash XXYq_10 \vdash XXY0q_1\square \end{aligned}$$

Nach dem letzten Schritt kann die Maschine nicht mehr weiterarbeiten, da $\delta(q_1, \square) = \varnothing$. Nachdem q_1 kein akzeptierender Endzustand ist, wird das Wort korrekterweise abgelehnt.

3.1.3 *Linear Beschränkte Automaten*

In Ihrer Standarddefinition enthält die Turing-Maschine zwei Komponenten, die eine Umsetzung mit realen physikalischen Mitteln erschweren bzw. aus prinzipieller Hinsicht betrachtet unmöglich machen:

Auch wenn es für die allermeisten praktischen Probleme möglich ist, ein endliches Band soweit zu verlängern, dass es für das spezifische Problem ausreicht, bleibt das prinzipielle Problem bestehen.

- ❏ Ein unendlich langes Band ist aufgrund der beschränkten Ressourcen des Universums prinzipiell nicht verfügbar. Die Simulation einer TM auf einem Computer ist aufgrund der endlichen Menge an RAM-Speicher, Festplattenspeicherplatz etc. daher nicht möglich.
- ❏ Nicht-Deterministische Übergänge können zu einer beliebig häufigen Aufspaltung des Berechnungsbaumes führen und sind daher einer Simulation mit endlichen Ressourcen ebenfalls nicht zugänglich.

In Abschnitt 3.3.4 werden wir zeigen, dass sich die Berechnungsmacht deterministischer und nicht-deterministischer Turing-Maschinen nicht voneinander unterscheidet; wir werden die Probleme des Nicht-Determinismus daher vorerst beiseite lassen und die Einschränkungen genauer untersuchen, wenn das Band einer Turing-Maschine auf eine endliche Länge beschränkt wird. Den dadurch entstehenden Maschinentypen bezeichnet man als *linear beschränkte Turing-Maschine* bzw. *linear bounded automation (LBA)*:

Achtung: Die Möglichkeit nicht-deterministischer Übergänge wird beibehalten, auch wenn ein unendlich langes Band weniger problematisch als ein nicht-deterministischer Mechanismus erscheint. Der Grund dafür ist, dass sich die entstehende Maschine als nützlich erweisen wird, wenn kontextsensitive Sprachen verarbeitet werden.

Linear beschränkte Turing-Maschine (LBA)

Eine nicht-deterministische Turing-Maschine M heißt linear beschränkt, wenn für alle Eingabestrings $a_1 a_2 \ldots a_{n-1} a_n \in \Sigma^+$ und für alle Konfigurationen $\alpha q \beta$ mit $q_0 \hat{a}_1 a_2 \ldots a_{n-1} \hat{a}_n \overset{*}{\vdash} \alpha q \beta$ gilt, dass $|\alpha\beta| = n$. Während der Laufzeit der Maschine gilt, dass

- ❏ das Band, auf dem sich die Eingabe befindet, nicht verlassen wird (konsequenterweise wird die Eingabe überschrieben, wenn Zwischenergebnisse festgehalten werden müssen).
- ❏ es Endmarkierungen auf beiden Seiten gibt, die mittels einer Erweiterung des Alphabets $\Sigma \rightarrow \Sigma' \equiv \Sigma \cup \{\hat{a} \mid a \in \Sigma\}$ realisiert werden.

Informell betrachtet bedeutet die Bedingung $|\alpha\beta| = n$ in der Definition, dass die Gesamtlänge aus ursprünglicher Eingabe und aller im Verlauf der Rechnung neu beschriebenen Bandsymbole den (festen) Grenzwert n nicht übersteigt.

3.2 *Satz von Kuroda*

3.2.1 *Maschinen für $\mathcal{L}_0$ und $\mathcal{L}_1$*

Die Vorgabe einer endlichen Bandlänge schränkt die Berechnungsmacht einer Turing-Maschine intuitiv ein. Folgender Satz präzisiert diese Einschränkung:

Satz von Kuroda

Die von linear beschränkten, nicht-deterministischen Turing-Maschinen akzeptierbaren Sprachen sind genau die kontextsensitiven Sprachen $\mathcal{L}_1$.

Die genaue Unterschied zwischen beiden Begriffen wird in Abschnitt 3.3.1 erläutert.

Es ist zu beachten, dass sich der Satz nur auf die *Akzeptanz* und nicht auf die *Entscheidbarkeit* von Sprachen bezieht! Wir beweisen an dieser Stelle lediglich die Vorwärtsrichtung; ein Beweis der formal komplizierteren Rückwärtsrichtung ist in den in Abschnitt 1.4 genannten Büchern zu finden.

Es soll untersucht werden, ob die Eingabe $\vec{w} = x_1 x_2 \cdots x_n$ mit Hilfe einer kontextsensitiven Grammatik G generiert werden kann oder nicht (dies beantwortet die Frage, ob $\vec{w} \in \mathcal{L}(G)$ gilt). Dazu wenden wir folgenden Algorithmus an, der das Wort durch schrittweise »Rückwärtsanwendung« der Produktionen von G auf das Startsymbol zurückführt, sofern dies möglich ist:

1. Nicht-deterministisch wird eine Regel $(u \to v) \in P$ gewählt.
2. Die Maschine sucht ein passendes Vorkommen der rechten Seite v auf dem Band. Wenn keines zu finden ist, endet der Berechnungszweig in einem nicht-akzeptierenden Zustand.
3. Wird ein passendes Stück Text gefunden, ersetzt die Maschine das Teilwort durch u. Wenn $|u| < |v|$ gilt, die Regel also zu einer Verlängerung des Wortes und daher in der Rückwärtsanwendung zu einer Verkürzung des Ableitungsstrings führt, wird das Restwort passend nach links verschoben.
4. Wenn nur mehr $\langle S \rangle$ auf dem Band zu finden ist, hält die Maschine an und akzeptiert das Wort; ansonsten zurück zu Schritt 1.

Um zu sehen, dass dieser Prozess für jedes Wort $x \in \mathcal{L}(G)$ mit $\mathcal{L}(G) \in \mathcal{L}_1$ zur Akzeptanz führt, betrachtet man folgende Umformung:

$$\begin{aligned} x \in \mathcal{L}(G) &\Leftrightarrow \exists \text{ Ableitungssequenz } S \Rightarrow \cdots \Rightarrow x \\ &\Leftrightarrow \exists \text{ Rechnung von } M, \text{ die Ableitung umgekehrt simuliert} \\ &\Leftrightarrow x \in \mathcal{L}(M) \end{aligned}$$

Nachdem ausschließlich Äquivalenztransformationen verwendet wurden, ist klar, dass Mitgliedschaft in der Sprache und Halten der Maschine gleichbedeutend sind – die Argumentation kann schließlich sowohl von links nach rechts als auch von rechts nach links gelesen werden. □

Der Satz von Kuroda kann leicht auf folgende Verallgemeinerung erweitert werden:

Satz: Turing-Maschinen und Phrasenstruktursprachen

Die durch allgemeine Turing-Maschinen akzeptierbaren Sprachen sind genau die Phrasenstruktursprachen $\mathcal{L}_0$.

Beweis: In der Vorwärtsrichtung reicht es, analog zum Satz von Kuroda zu argumentieren, allerdings ohne eine Beschränkung des Bandplatzes vorauszusetzen. Für die formal kompliziertere Rückwärtsrichtung verweisen wir auf die Literatur. □

3.2.2 *Zusammenfassung: Sprachen und Automaten*

Mit den Sätzen des vorhergehenden Abschnitts haben wir die Verknüpfung von Automaten und formalen Sprachen komplettiert: Jedes Maschinenmodell, das wir eingeführt haben, korrespondiert zu einer Sprachklasse der Chomsky-Hierarchie, was die engen strukturellen Zusammenhänge zwischen den auf den ersten Blick vollständig unterschiedlichen Konzepten »Sprache« und »Maschine« eindrucksvoll unterstreicht. Die Zusammenhänge sind in Tabelle 3.2 abschließend zusammengefasst.

Allerdings ist zu beachten, dass wir in Abschnitt 3.3.1 noch genauer zwischen dem Akzeptieren und dem Entscheiden von Sprachen unterscheiden werden müssen.

Sprachklasse	Maschinenmodell
Reguläre Sprachen $\mathcal{L}_3$	(Nicht)Deterministischer endlicher Automat
Kontextfreie Sprachen $\mathcal{L}_2$	Kellerautomat
Kontextsensitive Sprachen $\mathcal{L}_1$	Linear beschränkter Automat
Phrasenstruktur-Sprachen $\mathcal{L}_0$	Turing-Maschine

Tabelle 3.2: Zusammenhänge zwischen Automaten und Sprachklassen der Chomsky–Hierarchie

3.3 *Berechenbarkeit und Church-Turing-These*

Das Vertrauen der Informatik auf die Fähigkeiten von Turing-Maschinen ist enorm: Obwohl das Maschinenmodell vor mehr als einem halben Jahrhundert eingeführt wurde, konnte man kein einziges Problem entdecken, das in irgendeiner Form des automatischen Rechnens gelöst, aber nicht mit Hilfe einer Turing-Maschine umgesetzt werden kann.

Man vergleiche diese Zeitspanne mit den heutzutage üblichen halbjährlichen Innovationszyklen der Computerindustrie!

Definition: Turing-Berechenbarkeit Ein Problem, das in Form der Berechnung einer Funktion auf einer Turing-Maschine gelöst werden kann, bezeichnet man als *Turing-berechenbar*. ∎

Das Vertrauen geht sogar so weit, dass nach wissenschaftlicher Überzeugung kein Problem existiert, das nicht auf einer Turing-Maschine gelöst werden kann, sofern es überhaupt maschinell lösbar ist. Etwas systematischer formuliert man dies in einer These, die im Gegensatz zu mathematischen Sätzen zwar prinzipiell nicht bewiesen werden kann, aber dennoch praktisch axiomatischen Status in der Informatik genießt:

Unserer Erläuterungen stellen die Situation etwas vereinfacht dar; es gäbe noch zahlreiche philosophische und physikalische Details zu beachten.

Church-Turing-These

Die durch *formale Definition der Turing-Berechenbarkeit* erfasste Klasse von Funktionen stimmt genau mit der Klasse der im *intuitiven Sinne berechenbaren* Funktionen überein.

Alonzo Church (1903–1995)
TODO: Biographie von Church

Allerdings sind nicht nur Turing-Maschinen als Modell eines Universalrechners geeignet; unter anderem kann man zeigen, dass folgende Objekte genauso berechnungsmächtig wie eine Turing-Maschine sind:

- ❏ WHILE- und GOTO-Programme (einfache Programmiersprachen siehe Abschnitt 3.4).
- ❏ μ-rekursive Funktionen (nicht im Rahmen der Vorlesung behandelt).
- ❏ Registermaschinen (moderner »Computer«; im Rahmen der Vorlesung nicht behandelt).

Neben zahlreichen philosophischen Auswirkungen, auf nicht weiter eingegangen werden soll, haben die Äquivalenzen auch große praktische Bedeutung: Die einfachen Sprachen *While* und *Goto* können von jeder vernünftigen realen Programmiersprache simuliert werden. Damit werden *alle* Aussagen, die man über die Möglichkeiten und Grenzen von Turing-Maschinen beweist, auch automatisch von modernen Rechnern und Programmiersprachen »geerbt«. Oder, anders ausgedrückt: Wenn ein Problem auf einer Turing-Maschine zu lösen ist, kann man es auch mit einer »normalen« Programmiersprache auf einem Standardrechner abarbeiten. Kann eine Turing-Maschine ein Problem nicht lösen, ist es beweisbar sinnlos, dies auf einem normalen Computer zu versuchen.

Eine Programmiersprache, die eine Turing-Maschine simulieren kann (was über eine Simulation der `Goto`- oder `While`-Sprache nachgewiesen werden kann), bezeichnet man als Turing-vollständig. In den Kanon der Turing-vollständigen Sprachen reihen sich auch manche überraschenden Kandidaten wie TEX oder C++-Templates ein. Selbst die `mov`-Assembler-Anweisung des x86-Befehlssatzes ist bereits Turing-vollständig, und es wäre prinzipiell ausreichend, eine CPU mit nur diesem Befehl herzustellen! Das Konzept zeigt allerdings auch, warum es beispielsweise nicht sinnvoll ist, nicht Turing-vollständige Mechanismen wie HTML als Programmiersprache zu bezeichnen.

3.3.1 *Akzeptanz und Entscheidbarkeit*

DEAs besitzen ein sehr einfaches Verhalten bezüglich der Laufzeit, die zur Analyse eines Eingabewortes mit einer bestimmten Länge notwendig ist: Mit jedem Zeitschritt wird ein Buchstabe abgearbeitet, womit die Analyse eines Wortes $\vec{w}$ nach genau $|\vec{w}|$ Zeitschritten abgeschlossen ist. Turing-Maschinen teilen diese Eigenschaft nicht, da sich der Schreib/Lese-Kopf beliebig über das Band bewegen kann. Insbesondere ist es möglich, dass Turing-Maschinen in eine Endlosschleife geraten. Dies hat offenbar Implikationen auf die Frage, wann eine Rechnung erfolgreich beziehungsweise erfolglos beendet wurde. Grundlage der Überlegungen sind die beiden Konzepte »Akzeptanz« und »Entscheidung«:

Eine einfache Endlosschleife ist durch das Regelpaar $\delta(q_1,\sigma) \rightarrow (q_2,\sigma,\mathsf{L})$ und $\delta(q_2,\sigma) \rightarrow (q_1,\sigma,\mathsf{R})$ gegeben, das für alle $\sigma \in \Gamma$ gilt. Noch kürzer schreibt man $\delta(q,\sigma) = (q,\sigma,a)\forall\sigma$.

Definition: Akzeptanz

Eine Turing-Maschine M *akzeptiert* eine Sprache L, wenn M alle $x \in L$ akzeptiert, d.h. in einem Zustand $q \in F$ hält.

Achtung: Der Fall $x \notin L$ mit M hält *nicht* für x ist durchaus möglich! Eine Turing-Maschine, die eine Sprache nur akzeptiert, muss also nicht notwendigerweise anhalten, wenn ein Wort analysiert wird, das *kein* Element der untersuchten Sprache ist.

Definition: Entscheidbarkeit

Eine Turing-Maschine M *entscheidet* eine Sprache L, wenn M die Sprache L akzeptiert und für alle $x \notin L$ nach endlich vielen Schritten in einem Zustand $q \notin F$ hält.

Es gibt in der Literatur verschiedene Notationskonventionen, die Entscheidbarkeit und Akzeptanz von Sprachen betreffen und die sich vor allem aus historischen Gründen etabliert haben:

»Rekursiv« in den Definitionen hat nichts mit rekursiven Funktionen zu tun, die sich selbst aufrufen, sondern ist eine ältere mathematische Bezeichnung, die aus der Vor-Computer-Ära stammt.

- ❏ L ist *semi-entscheidbar* (*recognisable*, »rekursiv aufzählbar«, »recursively enumerable«) $\Leftrightarrow \exists M$, die L akzeptiert.
- ❏ L ist *entscheidbar* (*decidable*, »rekursiv«), $\Leftrightarrow \exists M$, die L entscheidet.

Viele praktische Aufgaben der Informatik beschäftigen sich mit Sprachverarbeitung. Ein genauso wichtiges Feld ist aber die Berechnung von Funktionen, angefangen bei einfachen Winkelfunktionen, die in Steuerungen

benötigt werden, bis hin zu komplexen numerischen Simulationen physikalischer Prozesse. Viele reale Probleme können in Form der Berechnung von Funktionen beschrieben werden – aber nicht alle Funktionen, die man berechnen möchte, kann man auch berechnen, sei es aus praktischen, sei es aus prinzipiellen Gründen. Die Menge von Funktionen, die auf einer Turing-Maschine berechnet werden können, ist durch folgende Definition festgelegt:

Definition: Turing-Berechenbarkeit Eine partielle Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist *Turing-berechenbar*, wenn es eine Turing-Maschine M gibt, die bei Eingabe von $x_1, x_2, \ldots, x_k$ die Ausgabe $y = f(x_1, x_2, \ldots, x_k)$ liefert und hält:

$$q_0 x_1 x_2 \cdots x_k \overset{*}{\vdash} qy \text{ mit } q \in F \wedge y \in \Gamma^* \wedge y = f(x_1, \ldots, x_k)$$

Falls M nicht hält, gilt $f(x_1, \ldots, x_k) = \perp$. ∎

Eine Funktion bezeichnet man als *partiell*, wenn nicht für jede Kombination aus Eingabewerten ein gültiger Ausgabewert berechnet wird. Diese Fälle werden über das Resultat $\perp$ beschrieben. Ein Beispiel für eine Funktion, die nicht auf allen Argumenten definiert ist, ist $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ mit $f(x, y) = x/y$. Die Rückgabewerte sind für $y = 0 \in \mathbb{R}$ undefiniert, der Funktionswert ist in diesem Fall $\perp$.

Man bezeichnet das Symbol »$\perp$« als Bottom-Symbol.

Das Resultat einer Division durch 0 existiert nur als Grenzwert, und ist selbst in dieser Sichtweise keine Zahl, sondern $\pm\infty$.

Zwei wichtige praktische Fragen (die allerdings keinen Einfluss auf die prinzipielle Berechenbarkeit von Funktionen haben) werden bei obiger allgemeiner Definition außen vorgelassen und müssen noch beantwortet werden:

- ❏ Wie werden die Ein- und Ausgabewerte genau auf dem Band kodiert?
- ❏ Wie viel Ressourcen (in Form von Rechenzeit und Bandplatz) werden verbraucht, um eine gegebene Funktion zu berechnen?

Betrachten wir zunächst die Frage der Informationskodierung: Das Standardalphabet der Informatik ist das binäre Alphabet $\Sigma = \{0, 1\}$, und entsprechend kann man sich – zusammen mit dem Arbeitsalphabet $\Gamma = \{0, 1, \square\}$, das zusätzlich das Blank-Symbol enthält – auf diese Kodierung als Standard für Turing-Maschinen einigen. Andere endliche Alphabete (lateinische Zeichen, griechische Buchstaben, Katakana, Smileys, …) können durch folgendes Abzählverfahren (jeder Buchstabe erhält einen eigenen Binärstring) durch ein binäres Alphabet dargestellt werden:

- ❏ Der i-te Buchstabe wird durch die Binärzahl bin(i) dargestellt.
- ❏ Da die maximale Anzahl an Binärziffern bekannt ist, können alle Darstellungen von links mit Nullen aufgefüllt werden, um die gleiche Länge zu erreichen, da die führenden Nullen nichts an numerischen Wert ändern.
- ❏ Alternativ (bei einer Kodierung mit variabler Länge) kann ein explizites Trennzeichen zwischen den Symbolkodierungen verwendet werden, beispielsweise das Blank-Symbol. Die Kodierung eines Strings verkürzt sich dadurch typischerweise, verkompliziert aber die Verarbeitung.

Die unäre Kodierung, bei der analog zu Strichlisten auf Bierdeckeln nur ein einziges Symbol verwendet wird, ist zur Kodierung ungeeignet, da exponentiell lange Zahlen im Vergleich zu Binärkodierung entstehen: Nachdem mit n Bits 2^n Zeichen dargestellt werden können, benötigt eine Zahl $N \in \mathbb{N}$ ungefähr $\log_2(N)$ binäre Ziffern, wogegen in unärer Darstellung N Ziffern notwendig sind – als exponentiell mehr, was die Kodierung sehr ineffizient macht.

Der ASCII-Zeichensatz basiert auf einer Kodierung fester Länge; die UTF-8-Kodierung von Unicode nutzt variable Längen.

Durch die Verwendung eines Standardalphabets kann die Definition der Turing-Berechenbarkeit weiter konkretisiert werden:

Turing-Berechenbarkeit

❏ Eine partielle Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist *Turing-berechenbar*, wenn es eine Turing-Maschine M gibt, die bei Eingabe von $x_1, x_2, \ldots, x_k$ die Ausgabe $y = f(x_1, x_2, \ldots, x_k)$ liefert und hält:

$$q_0 \,\mathrm{bin}(x_1)\square \,\mathrm{bin}(x_2)\square \cdots \square \,\mathrm{bin}(x_k) \overset{*}{\vdash} \square \cdots \square q \,\mathrm{bin}(y)\square \cdots \square$$

mit $q \in F \wedge y \in \Gamma^* \wedge y = f(x_1, \ldots, x_k)$.

❏ Falls M nicht hält, gilt $f(x_1, \ldots, x_k) = \perp$.

3.3.2 *Varianten von Turing-Maschinen*

Turing-Maschinen sind in der Standardform mit nur einem Band vergleichsweise schwierig zu programmieren, da häufige »Fahroperationen« notwendig sind, um auf Informationen zuzugreifen, die sich an unterschiedlichen Bandpositionen befinden. Was sind die Konsequenzen, wenn einfacher zu programmierende Varianten entwickelt werden? Nachdem Turing-Maschinen per Definitionem das mächtigste Berechnungsmodell sind, ist es nicht möglich, die Berechnungsmacht der Maschine weiter zu erhöhen. Allerdings stellt sich die Frage, ob erweiterte Maschinen zu fundamental relevanten Beschleunigungen (oder Verlangsamungen) führen.

Wie sich herausstellen wird, sind Turing-Maschinen sehr robust gegenüber (sinnvollen) Modifikationen der genauen Arbeitsdefinition, weshalb sie sich sehr gut als Ausgangspunkt für die Messung der Komplexität von Algorithmen – sprich: der notwendigen Laufzeit und dem verbrauchten Speicherplatz – eignen. Wir werden dies anhand von zwei Varianten demonstrieren:

Andere Größen wie beispielsweise die Anzahl der notwendigen elektronischen Maschinenkomponenten oder die während einer Rechnung verbrauchte Energie sind ebenfalls praxisrelevante Komplexitätsgrößen, die wir allerdings nicht weiter betrachten werden.

❏ Die Turing-Maschine wird mit k Spuren auf einem Band (und daher nur *einem* Schreib/Lesekopf) ausgestattet, wie Abbildung 3.2 darstellt.

In jedem Schritt wird entsprechend nicht mehr ein einziges Zeichen gelesen und geschrieben, sondern mehrere Zeichen auf einmal. Die Signatur der Transitionsfunktion ist entsprechend

$$\delta : Q \backslash F \times \Gamma^k \to Q \times \Gamma^k \times \{\mathrm{R}, \mathrm{L}, \mathrm{N}\}. \tag{3.1}$$

Die Äquivalenz zu einer Einspur-Maschine wird hergestellt, indem man ein neues Alphabet $\Gamma' = \Gamma^k$ definiert: Jedes Tupel aus k Buchstaben entspricht *einen* einzigen Buchstaben aus Γ'. Nachdem Γ endlich ist, ist $\Gamma' = \Gamma^k$ ebenso endlich, weshalb keine Probleme mit den Anforderungen an eine Turing-Maschine auftreten. Eine Maschine mit mehreren Spuren kann daher ohne Laufzeitnachteil durch eine Maschine mit einer Spur ersetzt werden.

TODO: Abbildung Mehrspurmaschine
Abbildung 3.2: Mehrspurige Einband-Turing-Maschine.

❏ Anstatt eines Bandes mit mehreren Spuren können auch k Bänder verwendet werden, die mit k *unabhängigen* Schreib/Leseköpfen ausgestattet sind (je einer pro Band); Abbildung 3.3 veranschaulicht das Szenario.

Die Transitionsfunktion einer Mehrband-Maschine ist gegeben durch

$$\delta : Q \backslash F \times \Gamma^k \to Q \times \Gamma^k \times \{L, N, R\}^k. \tag{3.2}$$

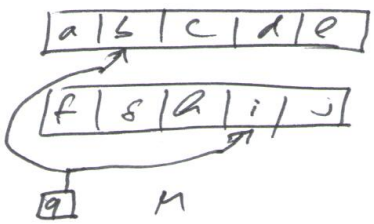


Abbildung 3.3: Mehrband-Turing-Maschine.

Mit welchem Overhead kann eine Mehrband-Maschine auf einer Einband-Maschine simuliert werden? Nachdem Mehrspur- und Einspur-Maschinen in Bezug auf die Laufzeit identisch sind, reicht es, eine Simulation auf einer Mehrspur-Maschine mit $2k + 1$ Bändern anzugeben – Abbildung 3.4 zeigt anhand eines Beispiels für zwei Bänder, wie der konkrete Fall $k = 2$ abgebildet wird.

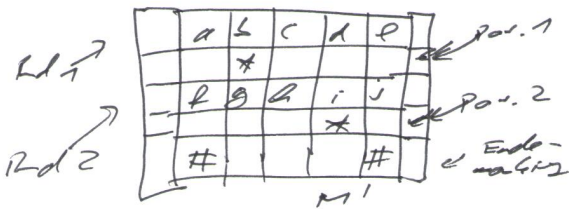


Abbildung 3.4: Simulation einer Mehrband–Turing–Maschine auf einer Mehrspur–TM.

Jedes Band erhält eine eigene Spur; die Position des jeweiligen Schreib/-Lesekopfes wird durch ein Sternchen auf der darunter liegenden Spur markiert. Die unterste Spur kennzeichnet die linke und rechte Grenze des Bandes, also die erste und letzte beschriebene Bandposition.

Jeder Zeitschritt (Transition) der Mehrband-Maschine M' wird durch folgende Sequenz an Übergängen auf der Mehrspur-Maschine M nachvollzogen:

- M läuft vom linken zum rechten Ende des Bandes, und »merkt« sich die über den Sternchen befindlichen Zeichen. Nachdem sowohl das Arbeitsalphabet wie auch die Anzahl der Bänder endlich sind, ist eine entsprechende Kodierung in den Maschinenzustand möglich. Damit erfährt M die Ergebnisse der Leseoperationen von M', die die simulierte Maschine in einem Zeitschritt erhält.
- Anhand der Übergangsfunktion von M' kann M die Änderungen nachvollziehen, die am Band von M' vorgenommen werden sollen, und wie sich die einzelnen Köpfe bewegen. M fährt nun von links nach rechts, ändert die Beschriftungen der einzelnen Spuren, und verschiebt die Sternchen je nach Bewegung der Köpfe von M'. Nach Bedarf werden auch die Bandende-Markierungen auf der letzten Spur angepasst.

Angenommen, die Mehrband-Maschine bearbeitet ein Wort mit n Zeichen und braucht dafür $t(n)$ Zeitschritte und $s(n)$ Bandelemente (die genaue Form der Funktionen $t(n)$ und $s(n)$ hängt natürlich vom bearbeiteten Problem und dem dafür eingesetzten Algorithmus ab, ist aber vorerst nicht relevant). Die simulierende Maschine M muss für jeden Zeitschritt der simulierten Maschine M' das Band zweimal abfahren (von links nach rechts und von rechts nach links); da M' maximal $s(n)$ Bandpositionen verbraucht, sind entsprechend maximal $2s(n)$ Zeitschritte notwendig. Zusätzlich kommen kurze »Zitterbewegungen« für das potentielle Verschieben der Sternchen mit hinzu, die wir über eine unbekannte, aber endliche Konstante c abschätzen. Nachdem M' maximal $t(n)$ Zeitschritte benötigt, kommt M entsprechend mit

$$t(n) = t(n)(2s(n) + c) \tag{3.3}$$

$$\leq c_1 t(n)s(n) \tag{3.4}$$
$$\leq c_1 t^2(n) \tag{3.5}$$

Zeitschritten aus. In Zeile (3.4) haben wir die additive Konstante durch einen multiplikativen Vorfaktor ersetzt, um den Ausdruck zu vereinfachen; Zeile (3.5) nutzt aus, dass die Anzahl der verwendeten Bandpositionen durch die Anzahl an Zeitschritten begrenzt ist, da eine Maschine nicht mehr Bandpositionen verbrauchen kann, als ihr Zeitschritte zur Verfügung stehen, weshalb die Schranke $s(n) \leq t(n)$ gilt.

3.3.3 Berechnungskomplexität

Nachdem wir den Aufwand für die Simulation einer Mehrband- und einer Einband-Turing-Maschine abgeschätzt haben, soll die Bestimmung der Komplexität von Algorithmen weiter systematisiert werden. Dabei geht es wie bereits angesprochen um die beiden Fragen

- ❏ Wie viel Bandplatz verbraucht eine Rechnung?
- ❏ Wie viele Rechenschritte sind notwendig?

Von besonderem Interesse ist der Aufwand im schlechtesten Fall, da dieser die obere Aufwandsgrenze für ein Problem absteckt. Zunächst ist es aber nur möglich, den Aufwand für *eine* konkrete Eingabe zu ermitteln, die einer Turing-Maschine M zur Abarbeitung übergeben wird:

Die Bandkomplexität bezeichnet man auch als space complexity, daher die Bezeichnung $s(\vec{x})$.

Definition: Zeitkomplexität $T_M(\vec{x})$ Die Funktion gibt die Anzahl der Zeitschritte einer Turing-Maschine wieder, es gilt $T_M(\vec{x})$ ≡ Anzahl der Schritte von M mit Eingabe $\vec{x} \in \Sigma^*$. ∎

Definition: Platzkomplexität $S_M(\vec{x})$ Es gilt $S_M(\vec{x})$ ≡ Anzahl *verschiedener* Zellen, die M bei Eingabe $\vec{x} \in \Sigma^*$ besucht . ∎

In der Praxis ist nicht das Verhalten für eine einzige Eingabe interessant, sondern das Wachstumsverhalten von Laufzeit (und Bandplatz) mit steigender Problemgröße. Bei der Verarbeitung von Strings gibt die Länge $n = |\vec{x}|$ der betrachteten Zeichenkette $\vec{x}$ die Größe des Problems an. Entsprechend macht man folgende Definitionen:

- ❏ Worst-Case-Laufzeit (*time complexity*):

$$T_M(n) \equiv \max_{x \in \Sigma^*, |\vec{x}| \leq n} T_M(\vec{x})$$

- ❏ Worst-Case-Platzverbrauch (*space complexity*):

$$S_M(n) \equiv \max_{x \in \Sigma^*, |\vec{x}| \leq n} S_M(\vec{x})$$

Bei der Betrachtung der Simulation von Mehrband- auf Einband-Maschinen haben wir vorausgesetzt, dass obere Grenzen für die verwendeten Ressourcen Zeit und Bandplatz bekannt sind. Mit Hilfe der vorhergehenden Definitionen können wir diese Beschränkungen nun formal sauber angeben:

Definition: Platz- und Zeitbeschränkung Gegeben $t, s : \mathbb{N} \times \mathbb{N}$. Eine Turing-Maschine M heißt $t(n)$-zeitbeschränkt und $s(n)$-platzbeschränkt, wenn für alle $n \in \mathbb{N}$ gilt:

$$T_M(n) \leq t(n) \wedge S_M(n) \leq s(n) \tag{3.6}$$

■

Ebenfalls können wir die Ergebnisse zum Simulations-Overhead formal sauberer angeben:

Satz: Einband- und Mehrband"

Seien $c_1, c_2 \in \mathbb{N}$. Jede $t(n)$zeit- und $s(n)$-platzbeschränkte k-Band-Turing-Maschine M_k kann durch eine $c_1 \cdot t(n) \cdot s(n)$-zeitbeschränkte und $c_2 \cdot s(n)$-platzbeschränkte Turing-Maschine M simuliert werden.

3.3.4 *Nicht-Determinismus bei Turing-Maschinen*

Die auf Seite 93 besprochene Church-Turing-These spezifiziert nicht explizit, ob sie sich auf deterministische oder nicht-deterministische Maschinen bezieht. Dies ist gleichgültig, sofern beide Maschinenmodelle gleich berechnungsmächtig sind, was allerdings noch zu zeigen ist. Es gilt dazu folgender Satz:

Die These beruft sich nicht auf Effizienz, sondern prinzipielle Berechenbarkeit.

Satz: Äquivalenz von DTM und NTM

Zu jeder nicht-deterministischen Turing-Maschine M gibt es eine äquivalente deterministische Turing-Maschine M'.

TODO: *Illustration: Siehe Tafel*

Zusammen mit den anderen Betrachtungen dieses Kapitels ergeben sich einige Konsequenzen bezüglich der verschiedenen Varianten von Turing-Maschinen:

- ❏ NTM und DTM besitzen die gleiche Berechnungsmacht, zeigen aber drastisch unterschiedliche Ressourcenaufwände (vermutlich exponentielle Verlangsamung im deterministischen Fall).
- ❏ DTM und k-Band-DTM besitzen die gleiche Berechnungsmacht; die Laufzeitunterschiede zwischen den Typen sind nicht substantiell.
- ❏ Deterministische und nicht-deterministische Maschinen dienen unterschiedlichen Einsatzzwecken:
  - Die *praktische Effizienz* eines Algorithmus wird an seiner *Laufzeit* auf einer DTM gemessen.
  - Die *prinzipielle Lösbarkeit* eines Problems ist über die *Entscheidbarkeit* auf einer NTM definiert, unabhängig von der dazu notwendigen Rechenzeit.

Natürlich muss die Rechenzeit echt weniger als unendlich viele Zeitschritte für eine endlichen Eingabe betragen, damit ein Problem prinzipiell lösbar ist.

3.4 *Alternative Berechnungsmodelle*

Wenn Turing-Maschinen das leistungsfähigste Berechnungsmodell sind, warum werden Sie dann nicht in praktischen Anwendungen benutzt? Die

komplizierte Programmierung ist der Stolperstein: TMs sind gut geeignet, um über Eigenschaften des Rechnens nachzudenken, strukturelle Aussagen zu beweisen und Grenzen zu ziehen. Sie sind aber denkbar ungeeignet, um große, les- und wartbare Programme zu erstellen. In der täglichen Arbeit werden in der Informatik daher Programmiersprachen wie C, Java, C# oder Python bevorzugt.

In diesem Abschnitt werden wir zeigen, dass Programmiersprachen einer Turing-Maschine in nichts nachstehen, dass sie also die gleiche Berechnungsmacht besitzen. Da realistische Sprachen sehr komplex und vielschichtig sind, werden wir den Äquivalenznachweis mit Hilfe zweier stark vereinfachter Sprachen – WHILE und GOTO – führen, die im Kern jeder modernen Sprache enthalten sind.

3.4.1 Die Sprache WHILE

Die Syntax der WHILE-Sprache ist durch folgende kontextfreie Grammatik definiert:

$$
\begin{aligned}
\langle const\rangle &\to 0 \mid 1 \mid 2 \mid \ldots \\
\langle var\rangle &\to x_0 \mid x_1 \mid x_2 \mid \ldots \\
\langle stmts\rangle &\to \langle stmt\rangle \mid \langle stmt\rangle ; \langle stmts\rangle \\
\langle stmt\rangle &\to \langle assignment\rangle \mid \langle loop\rangle \mid \langle while\rangle \\
\langle assignment\rangle &\to \langle var\rangle := \langle expr\rangle \\
\langle loop\rangle &\to \texttt{loop}\ \langle var\rangle\ \texttt{do}\ \langle stmts\rangle\ \texttt{end} \\
\langle while\rangle &\to \texttt{while}\ x_i \neq 0\ \texttt{do}\ \langle stmts\rangle\ \texttt{end} \\
\langle expr\rangle &\to \langle var\rangle \mid \langle const\rangle \mid \langle expr\rangle + \langle expr\rangle \mid \langle expr\rangle - \langle expr\rangle
\end{aligned}
\tag{3.7}
$$

Die Sprachelemente sind beinahe selbsterklärend; lediglich folgende Punkte bedürfen einer genaueren Erläuterung:

- Bei `loop`-Schleifen ist die Anzahl der Schleifendurchläufe *fest* bestimmt und durch den Wert der Schleifenvariablen festgelegt, den diese beim ersten Eintritt in die Schleife hatte.
- Die Durchlaufzahl von `while`-Schleifen ist dynamisch geregelt: In jeder Ausführung des Schleifenkörpers kann die Kontrollvariable geändert werden. Nach jedem Durchlauf wird geprüft, ob die Abbruchbedingung erfüllt ist; ist dies nicht der Fall, wird der Schleifenkörper ein weiteres Mal durchlaufen.

Die Sprache ist so entworfen, dass sie mit der kleinstmöglichen Anzahl von Anweisung auskommt. Dies ist vorteilhaft, wenn Aussagen über die Sprache gemacht werden sollen, da man nur wenige Sprachelemente berücksichtigen muss. Für die praktische Anwendung ist dies natürlich kontraproduktiv. Beispielsweise fehlt der While-Sprache eine konditionale Abfrage (`if`), die in ansonsten jeder Programmiersprache enthalten ist! Allerdings kann man `if`-Abfragen durch `loop`-Schleifen simulieren: Beispielsweise ist die Anweisung `if` x_0`=0 then P end` äquivalent zur Sequenz

x_1 `:= 1;`

```
loop x0 do x1 := 0 end;
loop x1 do P end;
```

`if`-Schleifen sind daher *syntaktischer Zucker*, der hilfreich für den Programmierer, aber nicht unbedingt notwendig für das Funktionieren der Sprache ist. Auch die Einschränkung der logischen Abfragen bei `while`-Schleifen auf den Vergleich $x_i \neq 0$ ist keine wirkliche Beschränkung; komplexere Abfragen können genauso durch kleine Teilprogramme ersetzt werden und brauchen nicht in die Kernsyntax der Sprache aufgenommen werden. Auch die Multiplikation ist syntaktischer Zucker, da sie durch in einer Schleife wiederholte Additionen ersetzt werden kann; ebenso können komplexere mathematische Operationen wie Modulo-Rechnungen mit kleinen Teilprogrammen erledigt werden und müssen nicht in der Kernsprache vorhanden sein.

Ein Beispiel für eine reale Programmiersprache, die aus einem extrem kleinen Sprachkern besteht, der durch viele Bibliotheksfunktionen und syntaktische Zusätze erweitert wird, ist Mathematica.

Die Grammatik (3.7) stellt die *syntaktischen* Regeln der WHILE-Sprache auf. Ein Programm, das als syntaktisch korrekt akzeptiert wird – beispielsweise von einem CYK-Parser – ist allerdings nichts weiter als eine Folge von Zeichenketten. Die *Semantik* des Programms, also die durch den Quelltext repräsentierte Rechnung, muss noch definiert werden. Wir benötigen eine Möglichkeit, um eine Zeichenkette, die ein WHILE-Programm repräsentiert, auf eine mathematische Funktion abzubilden.

Prinzipiell gesehen nimmt ein Programm eine bestimmte Anzahl von Eingaben entgegen und berechnet daraus eine bestimmte Zahl von Ausgabewerten. Die Semantik eines WHILE-Programms ist daher durch eine Funktion mit Signatur

$$\delta : \mathbb{N}^{n+1} \times x \to \mathbb{N}^{m-n} \text{ mit } x \in L_{\text{while}} \tag{3.8}$$

gegeben. Die Funktion kann systematisch durch eine *induktive Definition* über die Programmstruktur angegeben werden.

Die Verwendung natürlicher Zahlen für die Ein- und Ausgaben ist keine Beschränkung der Allgemeinheit, da die ganzen und rationalen Zahlen bijektiv auf die natürlichen Zahlen abgebildet werden können. Da es reelle Zahlen gibt, die unendlich viele Ziffern zu ihrer korrekten Darstellung brauchen, können Computer prinzipiell nicht mit reellen Zahlen rechnen. Selbst bei ganzen Zahlen rechnen Computer nicht über $\mathbb{N}$, sondern verwenden effektiv eine Teilmenge, die im endlichen Speicher repräsentiert werden kann. Diese Teilmenge ist aber bei Verwendung passender Datentypen so groß, dass das Problem in der Praxis häufig ignoriert werden kann.

Grundlage der Semantik ist ein *Speichervektor*

$$\vec{v} = (x_0, x_1, \ldots, x_n, x_{n+1}, \ldots, x_m), \tag{3.9}$$

der die Eingaben in $x_0, \ldots, x_n$ und die Ausgaben in $x_{n+1}, \ldots, x_m$ bereithält. Wir müssen für jedes Sprachelement von WHILE angeben, wie sich der Inhalt des Speichervektors verändert. Für die meisten Elemente ist dies mit wenig Aufwand möglich:

- Wertzuweisungen ersetzen den bisherigen Wert der Variablen im Speichervektor durch den neuen Wert, was man folgendermaßen ausdrückt:

$$\delta(\vec{v}, x_i := \langle expr \rangle) \equiv \vec{v}[x_i \leftarrow \langle expr \rangle] \tag{3.10}$$

- Bei der Konkatenation mehrerer Ausdrücke muss sichergestellt werden, dass alle Änderungen am Speichervektor, die die erste Anweisung hinterlassen hat, bei der Ausführung der zweiten Anweisung berücksichtigt werden:

$$\delta(\vec{v}, P_1; P_2) \equiv \delta(\delta(\vec{v}, P_1), P_2) \tag{3.11}$$

- `loop`-Schleifen sind nichts anderes als eine Anweisung, die mehrfach hintereinander ausgeführt wird:

$$\delta(\vec{v}, \texttt{loop } x_i \texttt{ do } P \texttt{ end}) \equiv \delta(\vec{v}, \underbrace{P; P; P; \ldots; P}_{x_i\text{-fach}}) \equiv \delta(\vec{v}, P^{x_i}) \tag{3.12}$$

Die `while`-Schleife ist schwieriger zu behandeln, da nicht a priori klar ist, wie viele Wiederholungen des Schleifenkörpers ablaufen müssen, bevor die Abbruchbedingung erreicht ist. Um die passende Anzahl herauszufinden, definiert man die *Terminierungsmenge* T_i durch

$$T_i \equiv \{k \in \mathbb{N}_0 \mid \delta(\vec{v}, P^k) = (x_0, \ldots, x_{i-1}, 0, x_{i+1}, \ldots) \text{ mit } x_l \in \mathbb{N}_0\} \tag{3.13}$$

Die Schleife `while` $x_i \neq 0$ `do` P `end` führt den Schleifenkörper `P` wiederholt aus, solange $x_i \neq 0$ gilt. Wenn x_i den Wert 0 erhält, wird die Schleife beendet. Die Aussage der Terminierungsmenge ist daher: Wie oft muss P wiederholt werden, damit x_i Null wird? Ob x_i gleich 0 ist, kann man anhand des Speichervektors feststellen. Die Belegung aller anderen Variablen ist für die Semantik der Schleife gleichgültig, lediglich der i-te Eintrag ist von Interesse.

Salopp gesprochen »probiert« die Terminierungsmenge, wie oft hintereinander `P` ausgeführt werden muss, bis x_i gleich 0 wird, und liefert die benötigte Anzahl an Wiederholungen. Dies ergibt allerdings ein Problem: Nachdem T_i als Menge über den natürlichen Zahlen und nicht als explizite Schleife aufgebaut ist, wird der Test implizit für alle möglichen Wiederholungsanzahlen durchgeführt. Je nach Konstruktion der Schleife kann es vorkommen, dass die Abbruchbedingung nicht nur einmal, sondern mehrere Male erreicht wird. Der Code

Die Modulo-Operation ist nicht Bestandteil der Sprachdefinition, sondern syntaktischer Zucker, den wir der Einfachheit halber verwenden.

```
1: x_2 := 2
2: while x_2 mod 5 ≠ 0 do
3:     x_2 := x_2 + 1
4: end while
```

terminiert beispielsweise nach drei Schleifendurchläufen (wenn x_2 den Wert 5 erreicht hat). Nach Definition von T_i wären aber auch acht Durchläufe zulässig, um die gewünschte Abbruchkonfiguration zu erreichen, da x_2 dann den Wert 10 hat und $10 \equiv 0 \bmod 5$ gilt. Die Terminierungsmenge nimmt für obigen Code daher den Inhalt

$$T_i = \{3, 8, 13, 18, \ldots\} = \{3 + i \cdot 5 \mid i \in \mathbb{N}_0\} \tag{3.14}$$

an. Die höherzahligen Durchläufe sind natürlich nur ein Artefakt der mathematische Definition und nicht die Intention der Sprachdefinition. Man wählt daher die kleinste Anzahl von Schleifendurchläufen als korrekte Lösung aus, die die Abbruchbedingung erfüllt, indem man das kleinste Element von T_i selektiert. Formal schreibt man dies als $\min(T_i)$; für das Beispiel gilt $\min(\{3 + i \cdot 5 \mid i \in \mathbb{N}_0\}) = 3$.

$\min(T_i) = 0$ ist ein mögliches Resultat, wenn die Schleife nie durchlaufen wird.

Zusätzlich muss der Sonderfall berücksichtigt werden, dass eine Schleife nicht terminiert, was beispielsweise bei folgendem Codefragment der Fall ist:

```
1: x_1 := 3
2: while x_1 ≠ 0 do
3:     x_1 := x_1 + 2
4: end while
```

Nachdem die Abbruchbedingung nie erfüllt ist, gibt es keinen Wert, der in die Terminierungsmenge aufgenommen wird, und es gilt $T_i = \varnothing$. Semantisch wird dieser Fall abgehandelt, indem der Speichervektor durch das Symbol $\perp$ für »ungültiges Ergebnis« ersetzt wird.

Die Semantik der `while`-Schleife ist zusammenfassend durch

$$\delta(\vec{v}, \texttt{while } x_i \neq 0 \texttt{ do } P \texttt{ end}) \equiv \begin{cases} \bot & \text{falls } T_i = \varnothing \\ \delta(\vec{v}, P^{\min(T_i)}) & \text{falls } T_i \neq \varnothing \end{cases} \tag{3.15}$$

bestimmt. Wir legen weiterhin $\delta(\bot, P) = \bot$ fest, da eine einzige nichtterminierende Schleife in einem Programm natürlich dazu führt, dass das gesamte Programm nicht terminiert.

Abschließend sei bemerkt, dass die Definition formaler Semantiken für Programme nicht nur eine akademische Spielerei der theoretischen Informatik ist, sondern bei Systemen wie Airbags oder Bremsassistenten, die auf funktionale Sicherheit abzielen, oftmals eine strikte Erfordernis ist. Auch bei der Implementierung von Computerchips spielen Methoden der formalen Semantik eine wichtige Rolle.

Beispielsweise hat die Firma Intel vor einigen Jahren gelernt, dass es billiger ist, die Funktionsweise von Prozessoren formal zu verifizieren und damit die Anzahl der Fehler so gering wie möglich zu halten, als Chips mit defekten Recheneinheiten auszuliefern. Die genauen Hintergründe zum Fall erhalten Sie mit einer Recherche nach dem FDIV-Bug im Pentium-Prozessor.

Mit Hilfe der formalen Semantik für WHILE-Programme können wir eine Definition der While-Berechenbarkeit aufstellen, die sich am bekannten Konzept der Turing-Berechenbarkeit orientiert:

While-Berechenbarkeit

Eine partielle Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist *While-berechenbar*, wenn es ein While-Programm P gibt, das bei Eingabe von $x_1, x_2, \ldots, x_k$ die Ausgabe $y = f(x_1, x_2, \ldots, x_k)$ liefert und terminiert:

$$\delta((x_1, x_2, \ldots, x_k, 0), P) = (x_1, \ldots, x_k, y)$$

Falls P nicht terminiert, gilt $f(x_1, \ldots, x_k) = \bot$.

Um zu zeigen, dass Programmiersprachen und Turing-Maschinen gleich leistungsfähig sind, müssen wir beweisen, dass sich Turing- und While-Berechenbarkeit gegenseitig implizieren. Dies wird in folgendem Satz festgehalten:

Satz: While-Programme und Turing-Maschinen

Jedes While-Programm kann auf einer Turing-Maschine simuliert werden.

Der Beweis der Vorwärtsrichtung ist trivial, da es unschwer möglich ist, sämtliche WHILE-Sprachelemente auf eine Mehrband-Turing-Maschine (die beispielsweise ein Band für jede Variable verwendet) abzubilden – wir verzichten daher auf eine formale Rechnung. Die Rückwärtsrichtung ist etwas komplizierter, weshalb wir zunächst eine weitere Programmiersprache einführen, die zur WHILE-Sprache berechnungsäquivalent ist, mit der aber Turing-Maschinen sehr einfach simuliert werden können.

3.4.2 Die Sprache GOTO

»Basic« steht für Beginner's all purpose symbolic instruction code und wurde entworfen, um den Einstieg in die Programmierung zu erleichtern. Mit Assembler werden Sie (unter anderem) mehr in der Vorlesung »Datenverarbeitungssysteme« erfahren.

Während sich die WHILE-Sprache vorwiegend an Hochsprachkonzepten orientiert, erinnert die GOTO-Sprache an gering strukturierte Ansätze wie Assembler oder Basic. Sie enthält folgende Anweisungen.

❑ Wertzuweisungen und arithmetische Operationen: $x_i := x_j \pm c$.

- ❏ Unkonditionale Sprünge: `goto` M_i. Die i-te Zeile im Quellcode ist implizit oder explizit mit dem Label M_i versehen, um sie im Sprung identifizieren zu können.
- ❏ Bedingte Sprünge: `if` $x_i = c$ `then goto` M_i. Achtung: Andere Anweisungen können nicht direkt konditional ausgeführt werden, da dies mit den vorhandenen Sprachmitteln durchgeführt werden kann.
- ❏ Die Anweisung `halt` beendet ein Programm. Die letzte Anweisung eines Programm ist implizit immer `halt`, wenn dies nicht explizit angegeben wird.

Die Sprachen sind gleichmächtig:

Satz

Eine Funktion ist genau dann While-berechenbar, wenn sie Goto-berechenbar ist.

P steht für eine beliebige Liste von Anweisungen.

Beweis: Die Vorwärtsrichtung »⇒« ist leicht zu zeigen: Jede While-Schleife `while` $x_i \neq 0$ `do` P `end` ist äquivalent zur GOTO-Konstruktion

M_1`: if` x_i `=` 0 `then goto` M_k`;`
P`;`
M_{k-1}`: goto` M_1`;`
M_k`:` …

Nachdem `loop`-Schleifen durch `while`-Schleifen simuliert werden können und alle anderen WHILE-Sprachelemente in GOTO enthalten sind, kann jedes WHILE-Programm in ein äquivalentes GOTO-Programm umgewandelt werden.

Um die Rückwärtsrichtung »⇐« zu zeigen, müssen wir eine Konstruktion angeben, die beliebige GOTO-Programme in WHILE-Programme umwandelt. Das GOTO-Programm hat dabei die allgemeine Form

M_1 : A_1;
M_2 : A_2;
…
M_n : A_n;

wobei A_i beliebige Anweisungen repräsentiert. Das Programm kann durch folgendes WHILE-Programm simuliert werden:

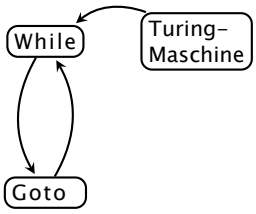
```
y := 1;
while y ≠ 0 do
  if y = 1 then A_1 end;
  if y = 2 then A_2 end;
  …
  if y = n then A_n end;
end
```

Die Variable y übernimmt die Rolle eines »Instruktionszeigers« und verweist auf die Zeile, die im GOTO-Programm ausgeführt wird. In jedem

GOTO-Element	WHILE-Simulation
`goto` M_j	$y := j$
`if` x_i `goto` M_j	$y := y + 1$; `if` x_i `then` $y := j$ `end`
`halt`	$y := 0$
A (sonstige Anweisungen)	A; $y := y + 1$

Tabelle 3.3: Abbildung zwischen Goto- und While-Sprachelementen

Durchlauf der `while`-Schleife wird (mindestens) eine Zeile des GOTO-Programms simuliert. Die einzelnen Sprachelemente werden wie in Tabelle 3.3 gezeigt aufeinander abgebildet.
Damit ist die Äquivalenz beider Sprachen gezeigt. □



3.4.3 *Simulation von Turing-Maschinen*

Um den Ringschluss zwischen Programmiersprachen und Turing-Maschinen zu beenden, bleibt noch folgender Satz zu zeigen:

Satz: Turing-Maschinen und Goto-Programme

Jede deterministische Turing-Maschine kann durch ein GOTO-Programm simuliert werden.

Beweis: Wir betrachten eine beliebige Turing-Maschine mit Zustandsmenge $Q = \{q_1, q_2, \ldots, q_n\}$ und Arbeitsalphabet $\Gamma = \{a_1, \ldots, a_N\}$. Beide Mengen sind endlich und können deshalb eindeutig numeriert werden; insbesondere kann man anstatt der Symbole a_i auch natürliche Zahlen verwenden. Dem i-ten Symbol wird die Zahl i zugewiesen; die Funktion idx(·) ermittelt die Zahl, die zu einem gegebenen Symbol gehört:

$$\mathrm{idx}(a_i) = i, i \geq 1. \tag{3.16}$$

Repräsentiert man die Zahlen in der Basis b mit $b > |\Gamma| = N$, genügt für jede auftretende Zahl eine einzige Ziffer.

Die aktuelle Konfiguration der Turing-Maschine wird wie bislang üblich angegeben, indem man den aktuellen Zustand zwischen den linken und rechten Bandteil schreibt:

$$\underbrace{a_{i_m} \ldots a_{i_1}}_{\text{Bandteil } \vec{x}}\, q_l\, \underbrace{a_{j_1} \ldots a_{j_n}}_{\text{Bandteil } \vec{y}} \tag{3.17}$$

Die beiden Bandteile $\vec{x}$ und $\vec{y}$ können bijektiv durch eine b-adische Zahl (Zahl zur Basis b) *ohne* Null repräsentiert werden, wenn man folgende Definition vornimmt:

$$x \equiv \underbrace{\langle i_m i_{m-1} \ldots i_2 i_1 \rangle}_{\text{Big Endian}} = \sum_{k=1}^{m} i_k b^{k-1}$$

$$y \equiv \underbrace{\langle j_1 j_2 \ldots j_{n-1} j_n \rangle}_{\text{Little Endian}} = \sum_{k=1}^{n} j_k b^{k-1}$$

Damit besteht die Konfiguration einer Turing-Maschine nicht mehr aus Zeichenketten, sondern aus drei Zahlen – je eine für den linken und rechten Bandinhalt, und eine für den aktuellen Zustand. Mit Zahlen kann die GOTO-Sprache problemlos umgehen. Um einen Konfigurationsübergang der Turing-Maschine zu simulieren, müssen folgende Rechnungen ausgeführt werden:

Ein Übergang ohne Kopfbewegung muss nicht separat berücksichtigt werden, da er durch eine passende Rechts–Links–Bewegung ersetzt werden kann, siehe Übung 5.43.

1. Der Übergang $\delta(q_i, \sigma) = (q_j, \sigma', \mathrm{L})$ wird durch folgende Rechensequenz implementiert:

   ❏ Erstes Zeichen in y ersetzen:

   $$y \leftarrow y \div b,$$
   $$y \leftarrow \mathrm{idx}(\sigma') + b \times y.$$

   ❏ Linksbewegung ausführen:

   $$y \leftarrow (x \bmod b) + b \times y,$$
   $$x \leftarrow x \div b.$$

❏ Zustand ändern: $q \leftarrow j$.

2. Der Übergang $\delta(q_i, \sigma) = (q_j, \sigma', \mathrm{R})$ wird durch folgende Rechensequenz implementiert:

   ❏ Erstes Zeichen in y ersetzen:

   $$y \leftarrow y \div b,$$
   $$y \leftarrow \mathrm{idx}(\sigma') + b \times y.$$

   ❏ Rechtsbewegung ausführen:

   $$x \leftarrow b \times x + (y \bmod b),$$
   $$y \leftarrow y \div b.$$

   ❏ Zustand ändern: $q \leftarrow j$.

Um die Korrektheit der Definition zu veranschaulichen, betrachten wir die Turing-Maschine $M = (\{q_0, q_1, q_2, q_f\}, \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}, \Sigma \cup \{\square\}, \delta, q_0, \{q_f\})$, deren Transitionsfunktion das Element $\delta(q_2, \mathrm{b}) = (q_0, \mathrm{c}, \mathrm{L})$ enthalten soll. Wir verwenden die (wohlbekannte) Basis $b = 10$; dies ist erlaubt, da $b = 10 > |\Gamma| = 5$ gilt. Die einzelnen Elemente von Γ erhalten die Ziffern

$$\begin{aligned} \square &\equiv 0, \\ \mathrm{a} &\equiv 1 & \mathrm{b} &\equiv 2, \\ \mathrm{c} &\equiv 3 & \mathrm{d} &\equiv 4 \end{aligned} \tag{3.18}$$

Angenommen, die Maschine befindet sich in der Konfiguration

$$\underbrace{\mathrm{abaacb}}_{\vec{x}}\, q_2\, \underbrace{\mathrm{bbaca}}_{\vec{y}}\,. \tag{3.19}$$

Dann wird der linke Bandteil $\vec{x}$ durch die Ziffernfolge 121132 repräsentiert, die den numerischen Wert $x = 121132$ besitzt. Der rechte Bandteil korrespondiert zur Ziffernfolge 22131, die aufgrund der Little-Endian-Interpretation den numerischen Wert 13122 hat.

Vektoren geben im Folgenden Ziffernfolgen, Symbole ohne Vektorpfeil Zahlen an.

Um den Übergang $\delta(q_2, \mathrm{b}) = (q_0, \mathrm{c}, \mathrm{L})$ zu simulieren, muss folgende Rechnung ausgeführt werden:

1. Erstes Zeichen ersetzen:

   $$y \leftarrow 13122 \div 10 = 1312 \tag{3.20}$$
   $$y \leftarrow 3 + 1312 \times 10 = 13120 + 3 = 13123 \tag{3.21}$$

2. Linksbewegung ausführen:

   $$y \leftarrow 121132 \bmod 10 + 10 \times 13123 = 2 + 131230 = 131232 \tag{3.22}$$
   $$x \leftarrow 121132 \div 10 = 12113 \tag{3.23}$$

Damit ergeben sich die Ziffernfolgen $\vec{x} = 12113$ und (wiederum unter Beachtung der Little-Endian-Konvention für den rechten Bandteil) $\vec{y} = 232131$, die zu den Zeichenketten $\vec{x} = \mathrm{abaac}$ und $\vec{y} = \mathrm{bcbaca}$ korrespondieren.

Die Konfiguration der Turing-Maschine nach dem Übergang ist damit wie erforderlich

$$\underbrace{\text{abaac}}_{\vec{x}}\, q_0\, \underbrace{\text{bcbaca}}_{\vec{y}}, \tag{3.24}$$

die Rechnung war also korrekt.

Mit Hilfe der Mechanik für Konfigurationsübergänge, die durch ein GOTO-Programm berechnet wird, können wir nun ein allgemeines GOTO-Programm angeben, das eine beliebige Turing-Maschine simuliert:

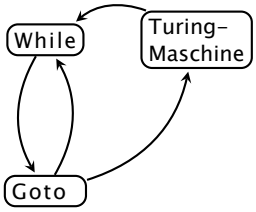
1. Variablen x, y, q initialisieren.
2. Transitionsfunktion simulieren:

```
M2: a := y  mod  b; // Aktuelles Zeichen in a
      if q=1 and a=1 then goto M1.1;
      if q=1 and a=2 then goto M1.2;
...
      if q=n and a=N then goto Mn.N;

M1.1: δ(q1, a1) simulieren
      goto M2;
M1.2: δ(q1, a2) simulieren
      goto M2;
...
Mn.N: δ(qn, aN) simulieren
      goto M2;
```

3. Ergebnis nach x_0 schreiben, wenn ein akzeptierender Endzustand erreicht wurde.

Essentiell bietet das Programm für jedes Element der Transitionsfunktion ein Label $M_{n.m}$ an, das den entsprechenden Übergang von $\delta(\cdot,\cdot)$ nachvollzieht. Der aktuelle Zustand der Maschine ist in q gespeichert. In jedem Simulationsschritt wird durch Berechnung von $y \mod b$ das aktuell gelesene Bandsymbol ermittelt; zusammen mit dem aktuellen Zustand kann festgestellt werden, welcher Übergang auszuführen ist. Am Ende der Rechnung wird das Resultat – kombinierter linker und rechten Bandinhalt – in die Variable x_0 geschrieben. □



Damit ist der Ringschluss zwischen Sprachen und Turing-Maschinen vollendet.

3.4.4 Universelle Turing-Maschinen

Im Vergleich mit »normalen« Computern, Tablets oder Mobiltelefonen besitzen Turing-Maschinen einen konzeptionellen Nachteil: Sie sind nur auf ein einziges Programm festgelegt, das durch die Übergangsfunktion δ festgelegt ist. Wenn ein zweites, unterschiedliches Programm ausgeführt werden soll, muss die Übergangsfunktion ersetzt werden, was bei einer physikalischen Umsetzung mit dem Austausch der Steuereinheit korrespondiert. Dies ist unpraktisch: Auf einem Computer kann ein Programm ersetzt werden,

indem der alte Code aus dem Speicher entfernt und durch einen neuen Code ersetzt wird. Ist der beschriebene Nachteil fundamentaler Natur? Nein, da wir in diesem Abschnitt eine Turing-Maschine konstruieren werden, deren Programm wie bei einem orthodoxen Computer austauschbar ist, ohne die Definition der eigentlichen Recheneinheit – des Steuerwerks – zu verändern.

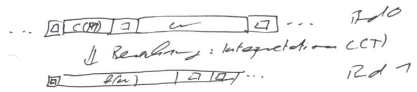


Abbildung 3.5: Universelle Turing-Maschine

Wir bezeichnen die Maschine als *universelle* Turing-Maschine; Abbildung 3.5 zeigt eine graphische Illustration. Eingabewort ω und abgearbeitetes Programm f werden auf dem ersten Band gespeichert. Die Maschine berechnet $f(\omega)$ und speichert das Resultat auf einem zweiten Band. Mehrband-Turing-Maschinen stehen bereits zur Verfügung; das wesentliche Problem ist daher, eine TM so zu kodieren, dass Ihre Beschreibung auf dem zweiten Band gespeichert werden kann. Die Idee ist, jede Turing-Maschine eindeutig durch eine Zahl zu repräsentieren.

Natürlich könnte man auch drei oder mehr Bänder verwenden: Eines für die Eingabe, eines für das abgearbeitete Programm, und eines oder mehrere für das Resultat und Zwischenergebnisse. Nachdem Ein- und Mehrbandmaschinen zueinander äquivalent sind, ist dies aber ein irrelevantes Detail.

Bekanntlich kann jede Turing-Maschine M ohne Beschränkung der Allgemeinheit durch

$$M = (\{q_0, \ldots, q_n, q_f\}, \{0,1\}, \{0,1,\square\}, \delta, q_0, \square, \{q_f\}) \tag{3.25}$$

repräsentiert werden. Im Rahmen der GOTO-Simulation haben wir bereits besprochen, wie die darin auftretenden Mengen und Alphabete durch Zahlen ersetzt werden können. Es bleibt lediglich die Transitionsfunktion durch eine Kodierung in Zahlen zu ersetzen. Strukturell ist $\delta(q,a) = (q', b, \{\mathrm{L}, \mathrm{R}, \mathrm{N}\})$ nichts anderes als ein 5-Tupel

$$(q, a, q', b, \{\mathrm{L}, \mathrm{R}, \mathrm{N}\}) \tag{3.26}$$

mit $a, b \in \{0,1\}$ und $q, q' \in Q$. Nachdem alle Elemente des Tupels als Zahlen kodiert werden können, kann man auch das gesamte Tupel durch eine (konkatenierte) Zahl darstellen. Kombiniert man die Kodierungen der Mengen Q, Σ und Γ mit der Kodierung der Übergangsfunktion, erhält man eine Zahl, die die Maschine M eindeutig charakterisiert. Eine Abbildung

$$c : M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F) \to \mathbb{N}, \tag{3.27}$$

die aus jeder Turing-Maschine eine natürliche Zahl erzeugt, bezeichnet man als *Gödelisierung*. Durch die Definition der Signatur ist die Abbildung nicht eindeutig festgelegt. Es gibt in der Tat viele verschiedene Möglichkeiten, eine Gödelisierung durchzuführen. Nachdem die Effizienz in unseren Überlegungen (momentan noch) keine Rolle spielt, geben wir eine *unäre* Kodierung als Beispiel an. Sie besteht aus folgenden Schritten:

Kurt Gödel (1906–1978)
TODO: Biographie Gödel. Bei dieser Gelegenheit sei Ihnen der hervorragende Essay Musil, Gödel, Wittgenstein und das Unendliche aus den Wiener Vorlesungen von Rudolf Taschner ans Herz gelegt, der für alle interessant ist, die sich auch für den kulturell-historischen Hintergrund ihres Faches interessieren.
Zur Erinnerung: Die unäre Kodierung ersetzt die Zahl N durch N Ziffern, ist also durch die Abbildung $1 \to 1$, $2 \to 11$, $3 \to 111$,...definiert.

Gödelisierung in unärer Kodierung

Q, Σ, Γ und $\{\mathrm{L}, \mathrm{R}, \mathrm{N}\}$ werden durch

$$Q \to \mathbb{N} \qquad \Sigma \to \mathbb{N} \tag{3.28}$$

$$\Gamma \to \mathbb{N} \qquad \{\mathrm{L}, \mathrm{R}, \mathrm{N}\} \to \mathbb{N} \tag{3.29}$$

eindeutig (dezimal) numeriert, und die Ergebnisse in unäre Zahlen

umgewandelt. Jedes Element des 5-Tupels von $\delta(\cdot,\cdot)$ wird unär kodiert und die einzelnen Elemente durch die Ziffer »0« getrennt; als Trennsymbol zwischen den Tupeln wird »00« verwendet.

TODO: *Beispiel: Siehe Tafel*

Natürlich ist nicht jede beliebige Funktion $c(M)$ eine sinnvolle Gödelisierung; beispielsweise bildet $c(M) = 1 = \text{const.}$ wie gefordert jede Maschine auf eine Zahl ab, aber leider auch jede Maschine auf die gleiche Zahl. Man stellt daher einige Anforderungen an eine Gödelisierung, die in folgender Definition zusammengefasst sind:

Definition: Gödelisierung Man bezeichnet eine Funktion $c : M \to \mathbb{N}$ als *Gödelisierung*, wenn sie folgende Eigenschaften besitzt:

- ❑ c ist injektiv – dies stellt sicher, dass jede Maschine auf genau eine Zahl abgebildet wird (allerdings repräsentiert nicht jede Zahl eine Maschine).
- ❑ Die Bildmenge $c(M)$ ist entscheidbar – dies stellt sicher, dass man algorithmisch entscheiden kann, ob eine gegebene Zahl eine Turing-Maschine kodiert oder nicht.
- ❑ Sowohl $c : M \to \mathbb{N}$ als auch $c^{-1} : \mathbb{N} = c(M) \to M$ sind berechenbar – dies stellt sicher, dass die Abbildung (und auch die Rückwärtstransformation) tatsächlich durchgeführt werden können. ∎

3.5 Das Halteproblem

Bei den Überlegungen zu endlichen Automaten haben wir uns im Wesentlichen mit Sprach- und damit mit Mengenproblemen beschäftigt; das Konzept der Turing-Berechenbarkeit ist hingegen auf Funktionen zentriert. Die Konzepte sind aber nicht disjunkt voneinander, sondern können über die charakteristische Funktion miteinander verknüpft werden:

Eine analoge Definition könnte man auch für Semi–Entscheidbarkeit angeben, worauf wir hier aber der Einfachheit halber verzichten.

Definition: Charakteristische Funktion Die charakteristische Funktion $\chi_A : \Sigma^* \to \{0,1\}$ einer Menge A ist definiert durch

$$\chi_A(\omega) = \begin{cases} 1 & \text{für } \omega \in A \\ 0 & \text{für } \omega \notin A. \end{cases} \tag{3.30}$$

Eine Menge $A \subseteq \Sigma^*$ heißt *entscheidbar*, falls die charakteristische Funktion χ_A berechenbar ist. ∎

Das Wortproblem wird dadurch, nachdem eine Sprache nichts anderes als eine Menge ist, auf die Berechnung einer Funktion zurückgeführt. Dieser formale Kniff scheint zunächst keine praktische Bedeutung zu haben. Man kann das Konzept allerdings nutzen, um ein interessantes Problem zu analysieren, das sich mit den Eigenschaften gödelisierter Turing-Maschinen beschäftigt.

3.5.1 Das spezielle Halteproblem

Um den Komplikationen zu entgehen, die mit der Injektivität der Gödelisierung einhergehen, vereinbaren wir zunächst eine Konvention zur

Erleichterung. Sei $\hat{M}$ eine beliebige TM. Dann ist M_ω definiert durch

$$M_\omega \equiv \begin{cases} M & \text{wenn } \omega \text{ ein Codewort von } M \text{ ist,} \\ \hat{M} & \text{sonst.} \end{cases} \tag{3.31}$$

Unter »Codewort« verstehen wir eine korrekte Kodierung unter einer gegebenen Gödelisierung $c(\cdot)$.

Mit Hilfe der Definition vermeiden wir, im Folgenden zwischen gültigen und nicht-gültigen Gödelzahlen unterscheiden zu müssen, da *jede* Zahl auf eine gültige Maschine abgebildet wird.

Als Gödelzahl oder Gödelnummer bezeichnet man die Zahl, die als Resultat der Gödelisierung entsteht.

Bei sicherheitskritischer Software möchte man idealerweise sicherstellen, dass ein Programm unter allen gegebenen Umständen korrekt funktioniert – beispielsweise soll ein Bremssystem in einem Zug immer dann die Bremswirkung auslösen, wenn es explizit (Betätigen des Bremshebels) oder implizit (Überfahren eines Haltesignals) gewünscht ist, und immer *nicht* auslösen, wenn kein entsprechender Grund vorliegt – beispielsweise weil sich ein ICE mit 357,28 km/h auf freier Strecke bewegt und daher zur Steigerung des Fahrkomforts keine grundlose Vollbremsung machen sollte. Aufgrund der hohen Anzahl unterschiedlicher Auslösesituationen ist es prinzipiell unmöglich, alle Szenarien systematisch zu testen, weshalb man die Korrektheit der Software nur durch eine (statische) Analyse des Programm-Quellcodes sicherstellen kann. Kann man einen Algorithmus oder eine Software konstruieren, die ein allgemeines Programm untersucht und prüft, ob dieses korrekt funktioniert?

Man unterscheidet in der Informatik zwischen Informationssicherheit (security) und funktionaler Sicherheit (safety). Wir beziehen uns in diesem Abschnitt vor allem auf letzteren Begriff.

Das beschriebene Problem ist zu allgemein, um es sinnvoll zu behandeln. Daher betrachten wir ein strukturell einfacheres Problem: Wenn eine Turing-Maschine mit einer Eingabe gestartet wird, hält sie an oder nicht? Offensichtlich ist diese Aufgabe leichter (da spezifischer) als das vorherige Problem; die Analogie zum Bremsbeispiel ist dennoch offensichtlich, wenn man »halten« mit »Bremsen« identifiziert. Ebenso ist klar, dass es nicht ausreicht, das getestete Programm einfach zu simulieren, da man eine Entscheidung der testenden Maschine in endlicher Zeit erhalten möchte, ein nicht-haltendes Programm aber unendlich lange läuft.

Die Kombination aus Maschine und Eingabe bezeichnen wir als »getestetes Programm« oder »analysierte Maschine«.

Daher versuchen wir, eine Analysemaschine M zu konstruieren, die eine beliebige andere Turing-Maschine analysiert und entscheidet, ob diese anhält oder nicht. Mit Hilfe der Gödelisierung ist sichergestellt, dass die analysierte Maschine als Zahl dargestellt und damit für die Analysemaschine M als Eingabeparameter verfügbar gemacht werden kann. Auch die analysierte Maschine benötigt eine Eingabe, auf der sie ausgeführt wird; wir legen fest, dass die Gödelisierung der Maschine selbst als Eingabeparameter verwendet wird (dies ist offensichtlich in den meisten Fälle keine sinnvolle Eingabe, aber letztlich genauso gut oder schlecht wie jede andere zufällige Eingabe, da nichts genaues über das Problem bekannt ist, das die analysierte Maschine repräsentiert.

Wir unterscheiden zwischen zwei Maschinen: Der analysierenden Maschine und der analysierten Maschine.

Das beschriebene Szenario kann man kompakt und etwas abstrakter durch folgenden Satz beschreiben:

Satz: Spezielles Halteproblem

Gegeben sei die Selbstanwendbarkeitsmenge

$$K \equiv \{\omega \in \{0,1\}^* \mid M_\omega \text{ angesetzt auf } \omega \text{ hält}\}.$$

Die Menge K ist nicht entscheidbar.

Der Satz enthält bereits das betrübliche Endergebnis: Es ist *nicht* möglich. Bevor wir einen formalen Beweis angeben, betrachten wir die Illustration in Abbildung 3.6. TODO: *Beschreibung.*

Abbildung 3.6: Unlösbarkeit des Halteproblems

TODO: *Abbildungen einscannen*

Nachdem die Überlegung anschaulich klar ist, führen wir noch eine formale Rechnung durch, die die Argumentation wasserdicht macht. Wir nehmen an, dass eine Maschine M existiert, die die charakteristische Funktion χ_K der Selbstanwendbarkeitsmenge K berechnet und damit das spezielle Halteproblem löst. Wir konstruieren eine zweite Maschine M', die M ausführt und wie folgt reagiert (das Ergebnis »nicht halten« wird realisiert, indem M' in eine Endlosschleife geht):

$$M' \begin{cases} \text{hält, wenn } M \text{ das Ergebnis 0 (da } \omega \notin K) \text{ liefert.} \\ \text{hält nicht, wenn } M \text{ das Ergebnis 1 (da } \omega \in K) \text{ liefert.} \end{cases} \tag{3.32}$$

Sei w' die Gödelisierung von M'. Wir betrachten die Konsequenzen, wenn M' auf w' angesetzt wird und nehmen an, dass die Maschine hält:

$$\begin{aligned} M' \text{ angesetzt auf } w' \text{ hält } &\Leftrightarrow M \text{ angesetzt auf } w' \text{ gibt 0 aus} \\ &\Leftrightarrow \chi_K(\omega') = 0 && (3.33) \\ &\Leftrightarrow \omega' \notin K && (3.34) \\ &\Leftrightarrow M_{\omega'} \text{ angesetzt auf } \omega' \text{ hält nicht} && (3.35) \\ &\Leftrightarrow M' \text{ angesetzt auf } \omega' \text{ hält nicht.} && (3.36) \end{aligned}$$

Die Folgerungen bis zu Zeile (3.33) sind Konsequenzen der Definition in Gleichung (3.32). In Zeile (3.34) haben wir die Definition von χ_K verwendet; Zeile (3.35) folgt aus der Definition der Selbstanwendbarkeitsmenge K. Zeile (3.36) resultiert schließlich aus das Tatsache, dass ω' die Gödelisierung von M' ist, weshalb $M_{w'} = M'$ gilt. Zusammenfassend stellen wir fest, dass

$$M' \text{ angesetzt auf } w' \text{ hält } \Leftrightarrow M' \text{ angesetzt auf } w' \text{ hält nicht} \tag{3.37}$$

gilt, was ein offensichtlicher Widerspruch ist. Daher ist die ursprüngliche Annahme falsch, die postulierte Maschine M existiert also nicht – und das spezielle Halteproblem ist unentscheidbar. □

Wir möchten besonders hervorheben, dass die Unentscheidbarkeit des speziellen Halteproblems weder vom aktuellen Stand der Technik noch von zukünftigen algorithmischen Entwicklungen abhängt: Schnellere Rechner werden auch in der Zukunft keine Entscheidung herbeiführen können, da wir im Beweis keine Effizienzüberlegungen angestellt haben. Ebenfalls kann in der Zukunft kein Algorithmus zur Lösung des Problems entdeckt werden, da wir keinen expliziten Algorithmus für die Maschine M angegeben haben, sondern den Widersprich nur unter Annahme der Existenz *irgendeiner* Maschine M (oder Äquivalent dazu der Existenz irgendeines Algorithmus) konstruiert haben! Der Beweis der Unentscheidbarkeit bezieht sich damit nicht auf eine konkrete Lösungsidee für das Problem, sondern auf das Problem an sich.

3.5.2 *Reduzierbarkeit*

Eine Maschine, die auf Ihre eigene Beschreibung angewendet wird, ist kurios – reale Anwendungen verwenden offenbar andere Eingaben. Allerdings ist intuitiv klar, dass die Unentscheidbarkeit des speziellen Halteproblems auch die Unentscheidbarkeit des allgemeineren Problems – hält eine Turing-Maschine, die auf eine beliebige Eingabe angewendet wird? – nach sich zieht, da letzteres Problem schwieriger als ersteres Problem ist. Als allgemeine Eingabe könnte man prinzipiell auch die Gödelisierung der Maschine verwenden, woraus ein bewiesen unentscheidbarer Fall folgt. Ein Problem ist aber auch dann unentscheidbar, wenn spezifische Teilprobleme nicht entscheidbar sind.

Um diese Überlegung mathematisch sauber durchzuführen, müssen wir das Konzept »leichterer« und »schwererer« Probleme formalisieren. Dazu verwendet man den Begriff der *Reduzierbarkeit*:

Definition: Reduzierbarkeit Seien $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ Sprachen. Dann heißt A auf B reduzierbar ($A \leq B$), wenn es eine totale berechenbare Abbildung $f : \Sigma^* \to \Gamma^*$ gibt, so dass $\forall x \in \Sigma^*$:

$$x \in A \Leftrightarrow f(x) \in B.$$

■

Die Definition besagt implizit auch, dass $x \notin A \Leftrightarrow f(x) \notin B$ gilt. Wörter, die nicht in A enthalten sind, müssen also notwendigerweise auf Wörter außerhalb der Sprache B abgebildet werden. Allerdings muss nicht jedes Wort aus B auf ein Wort aus A abgebildet werden; die Sprache B kann damit umfangreicher als Sprache A sein, aber nicht umgekehrt.

Essentiell ist ein Problem A ist auf ein (schwereres) Problem B reduzierbar, wenn jede Eingabe für Problem A auf eine Eingabe für Problem B abgebildet werden kann, die das Enthaltensein in der Sprache nicht verändert. Dies führt zu einer Möglichkeit, aus einer Lösungsstrategie für Problem B eine Lösung für Problem A zu ermitteln, die in folgendem Satz festgehalten ist:

Bei der Definition von f lassen wir Effizienzüberlegungen außen vor; die Funktion kann beliebig aufwendig zu berechnen sein, solange sie überhaupt zu berechnen ist.

Satz: Reduzierbarkeit und Entscheidbarkeit

Es gilt $A \leq B$ und B ist entscheidbar $\Rightarrow$ A ist entscheidbar.

Beweis: Wenn $A \leq B$ gilt, existiert nach Voraussetzung eine Abbildung f, die Eingaben von A in Eingaben von B transformiert. Die charakteristische Funktion χ_B ist berechenbar, da B entscheidbar ist. Die Komposition zweier berechenbarer Funktionen ist berechenbar, also auch $\chi_B \circ f$. Die Resultate von χ_A und χ_B sind verknüpft durch

$$\chi_A(x) = \begin{cases} 1 & \text{für } x \in A \\ 0 & \text{für } x \notin A \end{cases} = \begin{cases} 1 \text{ für } f(x) \in B \\ 0 \text{ für } f(x) \notin B \end{cases} = \chi_B(f(x)). \tag{3.38}$$

Damit ist χ_A berechenbar, wenn χ_B berechenbar ist. Daraus folgt wiederum, dass A entscheidbar ist. □

3.5.3 *Allgemeines Halteproblem und Satz von Rice*

Mit den mittlerweile zur Verfügung stehenden Hilfsmitteln ist es einfach, die Unentscheidbarkeit des allgemeinen Halteproblems zu zeigen, wie wir in folgendem Satz festhalten:

Satz: Allgemeines Halteproblem

Gegeben sei die Sprache

$$H \equiv \{w\#x \mid M_\omega \text{ angewandt auf } x \text{ hält }\}.$$

H ist nicht entscheidbar.

Beweis: Wir verwenden die Kontraposition des Reduzierbarkeitslemmas, die eine Aussage über unentscheidbare Probleme liefert. Wenn $A \leq B$ und A unentscheidbar ist, ist auch B unentscheidbar. Zur Transformation zwischen dem speziellen und dem allgemeinen Halteproblem verwenden wir als Abbildung die Funktion

Anschauliche Interpretation: Wenn B schwieriger als B ist, aber bereits A unentscheidbar ist, muss B unentscheidbar sein.

$$f(\omega) = \omega\#\omega : \omega \in K \Leftrightarrow f(\omega) \in H, \tag{3.39}$$

die offensichtlich total (da für jede Eingabe definiert) und berechenbar (da simple Konkatenation von Zeichenketten) ist. Nachdem K unentscheidbar ist und $K \leq H$ gilt, ist auch H unentscheidbar. □

…oder wenigstens für das Selbstverständnis der statischen Verifikation.

Die Aussage des allgemeinen Halteproblems ist für das Selbstverständnis der Informatik sehr negativ; schließlich zeigt es der Disziplin eine klare Grenzlinie, die noch dazu bereits bei einem sehr einfachen Problem gezogen wird. Die Situation stellt sich allerdings in Form des Satzes von Rice noch schlechter dar:

Satz von Rice

Sei $\mathcal{R}$ die Klasse aller Turing-berechenbaren Funktionen. Sei $\mathcal{S} \subset \mathcal{R}$ und $\mathcal{S} \neq \emptyset$. Dann ist die Sprache

$$C(\mathcal{S}) = \{\omega \mid \text{Die von } M_\omega \text{ berechnete Funktion liegt in } \mathcal{S}\}$$

nicht berechenbar.

Der Satz ist in dieser Form auf den ersten Blick nur schwer verständlich. Etwas anschaulicher ist folgende Variante:

Satz von Rice (anschauliche Formulierung)

Sei E eine nicht-triviale funktionale Eigenschaft von Turing-Maschinen, und sei M eine Turing-Maschine. Dann ist das Problem »Besitzt M die Eigenschaft T?« nicht entscheidbar.

Auch in der einfacheren Form ist es notwendig, den Satz genau zu lesen und die darin enthaltenen Feinheiten zu verstehen, vor allem die der Eigenschaft E zugeschriebenen Qualitäten:

Eigenschaften, die jeder Turing-Maschine zugeschrieben werden können, sind in diesem Zusammenhang nicht zielführend, da sich offenbar die Frage nicht stellt, ob eine gegebene Maschine die Eigenschaft besitzt oder nicht.

- *Nicht-trivial* bedeutet, dass mindestens eine Maschine die Eigenschaft *nicht* besitzt.
- *Funktional* kennzeichnet eine relevante Eigenschaft der Maschine. Beispielsweise ist es nicht sinnvoll zu fragen, ob eine Turing-Maschine gelb, freundlich oder überschall-schnell ist, da dies den Ausgang einer Rechnung nicht beeinflusst. Gültige funktionale Eigenschaften sind beispielsweise »Eine Turing-Maschine liefert eine durch 2 teilbare Zahl als

Die Geschwindigkeit der Maschine ist tatsächlich nicht interessant, da wir nur daran interessiert sind, ob ein Problem prinzipiell lösbar ist – nicht wie lange es dauert, dies zu lösen.

Ergebnis«, oder »Eine TM gibt mindestens zweimal das gleiche Zeichen hintereinander aus«.

TODO: *Beweis: Siehe Tafel*

Das Halteproblem und der Satz von Rice gelten für alle Turing-vollständigen Berechnungsmechanismen, also auch Programmiersprachen! Sie ziehen daher sehr praxisrelevante Implikationen nach sich, da beispielsweise folgende Aufgaben nicht erfüllt werden können:

- *Statische Verifikation*: Es ist nicht möglich, die Korrektheit eines Programms zu beweisen. Erst das tatsächliche Ausführen am Computer zeigt, ob die gewünschten Ergebnisse berechnet werden oder nicht (Hinweis: Erschöpfendes Testen aller Eingabeparameter ist aufgrund der kombinatorischen Explosion unterschiedlicher Szenarien nicht möglich).
- *Optimierende Compiler*: Es ist nicht möglich, optimalen Code bei der Übersetzung in Maschinencode zu erzeugen, da ein nicht-haltendes Programm in eine simple Endlosschleife aus wenigen Assembler-Anweisungen übersetzt werden müsste. Der entstehende Maschinencode liefert daher nie ein Ergebnis zurück und hält nicht. Um zu erkennen, welche Programme auf diese Art und Weise zu behandeln wären, müsste der Compiler allerdings das Halteproblem lösen!
- *Virenscanner*: Um zu erkennen, um ein Programm gut- oder böswillig ist, muss es ausgeführt werden – was man in letzterem Fall natürlich nicht möchte. Virenscanner verwenden daher in der Praxis heuristische Techniken oder suchen nach »Fingerabdrücken« bekannter Computerviren, können aber prinzipiell nicht auf bislang unbekannte Schadprogramme reagieren.

»Optimal« ist ohne Angabe eines Optimierungskriteriums nicht sinnvoll; wir verwenden hier Codegröße als Maß. Andere Kriterien wären beispielsweise Geschwindigkeit oder Energieeffizienz; die Argumentation verläuft anlog.

Das Marketing verschiedener Hersteller scheint diese simple Tatsache zu ignorieren, was schlimm genug ist. Wirklich verheerend ist die Tatsache, dass viele Softwaresysteme sich auf die Wirkung von Virenscannern verlassen und nicht damit nur die Symptome, nicht aber das eigentliche Problem lösen, nämlich Software von Grund auf sicher zu entwerfen und Einfallstore für Schadcode zu vermeiden, anstatt diese Tore von vornherein zu schließen.

Allerdings darf die Bedeutung des Satzes auch nicht überbewertet werden: Für das Programm

```
1: procedure SIMPLE(n)
2:     n ← 2 × n
3:     return n
4: end procedure
```

ist die Eigenschaft »berechnet ein durch 2 teilbares Resultat« beispielsweise trivial anhand des Quellcodes zu entscheiden. Steht dies im Widerspruch zum Satz von Rice? Nein. Der Satz bezieht sich darauf, die Gültigkeit einer Eigenschaft für *beliebige* Programme zu ermitteln, einschließlich sehr komplizierter Programme. Es kann also durchaus einfache Programme geben, für die eine bestimmte gewünschte Eigenschaft nachweisbar ist.

In safety-kritischen Programmen beschränkt man sich daher häufig auf leicht zu analysierende Teilmengen von Programmiersprachen, um mit Hilfe statischer Analysen weitreichende Aussagen treffen zu können. Beispielsweise regelt der MISRA-Standard Programmiervorschriften für C-Programme der Automobilindustrie, die auf komplexe und schwer zu analysierende Sprachmittel verzichten müssen.

3.5.4 *Das Post'sche Korrespondenzproblem*

Wir haben in Abschnitt 2.16.4 angesprochen, dass es nicht entscheidbare Sprachprobleme gibt, sind aber bislang den Beweis dafür schuldig geblieben. Um dies nachzuholen, müssen wir durch Reduktion eine Verbindung zwischen den Sprachproblemen und beweisbar nicht entscheidbaren Problemen herstellen. Die beiden bislang bekannten unentscheidbaren Probleme – das spezielle und das allgemeine Halteproblem – eigenen sich

Emil Post (1897–1954)
Die bekannteste Leistung von Post ist die Verbindung zwischen unlösbaren Problemen und Sprachen – der früh verstorbene Mathematiker, der zeitlebens mit manisch-depressiven Anfällen konfrontiert war, hat sich allerdings auch auf dem Gebiet der Aussagenlogik und durch die Definition eines universellen Rechenmechanismus, der noch einfacher als die Turing-Maschine ist, einen Platz in der Wissenschaftshistorie gesichert.

dazu aber nicht besonders dazu weshalb wir zunächst ein weiteres nicht entscheidbares Problem einführen:

PCP (Post's Correspondence Problem)

Gegeben sei eine endliche Menge von Wortpaaren

$$(x_1, y_1), \ldots, (x_n, y_n)$$

mit $x_i, y_i \in \Sigma^+$. Gibt es eine Indexfolge $i_1, i_2, \ldots, i_k \in \{1, 2, \ldots, n\}$, so dass gilt

$$x_{i_1}, x_{i_2}, \ldots, x_{i_k} = y_{i_1}, y_{i_2}, \ldots, y_{i_k}?$$

TODO: *Beispiel: Siehe Tafel*

Wir geben den Beweis der Unentscheidbarkeit des Problems nicht explizit an, sondern verweisen auf die Literatur. Die Struktur ist, für jede Turing-Maschine eine PCP-Instanz zu konstruieren und eine Verbindung zwischen haltenden Maschinen und lösbaren PCP-Instanzen herzustellen. Nachdem sich damit durch Lösen des PC-Problems auch das Halteproblem lösen lassen würde, ist ein Widerspruch konstruiert, und das PCP als nicht entscheidbar entlarvt.

Mit dem PCP-Problem als Hilfsmittel können verschiedene Unentscheidbarkeitsaussagen über Sprachen mit vergleichsweise geringem Aufwand bewiesen werden.

Satz: Schnittproblem für kfS

Das Schnittproblem für kontextfreie Sprachen ist nicht entscheidbar.

TODO: *Beweis: Siehe Tafel*

Satz: Mehrdeutigkeitsproblem für kfS

Das Mehrdeutigkeitsproblem für kontextfreie Sprachen ist nicht entscheidbar.

TODO: *Beweis: Siehe Tafel*

Satz: Leerheitsproblem für ksS

Das Leerheitsproblem für kontextsensitive Sprachen ist nicht entscheidbar.

TODO: *Beweis: Siehe Tafel*

4
Komplexitätstheorie

Bislang haben wir uns vorwiegend mit zwei Klassen von Problemen beschäftigt: lösbaren und unlösbaren. Es ist wichtig, zwischen diesen beiden Klassen zu unterscheiden, um sich der generellen Grenzen der Informatik bewusst zu werden. Allerdings korrespondieren die lösbaren Probleme *nicht* automatisch zu den mit heutiger Technik lösbaren Problemen; es gibt innerhalb der Klasse eine weitere Unterteilung, die zwischen *praktisch* und *prinzipiell* lösbaren Probleme unterscheidet. Letztere Klasse enthält grob gesprochen alle Probleme, für die man zwar Algorithmen angeben kann, die Berechnung des Resultats für hinreichend lange Eingaben aber länger dauert, als man fundamental warten kann, beispielsweise mehrere Jahrhunderte.

4.1 Ressourcenmodellierung

4.1.1 Laufzeitbestimmung

Um zwischen praktisch und nicht praktisch lösbaren Problemen zu unterscheiden, ist ein Maß für den Aufwand einer Rechnung notwendig. In Abschnitt 3.3.3 haben wir bereits die beiden Maße Zeit- und Platzkomplexität eingeführt, deren Definition wir daher an dieser Stelle nicht wiederholen, sondern nur mit der in Abbildung 4.1 gezeigten Illustration veranschaulichen. Insbesondere sei darauf hingewiesen, dass als Variationskriterium die Eingabelänge verwendet und der schlechteste Fall (mit der längsten Laufzeit) aller Eingaben bis zu dieser Länge als Resultat gewertet wird. Nachdem wir uns mit lösbaren Problemen beschäftigen, können wir ebenfalls voraussetzen, dass die betrachtete Sprache entscheidbar ist; der Fall einer nicht haltenden Maschine ist daher ausgeschlossen.

Formal betrachtet ist die Laufzeit eine Funktion $f : \mathbb{N} \to \mathbb{N}$, die die maximale Anzahl an Schritten von M angibt, bis ein Wort akzeptiert wird. Man verwendet die beiden sprachlichen Aussagen

- »M läuft in in Zeit $f(n)$.«
- »M ist eine $f(n)$-Turing-Maschine.«

Wenn ein Wort abgelehnt wird, definiert man 0 als Ergebnis von f. Bei der Definition der Laufzeit nichtdeterministischer Maschinen gibt es in der Literatur zwei Konventionen: Man kann der Funktion f entweder die kürzeste oder die längste Rechnung zugrundelegen. Wir entscheiden uns für die längste; im Buch von Schöning wird aber beispielsweise die alternative Konvention verwendet.

Die exakte Bestimmung der Laufzeit (und analog der verwendeten Bandpositionen) funktioniert, indem man die Gesamtzahl der Zeitschritte (analog: besuchten Bandsymbole) aufaddiert. Aufgrund von »Einmaleffekten« wie Initialisierungsschritten ergibt dies aber typischerweise sehr komplexe Ausdrücke. Zusätzlich können Konstanten, die in den Ausdrücken auftreten,

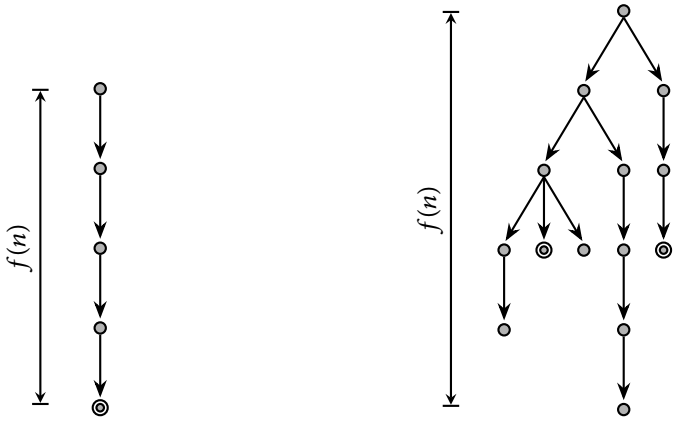


Abbildung 4.1: Messung der Zeitkomplexität bei deterministischen (links) und nicht-deterministischen (rechts) Turing-Maschinen

sich je nach Ressourcenmodell unterscheiden. Daher ist es nicht notwendig, den Ressourcenverbrauch »allzu« genau zu bestimmen, weil es zum einen unabhängig vom Rechenmodell (und nur auf den Algorithmus bezogen) ohnehin nicht möglich ist, und weil zum anderen statische Beiträge oder kleine Korrekturen für lange Eingaben irrelevant werden. Betrachten wir dies am Beispiel der Laufzeit

Die Laufzeit für kurze Eingaben ist nicht allzu interessant – es ist viel wichtiger zu wissen, wie sich ein Algorithmus mit immer länger werdenden Eingaben verhält.

$$f(n) = 6n^3 + 2n^2 + 20n + 45. \tag{4.1}$$

Bei einer Eingabe der Länge 1000 ergibt sich die Laufzeit

$$f(1000) = 6 \times 10^9 + 2 \times 10^6 + 20 \times 1000 + 45, \tag{4.2}$$

bei der der konstante Term 45 gegenüber dem höchstwertigen Term 6×10^9 offenbar vernachlässigbar ist. Selbst der zweithöchste, quadratische Term liefert nur einen Beitrag, der ungefähr 0.1‰ des kubischen Terms ausmacht – und ist daher ebenso wie der lineare Term ohne substantiellen Fehler vernachlässigbar. Wichtig ist nur die höchste Polynomordnung, in diesem Fall der kubische Term. Für eine grobe Klassifikation ist selbst der multiplikative Vorfaktor 6 nicht wesentlich, weshalb man sagt, der Ausdruck in Gleichung (4.1) ist von der Größenordnung n^3. Formal ausgedrückt schreibt man

$$f(n) = \mathcal{O}(n^3), \tag{4.3}$$

und meint damit, dass der Ausdruck »π-mal-Daumen« eine (höchstens) kubisch skalierende Anzahl von Schritten beschreibt.

Vorsicht ist aufgrund des »=«-Symbols geboten! Es handelt sich natürlich um keine Gleichheit im üblichen Sinn, sondern nur um eine grobe Zuordnung.

4.1.2 *Asymptotische Notation*

Sie vermuten korrekt, dass die Argumentation mit »π-mal-Daumen« nicht die endgültige Grenze der Exaktheit beim Vergleich von Funktionen ist. Man kann die beschriebenen Zusammenhänge über das *Landau-Symbol* formal sauber und vor allem eindeutig ausdrücken:

Definition: Landau-Symbol Gegeben $f, g : \mathbb{N} \to \mathbb{R}^+$. Man sagt $f(n) = \mathcal{O}(g(n))$, wenn $\exists c, n_0 \in \mathbb{N}^+$, so dass $\forall n \geq n_0$

$$f(n) \leq c \cdot g(n).$$

■

TODO: *Beispiel angeben*

Bezeichnung	$C = \lim_{n\to\infty} \frac{f(n)}{g(n)}$	»Grob gesprochen«
$f(n) = \mathcal{O}(g(n))$	$C < \infty$	$f \leq g$
$f(n) = \Omega(g(n))$	$C > 0$	$f \geq g$
$f(n) = \Theta(g(n))$	$0 < C < \infty$	$f = g$
$f(n) = o((g(n))$	$C = 0$	$f < g$
$f(n) = \omega(g(n))$	$C = \infty$	$f > g$

Tabelle 4.1: Klassifikation des Wachstumsverhaltens von Funktionen

4.2 *Komplexitätsklassen*

Um die Schwierigkeit unterschiedlicher Algorithmen miteinander zu vergleichen oder, allgemeiner, Algorithmen in unterschiedliche Schwierigkeitsklassen zu kategorisieren, benötigt man ein Maß, das als Basis der Klassifikation dient. Je nach Fragestellung können sind unterschiedliche Maße sinnvoll; häufig ist man aber an den beiden elementaren Ressourcen »Zeit« und »Speicherplatz« interessiert, die sich als üblicherweise betrachtete Maße etabliert haben.

4.2.1 *Die Komplexitätsklasse **P***

Nachdem wir im vorhergehenden Kapitel bereits explizite Definitionen zur Messung beider Kriterien angegeben haben, ist es leicht, darauf aufbauend eine ganze *Klasse* von Algorithmen vergleichbarer Komplexität zu definieren (wir konzentrieren uns im Folgenden auf die Zeit als Maß; die meisten Definitionen lassen sich analog auf den Speicherplatzbedarf übertragen):

Definition: Zeitkomplexitätsklasse TIME Sei $t : \mathbb{N} \to \mathbb{R}^+$. Die Zeitkomplexitätsklasse **TIME**$(t(n))$ enthält alle Sprachen, die von einer $\mathcal{O}(t(n))$-DTM entschieden werden. ∎

Die Klasse **TIME**(n^2) enthält beispielsweise alle Sprachen, deren Laufzeit maximal polynomial in der Eingabelänge skaliert. Lineare Algorithmen (oder Algorithmen mit konstanter Laufzeit) sind darin ebenfalls enthalten (also **TIME**$(n) \subseteq$ **TIME**(n^2)), Algorithmen mit kubischer Laufzeit sind dies allerdings nicht (also **TIME**$(n^3) \not\subseteq$**TIME**(n^2)).

Polynome unterschiedlicher Ordnung unterscheiden sich substantiell in ihrem Wachstumsverhalten; während n^2 selbst für große Werte von n noch zu akzeptablen Laufzeiten führt, ist dies für n^{25} eher nicht der Fall. Man wäre deshalb versucht, eine bestimmte Polynomordnung (beispielsweise n^4) als obere Grenze für effiziente Algorithmen zu wählen und alles, was darüber liegt (n^5, n^6, …) per Definition als nicht effizient zu betrachten.

Genau genommen bezieht man sich nicht auf ein Problem, sondern auf einen Algorithmus zur Lösung eines Problems.

Dies führt zu Problemen: Die Frage, ob ein Problem leicht oder schwer zu berechnen ist, sollte sich nur auf das Problem selbst beziehen, nicht aber auf das Maschinenmodell abhängen, auf dem das Problem betrachtet wird – also auch nicht davon, ob auf einer Ein- oder Mehrband-Turing-Maschine gerechnet wird. Allerdings haben wir bereits nachgewiesen, dass es für jede $\mathcal{O}(t(n))$-Mehrband-TM eine gleichwertige $\mathcal{O}(t^2(n))$-Einband-TM) – ein Algorithms, der auf ersterer Maschine in n^3 läuft und nach obiger Definition effizient ist, benötigt auf einer Einband-Maschine $(n^3)^2 = n^6$ Schritte ist damit plötzlich ineffizient, obwohl die Simulation der Einband-Maschine

aufgrund des quadratischen Verhaltens per Definition effizient ist. Eine spezifische Polynomordnung als Trenner zwischen »effizient« und »nicht effizient« ergibt also keine zufriedenstellende Unterscheidung.

Um dem beschriebenen Problem zu entgehen, macht man folgende Definition:

Definition: Komplexitätsklasse P Die Komplexitätsklasse **P** aller effizient in polynomialer Zeit berechenbaren Algorithmen ist durch

$$\mathbf{P} \equiv \bigcup_k \mathbf{TIME}(n^k) \tag{4.4}$$

definiert. ∎

Die Klasse **P** enthält beispielsweise auch Algorithmen mit Skalierungsverhalten $\mathbf{TIME}(1)$, $\mathbf{TIME}(\log n)$, $\mathbf{TIME}(n \log n)$, … Sie ist wie gewünscht invariant gegenüber polynomialen Beschleunigungen und Verlangsamungen, auf Kosten der Tatsache, dass auch pathologische Skalierungsverhalten wie n^{42}, die nur in konstruierten Problemen und nicht in der Praxis nicht auftreten, in der Klasse enthalten ist.

Algorithmen in P

Folgende weit verbreiteten Algorithmen sind in der Komplexitätsklasse **P** enthalten:

- ❏ Fast Fourier-Transformation
- ❏ Sortieralgorithmen (Merge Sort, Quicksort, Bubblesort, …)
- ❏ Binäre Suche
- ❏ Primzahl-Test
- ❏ Kontextfreie Sprachen

Ein Rechner mit einer Taktfrequenz von 1 MHz entspricht sicher nicht dem aktuellen Stand der Technik; nachdem der Graph aber bis in den Ursprung des Universums zurückreicht, zu dem der Rechner bereits existiert haben muss, ist dies kein schlechter Wert – vor allem wenn man bedenkt, dass dies ziemlich genau dem Takt des vor 25 Jahren weit verbreiteten C64 entspricht.

Man erkennt, dass die Festlegung auf polynomial skalierende Algorithmen als effiziente Algorithmen sinnvoll ist, wenn man Abbildung 4.2 betrachtet, in dem Polynome anderen Wachstumsfunktionen gegenübergestellt werden.

Auf der linken Seite ist die Anzahl von Maschinentakten auftragen, die zur Lösung von Problemen mit variierenden Eingabelängen benötigt werden. Nachdem Einheiten in μs nicht unbedingt leicht zu interpretieren sind, sind im Graph anschauliche Zeiteinheiten aufgetragen. Man erkennt, dass alle polynomial skalierenden Algorithmen für den betrachteten Längenbereich (der bis 100 reicht, was für moderne Maschinen und im Zeitalter von Terrabyte-Speichern keine furchteinflößende Größe darstellt) ihr Arbeit in deutlich unter einer Minute beenden. Algorithmen mit exponentiellem (2^n) oder gar kombinatorischem ($n!$) Skalierungsverhalten haben aber selbst dann, wenn sie zu Anbeginn des Universums gestartet worden sind, ihre Arbeit bis heute nicht beendet! Zwischen diesen beiden Klassen und Polynomen liegen also fundamentale Unterschiede, die weder durch eine Erhöhung der Taktzahl noch eine Vergrößerung der CPU-Anzahl substantiell zu beeinflussen sind, weshalb die Definition der Klasse **P** sinnvoll ist.

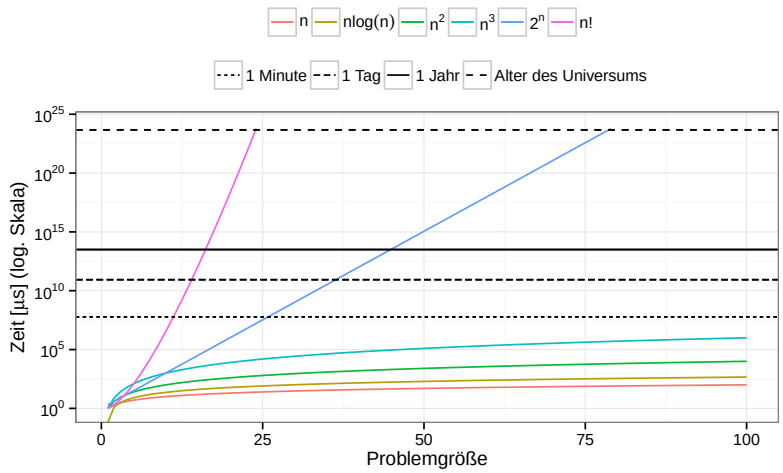


Abbildung 4.2: Skalierungsverhalten unterschiedlicher Funktionen in Relation zur Realzeit. Berechnungsgrundlage ist ein Rechner mit einem Taktzyklus von 1 μs (entsprechend einer Frequenz von 1 MHz).

4.2.2 *Die Komplexitätsklasse **NP***

Zwischen den effizienten Algorithmen mit polynomialem Skalierungsverhalten und den nicht-polynomialen Algorithmen mit Laufzeit 2^n bzw. $n!$ liegt ein sehr großer Abstand, wie aus Abbildung 4.2 klar wurde. Die Grenze der Praktikabilität beginnt daher nicht erst bei exponentiellem Skalierungsverhalten, sondern muss bereits früher starten. Es hat sich bereits gezeigt, dass die explizite Funktion zur Unterscheidung zwischen beiden Varianten keine gute Klassifikationsmöglichkeit ist. Man kann die Grenze allerdings dennoch etwas weiter präzisieren, indem man das Berechnungsmodell wechselt. Neben deterministischen Rechnungen auf einer DTM haben wir auch das Konzept des Nicht-Determinismus kennengelernt, das sich auch auf Turing-Maschinen anwenden lässt. Ebenfalls haben wir festgestellt, dass eine NTM mit einem sehr einfachen, nicht notwendigerweise optimalen Algorithmus mit exponentiellem Aufwand auf einer DTM simuliert werden kann. Eine formal elegante Charakterisierung nicht praktisch berechenbarer Funktionen ergibt sich durch die neue Komplexitätsklasse **NP**, die in starker Analogie zu **P** definiert ist:

$$\mathbf{NP} \equiv \bigcup_k \mathbf{NTIME}(n^k) \tag{4.5}$$

NTIME ist das Äquivalent von **TIME** für eine nicht-deterministische Turing-Maschine – dies bedeutet, dass Algorithmen der Klasse in polynomialer Zeit ablaufen, allerdings darf nun eine nicht-deterministische Turing-Maschine verwendet werden. Das »N« in der Definition steht für »Nicht-Deterministisch«. Per Konvention (und nach der Erfahrung der letzten Jahrzehnte) sind Probleme, die in **NP**, aber nicht in **P** enthalten sind, nicht mit praktikablem Aufwand zu lösen.

Achtung: NP bedeutet nicht Nicht-Polynomial, da die Klasse ja genau auf Basis einer polynomialen Laufzeit definiert ist!

Achtung: Wenn ein Problem in **NP** enthalten ist, bedeutet dies nicht notwendigerweise, dass kein effizienter Algorithmus existiert! Zum einen sind alle Probleme aus **P** auch in **NP** enthalten, da eine DTM trivial durch eine NTM ersetzt werden kann. Zum anderen gibt es bislang kein Problem, das in **NP**, aber *beweisbar* nicht in **P** liegt. Im Gegensatz zur fundamentalen Trennlinie zwischen berechenbaren und nicht-berechenbaren Problemen

ist es bei Problemen aus **NP** also durchaus möglich, durch trickreiche neue Algorithmen auf einfachere Lösungsmöglichkeiten zu stoßen, was in der Vergangenheit auch bereits des öfteren passiert ist.

Ein sehr bekanntes Beispiel ist ein Algorithmus zum Testen, ob eine gegebene Zahl eine Primzahl ist: Erst nach jahrzehntelanger Forschungsarbeit konnte 2002 das als AKS-Algorithmus bekanntgewordene Resultat gefunden werden, das das Problem mit polynomialem Aufwand löst. Vorher waren lediglich randomisierte oder nicht-deterministische Verfahren mit polynomialer Skalierung bekannt.

*4.2.3 Polynomiale Reduzierbarkeit und Struktur von **NP***

*4.2.4 Das **Sat**-Problem*

5
Übungsaufgaben

5.1 Aufgabe: Grundlegendes über DEAs

Betrachten Sie den in Abbildung 5.1 definierten deterministischen endlichen Automaten M.

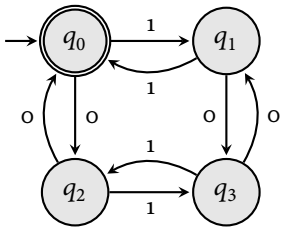


Abbildung 5.1: Deterministischer endlicher Automat, dargestellt durch einen Graphen.

1. Geben Sie eine vollständige formale Charakterisierung des Automaten M mit Hilfe des 5-Tupels $(Q, \Sigma, \delta, q_0, F)$ an.

   Eine vollständige formale Charakterisierung des Automaten ist durch folgendes 5-Tupel gegeben:

   $$Q = \{q_0, q_1, q_2, q_3\}$$
   $$\Sigma = \{0, 1\}$$

   Die Übergangsfunktion δ ist durch Tabelle 5.1 spezifiziert.

Σ \ Q	q_0	q_1	q_2	q_3
0	q_2	q_3	q_0	q_1
1	q_1	q_0	q_3	q_2

Tabelle 5.1: Transitionsfunktion für den DEA aus Abbildung 5.1

   Weiterhin gilt, dass der Startzustand durch q_0 und die Menge der akzeptierenden Endzustände durch $F = \{q_0\}$ gegeben ist.

2. Finden Sie drei Wörter, die der Automat akzeptiert, und weitere drei Wörter, die der Automat ablehnt.

   Akzeptiert werden beispielsweise die Wörter »0101«, »100001« und »00«. Abgelehnt werden beispielsweise die Wörter »110«, »01011« und »1«, da der Zustand nach Abarbeitung der Kette nicht q_0 ist.

   Es gibt (unendlich) viele weitere Wörter, die akzeptiert und abgelehnt werden – endlich ist nur die Anzahl der Zustände.

5.2 *Aufgabe: Komplexes Verhalten einfacher Funktionen*

Ziel der Aufgabe ist zu verstehen, dass selbst einfachste Algorithmen ein sehr komplexes dynamisches Verhalten zeigen können. Gegeben sei die Funktion

$$f_\mu(x) = \mu \cdot \min\{x, 1 - x\}, \tag{5.1}$$

wobei $x, \mu \in \mathbb{R}$ mit $0 < \mu \leq 2$ und $x \in [0, 1]$. Betrachten Sie die Sequenz

$$x_{n+1} = f_\mu(x_n) \tag{5.2}$$

mit $n \in \mathbb{N}$.

1. Vervollständigen Sie die Gleichung

$$x_{n+1} = f_\mu(x_n) = \begin{cases} \qquad & \text{für } x < \frac{1}{2} \\ \qquad & \text{für } \frac{1}{2} \leq x_n, \end{cases} \tag{5.3}$$

   so dass ein zu Gleichung 5.1 identisches Ergebnis berechnet wird. Der Startwert x_0 soll vorgebbar sein.

   Die vollständige Gleichung lautet

$$x_{n+1} = f_\mu(x_n) = \begin{cases} \mu \times x_n \text{ für } x < \frac{1}{2}, \\ \mu(1 - x_n) \text{ für } \frac{1}{2} \leq x_n. \end{cases}$$

2. Schreiben Sie ein Programm in Ihrer bevorzugten Programmiersprache, das die Sequenz $\{x_n\}$ für $n = 1, \ldots, N$ berechnet.

   Das Programm kann beispielsweise in GNU R[1] implementiert werden:

[1] Eine statistische Programmiersprache, die auch mit vielen Möglichkeiten zur graphischen Darstellung von Resultaten ausgestattet ist, siehe `www.r-project.org`

```
library(ggplot2)

tent.map <- function(mu, x) {
    if (x < 0.5) {
        return (mu*x)
    }

    return(mu*(1-x))
}

do.tent.map <- function(mu, x.list, n) {
    if (n == 1) {
        return (x.list)
    }

    x <- x.list[[length(x.list)]]

    return(c(do.tent.map(mu, c(x.list, tent.map(mu,
    x)), n-1)))
}

gen.series <- function(mu, x0, n) {
```

```
    return(data.frame(x0=x0, x=1:n,
    y=as.numeric(do.tent.map(mu, as.list(x0), n))))
}

dat <- gen.series(1.5, 0.6001, 70)
dat <- rbind(dat, gen.series(1.5, 0.6, 70))
dat <- rbind(dat, gen.series(1.5, 0.5999, 70))
dat$x0 <- as.factor(dat$x0)
g <- ggplot(dat, aes(x=x, y=y, colour=x0)) +
    geom_line()
ggsave("tentmap.pdf", g, width=7, height=4)
```

3. Berechnen Sie jeweils drei Sequenzen und zeichnen Sie den Graph (x-Achse: n, y-Achse: x_n) der Funktion für $n \in [0, 70]$ in ein gemeinsames Diagramm. Verwenden Sie $\mu = 1.5$ mit den Startwerten $x_0 = 0.599$, $x_0 = 0.6$ und $x_0 = 0.601$.

   Das Programm erzeugt die in Abbildung 5.2 gezeigte Ausgabe.

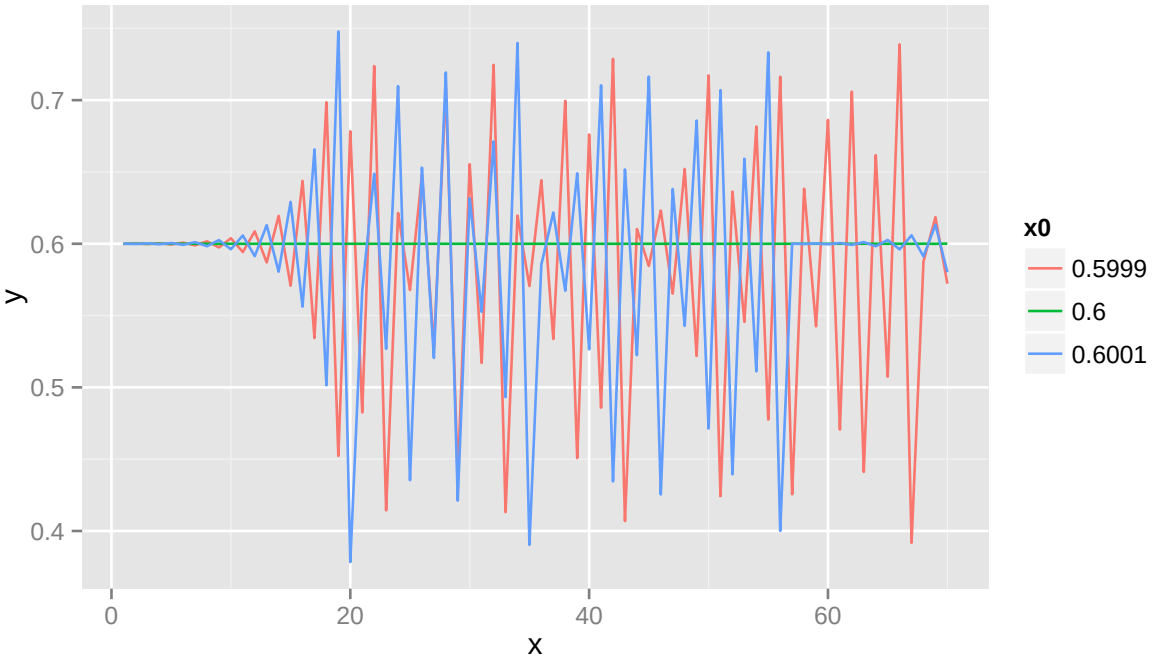


Abbildung 5.2: Drei Ausgabebeispiele für Gleichung 5.3 mit minimal unterschiedlichen Startwerten.

Trotz der extrem nahe beieinanderliegenden Startwerte (Differenz: $\pm 0,0001$) ist das dynamische Verhalten ist vollständig unterschiedlich: Für $\mu = 0.6$ erhält man eine konstante Funktion als Ausgabe, die beiden anderen Startwerte erzeugen chaotische Reihen, die keine Korrelationen zueinander aufweisen. Moral der Geschichte: Selbst einfachste Programme mit einfachen numerischen Formeln können ein sehr reiches und nur schwer analysierbares Verhalten erzeugen – entsprechend wichtig ist es, möglichst einfache Modelle des automatischen Rechnens zu finden, um überhaupt quantitative Aussagen über Programme treffen zu können. Die Suche nach solchen Modellen ist daher einer der Hauptaspekte der theoretischen Informatik.

5.3 *Aufgabe: Umgekehrte Polnische Notation*

Arithmetische Ausdrücke sind eine formale Sprache über dem Alphabet

$$\Sigma_1 = \{0, 1, \ldots, 9, +, -, \times, \div, (,)\}.$$

Beispielsweise sind $5 + 3 \times 7$ oder $(8 + 2) \times 3 + 2$ arithmetische Ausdrücke.

Die UPN wurde traditionellerweise in HP-Taschenrechnern verwendet; heute ist sie vor allem in der Seitenbesschreibungssprache PostScript (dem Vorläufer von PDF) oder in der Nischenprogrammiersprache Forth im Einsatz.

Eine alternative Schreibweise (die *umgekehrte polnische Notation* (UPN); sie kommt ohne Klammern aus!) besteht darin, zunächst die Operanden (Zahlen) und anschließend den Operator (Rechenoperation) anzugeben. Obige Ausdrücke finden folgende Entsprechung:

- $(8 + 2) \times 3 + 2 \rightarrow 8\ 2\ +\ 3\ \times\ 2\ +$
- $5 + 3 \times 7 \rightarrow 3\ 7\ \times\ 5\ +$

Zur Auswertung der Ausdrücke gelten folgende Regeln:

- Der Ausdruck wird von links nach rechts gelesen
- Trifft man auf einen Operator, wird er auf die beiden links von ihm stehenden Zahlen angewendet. Beide Zahlen und der Operator werden durch das Resultat ersetzt.

Berechnen Sie den Wert folgender Ausdrücke und geben Sie die dazu verwendeten Zwischenschritte an:

1. $1\ 2\ +\ 3\ +$
2. $2\ 3\ \times\ 4\ \times$
3. $9\ 3\ 7\ \times\ 5\ \times\ +$
4. $5\ 7\ 8\ +\ \times$
5. $5\ 7\ 8\ \times\ +$

Formen Sie folgende Ausdrücke in die umgekehrte polnische Notation um:

1. $1 + 2 + 3$
2. $5 + 3 \times (2 + 2)$
3. $5 \times (2 + 2) \times (1 + 2)$

1. Die Werte der Ausdrücke sind wie folgt gegeben:
   (a) $1\ 2\ +\ 3\ + \Rightarrow 3\ 3\ + \Rightarrow 6$
   (b) $2\ 3\ \times\ 4\ \times \Rightarrow 6\ 4\ \times \Rightarrow 24$
   (c) $9\ 3\ 7\ \times\ 5\ \times\ + \Rightarrow 9\ 21\ 5\ \times + \Rightarrow 9\ 105\ + \Rightarrow 114$
   (d) $5\ 7\ 8\ +\ \times \Rightarrow 5\ 15\ \times \Rightarrow 75$
   (e) $5\ 7\ 8\ \times\ + \Rightarrow 5\ 56\ + \Rightarrow 61$
2. Die Ausdrücke werden wie folgt umgewandelt:

(a) $1 + 2 + 3 \Rightarrow 1\ 2\ 3\ +\ +$ oder $1\ 2\ +\ 3 +$ oder $3\ 2\ 1\ +\ +$. Nachdem nur assoziative Operatoren gleicher Priorität verwendet werden, kann der Ausdruck beliebig umgeklammert werden.

(b) $5 + 3 \times (2 + 2) \Rightarrow 2\ 2\ +\ 3\ \times\ 5 +$

(c) $5 \times (2 + 2) \times (1 + 2) \Rightarrow\ 2\ 2\ +\ 5\ \times\ 1\ 2\ +\ \times$

5.4 *Aufgabe: Konstruktion einer Grammatik*

1. Geben Sie eine Grammatik mit Alphabet $\Sigma = \{a, b, c\}$ an, die die Sprache

$$L = \{x \in \Sigma^* \mid x \text{ enthält } \textit{nicht } ab\}$$

erzeugt.

2. Leiten Sie ein Beispielwort $w \in L$ mit $|w| = 5$ ab, und geben Sie den zugehörigen Ableitungsbaum an.

Um eine passende Grammatik mit den textuell beschrieben Einschränkungen zu erhalten, verwenden wir folgendes Konstruktionsprinzip: Nach Erzeugung des Buchstabens »a« werden alle Buchstaben erlaubt, der Buchstabe »b« aber vermieden. Nach jeden Buchstaben außer »a« darf ein beliebiger anderer Buchstabe auftreten. Dies beiden Regeln übersetzt man formal in eine Grammatik mit Startsymbol $\langle S \rangle$):

$$\begin{aligned} \langle S \rangle &\rightarrow \mathtt{a}\langle A \rangle \mid \mathtt{b}\langle S \rangle \mid \mathtt{c}\langle S \rangle \mid \epsilon \\ \langle A \rangle &\rightarrow \mathtt{a}\langle A \rangle \mid \mathtt{c}\langle S \rangle \mid \epsilon \end{aligned} \tag{5.4}$$

Das Wort »baacb« ist beispielsweise Mitglied der Grammatik; der zugehörige Ableitungsbaum findet sich in Abbildung 5.3.

5.5 *Aufgabe: Vollständige Induktion*

Die vollständige Induktion ist ein wichtiges Beweisprinzip der theoretischen Informatik. Sie soll deshalb an einem einfachen arithmetischen Beispiel verdeutlicht werden. Dazu sei $S(n)$ die Summe der ersten n natürlichen Zahlen, d.h.

$$S(n) = 1 + 2 + \cdots + n = \sum_{k=1}^{n} k, \tag{5.5}$$

und $C(n)$ die Summe

$$C(n) = 1^3 + 2^3 + 3^3 + \cdots + n^3 = \sum_{k=1}^{n} k^3. \tag{5.6}$$

Wir behaupten, dass

$$\begin{aligned} S(n) &= \frac{1}{2} n(n+1), \\ C(n) &= \frac{1}{4} n^2 (n+1)^2 \end{aligned}$$

gilt.

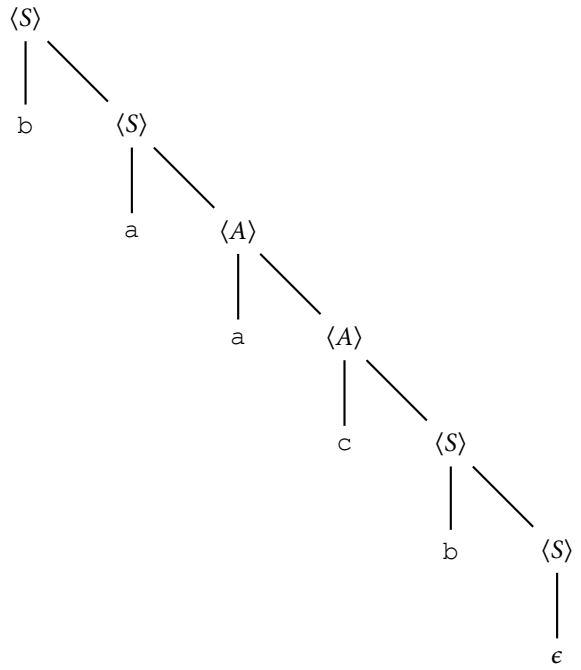


Abbildung 5.3: Ableitungsbaum für das Wort »baacb«, das durch Grammatik 5.4 erzeugt wird.

.

1. Beweisen Sie beide Aussagen jeweils durch vollständige Induktion. Zur Erinnerung: Sie müssen den Induktionsanfang ($n = 1$) beweisen, und anschließend zeigen, dass die Aussage für $n + 1$ unter Verwendung des Ergebnisses für n gilt.

2. Zeigen Sie, dass
$$\left(\sum_{k=1}^{n} k\right)^2 = \sum_{k=1}^{n} k^3$$
gilt.

DER BEWEIS für die erste Behauptung verläuft wie folgt:

1. Induktionsanfang $n = 1$:
$$\sum_{k=1}^{n} k = 1$$
$$S(n = 1) = \frac{1}{2}1(1 + 1) = 1 \checkmark$$

2. Induktionsschritt $n \to 1$:
$$\begin{aligned}\sum_{k=1}^{n+1} k &= (n + 1) + \sum_{k=1}^{n} k \\ &= (n + 1) + \frac{1}{2}n(n + 1) \\ &= \left(\frac{1}{2}n + 1\right)(n + 1) = S(n + 1) \checkmark\end{aligned}$$

□

Die Rechnung für die zweite Behauptung ist mechanisch aufwendiger, aber strukturell ähnlich:

- Induktionsanfang $n = 1$:

$$\sum_{k=1}^{n} k^3 = 1 = C(n = 1) \checkmark$$

- Induktionsschritt $n \to n + 1$:

$$\begin{aligned} C(n+1) &= \frac{1}{4}(n+1)^2(n+2)^2 \checkmark \\ \sum_{k=1}^{n+1} k^3 &= (n+1)^3 + \sum_{k=1}^{n} k^3 \\ &= (n+1)(n+1)^2 + \frac{1}{4}n^2(n+1)^2 \\ &= \left(\frac{1}{4}n^2 + (n+1)\right)(n+1)^2 \\ &= \left(\frac{1}{2}n + 1\right)^2 (n+1)^2 = \left(\frac{1}{2}(n+2)\right)^2 (n+1)^2 \\ &= \frac{1}{4}(n+2)^2(n+1)^2 = C(n+1) \checkmark \end{aligned}$$

□

5.6 *Aufgabe: Vollständige Induktion auf Sprachen*

Das Beweisprinzip der vollständigen Induktion funktioniert nicht nur mit arithmetischen Ausdrücken, sondern kann beispielsweise auch auf Sprachen angewendet werden. Zeigen Sie daher mittels vollständiger Induktion, dass die Sprache

$$L = \{(ab)^n \mid n \in \mathbb{N}, n > 0\}$$

durch die Grammatik $G(V, \Sigma, P, S)$ mit

$$\begin{aligned} V &= \{S, B, C\} \\ \Sigma &= \{a, b\} \\ P &= \{S \to aB, B \to bC, C \to \epsilon | aB\} \end{aligned}$$

erzeugt wird. *Hinweis*: Betrachten Sie als Basis den Fall $n = 1$, und führen Sie die Induktion nach der Länge des Wortes. Beachten Sie im Induktionsschritt den letzten ϵ-Übergang, der zur Ableitung des Wortes der Länge n angewendet wurde *musste*.

Um den Beweis zu führen, gehen wir folgendermaßen vor:

- Betrachte Induktionsbasis, d.h. $n = 1$. Durch die Ableitung

$$S \Rightarrow aB \Rightarrow abC \Rightarrow ab \checkmark$$

  erhält man das gewünscht Wort.

- Induktionsschritt: $n \to n + 1$. Gegeben sei ein Wort $(ab)^n$, also ab … ab mit n-facher Wiederholung von ab. Dann muss es einen Ableitungsschritt

$$ab \ldots abC \Rightarrow ab \ldots ab \text{ mit } C \to \epsilon$$

  gegeben haben. Ersetzt man die angewendete Regel $C \to \epsilon$ durch $C \to aB$, kann man $C \Rightarrow aB \Rightarrow abC \Rightarrow ab$ ableiten. Dadurch ist aus dem bestehenden Wort der Länge n ein Wort der Länge $n + 1$ generiert. ✓

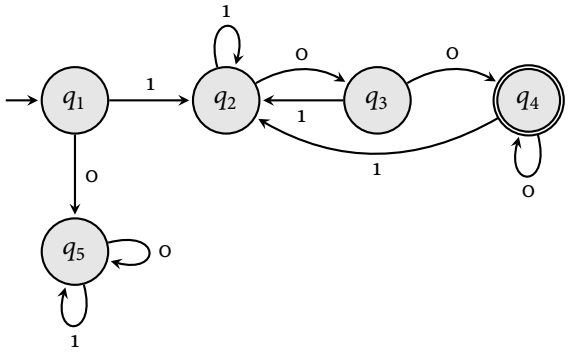
5.7 Aufgabe: Zahlenerkennung durch DEAs

1. Geben Sie jeweils den Graph eines endlichen Automaten an, der
   - alle Binärzahlen akzeptiert, die ohne Rest durch 4 teilbar[2] sind.
   - alle Binärzahlen akzeptiert, die ohne Rest durch 8 teilbar sind.
2. Geben Sie die allgemeine Definition (ohne graphische Darstellung) eines endlichen Automaten an, der Binärzahlen n akzeptiert, für die gilt $n \equiv 0 \mod 2^k \; \forall k \in \mathbb{N}\backslash\{0\}$.

[2] Alternative mathematische Ausdrucksweise für diese Aussage: Alle Zahlen n, für die $n \equiv 0 \mod 4$ gilt.

Achtung: Die Angabe wird so interpretiert, dass die Zahl mindestens eine 1 besitzen muss und diese am Anfang steht, um unterschiedlich kodierte, aber ansonsten identische Zahlen wie 010, 0010 etc. auszuschließen. Weiterhin sind die trivialen Lösung 0, 00, 000 etc. ausgeschlossen.

Zahlen, die ohne Rest durch $4 = 2^2$ teilbar sind, müssen in ihrer Binärdarstellung (mindestens) zwei aufeinanderfolgende Nullen am Schluss besitzen. Dies kann durch folgenden DEA erkannt werden:



Der Automat ist wie folgt formal definiert:

$$\Sigma = \{0, 1\}; \quad Q = \{q_1, q_2, q_3, q_4, q_5\}; \quad q_0 = q_1; \quad F = \{q_5\}$$

Die Übergänge finden sich in Tabelle 5.2.

Tabelle 5.2: Übergänge für einen DEA zur Erkennung durch 4 teilbarer Zahlen.

q \ Σ	0	1
q_1	q_5	q_2
q_2	q_3	q_2
q_3	q_4	q_2
q_4	q_4	q_2
q_5	q_5	q_5

- Die Erweiterung auf $8 = 2^3$ wird durch Einführen weiterer »Innenzustände« durchgeführt; diese Zustände stellen sicher, dass genügend viele Nullen (drei oder mehr) am Schluss stehen.
- Allgemein: Teilbar durch $2^k \rightarrow$ k-tes Bit ist 1 oder 0, die Bits $1 \ldots k-1$ sind 0 (von rechts gezählt). Der Trap-Zustand q_{k+3} ist gegeben durch

$$\delta(q_{k+3}, 0) = \delta(q_{k+3}, 1) = q_{k+3} \tag{5.7}$$

  Folgende Regeln gelten für die Anfangsziffern:

$$\delta(q_1, 0) = q_{k+3} \tag{5.8}$$

$$\delta(q_1, 1) = q_2 \tag{5.9}$$
$$\delta(q_2, 1) = q_2 \tag{5.10}$$
$$\delta(q_2, 0) = q_3 \tag{5.11}$$

Nach diesem Schema werden für $2 < m < k + 2$ die Regeln

$$\delta(q_m, 1) = q_2 \tag{5.12}$$
$$\delta(q_m, 0) = q_{m+1} \tag{5.13}$$

definiert, die durch die Regeln

$$\delta(q_{k+2}, 0) = q_{k+2} \tag{5.14}$$
$$\delta(q_{k+2}, 1) = q_2 \tag{5.15}$$

abgerundet werden.

5.8 *Aufgabe: Grammatik für PL/0*

Die Produktionen der Programmiersprache PL/0 in EBNF-Form sind durch folgende Grammatik gegeben:

Hinweis: Das frei verfügbare eBook `www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf` geht auf die Konstruktion eines Compilers für die verwandte, aber etwas umfangreichere Sprache Oberon-0 ein.

⟨*program*⟩ → ⟨*block*⟩

⟨*block*⟩ → [`const` ⟨*ident*⟩ = ⟨*number*⟩ { `,` ⟨*ident*⟩ = ⟨*number*⟩ }`;`]
[`var` ⟨*ident*⟩ {`,` ⟨*ident*⟩ }`;`]
{ `procedure` ⟨*ident*⟩`;` ⟨*block*⟩`;` } ⟨*statement*⟩

⟨*statement*⟩ → [⟨*ident*⟩`:=` ⟨*expression*⟩ | `call` ⟨*ident*⟩ |
`begin` ⟨*statement*⟩{`;` ⟨*statement*⟩ } `end` |
`if` ⟨*condition*⟩ `then` ⟨*statement*⟩ |
`while` ⟨*condition*⟩ `do` ⟨*statement*⟩]

⟨*condition*⟩ → `odd` ⟨*expression*⟩ | ⟨*expression*⟩ (= | ≠ |<| ≤ | > | ≥) ⟨*expression*⟩

⟨*expression*⟩ → [+|−] ⟨*term*⟩ { (+|−) ⟨*term*⟩ }

⟨*term*⟩ → ⟨*factor*⟩ { (∗|/) ⟨*factor*⟩ }

⟨*factor*⟩ → ⟨*ident*⟩ | ⟨*number*⟩ | `(` ⟨*expression*⟩ `)`

⟨*ident*⟩ steht für einen beliebigen Variablennamen aus Buchstaben; ⟨*number*⟩ gibt eine Zahl aus Ziffern an. Beide Produktionen werden nicht explizit spezifiziert und dürfen als gegeben vorausgesetzt werden. Klammern () in Normalschrift werden verwendet, um Auswahlgruppen zu kennzeichnen und haben keine syntaktische Bedeutung.

Gegen Sie den Ableitungsbaum für folgendes Programm an (Sie brauchen die Terminalsymbole »`,`« und »`;`« nicht im Baum zu berücksichtigen; spalten Sie bei Platzproblemen den Baum geeignet in mehrere Teile auf):

```
procedure multiply;
var a, b;

begin
  a := x;
  b := y;
```

```
  z := 0;
  while b > 0 do begin
    if odd b then z := z + a;
    a := 2 * a;
    b := b / 2
  end
end;
```

Der Syntaxbaum für das Programm ist in der Tat sehr umfangreich und deshalb über die fünf Abbildungen 5.4, 5.5, 5.6, 5.7 und 5.8 verteilt.

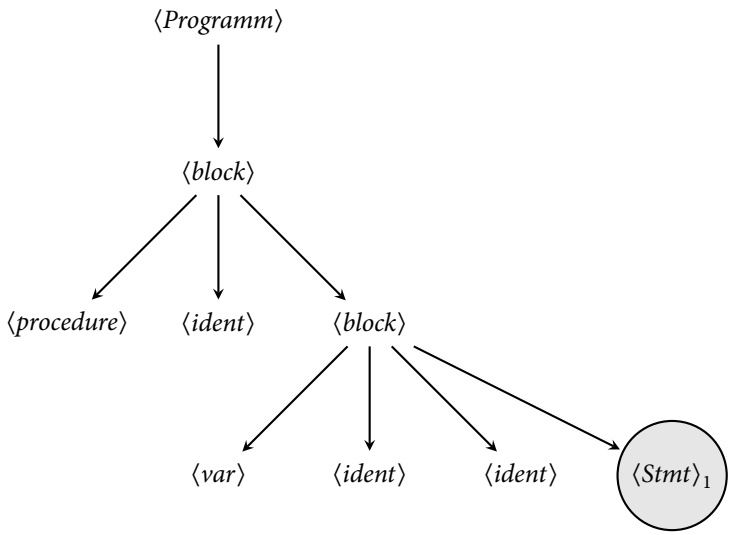


Abbildung 5.4: Syntaxbaum für das PL/0-Programm aus Abschnitt 5.8, Teil 1 (eingekreiste Knoten zeigen an, dass der Baum durch weitere Teilbaum forgesetzt wird).

5.9 *Aufgabe: Potenzmengen*

Das Konzept der Potenzmenge ist bei der Betrachtung nicht-deterministischer Automaten wichtig.

- Geben Sie die Potenzmenge $\mathcal{P}(M)$ von $M = \{a, b, c, 0, 1\}$ an.
- Geben Sie einen Algorithmus in Pseudocode oder Ihrer bevorzugten Programmiersprache an, der für eine gegebene Menge M die Potenzmenge $\mathcal{P}(M)$ konstruiert.

Die Grundmenge besteht aus 5 Elementen. Für jedes Element der Potenzmenge – eine Menge! – gilt, dass ein Element der Grundmenge entweder darin enthalten sein kann oder nicht. Insgesamt gibt es deshalb $2^5 = 32$ Elemente.

1. Alle Elemente der Potenzmenge $\mathcal{P}(\{a, b, c, 0, 1\})$ sind: (), (0), (1), (a), (b), (c), (0, 1), (0, a), (0, b), (0, c), (1, a), (1, b), (1, c), (a, b), (a, c), (b, c), (0, 1, a), (0, 1, b), (0, 1, c), (0, a, b), (0, a, c), (0, b, c), (1, a, b), (1, a, c), (1, b, c), (a, b, c), (0, 1, a, b), (0, 1, a, c), (0, 1, b, c), (0, a, b, c), (1, a, b, c), (0, 1, a, b, c).
2. Um alle Elemente einer Potenzmenge $\mathcal{P}(M)$ systematisch zu konstruieren, wird eine Bitkette so interpretiert, dass das i-te Bit anzeigt, ob das i-te Element der Menge in einem Potenzmengenelement enthalten ist oder nicht. Jedem Element der Grundmenge M wird eine eindeutige Kennziffer

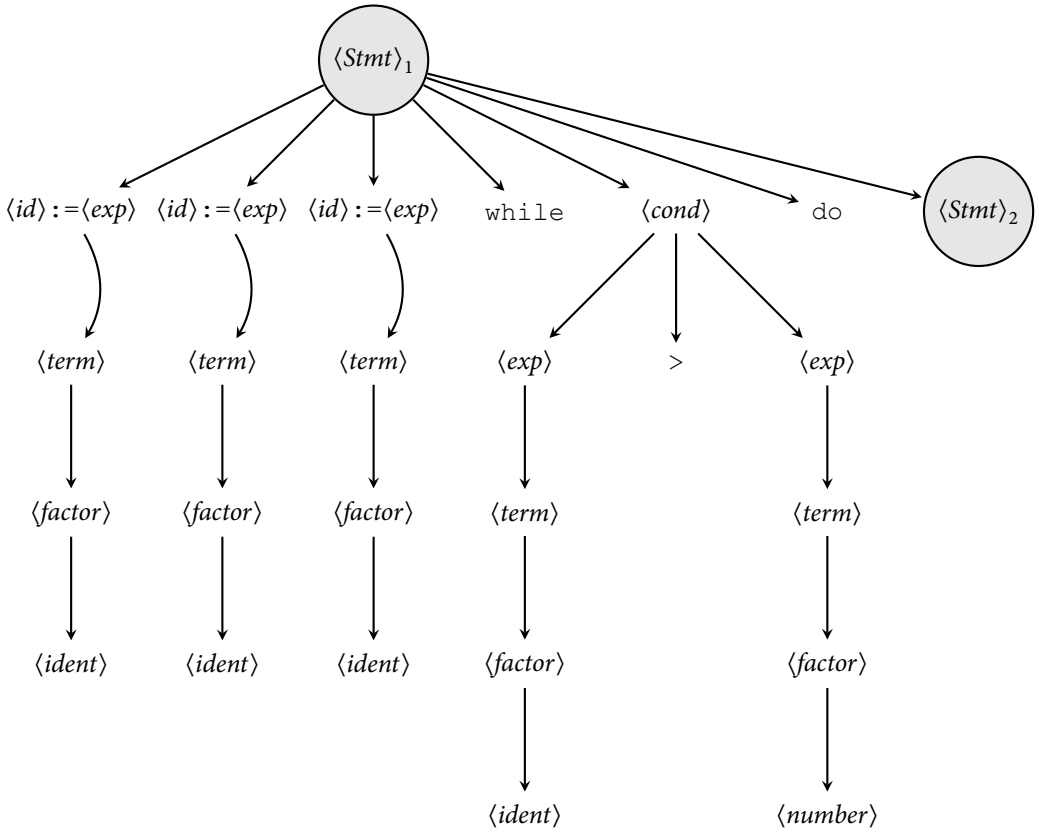


Abbildung 5.5: Syntaxbaum für das PL/0-Programm aus Abschnitt 5.8, Teil 2 (eingekreiste Knoten zeigen an, dass der Baum durch einen weiteren Teilbaum forgesetzt wird bzw. als Teilbaum eines anderen Baumes aufzufassen ist).

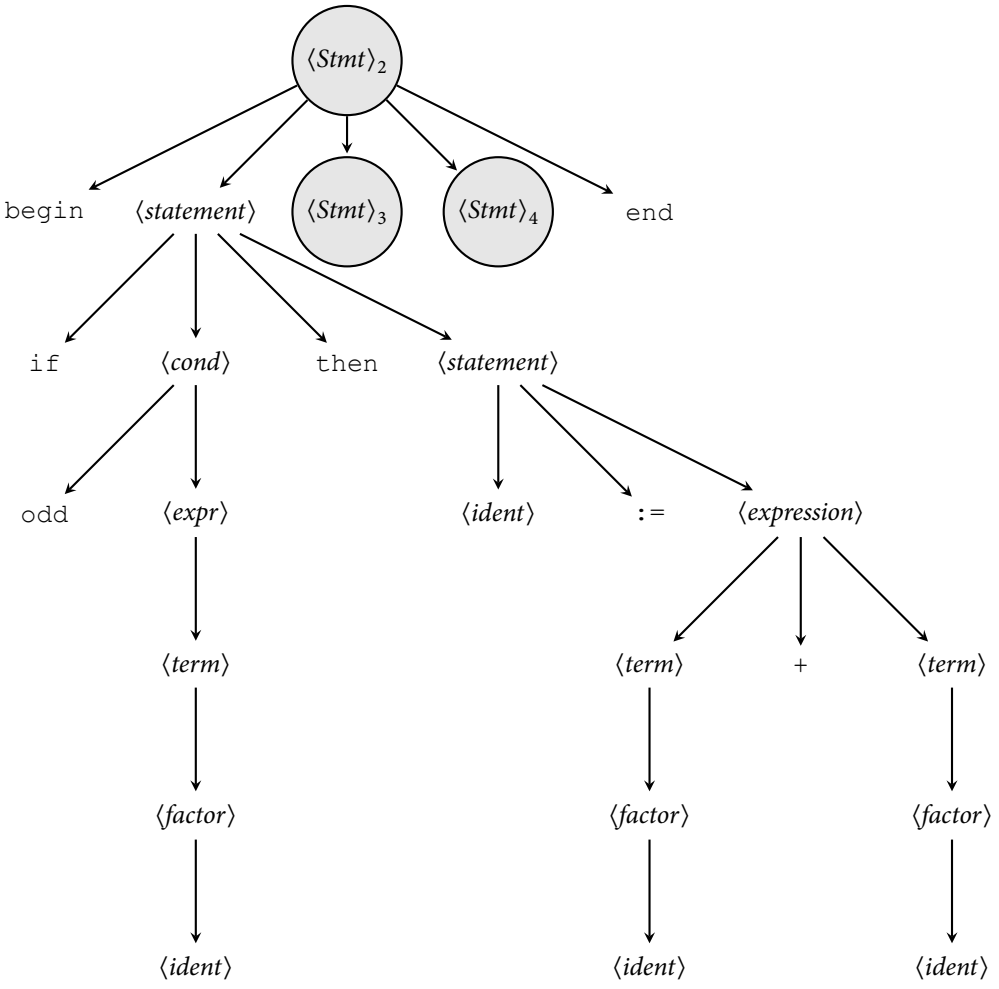


Abbildung 5.6: Syntaxbaum für das PL/0-Programm aus Abschnitt 5.8, Teil 3 (eingekreiste Knoten zeigen an, dass der Baum durch einen weiteren Teilbaum forgesetzt wird bzw. als Teilbaum eines anderen Baumes aufzufassen ist).

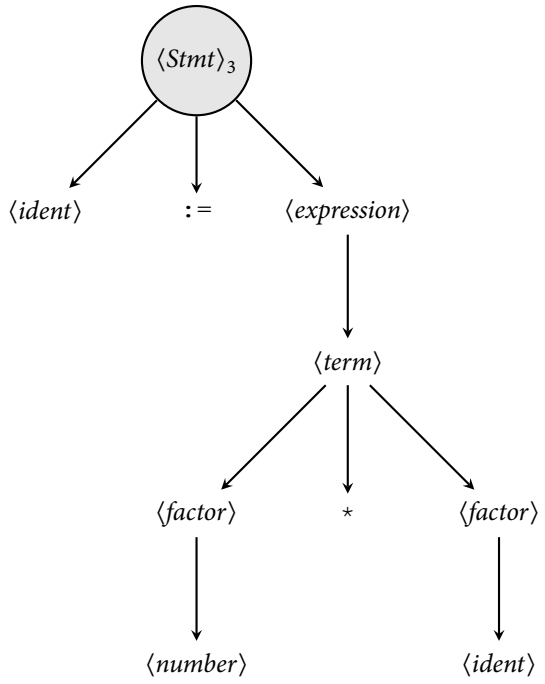


Abbildung 5.7: Syntaxbaum für das PL/0-Programm aus Abschnitt 5.8, Teil 4 (eingekreiste Knoten zeigen an, dass der Baum durch einen weiteren Teilbaum forgesetzt wird bzw. als Teilbaum eines anderen Baumes aufzufassen ist).

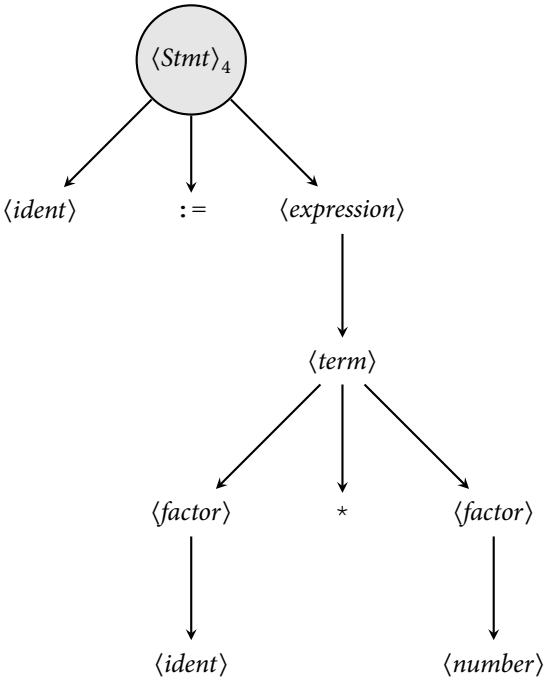


Abbildung 5.8: Syntaxbaum für das PL/0-Programm aus Abschnitt 5.8, Teil 5 (eingekreiste Knoten zeigen an, dass der Baum durch weitere Teilbaum forgesetzt wird bzw. einen als Teilbaum eines anderen Baumes aufzufassen ist).

zugewiesen. Für die Elemente der Menge $M = \{a, b, c\}$ verwendet man beispielsweise die Numerierung »a« ≙1, »b« ≙2, »c« ≙3.

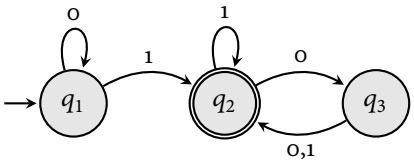
Die Teilmenge $\{a, b, c\}$ wird durch die Bitkette 111 repräsentiert; die leere Menge {} durch 000 und $\{a, c\}$ durch 101.

Ein systematischer Algorithmus kann daher über alle möglichen Belegungen einer n-Bit-Zahl iterieren und für jede Belegung die jeweilige Teilmenge ausgeben TODO: *Durch schöneren Pseudocode ersetzen*:

```
unsigned elements;
for (elements=0;elements<2^|elements|;elements++){
  print-set(elements);
}
void print-set(unsigned elements){
  if (elements & 0x1)
    print 'A';
  if (elements & 0x2)
    print 'B';
  if (elements & 0x4)
    print 'C';
}
```

5.10 *Aufgabe: DEAs und Grammatiken*

Wandeln Sie den DEA



in eine reguläre Grammatik um. Geben Sie für beide Konstrukte die vollständige formale Definition an, und zeigen Sie die Ableitung eines Wortes w Ihrer Wahl mit $|w| = 5$ aus der Grammatik.

Die Grammatik zur Erzeugung der Sprache ist wie folgt gegeben:

$$\langle Q\rangle_1 \rightarrow \mathtt{0}\langle Q\rangle_1 \mid \mathtt{1}\langle Q\rangle_2$$
$$\langle Q\rangle_2 \rightarrow \mathtt{1}\langle Q\rangle_2 \mid \mathtt{0}\langle Q\rangle_3 \mid \epsilon$$
$$\langle Q\rangle_3 \rightarrow \mathtt{0}\langle Q\rangle_2 \mid \mathtt{1}\langle Q\rangle_2$$
$$\Sigma = \{\mathtt{0}, \mathtt{1}\}$$
$$S = \langle Q\rangle_1$$
$$V = \{\langle Q\rangle_1, \langle Q\rangle_2, \langle Q\rangle_3\}$$

Der deterministische endliche Automat ist formal definiert durch

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

5.11 Aufgabe: Relationen

Sei $R = \{(1, 3), (2, 3), (3, 4), (4, 5)\}$ eine zweistellige Relation auf der Menge $M = \{1, 2, 3, 4, 5\}$.

- Visualisieren Sie R als gerichteten Graph, in dem die Elemente von M als Knoten und die Elemente R als Kanten dargestellt werden.
- Geben Sie R^2 und R^3 als Menge von 2-Tupeln an, und visualisieren Sie beide Mengen als gerichtete Graphen.
- Geben Sie die reflexiv-transitive Hülle der Relation als Menge von 2-Tupeln an.

Der Graph der Relation R ist in Abbildung 5.9 gegeben.

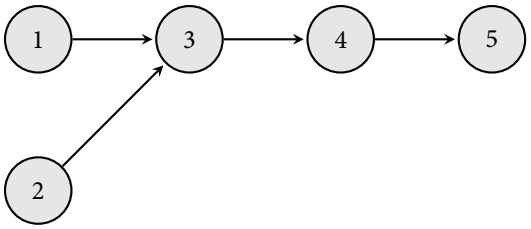


Abbildung 5.9: Graph der Relation R aus Aufgabe 5.11.

Die iterierten Relationen R^2 und R^3 erweitern ihre jeweiligen Vorgängermengen:TODO: *Besser einzeln berechnen, ohne die Vorgänger enthaltens ein zu lassen? Entspricht eher der Literaturkonvention.*

$$\mathrm{R}^2 = R \cup \{(1,4),(2,4)(3,5)\}$$
$$\mathrm{R}^3 = R^2 \cup \{(1,5),(2,5)\}$$

Die entsprechenden Graphen finden sich in Abbildung 5.10.

5.12 Aufgabe: Paritätscode

DEAs können als einfaches Rechenmodell leicht in billiger Elektronik implementiert werden. Nachdem der Paritätscode durch einen DEA umgesetzt werden kann, findet er sich in zahllosen Produkten.

Nehmen Sie an, ein Übertragungskanal kann in seltenen Fällen Bits »umdrehen«, d.h. eine verschickte »0« wird als »1« empfangen, und eine verschickte »1« erreicht den Empfänger als »0«. Ein *Paritätscode* gruppiert Daten vor dem

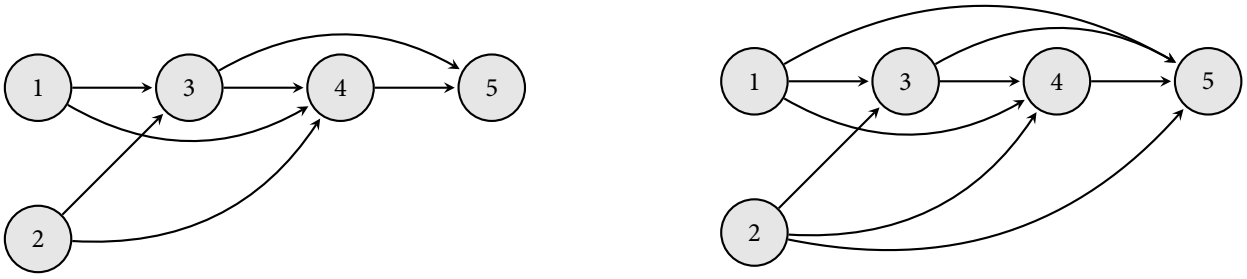


Abbildung 5.10: Graph der Relationen R^2 (linke Seite) und R^3 (rechte Seite) zu Aufgabe 5.1

Versand in Gruppen von vier Bits (x_0, x_1, x_2, x_3). Zu diesen 4 Bits wird ein fünftes Bit x_4 hinzugefügt wird. x_4 wird so gewählt, dass der resultierende 5-Bit-String eine gerade Anzahl von Einsen enthält.

1. Geben Sie fünf Beispiele für 5-Bit-Strings an, die obige Voraussetzungen erfüllen.
2. Geben Sie einen DEA an, der nur 5-Bit-Strings akzeptiert, die dieser Konvention entsprechen, und alle anderen Strings ablehnt.
3. Kann der Automat fehlerhafte Bitstrings erkennen, in denen an zwei oder mehr Positionen Bitdreher aufgetreten sind?

1. Fünf Beispiele für 5-Bit-Strings mit $\sum_{i=0}^{4} x_i \equiv 0 \mod 2$: 00000, 01001, 11110, 10111, 11000.
2. Ein DEA, der den Paritätscode umzusetzt, ist in Abbildung 5.11 zu sehen.

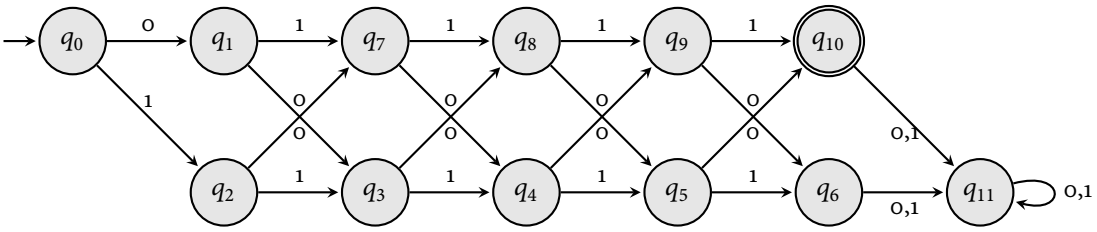
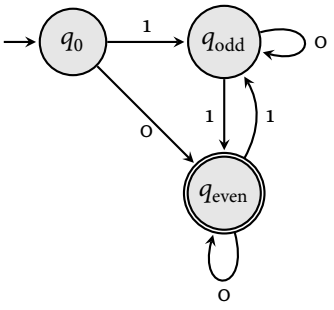
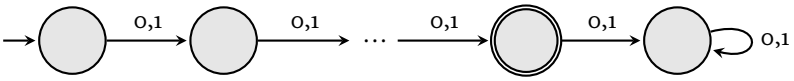


Abbildung 5.11: DEA zur Umsetzung des Paritätscodes für 4-Bit-Nachrichten.

Als alternative Lösung ist ein allgemeiner Paritätschecker möglich, wenn man die Aufgabe so interpretiert, dass auch Strings anderer Länge akzeptiert werden, die das beschriebene Kriterium erfüllen.



Um in diesem Fall sicherzustellen, dass die Länge des Wortes korrekt ist, kann ein zusätzlicher Automat eingesetzt werden:



3. Verhalten des Automats, wenn verschiedene Anzahlen von Bits geflippt werden:
   - 1 Bit geflippt: Wird erkannt
   - 2 Bits geflippt: Nicht erkannt (gleiche Summe an Einsen) $01 \rightarrow 10$, $00 \rightarrow 11$, $11 \rightarrow 00$
   - 3 Bits geflippt: Erkannt (allerdings sehr unwahrscheinlich)
   - 4 Bits geflippt: Nicht erkannt
   - …

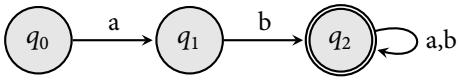
5.13 Aufgabe: Partielle Transitionsfunktionen

Deterministische endliche Automaten werden typischerweise nur mit vollständigen Übergangsfunktionen verwendet, die in jedem Zustand $q \in Q$ für jedes $\sigma \in \Sigma$ definiert sind. Nehmen Sie nun an, dass die Definition von DEAs partielle Übergangsfunktionen zulässt, in denen für $q \in Q, \sigma \in \Sigma$ gelten darf

$$\delta(q, \sigma) = \perp.$$

- Geben Sie ein Beispiel für einen DEA mit partieller Übergangsfunktion an.
- Definieren Sie eine Übergangsfunktion $\tilde{\delta}$, die δ auf eine totale Übergangsfunktion erweitert, die erkannte Sprache des Automaten aber unverändert lässt. Hinweis: Überlegen Sie, wie Sie Übergänge auf $\perp$, die zur sicheren nicht-Akzeptanz der Eingabe führen, in einem totalen DEA abbilden können.

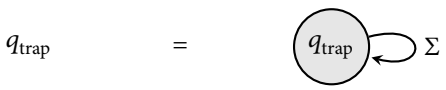
1. Ein DEA, der die Zeichenkette »ab« am Anfang einer längeren Zeichenkette erkennt, ist durch folgenden Graph gegeben:



   Die Übergangsfunktion ist für folgende Eingaben undefiniert (dargestellt durch das Symbol $\perp$):

$$\delta(q_0, b) = \delta(q_1, a) = \perp$$

2. Konstruktion: Alle Regeln, die in der Form $\delta(q, 0) = \perp$ gegeben sind, werden durch $\delta(q, 0) = q_{\text{trap}}$ mit:



   ersetzt. Formale Definition:

$$\delta(q_{\text{trap}}, \sigma \in \Sigma) = q_{trap}, q_{trap} \notin F$$

5.14 Aufgabe: Transformation Grammatik zu NEA

Gegeben sei die reguläre Grammatik G mit den Produktionen

$$\langle A\rangle \to \mathtt{b} \mid \mathtt{a}\langle A\rangle \mid \epsilon \mid \mathtt{c}\langle B\rangle$$
$$\langle B\rangle \to \mathtt{b}\langle B\rangle \mid \mathtt{b}\langle C\rangle$$
$$\langle C\rangle \to \mathtt{c} \mid \epsilon$$

über dem Alphabet $\Sigma = \{a, b, c\}$ mit Startsymbol $S = A$. Konstruieren Sie einen nicht-deterministischen endlichen Automaten M, für den gilt $\mathcal{L}(M) = \mathcal{L}(G)$.

Es werden die bekannten Konstruktionsvorlagen zur Umwandlung angewendet: $A \to b$ wird umgewandelt zu $A \to bA'$, $A' \to \epsilon$, $A' \in F$. Weiterhin wird die Regel $C \to c \mid \epsilon$ umgewandelt zu $C \to cC'$, $C' \to \epsilon$, $cC' \in F$. Dies ergibt den in Abbildung 5.12 gezeigen NEA:

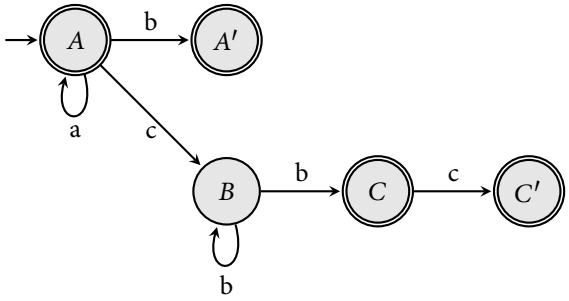
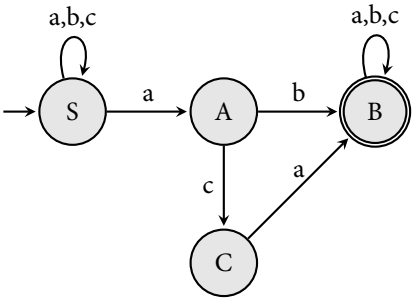


Abbildung 5.12: NEA zur Grammatik aus Aufgabe 5.14.

5.15 Aufgabe: Mechanik von NEAs

Gegeben sei der nicht-deterministische endliche Automat M:



- Geben Sie eine formale Definition der Transitionsfunktion $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ an. Finden Sie drei Wörter, die der Automat akzeptiert, und drei weitere Wörter, die der Automat nicht akzeptiert.
- Zeichnen Sie den Berechnungsbaum für den Eingabestring »abaca«, und markieren Sie alle akzeptierenden und nicht-akzeptierenden Endzustände.
- Verwenden Sie die in der Vorlesung definierte Funktion $\hat{\delta} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$, um die Menge der Endzustände nach Analyse der Zeichenkette »abc« schrittweise durch Anwendung von δ abzuleiten.

1. DIE FORMALE DEFINITION der Transitionsfunktion ist folgendermaßen gegeben:

Hinweis: Ein einzelner Zustand ist eine Ein-elementige Teilmenge von Q!

$$\begin{aligned}
\delta(S,a) &= \{S,A\}\\
\delta(S,b) &= \delta(S,c) = \{S\}\\
\delta(A,b) &= \{B\}\\
\delta(A,c) &= \{C\}\\
\\
\delta(C,a) &= \{B\}\\
\delta(B,a) &= \delta(B,b) = \delta(B,c) = \{B\}
\end{aligned}$$

Beispiele für Wörter, die akzeptiert bzw. abgelehnt werden:

- $x \in \mathcal{L}(M)$: abc, accab, aca
- $x \notin \mathcal{L}(M)$: aa, aacb, a

2. Der Berechnungsbaum für das Wort »abaca« ist in Abbildung 5.13 zu sehen.

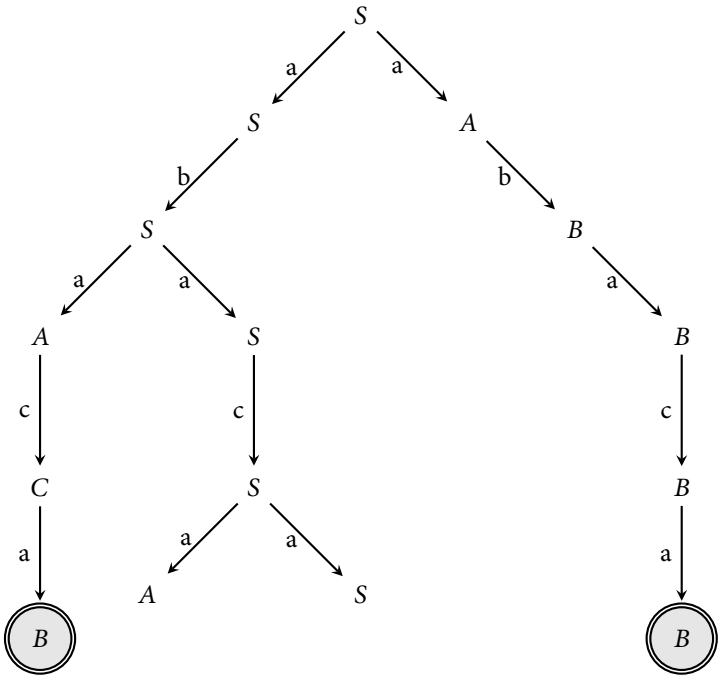


Abbildung 5.13: Berechnungsbaum des NEAs aus Aufgabe 5.15 bei der Erkennung des Wortes »abaca«.

3. Systematische Ableitung der Menge aller Endzustände:

$$\begin{aligned}
\hat{\delta}(S,abc) &= \hat{\delta}(\underbrace{\hat{\delta}(S,a)}_{\{S,A\}},bc) = \bigcup_{q\in\{S,A\}} \hat{\delta}(q,bc)\\
&= \hat{\delta}(S,bc) \cup \hat{\delta}(A,bc)\\
&= \hat{\delta}(\underbrace{\hat{\delta}(S,b)}_{\{S\}},c)) \cup \hat{\delta}(\underbrace{\hat{\delta}(A,b)}_{\{B\}},c))\\
&= \hat{\delta}(S,c) \cup \hat{\delta}(B,c)\\
&= \{S\} \cup \{B\} = \{S,B\}
\end{aligned}$$

5.16 Aufgabe: Lokale Sprachen

In einer lokalen Sprache kann die Mitgliedschaft eines Wortes entschieden werden, indem ein (durch das Wort gleitendes) Fenster der Länge 2 betrachtet wird. Besitzt eine Sprache diese Eigenschaft, kann sie von einem sogenannten lokalen Automaten, einer Unterklasse der deterministischen endlichen Automaten, erkannt werden. Das Maschinenmodell ist in wesentlichem von theoretischem Interesse; Ziel dieser Übung ist das genaue Verständnis regulärer Ausdrücke.

Eine reguläre Sprache L_1 über einem Alphabet Σ ist *lokal*, wenn es $P, S \subseteq \Sigma$ und $N \subseteq \Sigma^2$ gibt, dass $L_1 \equiv (P\Sigma^* \cap \Sigma^* S) \backslash \Sigma^* N \Sigma^*$ gilt.

1. Geben Sie anhand eines expliziten Beispiels für das Tupel (Σ, S, P, N) eine lokale reguläre Sprache an. Geben Sie drei konkrete Beispielwörter der Sprache an, und drei konkrete Wörter $w_i \in \Sigma^*$, die *nicht* aus L sind.
2. Die Definition $L_2 = PM^*S$ mit $M = \Sigma^2 \backslash N$ scheint auf den ersten Blick äquivalent zu obiger Definition zu sein, was allerdings nicht der Fall ist. Zeigen Sie dies, indem Sie ein Wort w finden, für das $w \in L_2$, aber $w \notin L_1$ gilt.
3. Geben Sie einen DEA mit *partieller* Übergangsfunktion und $Q = \Sigma \cup \epsilon$ für L_1 an. Alle Kanten mit gleicher Beschriftung sollen auf den selben Knoten führen.

1. Eine lokale reguläre Sprache ist beispielsweise gegeben durch

$$\begin{aligned} \Sigma &= \{a, b, c\} \\ P &= \{a\} \\ S &= \{c\} \\ N &= \{ab, bc, cc\} \\ M &= \Sigma^2 \backslash N = \{aa, ba, ca, bb, cb, ac\} \end{aligned}$$

   Gemäß der Definition von L_1 beginnt das Wort mit einem Buchstaben aus P, endet mit einem Buchstaben aus S, und enthält *keine* Zwei-Zeichen-Kombinationen aus der Menge N. Dies erklärt auch die oben beschriebene Eigenschaft.

   Beispielwörter, die Mitglieder der Sprache sind: »acbac«, »acacbac«, »ac«. Die Wörter »ab«, »cccc« und »abab« sind nicht Mitglied der Sprache, da sie nicht erlaubte Mittelteile enthalten.

2. Beispielsweise gilt abbc $\in L_2$, aber abbc $\notin L_1$. Das Wort enthält keine Zwei-Buchstaben-Kombination aus N im Mittelteil (d.h. ohne Anfangs- und Endbuchstabe); durch Kombination des Mittelteils »bb« mit dem Anfangsbuchstaben »a« und dem Endbuchstaben »b« entstehen allerdings die beiden Teilstrings »ab« und »bc«, die beide in N enthalten sind und damit nicht in L_1 auftreten dürfen. Dies verdeutlicht den Unterschied zwischen den beiden auf den ersten Blick ähnlich wirkenden regulären Ausdrücken, die zur Definition von L_1 und L_2 verwenden werden.
3. Ein DEA für L_1 ist in Abbildung 5.14 angegeben.

5.17 Aufgabe: NEAs und Startzustände

Alle NEAs in dieser Aufgabe verwenden *keine* ϵ-Übergänge.

1. Illustrieren Sie anhand des NEAs in Abbildung 5.15, dass ein NEA mit mehreren Startzuständen immer durch einen NEA M_2 mit einem Startzustand ersetzt werden kann, so dass $\mathcal{L}(M_1) = \mathcal{L}(M_2)$ gilt:

Abbildung 5.14: DEA für die lokale reguläre Sprache aus Aufgabe 5.16.

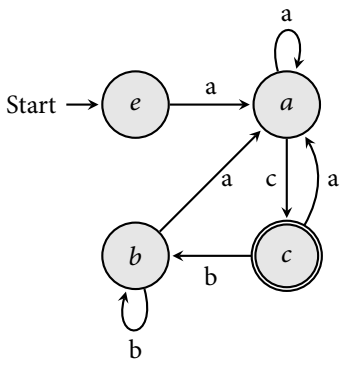
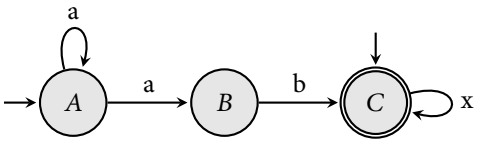


Abbildung 5.15: NEA mit mehreren Startzuständen.



Hinweis: Wenden Sie die Technik von Rabin/Scott auf die Startzustände an.

2. Verallgemeinern Sie Ihre Konstruktion, indem Sie angeben, wie ein NEA $M_1 = (Q, \Sigma, \delta, E, F)$ mit $|E| > 1$ in einen NEA $M_2 = (Q', \Sigma, \delta', E', F')$ mit $|E'| = 1$ überführt werden kann.

Der Automat wird wie in Abbildung 5.16 umgeschrieben, um mit einem einizgen Startzustand auszukommen.

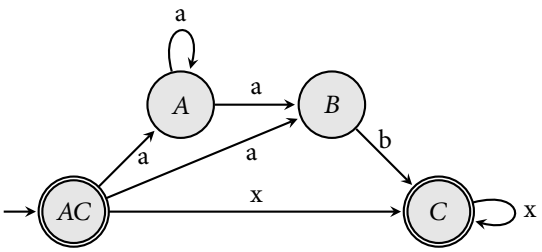


Abbildung 5.16: Ein zu Abbildung 5.15 äquivalenter NEA mit einem Startzustand.

Der neue Startzustand $E_{\text{neu}} = AC$, der aus der Menge der bisherigen Startzustände besteht, wird wie in der Konstruktion von Rabin/Scott als ein einziger Zustand interpretiert. Es werden folgende Übergangsfunktionen eingeführt:

$$\delta(E, \sigma) = q \; \forall \sigma \in \Sigma \; \forall q \in E : \delta(q, \sigma) \notin \perp$$

Interpretation: Jeder von einem bisherigen Anfangszustand ausgehende Übergang wird auch für den neuen, vereinigten Anfangszustand E_{neu} eingeführt. Nicht definierte Übergänge aus den Anfangszuständen heraus werden natürlich auch in der neuen Definition nicht eingeführt.

5.18 Aufgabe: Reguläre Ausdrücke

Geben Sie reguläre Ausdrücke für folgende Sprachen an:

1. Eine Liste mit Vornamen, die durch Kommas getrennt sind. Hinter dem letzten Namen der Liste soll *kein* Komma stehen. Ein Name setzt sich zusammen aus einem Großbuchstaben gefolgt von einer beliebigen Anzahl von Kleinbuchstaben.

2. Alle Wörter $\vec{\omega} \in \{a, b, c, \ldots, x, y, z\}^*$, deren Buchstaben alphabetisch sortiert sind, d.h. ω_n steht im Alphabet vor oder an gleicher Position wie ω_{n+1} $\forall n \in [1, |\omega| - 1]$. Die Eigenschaft trifft beispielsweise auf »*alt*« zu, aber nicht auf »*baby*«.

3. Alle Hexadezimalzahlen, die 32-Bit-Speicheradressen repräsentieren.

4. $L \equiv \{x \in \mathbb{N} \mid x \text{ ist Primzahl } \wedge x \leq 20\}$

Die beschriebenen Sprachen werden durch folgende regulären Ausdrücke erkannt:

1. $((\mathrm{A} - \mathrm{Z})(\mathrm{a} - \mathrm{z})^+,)^*(\mathrm{A} - \mathrm{Z})(\mathrm{a} - \mathrm{z})^+$

2. $\mathrm{a}^*\mathrm{b}^*\mathrm{c}^* \ldots \mathrm{x}^*\mathrm{y}^*\mathrm{z}^*$. *Achtung*: Die Lösung ist nicht korrekt, wenn + anstelle von $*$ verwendet wird, da das Wort in diesem Fall *jeden* Buchstaben des Alphabets mindestens einmal enthalten müsste.

3. $0\mathrm{x}(0 - 9 \mid \mathrm{A} - \mathrm{F})\{1, 8\}$

4. 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19. *Hinweis*: Die Menge aller Primzahlen bis zu einer bestimmten Maximalgröße kann als regulärer Ausdruck angegeben werden, da die Menge endlich ist. Es ist nicht möglich, alle Primzahlen *ohne* Obergrenze als regulären Ausdruck anzugeben.

5.19 Aufgabe: Grammatiken und DEAs

Gegeben sei eine reguläre Grammatik mit den Produktionsregeln

$$\begin{aligned}
\langle Q_0 \rangle &\rightarrow \mathtt{a}\langle Q_1 \rangle \mid \mathtt{a}\langle Q_2 \rangle \mid \mathtt{b}\langle Q_2 \rangle \mid \mathtt{a}\langle Q_3 \rangle \\
\langle Q_1 \rangle &\rightarrow \mathtt{a}\langle Q_1 \rangle \mid \mathtt{c}\langle Q_3 \rangle \\
\langle Q_2 \rangle &\rightarrow \mathtt{a}\langle Q_1 \rangle \mid \mathtt{b}\langle Q_3 \rangle \\
\langle Q_3 \rangle &\rightarrow \mathtt{b}\langle Q_0 \rangle \mid \mathtt{b}\langle Q_3 \rangle \mid \epsilon
\end{aligned}$$

Vorsicht: Die in der Vorlesung besprochene Konstruktion konvertiert zwischen reguläre Grammatiken und nicht-deterministischen endlichen Automaten; hier ist ein deterministischer Automat gesucht!

und Startsymbol $\langle Q_0 \rangle$. Geben Sie einen *deterministischen* endlichen Automaten mit partieller Übergangsfunktion an, der die Sprache akzeptiert, die von der Grammatik produziert wird.

Um die Grammatik in einen deterministischen endlichen Automaten umzuwandeln, geht man über den Zwischenschritt eines NEAs:

1. Grammatik in NEA umwandeln (direkte Konversion)

2. NEA in DEA umwandeln (Algorithmus von Rabin und Scott)

Die Vorgehensweise ist für jede reguläre Grammatik möglich. Für die gegebene Grammatik erhält man den in Abbildung 5.17 gezeigten Automaten.

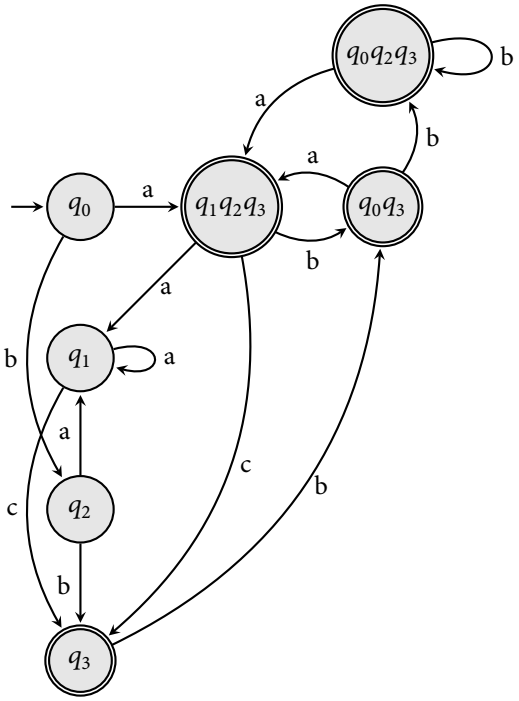


Abbildung 5.17: DEA für die Grammatik aus Aufgabe 5.19.

5.20 Aufgabe: Reguläre Sprachen

Wir betrachten Sprachen über dem Alphabet $\Sigma = \{a, b\}$.

1. Geben Sie reguläre Grammatiken für folgende Sprachen an:
   - $L_1 \equiv \{w \in \Sigma^+ \mid w \text{ beginnt mit } a \text{ und endet mit } b\}$
   - $L_2 \equiv \{w \in \Sigma^* \mid \text{Die Anzahl der Buchstaben »b« in } w \text{ ist ein Vielfaches von 3}\}$
2. Verallgemeinern Sie die Sprache L_2 so, dass die Anzahl der Buchstaben »b« im Wort ein Vielfaches von $k, k \in \mathbb{N}$ betragen muss. Geben Sie eine vollständige formale Definition der zugehörigen Grammatik an.

1. Die Sprache L_1 ist gegeben durch die Produktionen

$$\langle S\rangle \to \mathtt{a}\langle B\rangle$$
$$\langle B\rangle \to \mathtt{b}\langle B\rangle \mid \mathtt{a}\langle B\rangle \mid \mathtt{b}$$

   L_2 verwendet die Produktionen

$$\langle S\rangle \to \mathtt{a}\langle S\rangle \mid \mathtt{b}\langle N\rangle_1 \mid \epsilon$$
$$\langle N\rangle_1 \to \mathtt{b}\langle N\rangle_2 \mid \mathtt{a}\langle N\rangle_1$$
$$\langle N\rangle_2 \to \mathtt{b}\langle S\rangle \mid \mathtt{a}\langle N\rangle_2$$

   Sobald das erste Zeichen »b« produziert wurde, muss der komplette Zyklus $\langle N\rangle_1, \langle N\rangle_2, \langle S\rangle$ durchlaufen werden, bevor die Produktion gestoppt werden kann. Der Zyklus enthält das Zeichen »b« sicher dreimal. Damit ist garantiert ist, dass $\#_{\mathtt{b}}(w) \equiv 0 \mod 3$ gilt (der Zyklus kann beliebig oft durchlaufen werden).

2. Die Verallgemeinerung auf ein Vielfaches von k erfolgt, indem der Zyklus verlängert wird, der durchlaufen werden muss, um nach Erzeugung des ersten »b« zu einem fertigen Wort zu gelangen:

$$\langle S\rangle \to \mathtt{a}\langle S\rangle \mid \mathtt{b}\langle N\rangle_1 \mid \epsilon$$
$$\langle N\rangle_i \to \mathtt{b}\langle N\rangle_{i+1} \mid \mathtt{a}\langle N\rangle_i \; \forall i = 1 \ldots k-2$$
$$\langle N\rangle_{k-1} \to \mathtt{b}\langle S\rangle \mid \mathtt{a}\langle N\rangle_{k-1}$$

5.21 *Aufgabe: NEAs und DEAs*

Wandeln Sie den in Abbildung 5.18 gezeigten nicht-deterministischen endlichen Automaten

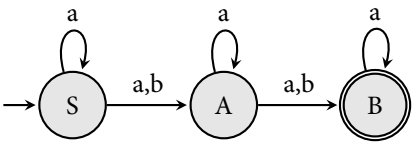


Abbildung 5.18: NEA, der in einen DEA umgewandelt werden soll.

in einen äquivalenten deterministischen endlichen Automaten um.

~ ~ ~

Die Umwandlung erfolgt mit dem Algorithmus von Rabin und Scott und führt zum in Abbildung 5.19 gezeigten Ergebnis.

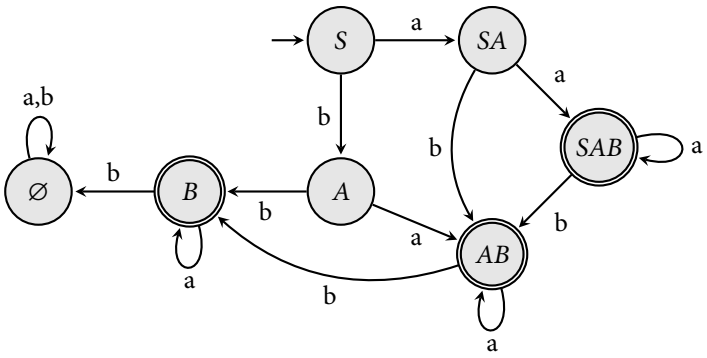


Abbildung 5.19: Zum NEA aus Aufgabe 5.21 äquivalenter DEA.

5.22 *Aufgabe: Kombinationen regulärer Sprachen*

Reguläre Sprachen, die geeignet miteinander verbunden werden, können oftmals wiederum durch eine reguläre Grammatik beschrieben werden.

1. Gegeben sei ein Alphabet Σ und eine Operation $\square_1 : \Sigma^* \times \Sigma^* \to \Sigma^*$, die für $v, w \in \Sigma^*$ und $a, b \in \Sigma$ definiert ist durch

$$\epsilon \square_1 w = w$$
$$v \square_1 \epsilon = v$$
$$av \square_1 bw = ab(v \square_1 w)$$

   Sei $\Sigma = \{0, 1\}$. Berechnen Sie schrittweise $01\square_1 10$, $01\square_1 1010$ und $0000\square_1 11$.

2. Seien $L_1, L_2 \in \mathrm{Reg}_\Sigma$ reguläre Sprachen, und sei $\square_2 : \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ definiert durch

$$L_1 \square_2 L_2 \equiv \{v \square_1 w \mid v \in L_1, w \in L_2\}$$

Beweisen Sie, dass $L_1 \square_2 L_2$ eine reguläre Sprache ist. *Hinweis:* Konstruieren Sie einen Automaten, der geeignet zwischen den Ausgangssprachen alterniert.

1. Die Rechnung verläuft in folgenden Schritten (*Hinweis*: Achten Sie darauf, die Gleichheitszeichen korrekt zu setzen, keine Teile der Rechnung im Verlauf zu »verlieren«, und nur die tatsächlich angegebenen Regeln zu verwenden):

$$\begin{aligned} 01\square 10 &= 01(1\square 0) = 0110 \\ 01\square 1010 &= 01(1\square 010) = 0110(\epsilon\square 10) = 011010 \\ 0000\square 11 &= 01(000\square 1) = 0101(00\square\epsilon) = 010100 \end{aligned}$$

2. Nachdem die Sprache durch

$$L\{u\square v \mid u \in L_1, v \in L_2\}$$

ist und es gilt, dass $L_1, L_2 \in \mathrm{Reg}_\Sigma$, gibt es endliche Automaten A_1, A_2 mit $\mathcal{L}(A_1) = L_1$ und $\mathcal{L}(A_2) = L_2$, die als Bauteile zur Erkennung der »Gesamtsprache« L verwendet werden können. Die Automaten sind durch folgende Komponenten gegeben:

$$\begin{aligned} A_1 &= (\Sigma, Q_1, \delta_1, q_0^1, F_1) \\ A_2 &= (\Sigma, Q_2, \delta_2, q_0^2, F_2) \end{aligned}$$

Definiere A' mit *einem* einzigen Endzustand, der nach Lesen von ϵ aus den bisherigen Endzuständen erreicht werden kann:

$$\begin{aligned} \Sigma' &= \Sigma \\ Q_i' &= Q_i \cup \{q_i^\mathrm{f}\} \text{ (neuer Endzustand)} \\ F_i' &= \{q_i^\mathrm{f}\} \\ \delta_i' &= \delta_i \cup \delta(q, \epsilon) = q_i^\mathrm{f}\ \forall q \in F_i \end{aligned}$$

Definiere eine »Produktmaschine« $X : (\Sigma, Q, \delta, q_0, F)$, die abwechselnd Zeichen aus $x \in L_1$ und $y \in L_2$ gemäß der Operationen $\square$ liest.

$$Q : (q_{A_1}, q_{A_2}, \{\{1, 2\}\})$$

Die Menge $\{1, 2\}$ gibt an, welche Maschine das nächste Zeichen liest. Dies muss explizit festgehalten werden, da die Frage nicht implizit aus dem Zustand zu beantworten ist. Die Übergangsfunktion ist wie folgt definiert:

$$\delta((q_1, q_2, 1), a) = \begin{cases} (\delta(q_1, a), q_2, 2) & \text{für } q_2 \neq q_2^\mathrm{f} \\ (\delta(q_1, a), q_2, 1) & \text{für } q_2 = q_2^\mathrm{f} \end{cases}$$

Wenn das Wort aus L_2 noch nicht fertig gelesen wurde, alterniert die Produktmaschine zwischen 1 und 2, ansonsten bleibt sie bei 1.

Analoge Definition für Maschine 2:

$$\delta((q_1, q_2, 2), a) = \begin{cases} (q_1, \delta_2(q_2, a), 1) & \text{für } q_1 \neq q_1^{\mathrm{f}} \\ (q_1, \delta_2(q_2, a), 2) & \text{für } q_1 = q_1^{\mathrm{f}} \end{cases}$$

5.23 *Aufgabe: Regulär oder nicht?*

Gegeben seien die Sprachen

$$L_1 \equiv \{a^m b^n \mid m, n \in \mathbb{N}, m + n \leq 42\}$$
$$L_2 \equiv \{a^m b^n \mid m, n \in \mathbb{N}, m - n \leq 42\}$$
$$L_3 \equiv \{a^m b^n \mid m, n \in \mathbb{N}, m - n = 0\}$$

Zeigen oder widerlegen Sie, dass die Sprachen regulär sind.

~ ~ ~

Typischerweise ist es leichter, die Regularität einer Sprache zu zeigen als das Gegenteil – es reicht im ersten Fall, eine spezifische reguläre Grammatik oder einen spezifischen regulären Ausdruck anzugeben, der beweist, dass die Sprache regulär ist. Betrachten wir die einzelnen Sprachen genauer:

1. L_1 ist endlich (es gibt nur endlich viele natürliche Zahlen, die in Summe 42 ergeben), also regulär.

2. Sei $k \in \mathbb{N}$ die Pumping-Lemma-Konstante. Wähle das Wort $z = a^k b^k$ mit $|z| = 2k \geq k$. Die Bedingung des Pumping-Lemmas ist damit erfüllt. Es gilt außerdem $k - k = 0 \leq 42$, das Wort ist daher Mitglied der Sprache L_2.

   Das Wort kann entsprechend zerlegt werden in $z = uvw$. Da $|\, uv \,| \leqslant k$, folgt $u, v \in a^*$. Entsprechend hat z die Form $z = a^p a^q a^r b^k$ mit

   $$\begin{aligned} p + q + r &= k \\ q &\geq 1 \\ p + q &\leq k. \end{aligned}$$

   Aufgrund des Pumping-Lemmas gilt $uv^N w \in L$ für jedes $N \in \mathbb{N}$, woraus folgt

   $$\begin{aligned} z' &= a^P a^{Nq} a^r b^k \\ &= \underbrace{a^p a^q a^r}_{=a^k} a^{(N-1)q} b^k \\ &= a^k a^{(N-1)q} b^k \end{aligned}$$

   Es muss gelten: $k + (N-1)q - k \leq 42 \Leftrightarrow (N-1)q \leq 42$.

   Schlechtester Fall: $q = 1 \Rightarrow (N-1) \leq 42$. Wähle $N = 44 \Rightarrow (44 - 1) = 43 \leq 42$. Dies ist offenbar ein Widerspruch: Die Sprache ist also nicht regulär.

3. Die Sprache entspricht $\{a^n b^n \mid n \in \mathbb{N}\}$. In der Vorlesung wurde ausführlich besprochen, warum die Sprache nicht regulär ist. Wesentlich für die Übung ist die Erkenntnis, dass die Sprache L_3 in gezeigter Form dargestellt werden kann!

5.24 *Aufgabe: Kombination nicht-regulärer Sprachen*

Gegeben seien zwei *nicht*-reguläre Sprachen L_1 und L_2.

1. Ist die Sprache $L_1 - L_2$ ebenfalls nicht regulär?
2. Ist die Sprache $L_1 \cup L_2$ ebenfalls nicht regulär?

Begründen Sie jeweils Ihre Antworten.

1. Wir betrachten den Sonderfall $L_1 = L_2$, d.h. die beiden Sprachen sind identisch. Dann gilt $L_1 - L_2 = \varnothing$. Die leere Menge ist eine reguläre Sprache, die Differenz zwischen zwei nicht-regulären Sprachen muss also *nicht* nicht-regulär sein.

   Die Beschränkung auf einen Sonderfall ist in diesem Fall erlaubt, da er die Aussage widerlegt. Die Argumentation wäre aber nicht korrekt, wenn man beispielsweise zeigen möchte, dass die Differenz immer regulär ist.

2. Sei $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ und $L_2 = \{a^k b^l \mid k \neq l\}$. Es gilt

$$L_1 \cup L_2 = \{a^k b^m \mid k, m \in \mathbb{N}\}$$

   Nachdem k und m unabhängig voneinander sind, kann die Menge durch den regulären Ausdruck $a^+ b^+$ beschrieben werden. $L_1 \cup L_2$ ist somit regulär, womit gezeigt ist, dass die Vereinigung zweier Sprachen im Allgemeinen *nicht* nicht-regulär ist.

5.25 *Aufgabe: Induktion auf regulären Sprachen*

Zeigen Sie per vollständiger Induktion über die Länge der erzeugten Wörter, dass die Sprache

$$L \equiv \{vw \mid v, w \in \{a, b\}^*, \#_a(v) = \#_b(w)\}$$

regulär ist, wobei $\#_\zeta(\vec{x})$ die Anzahl des Buchstabens ζ im Wort $\vec{x}$ angibt.

Hinweis: Zeigen Sie in Ihrem Beweis, dass für jedes Wort $\vec{x} \in \{a, b\}^*$ auch $x \in L$ gilt.

Wir verwenden folgende Strategie: Zunächst zeigen wir, dass jedes Wort $x \in \{a, b\}^*$ in der Form der Sprache L dargestellt werden kann. Da $\{a, b\}^*$ ein regulärer Ausdruck ist, ist die Sprache dann offenbar regulär. Der Beweis erfolgt durch vollständige Induktion nach Länge des Wortes.

- Induktionsanfang: $|z| = |xy| = 0$. Also gilt $z = \epsilon\epsilon$, und folglich $\#_\mathrm{a}(x) = \#_\mathrm{b}(y) = 0$ ✓
- Induktionsschritt: Das Wort z' mit Länge $|z'| = n + 1$ kann die Form $z' = za$ oder $z' = zb$ haben. Es gilt $|z| = n, z = xy$ mit $\#_\mathrm{a}(x) = \#_\mathrm{b}(y)$ nach Induktionsvoraussetzung. Folgende Fälle sind nun zu unterscheiden:

1. $z' = za = xya \Rightarrow \#_b(ya) = \#_b(y) \overset{\text{IV}}{=} \#_a(x)$.
2. Wenn gilt $z' = zb = xyb$, sind zwei weitere Unterfälle zu betrachten:
   - Sei $y = \epsilon$: $\#_b(y) = 0$. $\#_a(x)$ muss daher 0 sein, da $\#_a(x) = \#_b(y)$. Folglich gilt $\#_a(xyb) = \#_b(\epsilon) = 0$. Daher:

     $$x' = xyb$$
     $$y' = \epsilon \checkmark$$

   - Wenn $y \neq \epsilon \Rightarrow z' = xyb$.
     Fall 1: $y = a\bar{y} \Rightarrow z' = \underbrace{xa}_{x'}\,\underbrace{\bar{y}b}_{y'}$.

     $$\#_a(x) = \#_b(a\bar{y})$$
     $$\#_b(\bar{y}b) = \#_b(a\bar{y}b) = \#_b(a\bar{y}) + 1 = \#_b(y) + 1$$
     $$\#_a(xa) = \#_a(x) + 1$$

     Also gilt $x' = xa$, $y' = \bar{y}b$ ✓
     Fall 2:

     $$y = b\bar{y} \Rightarrow z' = xyb = \underbrace{xb}_{x'}\,\underbrace{\bar{y}b}_{y'} = x'y'$$

     Es gilt $\#_a(x) = \#_b(y)$ nach IV. Daher:

     $$\#_a(x') = \#_a(xb) = \#_a(x)$$
     $$\#_b(y') = \#_b(\bar{y}b) = \underbrace{\#_b(\bar{y})}_{\#_b(y)-1} + 1$$
     $$= \#_b(y) - 1 + 1 = \#_b(y) = \#_a(x)$$

     $\Rightarrow \#_a(x') = \#_b(y')$ ✓ □

5.26 *Aufgabe: Satz von Myhill und Nerode*

Verwenden Sie den Satz von Myhill und Nerode, um zu zeigen, dass folgende Sprachen regulär oder nicht-regulär sind:

1. $L_1 \equiv \{a^n b^n \mid n \in \mathbb{N} \cup 0\}$
2. $L_2 \equiv \{w \in \{0,1\}^* \mid$ Anzahl von »1« in w ist ohne Rest durch 3 teilbar $\}$
3. $L_3 \equiv \{abc, def, ghi\}$

Die Äquivalenzklassen werden durch folgende Überlegungen identifiziert:

1. Sprache L_1:

Repräsentant	Vervollständigt durch Suffix
[a]	b, abb, aabbb, …
[ϵ]	ab, aabb, …
[aa]	bb, abbb, …
[aaa]	bbb, abbbb, …
…	…
[a^i]	b^i, ab^{i+1}, …

☞ Es gibt ∞ viele Äquivalenzklassen, die Sprache ist als *nicht* regulär.

2. Betrachte Sprache L_2:

Repräsentant	Enthaltene Wörter
$[\epsilon]$	0, 00, 00, 000, 1110, 0111, …($\#1 \equiv 0 \mod 3$)
[1]	$\#1 \equiv 1 \mod 3$
[2]	$\#1 \equiv 2 \mod 3$

   Folglich gilt $\text{Index}(L_2) = 3$, die Sprache ist also *regulär*.

3. Betrachte Sprache L_3. Es gibt die Äquivalenzklassen [a], [ab], [abc], [d], [de], [def], [g], [gh], [ghi]. Der Index ist endlich, L_3 damit *regulär*.

5.27 Aufgabe: Minimalautomat

Betrachten Sie den nicht-minimierten DEA in Abbildung 5.20.

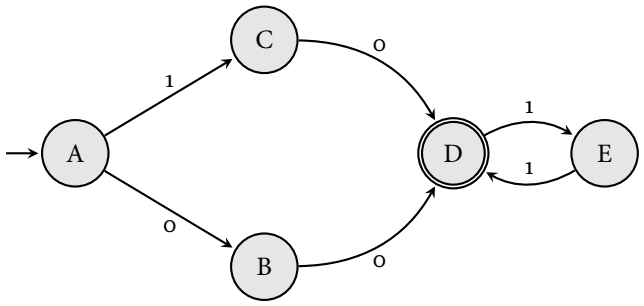


Abbildung 5.20: DEA mit mehr Zuständen als unbedingt notwendig.

1. Konstruieren Sie den dazugehörigen Minimalautomaten, indem Sie die Table-Filling-Konstruktion explizit ausführen und den Graphen des resultierenden Automaten zeichnen. *Hinweis:* Vervollständigen Sie den Automaten zuvor geeignet.

2. Geben Sie die erzeugte Sprache an, und berechnen Sie die Äquivalenzklassen bezüglich der Relation R_L. Welche Äquivalenzklasse gehört zu welchem Zustand des Minimalautomaten?

Zur Vervollständigung des Automaten werden für alle nicht-definierten Übergänge die Regeln $\delta(x, \sigma) = \perp$ und $\delta(\perp, \sigma) = \perp \; \forall \, \sigma \in \Sigma$ eingeführt (siehe Aufgabe 5.13). Dies ist notwendig, da der *Table Filling*-Algorithms nur mit vollständig definierten Übergangsfunktionen (im Gegensatz zu partiell definierten Übergangsfunktionen) korrekt arbeitet. Der resultierende Automat ist durch den in Abbildung 5.21 gezeigten Graphen gegeben.

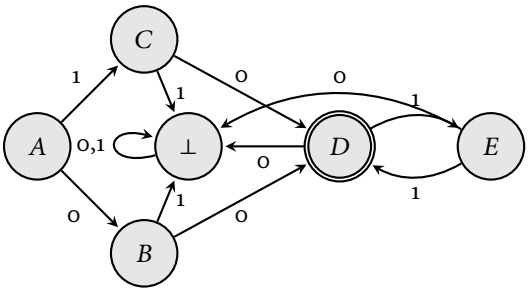


Abbildung 5.21: Vervollständigter Automat für den DEA aus Abbildung 5.20.

Um den TF-Algorithmus anzuwenden, geben wir in einem Dreiecksschema alle Kombinationen aus Zuständen an. Kombinationen identischer Zustände wie (A, A) und doppeltes Auftreten von Kombinationen, beispielsweise (A, B) und (B, A), die nicht notwendig respektive redundant sind, werden durch die Vorgehensweise vermieden. Das Resultat findet sich in Tabelle 5.3.

Tabelle 5.3: Dreiecksschema für Table-Filling-Algorithmus.

B	1				
C	1				
D	x	x	x		
E	1	1	1	x	
⊥	2	1	1	x	1
	A	B	C	D	E

Die dabei verwendete Symbole sind:

- Eintrag initial befüllt: x
- Befüllt im ersten Schritt: 1
- Befüllt im zweiten Schritt: 2

Es existieren folgende Zustandskombinationen, die bei der Betrachtung von Übergängen berücksichtigt werden müssen, wie in Tabelle 5.4 dargestellt.

AB	AC	AD	AE	A⊥
BC	BD	BE	B⊥	
CD	CE	C⊥		
DE	D⊥			
E⊥				

Tabelle 5.4: Zustandskombinationen für den TF-Algorithmus.

Wir betrachten nun alle möglichen Übergänge der Transitionsfunktion. Nachdem es die Buchstaben 0 und 1 im Alphabet gibt, müssen zwei Möglichkeiten betrachtet werden:

- Nach Übergang »0« verändern sich die Zustände wie in Tabelle 5.5 gezeigt.
- Nach Übergang »1« verändern sich die Zustände wie in Tabelle 5.6 gezeigt.

BD	BD	B⊥	B⊥	B⊥
DD	D⊥	D⊥	D⊥	
D⊥	D⊥	D⊥		
⊥⊥	⊥⊥			
⊥⊥				

Tabelle 5.5: Zustandskombinationen für den TF-Algorithmus nach Übergang »0«.

C⊥	C⊥	CE	CD	C⊥
⊥⊥	⊥E	⊥D	⊥⊥	
⊥E	⊥D	⊥⊥		
ED	E⊥			
D⊥				

Tabelle 5.6: Zustandskombinationen für den TF-Algorithmus nach Übergang »1«.

Achtung: Es reicht im Allgemeinen nicht, nur einen Lauf des Algorithmus durchzuführen. Beispielsweise sind für den gegebenen Automaten zwei Läufe erforderlich, um zu einem konvergenten Ergebnis zu gelangen.

Folgende Vorgehensweise wird zum Befüllen des Dreieckschemas verwenden: Prüfen, welche Kombinationen aus den Tabellen für die Übergänge bereits markiert sind. Entsprechende Paare im Dreiecksschema markieren. So lange wiederholen, bis sich Tabelle nicht mehr ändert. Das Paar (B, C) bleibt unmarkiert und kann verschmolzen werden, was zu dem in Abbildung 5.22 gezeigten minimierten Automaten führt.

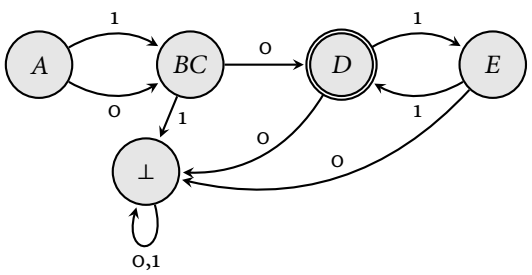


Abbildung 5.22: Minimierter Automat für den DEA aus Abbildung 5.21.

Die erzeugte Sprache hat eine »0« an zweiter Stelle und $2n$ (also 0, 2, 4, …) Einsen am Schluss.

Äquivalenzklassen und die korrespondierenden Zustände in obigem Graph:

1. $[\epsilon] \hat{=} \text{A}$
2. $[01] \rightarrow \text{Reject} \hat{=} \perp$
3. $[1] = [0] \hat{=} \text{BC}$
4. $[0011] \hat{=} \text{D}$
5. $[001] \hat{=} \text{E}$

5.28 *Aufgabe: Pumping-Lemma*

Zeigen Sie mit Hilfe des Pumping-Lemmas, dass folgende Sprachen *nicht* regulär sind:

- $L_1 = \{ww^R \mid w \in \Sigma^*\}$, wobei $w^R = (a_1 a_2 \dots a_n)^R \equiv (a_n a_{n-1} \dots a_2 a_1)$ die Umkehrung einer Zeichkette angibt und $|\Sigma| \geq 2$ gelten soll.
- $L_2 = \{0^n 1^m 0^n \mid n, m \in \mathbb{N}\}$

1. Wie so oft sei sei n die Pumping-Lemma Konstante. Wähle $w = a_1 a_2^n a_1$, wobei $a_1, a_2 \in \Sigma$, $a_1 \neq a_2$. Damit gilt $w^{\text{R}} = w$, das Wort ist also ein Palindrom.

   $\vec{w} = ww^{\text{R}} = ww \stackrel{!}{=} xyz$, da $|\vec{w}| = 2n+4 \geq n$. Gemäß des Pumping-Lemmas gilt $|xy| \leq n, |y| \geq 1$. $\Rightarrow xy = a_1 a_2^k$ mit $0 \leq k \leq n-1$.

   Es gibt zwei Möglichkeiten für x:

   $$x = \begin{cases} \epsilon \\ a_1 a_2^{k-|y|} \end{cases}$$

   Da $xyz \in L \Rightarrow xz \in L$ nach PL. Das Wort ist wie folgt zusammengesetzt:

   $$xyz = \underbrace{\overbrace{xy}^{a_1 a_2^k} a_2^{n-k} a_1}_{w} \underbrace{a_1 a_2^n a_1}_{w}$$

   Damit ergibt sich folgende Zusammensetzung für das verkürzte Wort xz:

   $$\Rightarrow xz = \begin{cases} \epsilon a_2^{n-k} a_1 a_1 a_2^n a_1 \Rightarrow \#(a_1) = 3 \Rightarrow xz = ww \text{ unmöglich.} \\ a_1 a_2^{k-|y|} a_2^{n-k} a_1 a_1 a_2^n a_1 \end{cases}$$

   Den unteren Fall kann man weiter auflösen:

   $$\begin{aligned} a_1 a_2^{k-|y|} a_2^{n-k} a_1 a_1 a_2^n a_1 &= a_1 a_2^{k-|y|+n-k} a_1 a_1 a_2^n a_1 \\ &= a_1 a_2^{n-|y|} a_1 a_1 a_2^n a_1 \\ &= a_2^{n-|y|} \neq a_2^n, \text{ da } |y| \geq 1 \end{aligned}$$

   Daher gilt in beiden Fällen $xz \notin L$, also Widerspruch! Entsprehend ist L nicht regulär.

2. Sei k die Pumping-Lemma-Konstante. Wähle $w = 0^k10^k$, $|w| = 2k + 1 \geq k$. $w = yxz$, $|xy| \leq k$, $|y| \geq 1$. xy besteht also nur aus Nullen, d.h.$xy = 0^p0^q$ mit $p + q \leq k$.

   Das Gesamtwort ist damit gegeben durch $xyz = 0^p0^q0^r10^k$ mit den Randbedingungen $p + q + r = k$. Das Pumping-Lemma garantiert, dass $xyz \in L \Rightarrow xz \in L$. Das verkürzte Wort besitzt die Form $xz = 0^p0^r10^k$ mit $p + r = k - q$. Daraus folgt $p + r < k$, da $q > 1$. Ergo $xz = 0^l10^k$ mit $l < k$, also $xz \notin L$. Widerspruch! ☞ L ist nicht regulär.

5.29 *Aufgabe: Reguläre Komplementärsprachen*

Sei R ein regulärer Ausdruck für eine Sprache L. Geben Sie mittels Pseudocode einen Algorithmus zur Konstruktion eines regulären Ausdrucks für die Komplementärsprache an. Nehmen Sie dabei an, dass Funktionen zur Konvertierung der bekannten äquivalenten Formulierungen regulärer Sprachen zur Verfügung stehen.

Als Grundidee der Konstruktion führen wir die Konvertierungskette

$$\text{RegExp} \rightarrow \text{NEA} \rightarrow \text{DEA}$$

durch, ersetzen akzeptierende und nicht akzeptierende Endzustände mittels $F' = Q - F$ (Automat mit invertierten Akzeptanz-Zuständen erkennt Komplementsprache), und konvertieren den DEA in einen RegExp. Dies führt zu folgendem Algorithmus in Pseudocode (die Prozeduren Nea2Dea, Dea2Rxp und Rxp2Nea stehen gemäß Angabe zur Verfügung und brauchen nicht definiert zu werden):

```
1: procedure RxpInvert(r)
2:     (Q, Σ, δ, E, F) = Rxp2Nea(r)
3:     (Q', Σ, δ', q0, F') = Nea2Dea(Q, Σ, δ, E, F)

4:     F'' ← Q' − F'                          ▷ Invertieren der Endzustände
5:     r' = Dea2Rxp(Q', Σ, δ', q0, F'')

6:     return r'
7: end procedure
```

5.30 *Aufgabe: Chomsky-Normalform*

Gegeben sei eine Grammatik $G = (V, \Sigma, P, S)$ mit $V = \{S, T\}$, $\Sigma = \{a, b\}$ und den Regeln

$$\begin{aligned}
\langle S\rangle &\rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{aa} \mid \texttt{bb} \\
\langle S\rangle &\rightarrow \texttt{a}\langle T\rangle\texttt{a} \mid \texttt{b}\langle T\rangle\texttt{b} \\
\langle T\rangle &\rightarrow \texttt{a}\langle T\rangle \mid \texttt{b}\langle T\rangle \mid \texttt{a} \mid \texttt{b}
\end{aligned}$$

Konstruieren Sie eine äquivalente Grammatik G' in Chomsky-Normalform. Geben Sie explizit alle Konvertierungsschritte mit an, um die Transformation zu veranschaulichen.

Zur Konvertierung der Grammatik in die Chomsky-Normalform werden die fünf Schritte des in der Vorlesung diskutierten Algorithmus durchlaufen:

Bei einfachen Grammatiken kann es auch einfachere bzw. direktere Wege geben, um die CNF zu konstruieren. Der ausgeführte Algorithmus garantiert allerdings, dass die Konvertierung für beliebig komplizierte Grammatiken funktioniert und prinzipiell auch maschinell durchgeführt werden kann.

1. Zyklen entfernen: Nicht notwendig, da keine vorhanden sind.
2. Sortieren: Trivialerweise wählt man die Ordnung $(\langle S\rangle, \langle T\rangle) \to (\langle A\rangle_0, \langle A\rangle_1)$
3. Einzelregeln eliminieren: Nicht notwendig, da keine entsprechenden Regeln vorhanden
4. Terminalsymbole ersetzen: Wir führen die Regeln

   $$\langle A\rangle \to \mathtt{a}, \langle B\rangle \to \mathtt{b}$$

   ein, um Möglichkeiten zur Erzeugung der Terminalsymbole gemäß CNF zu haben. Die Regeln $\langle A\rangle_1 \to \mathtt{a}, \langle A\rangle_1 \to \mathtt{b}, \langle A\rangle_0 \to \mathtt{a}, \langle A\rangle_0 \to \mathtt{b}$ werden beibehalten. Allerdings können diese Regeln *nicht* an anderen Stellen der Grammatik zur Erzeugung von Nicht-Terminalen eingesetzt werden, da dies möglicherweise die erzeugte Sprache verändert. Die Grammatik hat damit die Form

   $$\begin{aligned}\langle A\rangle_0 &\to \mathtt{a} \mid \mathtt{b} \mid \langle A\rangle\langle A\rangle \mid \langle B\rangle B\\ \langle A\rangle_0 &\to \langle A\rangle\langle A\rangle_1\langle A\rangle \mid \langle B\rangle\langle A\rangle_1\langle B\rangle\\ \langle A\rangle_1 &\to \langle A\rangle\langle A\rangle_1 \mid \langle B\rangle\langle A\rangle_1 \mid \mathtt{a} \mid \mathtt{b}\end{aligned}$$

5. Die endgültige Transformation in CNF erfolgt, indem alle Regeln mit $|r| > 2$ durch eine Kette kürzerer Regeln ersetzt werden. Die Regel $\langle A\rangle_0 \to \langle B\rangle\langle A\rangle_1\langle B\rangle$ wird transformiert in

   $$\begin{aligned}\langle A\rangle_0 &\to \langle B\rangle\langle C\rangle_1\\ \langle C\rangle_1 &\to \langle A\rangle_1\langle B\rangle\end{aligned}$$

   Die Regeln $\langle A\rangle_0 \to \langle A\rangle\langle A\rangle_1\langle A\rangle$ wird nach dem gleichen Schema umgewandelt zu

   $$\begin{aligned}\langle A\rangle_0 &\to \langle A\rangle_1\langle A\rangle\langle C\rangle_2\\ \langle C\rangle_2 &\to \langle A\rangle_1\langle A\rangle\end{aligned}$$

5.31 Aufgabe: Pumping-Lemma

Zeigen Sie mit Hilfe des Pumping-Lemmas, dass folgende Sprachen nicht kontextfrei sind:

1. $L_1 \equiv \{a^n | n = m^2, m \in \mathbb{N}\}$
2. $L_2 \equiv \{ww | w \in \{0,1\}^*\}$

1. Wie immer bei PL-Beweise gilt zunächst die Annahme, dass L_1 kontextfrei ist. ☞ $\vec{x} \in L$ kann für $|\vec{x}| \geqslant n$ (n ist die Pumping-Konstante) in der Form $\vec{x} = uvwxyz$ geschrieben werden.

Wähle $\vec{x} = a^{n^2}$. Das Wort ist in der Sprache enthalten und erfüllt die Längenanforderung. Entsprechend ist auch das aufgepumpte Wort uv^2wx^2y in der Sprache L enthalten. Man folgert

$$n^2 = |uvwxy| < |uv^2wx^2y|,$$

da $|vx| \geq 1 \Rightarrow$ entweder v oder w nicht-leer, eine Verdopplung des Mittelteils führt zu einer echten Wortverlängerung. Darauf aufbauend rechnet man

$$\begin{aligned} n^2 = |uvwxy| &< |uv^2wx^2y| \\ &= |v^2x^2uwy| = \underbrace{|v^2x^2|}_{=2|vx|\leq 2|vwx|\leq 2n} + |uwy| \\ &\leq 2n + \underbrace{|uwy|}_{<n^2} \\ &< n^2 + 2n < n^2 + 2n + 1 = (n+1)^2 \\ \Rightarrow n^2 < |uv^2wx^2y| &< (n+1)^2 \end{aligned}$$

☞ Die Länge des Wortes liegt echt zwischen zwei aufeinanderfolgenden Quadratzahlen. ☞ $uv^2wx^2y \notin L$ ☞ Widerspruch, die Sprache ist nicht kontextfrei.

Hinweis: In der Vorlesung wurde eine alternative Lösung betrachtet, die etwas anschaulicher argumentiert.

2. Sei n die PL-Konstante. Betrachte das Wort $\vec{x} = 0^n1^n0^n1^n$. Es gilt $\vec{x} \in L_2$, da $\vec{x} = ww$ mit $w = 0^n1^n$. Die Bedingung des Pumping-Lemmas ist erfüllt, da $|\vec{x}| = 4n > n$. ☞ Aufteilen in $\vec{x} = uvwxy$ mit $|vwx| \leq n, |vx| \geq 1$ (alternative Schreibweise für letzte Aussage: vx $\neq \epsilon$).

   Pumping-Lemma ☞ $uv^0wx^0y = uwy \in L_2$. Es gilt $|uwy| \geq 3n$, wenn $|uwx| \leq n$.

   Wenn $uwy = w'w'$ (nach Voraussetzungen), dann gilt$|uvy| \geq 3n \Rightarrow |w'| \geqslant \frac{3n}{2}$. Wir betrachten nun folgende Fälle, wie sich vwx im Wort befinden kann:

   (a) vwx liegt innerhalb des ersten Nuller-Blocks von $\vec{x}$
   ☞ v, w und x bestehen nur aus Nullen. Damit gilt folgendes:

   - $|vx| = k, k > 0, k \leq n$
   - uwy beginnt mit $n - k$ Nullen: ($uwy = 0^{n-k}1^n \dots$)
   - $|uwy| = 4n - |vx| = 4n - k$
   - $|w'|\frac{4n-k}{2} = 2n - \frac{k}{2}$
   - $uwy = w'w' = 0 \dots 01 \dots 10 \dots 01 \dots 1$

   Da $|0^{n-k}1^n| = 2n - k$, aber $|w'| = 2n - \frac{k}{2}$ ☞ w' endet mit Null ☞ $w'w'$ endet mit Null. Aber: $uwy \in L$ endet mit 1 ☞ Widerspruch ☞ $uwy \neq w'w'$.

   (b) vwx reicht in den ersten Einser-Block:

$$\begin{aligned} \vec{x} &= \overbrace{0\cdots0}^{n}\overbrace{1\cdots1}^{n}\overbrace{0\cdots0}^{n}\overbrace{1\cdots1}^{n} \\ &= 0\cdots\underbrace{0\cdots01\cdots1}_{vxw}\cdots1\overbrace{0\cdots0}^{n}\overbrace{1\cdots1}^{n} \end{aligned}$$

   Folgende Fälle müssen unterschieden werden:

   - $x = \epsilon \Rightarrow vx$ kann nur aus Nullen bestehen. ☞ Argumentation wie im Fall a)

- Wenn vx mindestens eine Eins enthält
  ☞ w' endet mit 1^n, da uwy mit 1^nendet. Aber: Kein passender Block außer des letzten vorhanden ☞ w' kann nicht mehrfach in uwy vorkommen.

(c) vwx ist im ersten Einser-Block enthalten ⇒ Argumentation entsprechend Fall b)

(d) vwx reicht vom ersten Einser-Block bis in den zweiten Nullen-Block. Folgende Fälle sind zu unterscheiden:

- vx enthält keine Nullen ☞ Argumentation analog zu vwx im ersten Einser-Block.
- vx enthält mindestens eine Null ☞ uwy beginnt mit Block von n Nullen, w' beginnt mit Block von n Nullen, wenn $uvw = w'w'$ Aber: Kein zweiter Block von Nullen in uvw enthalten, der für die zweite w'-Kopie notwendig ist. ☞ $\vec{x} \notin L$

(e) vwx befindet sich in zweiter Hälfte von $\vec{x}$: Symmetrische Argumentation wie bei erster Hälfte.

5.32 *Aufgabe: Pathologische Grammatik*

Betrachten Sie die Grammatik $G = (V, \Sigma, P, S)$ mit $\Sigma = \{\zeta, \lambda, \theta, \Gamma\}$, $V = \{\xi_i\}_{i\in[0,N]}$, $S = \xi_0$ und den Produktionen $\xi_i \to \xi_{(i+1) \mod N}$ für $0 \leq i < N$.

1. Warum ist die Grammatik nicht regulär?
2. Was ist die erzeugte Sprache $\mathcal{L}(G)$? Gilt $\epsilon \in \mathcal{L}(G)$? (bitte beides begründen)
3. Geben Sie eine kontextfreie Grammatik G' in Chomsky-Normalform an, so dass $\mathcal{L}(G) = \mathcal{L}(G')$ gilt.

1. Schreibt man die allgemein definierten Regeln explizit aus, erkannt man, dass sie zyklisch sind:

$$\begin{aligned} \xi_0 &\to \xi_1 \\ \xi_0 &\to \xi_1 \\ &\dots \\ \xi_{N-1} &\to \xi_0 \end{aligned}$$

   Die Grammatik ist *nicht* regulär, da Regeln auftreten, die nicht die Form $\langle A\rangle \to \mathtt{a}$ oder $\langle A\rangle \to \mathtt{a}\langle A\rangle$ haben.

2. Es kann keine Ableitung gefunden werden, die ein Wort mit einem Terminalzeichen ergibt.

   Anwenden der CNF-Transformation: $\xi_i \to \langle A\rangle \; \forall\, i$. Nachdem $\langle A\rangle$ undefiniert ist, kann der Zyklus kann entfernt werden. ☞ Nachdem keine Regeln verbleiben, gilt $\mathcal{L}(G) = \varnothing$.

   Achtung: Dies bedeutet auch, dass $\epsilon \notin \delta(G)$, da es keine gültigen Produktionen gibt.

3. Eine äquivalente Grammatik in Chomsky-Normalform ist gegeben durch

$$G = G(V, \Sigma, P, \delta), V = \{S\}, P = \{\}$$

Hinweis: Nachdem die Produktionsmenge P leer ist, verletzt auch keine Regel die Forderungen der CNF.

5.33 *Aufgabe: CYK-Algorithmus*

Hinweis: Die Grammatik und das Beispielwort sind der deutschen Ausgabe der Wikipedia entnommen, siehe `de.wikipedia.org/wiki/Cocke-Younger-Kasami-Algorithmus`.

Gegeben sei die Grammatik $G = (V, \Sigma, P, S)$ mit den Regeln

$$\langle S\rangle \to \langle A\rangle\langle B\rangle \mid \langle B\rangle\langle C\rangle$$
$$\langle A\rangle \to \langle B\rangle\langle A\rangle \mid \mathtt{a}$$
$$\langle B\rangle \to \langle C\rangle\langle C\rangle \mid \mathtt{b}$$
$$\langle C\rangle \to \langle A\rangle\langle B\rangle \mid \mathtt{a}$$

Der CYK-Algorithmus wird verwendet, um zu testen, ob ein Wort $\vec{w}$ von einer kontextfreien CNF-Grammatik G erzeugt werden kann. Als Datenstruktur kommt ein zweidimensionales Feld $T_{i,j}$ zum Einsatz, das in seinen Einträgen Mengen von Nichtterminalsymbolen (Beispiel: $T_{3,4} = \{V, S, A\}$) sammelt. Alle Einträge $T_{i,j}$ enthalten ursprünglich die leere Menge. Der Algorithmus ist auf Seite 83 gegeben.

1. Leiten Sie das Wort $w = w_1\vec{w}_2 \cdots w_n =$ bbabaa aus der Grammatik ab, und zeichnen Sie den Ableitungsbaum.

2. Führen Sie den Algorithmus für das Wort $\vec{w}$ aus. Gilt $S \in T_{1,n}$? *Hinweis:* Verwenden Sie eine 6×6-Tabelle für $T_{i,j}$, um die Ergebnisse der einzelnen Rechenschritte festzuhalten.

1. Über folgende Sequenz kann das Wort »bbabaa« aus dem Startsymbol abgeleitet werden:

$$\begin{aligned}\langle S\rangle &\Rightarrow \langle B\rangle\langle C\rangle \Rightarrow \langle B\rangle\mathtt{a} \Rightarrow \langle C\rangle\langle C\rangle\mathtt{a} \Rightarrow \langle C\rangle\mathtt{aa}\\ &\Rightarrow \langle A\rangle\langle B\rangle\mathtt{aa} \Rightarrow \langle A\rangle\mathtt{baa} \Rightarrow \langle B\rangle\langle A\rangle\mathtt{baa}\\ &\Rightarrow \mathtt{b}\langle A\rangle\mathtt{baa} \Rightarrow \mathtt{b}\langle B\rangle\langle A\rangle\mathtt{baa} \Rightarrow \mathtt{bbabaa}\end{aligned}$$

Der Ableitungsbaum ist in Abbildung 5.23 gezeigt.

2. Ausführen des CYK-Algorithmus führt zu Tabelle T in Abbildung 5.7 (die Tabelleneinträge sind die Elemente $T_{i,j}$; Index i wächst von oben nach unten, Index j von links nach rechts).

   Die Vorgehensweise zum Ausfüllen der Tabelle ist wie folgt:

   (a) Initialisierungsschritt, d.h. $i = 1$: Die Tabellenelemente $T_{i,1} = T_{1,1}, T_{2,1}, \ldots, T_{6,1}$ werden befüllt, indem man überprüft, aus welchen Regeln der i-te Buchstabe des Wortes erzeugbar ist. Für »a« sind dies die Regeln $\langle A\rangle$ und $\langle C\rangle$; für »b« ist dies die Regel $\langle B\rangle$.

   (b) Verschachtelte Schleifen. Betrachte $j = 2$, also läuft i von 1 bis $n - j + 1 = 6 - 2 + 1 = 5$.

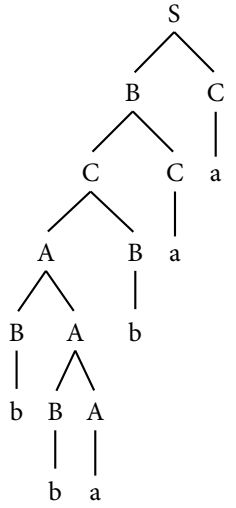


Abbildung 5.23: Ableitungsbaum für das Wort »bbabaa« über der Grammatik von Aufgabe 5.33.

Tabelle 5.7: Vom CYK-Algorithmus befüllte Tabelle zum Parsen des Wortes »bbabaa« über der Grammatik aus Aufgabe 5.33.

i \ j		1	2	3	4	5	6
1	b	B	∅	A	S, C	B	A, S
2	b	B	S, A	S, C	B	A, S	
3	a	A, C	S, C	B	S, A		
4	b	B	A, S	∅			
5	a	A, C	B				
6	a	A, C					

- $j = 2, i = 1, k = 1$, als Feld $T_{1,2}$: Betrachte $T_{1,1} = B$ und $T_{2,1} = B$. Es gibt keine Regel mit $\langle B\rangle\langle B\rangle$ auf der rechten Seite, das Feld wird mit ∅ befüllt.
- $j = 2, i = 2, k = 1$, als Feld $T_{2,2}$: Betrachte $T_{2,1} = B$ und $T_{3,1} = \{A, C\}$. Es gibt die passenden Regeln $\langle A\rangle \to \langle B\rangle\langle A\rangle$ und $\langle S\rangle \to \langle B\rangle\langle C\rangle$, also wird das Feld mit $\{S, A\}$ befüllt.
- …

Die weiteren Schritte des Algorithmus für $j = 3, \ldots, 6$ verlaufen analog, bis alle Felder ausgefüllt sind (*Achtung*: Es wird nur ein Teil der Matrix, das Dreieck links oben, befüllt).

Nachdem $\langle S\rangle \in T_{1,6}$, gilt $w \in \mathcal{L}(G)$, da das komplette Wort aus dem Startsymbol abgeleitet werden kann.

Hinweis: Man kann aus der Tabelle auch die Ableitung des Wortes nachvollziehen: Aus dem Inhalt von Feld $T_{1,6}$ folgt die Kette

$$
\begin{aligned}
T_{1,6} &\Rightarrow T_{2,5} + T_{1,1}\\
T_{2,5} &\Rightarrow T_{2,1} + T_{3,4}\\
T_{3,4} &\Rightarrow T_{3,3} + T_{6,1}\\
T_{3,3} &\Rightarrow T_{3,2} + T_{5,1}\\
T_{3,2} &\Rightarrow T_{3,1} + T_{4,1}
\end{aligned}
$$

Regeln, die nur einen Buchstaben umfassen ($T_{i,1}$), brauchen bzw. können nicht weiter aufgelöst werden.

5.34 *Aufgabe: Kellerautomaten I*

Geben Sie einen Pushdown-Automaten an, der die Sprache

$$L \equiv \{a^i b^j c^k \mid i, j, k \geq 0 \wedge (i = j \vee j = k)\}$$

erkennt. Erläutern Sie, warum eine Erweiterung auf $i = j = k$ nicht möglich ist.

Hinweis: Die Aufgabe basiert auf Sipser, Abbildung 2.17.

Wir setzen folgende Konstruktionsidee ein: Zuerst werden alle Buchstaben »a« gelesen und für jeden gelesenen Buchstaben einen

Buchstaben »A« auf dem Stapel aufgebaut (aus Gründen der leichteren Unterscheidbarkeit werden Kleinbuchstaben im Band- und Großbuchstaben im Stapelalphabet verwendet, dies ist aber nicht esentiell für die Arbeitsweise). Anschließend wird nicht-deterministisch zwischen folgenden Schritten gewählt:

- Alle Buchstaben »b« lesen. Sicherstellen, dass $\#_a = \#_b$ gilt, indem für jedes gelesene »b« ein Buchstabe »B« auf dem Stapel abgebaut wird, und danach eine beliebige Anzahl von Buchstaben »c« lesen.
- Alle Buchstaben »b« lesen und buchstabenweise als »B«s auf dem Stapel aufbauen. Anschließend alle Buchstaben »c« lesen und prüfen (durch simultanen Abbau von »B«s vom Stapel), dass $\#_b = \#_c$, dann die vorhandenen Buchstaben »A« auf dem Stapel abbauen.

  Die Strategie wird durch folgende Zustände umgesetzt:

- q_0: Buchstabe »a« lesen und »A« auf Stack aufbauen
- q_1: Buchstabe »b« lesen, für jeden Buchstaben ein »A« vom Stapel abbauen
- q_2: Buchtabe »b« lesen und »B« auf Stack aufbauen
- q_3: Buchstabe »c« lesen, ein »B« vom Stapel abbauen (gleiche Anzahl). Anschießend beliebige Anzahl »A«s vom Stack abbauen

  Die expliziten Regeln:

1. »a« lesen und »A« auf dem Stapel aufbauen. Dies muss bei leerem Stapel funktionieren und auch, wenn bereits das Zeichen »A« oben am Stapel liegt. Ein anderer Buchstabe darf *nicht* am Stapel obenauf liegen, da das Wort ansonsten ungültig wäre und abgelehnt werden muss).

$$q_0\mathrm{a}\# \to q_0\mathrm{A}\#$$
$$q_0\mathrm{aA} \to q_0\mathrm{AA}$$

2. Wenn das erste Zeichen »b« gelesen wird, wechselt der Automat in den Zustand q_1, in dem »A«s *ab*gebaut werden (die dritte Transition behandelt den Fall $i = j$, keine »c«s):

$$q_0\mathrm{bA} \to q_1\epsilon$$
$$q_1\mathrm{bA} \to q_1\epsilon$$
$$q_1\epsilon\# \to q_1\epsilon$$

3. Wenn das erste Zeichen »b« gelesen wird, wechselt der Automat in den Zustand q_2, in dem »A«s *auf*gebaut werden (der zweite Übergang behandelt den Fall, dass keine »a«s auf dem Band vorkommen):

$$q_0\mathrm{bA} \to q_2\mathrm{BA}$$
$$q_0\mathrm{b}\# \to q_2\mathrm{B}\#$$
$$q_2\mathrm{bB} \to q_2\mathrm{BB}$$

4. Nachdem das erste »c« gelesen wurde, beginnt der Automat mit dem Abbau der »B«s auf dem Stapel. Wenn alle »B«s abgebaut wurden, werden die verbliebenen »A«s vom Stapel abgebaut:

$$q_2\mathrm{cB} \rightarrow q_3\epsilon$$
$$q_3\mathrm{cB} \rightarrow q_3\epsilon$$
$$q_3\epsilon\mathrm{A} \rightarrow q_3\epsilon$$
$$q_3\epsilon\# \rightarrow q_3\epsilon$$

Der nicht-deterministische Übergang findet im Zustand q_0 beim Wechsel von »a« nach »b« statt:

$$q_0bA \begin{cases} \rightarrow q_1\epsilon \\ \rightarrow q_2\mathrm{BA} \end{cases}$$

Eine Erweiterung auf die striktere Bedingung $i = j = k$ ist nicht möglich, da nur der Auf/Abbau eines Buchstabenpaares möglich ist. Abgesehen davon wurde bereits mit dem Pumping-Lemma gezeigt, dass die Sprache nicht kontextfrei ist und daher mit einem Kellerautomaten prinzipiell nicht erkannt werden kann.

5.35 Aufgabe: Kellerautomaten II

Sei M ein Kellerautomat für die in der Vorlesung besprochene Sprache

$$L \equiv \{a_1a_2 \ldots a_na_na_{n-1} \ldots a_1 \mid a_i \in \Sigma\},$$

und sei $\Sigma = \{a, b, c\}$. Erweitern Sie die in der Vorlesung besprochene Maschine auf das neue Alphabet, und geben Sie den Konfigurationsbaum für die Erkennung des Wortes »baaccaab« an.

Band- und Kelleralphabet müssen erweitert werden, um den neuen Buchstaben repräsentieren zu können:

$$\Gamma \rightarrow \{A, B, C\}$$
$$\Sigma \rightarrow \{a, b, c\}$$

Die Regeln zum Stackaufbau müssen den neuen Buchstaben berücksichtigen:

$$q_0\mathrm{aC} \rightarrow q_0\mathrm{AC}$$
$$q_0\mathrm{bC} \rightarrow q_0\mathrm{BC}$$
$$q_0\mathrm{c\#} \rightarrow q_0\mathrm{C\#}$$
$$q_0\mathrm{cA} \rightarrow q_0\mathrm{CA}$$
$$q_0\mathrm{cB} \rightarrow q_0\mathrm{CB}$$
$$q_0\mathrm{cC} \rightarrow q_0\mathrm{CC}$$

Ebenso werden die Regeln zum Übergang in der Wortmitte und zum Abbau des Stacks erweitert:

$$q_0\mathrm{cC} \rightarrow q_1\epsilon$$

$$q_1 cC \to q_1 \epsilon$$

Der Konfigurationsbaum für das Wort »baaccaab« ist in Abbildung 5.24 dargestellt.TODO: *Sauber mit tikz setzen*

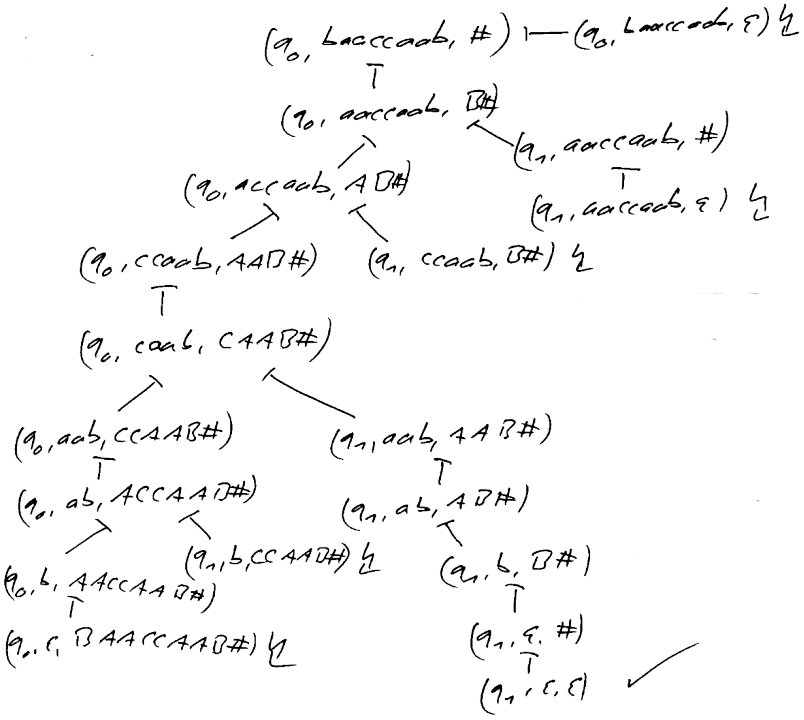


Abbildung 5.24: Konfigurationsbaum für die Ableitung des Wortes »baaccaab« mit dem Kellerautomat aus Aufgabe 5.35.

5.36 *Aufgabe: Transduktoren*

Die bislang betrachteten Automatenmodelle dienten lediglich zum Erkennen von Wörtern aus einer gegebenen Sprache L. In der Praxis werden aber auch Automaten benötigt, die unterschiedliche Darstellungen von Informationen konvertieren. Erweitern Sie das Modell des deterministischen endlichen Automaten so, dass in jedem Zustandsübergang nicht nur ein Zeichen gelesen wird, sondern auch ein Zeichen (einschließlich des leeren Wortes) auf ein Ausgabeband geschrieben werden kann. Das geschriebene Zeichen kann dabei vom aktuellen Zustand und dem zuletzt gelesenen Zeichen abhängen. Da das entstehende Automatenmodell zwischen zwei Sprachen übersetzen kann, bezeichnet man es als *Transduktor*.

1. Beschreiben Sie, wie die Definition des DEAs erweitert werden muss, um einen Transduktor zu erhalten.

2. Geben Sie eine formale Definition für Transduktoren an.

3. Die *Binary Coded Decimal*-Kodierung (BCD) kodiert eine Dezimalzahl, indem jede Ziffer durch 4 Zeichen lange Bitstrings repräsentiert wird (verwenden Sie die übliche Kodierung für vorzeichenlose Ganzzahlen, um die einzelnen Dezimalziffern zu repräsentieren). Die entstehenden Ziffern werden in der üblichen Weise konkateniert, um längere Zahlen zu erhalten.

Geben Sie einen Transduktor an, der eine Zahl in Binärdarstellung einliest und die dazu korrespondierende Dezimalzahl erzeugt. Geben Sie auch die Ein- und Ausgangssprache des resultierenden Automaten an.

1. UM DIE DEFINITION ZU ERWEITERN, sind folgende Schritte notwendig:

   - Ein Ausgabealphabet (das keine Übereinstimmung mit dem Bandalphabet zu haben braucht) muss spezifiziert werden.
   - Die Übergangsfunktion muss so erweitert werden, dass ein Zeichen auf das Band geschrieben werden kann.
   - Akzeptierende Endzustände werden überflüssig, da das eigentliche Resultat nicht mehr die Frage nach einer akzeptierenden oder nicht akzeptierenden Berechnung, sondern das als Ausgabe erzeugte Wort ist. Man kann das Konzept der Endzustände allerdings auch beibehalten.

2. Formale Definition: Ein Transduktor M ist ein Tupel $M = (Q, \Sigma, \Gamma, q_0, \delta)$; $\Gamma = \Gamma' \cup \epsilon$)

   - mit Γ': Bandalphabet
   - und $\delta : Q_* \times \Sigma \to Q \times \Gamma$

   Durch die Forderung $\Gamma = \Gamma' \cup \epsilon$ wird sichergestellt, dass es auch möglich ist, in einem Übergang *kein* Zeichen zu schreiben.

3. Abbildung 5.25 gibt eine bildliche Darstellung des Automaten ($\xrightarrow{a,b}$ in einer Transition bedeutet: Zeichen »a« gelesen, Zeichen »b« ausgegeben).

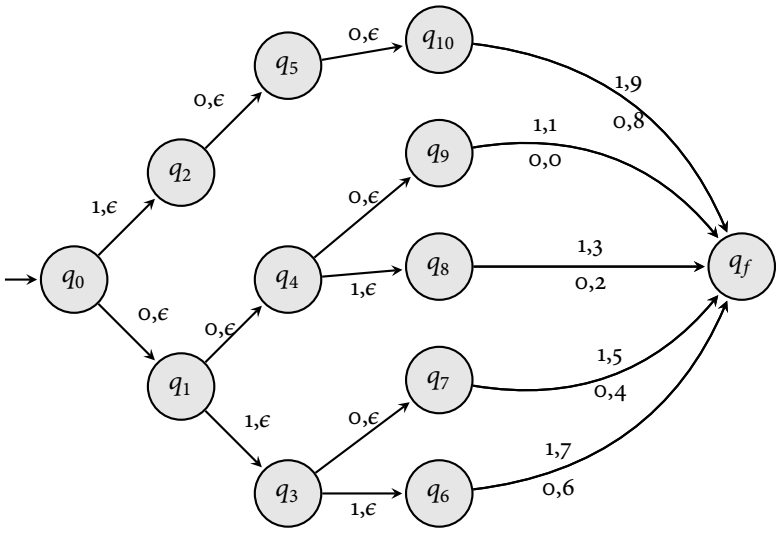


Abbildung 5.25: Der in Aufgabe 5.36 betrachtete Transduktor.

Ein Sonderfall tritt auf, wenn das erste Zeichen »1« ist, da dann nicht alle Folgekombinationen definiert sind. *Achtung*: Erst im letzten Schritt wird ein Zeichen ausgegeben, alle anderen Übergänge schreiben ϵ (und damit kein Zeichen).

5.37 Aufgabe: Parser

Hinweis: Die Aufgabe orientiert sich an Hopcroft et al., Übung 6.3.2

Gegeben sei die Grammatik G, deren Produktionen durch

$$\langle S\rangle \to \mathtt{a}\langle A\rangle\langle A\rangle$$
$$\langle A\rangle \to \mathtt{a}\langle S\rangle \mid \mathtt{b}\langle S\rangle \mid \mathtt{a}$$

definiert sind.

1. Erläutern Sie, warum G kontextfrei, aber nicht regulär ist.
2. Geben Sie eine äquivalente Grammatik in Chomsky-Normalform an.
3. Geben Sie einen *Parser* in Form eines Kellerautomaten an, der alle Wörter der Sprache erkennt. Macht es einen Unterschied, ob sie als Basis die Grammatik in Originalform oder in CNF verwenden?
4. Ist der entstehende Kellerautomat nicht-deterministisch? Bitte begründen Sie Ihre Aussage.

1. Für die Regel $\langle S\rangle \to \mathtt{a}\langle A\rangle\langle A\rangle$ gilt $|\mathtt{a}\langle A\rangle\langle A\rangle| = |r| = 3$, die CNF-Bedingungen sind also nicht erfüllt (allerdings ist die Regel kontextfrei). Auch Regeln $\langle A\rangle \to \mathtt{a}\langle S\rangle$ und $\langle A\rangle \to b\langle S\rangle$ sind kontextfrei, aber nicht in der von der CNF geforderten Struktur $\langle N\rangle \to \langle X\rangle\langle Y\rangle$.
2. Eine äquivalente Grammatik in CNF ist gegeben durch

$$\langle A'\rangle \to \mathtt{a}$$
$$\langle B'\rangle \to \mathtt{b}$$
$$\langle S\rangle \to \langle A'\rangle\langle B\rangle$$
$$\langle B\rangle \to \langle A\rangle\langle A\rangle$$
$$\langle A\rangle \to \langle A'\rangle\langle S\rangle \mid \langle B'\rangle\langle S\rangle \mid \mathtt{a}$$

3. Es gilt $\Sigma = \{a, b\}$, $V = \{A', B', A, B, S\}$ und $\Gamma = \Sigma \cup V$. Die Regeln lauten:

$$\delta(q, a, a) \ni (q, \epsilon)$$
$$\delta(q, b, b) \ni (q, \epsilon)$$
$$\delta(q, \epsilon, S) \ni (q, aAA)$$
$$\delta(q, \epsilon, A) \ni (q, aS)$$
$$\delta(q, \epsilon, A) \ni (q, bS)$$
$$\delta(q, \epsilon, A) \ni (q, a)$$

   Interpretation: Die ersten beiden Regeln komsumieren fertig abgeleitete Terminalsymbole. Alle anderen Regeln wenden Produktionsregeln der Grammatik an.

   Die Automaten unterscheiden sich je nach Form der Grammatik (CNF oder Originaldefinition), die erkannte Sprache bleibt aber gleich, da die CNF-Transformation die von der Grammatik erzeugte Sprache unverändert lässt: $\mathcal{L}(G) = \mathcal{L}(G')$.

4. Der Automat ist *nicht-deterministisch*, da

$$q, \epsilon, A \rightarrow \begin{cases} q, bS \\ q, a \end{cases}$$

ein nicht-deterministischer Übergang ist.

5.38 Aufgabe: Normalform für Kellerautomaten

Hinweis: Die Aufgabe orientiert sich an einem Übungsblatt von Prof. Dr. Kurt-Ulrich Witt.

Ein Kellerautomat sei in Normalform, wenn für jede Regel $\delta(q, a, A) \ni (q', \alpha)$ gilt, dass $|\alpha| \leq 2$.

1. Geben Sie zwei Beispiele für Regeln in Normalform an, und zwei Beispiele, die nicht in Normalform sind.
2. Zeigen Sie, dass die Menge der erkennbaren Sprachen von Kellerautomaten und Kellerautomaten in Normalform identisch sind.

1. Folgende Regeln sind in der beschriebenen Normalform:

$$\delta(q_0, a, A) \ni (q_1, AB)$$
$$\delta(q_0, a, B) \ni (q_0, \epsilon)$$

Nicht in Normalform:

$$\delta(q_0, a, A) \ni (q_0, AAA)$$
$$\delta(q_0, a, B) \ni (q_1, ABCA)$$

Hinweis: Es gibt für verschiedene Automatenmodelle typischerweise viele verschiedene mögliche Normalformen. Sinn und Zweck einer Normalform ist es, mögliche weitergehende Beweise zu vereinfachen, da nur noch wenige Regelformen berücksichtigt werden müssen. Allerdings darf die Berechnungsmächtigkeit eines Automaten nicht durch die Normalform beschränkt werden – jede Regel muss auch in der Normalform auszudrücken sein. Für Kellerautomaten ist die gewählte Normalform die kleinstmögliche; eine Beschränkung auf $|\alpha| = 1$ würde es beispielsweise unmöglich machen, den Stapel aufzubauen, da nur das implizit heruntergenommene Zeichen ersetzt, aber kein neues Zeichen hinzugefügt werden könnte.

2. Sei M ein Kellerautomat und M' ein analoger Kellerautomat mit Regeln in Normalform. Es ist zu zeigen, dass $\mathcal{L}(M) = \mathcal{L}(M')$.

   Das Problem hängt allerdings von der spezifischen Sprache ab, weshalb man die Äquivalenz für jede Sprache M zeigen müsste. Um den Beweis allgemein durchzuführen, betrachten wir die Klasse aller durch Kellerautomaten erkennbaren Sprachen $\mathcal{L}_2$ und definieren $\mathcal{L}_2^{NF}$ als Klasse aller Sprachen, die durch Normalform-PDA's zu erkennen sind.

   Es ist zu zeigen, dass $\mathcal{L}_2^{\mathrm{NF}} = \mathcal{L}_2$. Die Aussage wird wie üblich in zwei Richtungen aufgespalten:

$$\underbrace{\mathcal{L}_2^{\mathrm{NF}} \subseteq \mathcal{L}_2}_{\text{Trivial, da jeder NF-PDA auch ein PDA ist.}} \wedge \mathcal{L}_2 \subseteq \mathcal{L}_2^{\mathrm{NF}}$$

Beweis $\mathcal{L}_2 \subseteq \mathcal{L}_2^{NF}$: Jeder Übergang

$$\delta(q, a, A) \ni (q', B_n \dots B_m)$$

mit $m \geq 3$ wird simuliert durch die Kette

$$\begin{aligned}\delta(q, a, A) &\ni (S_1, B_{m-1}B_m)\\ \delta(S_1, \epsilon, B_{m-1}) &\ni (S_2, B_{m-2}B_{m-1})\\ &\dots\\ \delta(S_{m-2}, \epsilon, B_2) &\ni (q', B_1B_2)\end{aligned}$$

Durch Erweiterung $Q \to Q \cup \{S_1, \dots, S_{m-2}\}$ und Hinzunahme der neuen Regeln zu δ (und gleichzeitigem Entfernen der alten Regel) kann der Automat auf Normalform gebracht werden. Die Sprache wird nicht verändert, da die neuen »Zwischenzustände« nur innerhalb der Ableitungskette erreichbar sind, die vom ursprünglichen Zustand aus angestoßen wurde.

5.39 *Aufgabe: Normalform für kontextsensitive Sprachen*

Hinweis: Die beschriebene Normalform ist in der Literatur als Kuroda-Normalform bekannt, siehe beispielsweise Schöning S. 79/80.

Jede kontextsensitive Grammatik G kann in einer Normalform geschrieben werden, in der alle Regeln eine der Formen

$$\begin{aligned}\langle A\rangle\langle B\rangle &\to \langle C\rangle\langle D\rangle\\ \langle A\rangle &\to \langle B\rangle\langle C\rangle\\ \langle A\rangle &\to \langle B\rangle\\ \langle A\rangle &\to \mathtt{a}\end{aligned}$$

besitzen.

1. Erläutern Sie, welche der obigen Regeln nicht kontextfrei sind.
2. Geben Sie einen Algorithmus an, um eine gegebene kontextsensitive Grammatik G in eine äquivalente Grammatik G' mit $\mathcal{L}(G) = \mathcal{L}(G')$ in Normalform zu verwandeln. *Hinweis:* Orientieren Sie sich an der CNF-Transformation, und überlegen Sie, welche Regelformen im kontextsensitiven Fall zusätzlich vorkommen können.

1. Die Regel $\langle A\rangle\langle B\rangle \to \langle C\rangle\langle D\rangle$ ist nicht kontextfrei, da die linke Seite länger als ein Zeichen ist. $\langle A\rangle \to \langle B\rangle$ ist kontextfrei, aber nicht in CNF.
2. Wie bei der Transformation zur Chomsky-Normalform werden zunächst Zyklen entfernt und Sonderregeln $\langle A\rangle \to \mathtt{a}$ für jedes Terminalsymbol hinzugefügt.

   Regeln der Form $\langle A\rangle \to \langle B\rangle\langle C\rangle\langle D\rangle \dots \langle X\rangle$ werden wie bei der CNF-Transformation aufgebrochen zu

   $$\langle A\rangle \to \langle B\rangle\langle C\rangle_1, \langle C\rangle_1 \to \langle C\rangle\langle C\rangle_2, \dots$$

☞ Es verbleiben Regeln der Form $\langle A\rangle\langle B\rangle \ldots \langle X\rangle \rightarrow \langle F\rangle\langle G\rangle\langle H\rangle \ldots \langle X\rangle$; oder etwas systematischer ausgedrückt:

$$\langle A\rangle_1, \langle A\rangle_2 \ldots \langle A\rangle_m \rightarrow \langle B\rangle_1\langle B\rangle_2 \ldots \langle B\rangle_n$$

mit $2 \leq m \leq n$ (da $|r| \geq |l|$ nach Definition der kontextsensitiven Sprachen $\mathcal{L}_1$; der Fall $m = 1$ wurde bereits behandelt). Die Regeln werden ersetzt durch die Kette

$$\begin{aligned}\langle A\rangle_1\langle A\rangle_2 &\rightarrow \langle B\rangle_1\langle C\rangle_2\\ \langle C\rangle_2\langle A\rangle_3 &\rightarrow \langle B\rangle_2\langle C\rangle_3\\ &\ldots\\ \langle C\rangle_{m-1}\langle A\rangle_m &\rightarrow \langle B\rangle_{m-1}\langle C\rangle_m\end{aligned}$$

und die darauffolgende Kette

$$\begin{aligned}\langle C\rangle_m &\rightarrow \langle B\rangle_m\langle C\rangle_{m+1}\\ \langle C\rangle_{m+1} &\rightarrow \langle B\rangle_{m+1}\langle C\rangle_{m+2}\\ &\ldots\\ \langle C\rangle_{n-1} &\rightarrow \langle B\rangle_{n-1}\langle B\rangle_n\end{aligned}$$

Der erste Teil arbeitet Nichtterminale auf beiden Seiten der ursprünglichen Regel ab; wenn m Nichtterminale bearbeitet wurden, sind möglicherweise noch Nichtterminale auf der rechten Seite verblieben, die mit der zweiten Kette verarbeitet werden.

Beide Ketten hintereinander ausgeführt liefern eine zur ursprünglichen Regeln äquivalente Regel, wie man sich beispielsweise für $n = 5, m = 3$ klarmachen kann: $\langle A\rangle_1\langle A\rangle_2\langle A\rangle_3 \rightarrow \langle B\rangle_1\langle B\rangle_2\langle B\rangle_3\langle B\rangle_4\langle B\rangle_5$ wird aufgespalten in

$$\begin{aligned}\langle A\rangle_1\langle A\rangle_2 &\rightarrow \langle B\rangle_1\langle C\rangle_2\\ \langle C\rangle_2\langle A\rangle_3 &\rightarrow \langle B\rangle_2\langle C\rangle_3\\ \langle C\rangle_3 &\rightarrow \langle B\rangle_3\langle C\rangle_4\\ \langle C\rangle_4 &\rightarrow \langle B\rangle_4\langle B\rangle_5\end{aligned}$$

Die durch die neuen Regeln induzierte Ableitungskette ist

$$\begin{aligned}\langle A\rangle_1\langle A\rangle_2\langle A\rangle_3 &\Rightarrow \langle B\rangle_1\langle C\rangle_2\langle A\rangle_3 \Rightarrow \langle B\rangle_1\langle B\rangle_2\langle C\rangle_3\\ &\Rightarrow \langle B\rangle_1\langle B\rangle_2\langle B\rangle_3\langle C\rangle_4 \Rightarrow \langle B\rangle_1\langle B\rangle_2\langle B\rangle_3\langle B\rangle_4\langle B\rangle_5\end{aligned}$$

und entspricht in ihrer Wirkung obiger Originaldefinition. Nachdem die neuen Regeln nur innerhalb der Ableitungskette verfügbar sind, wird die erzeugte Sprache nicht verändert.

5.40 Aufgabe: Kellerautomat mit zwei Stapeln

Betrachten Sie einen Kellerautomaten, der mit *zwei* Stapeln arbeitet. Stapeloperationen sollen in jedem Rechenschritt unabhängig voneinander auf jedem Stapel durchgeführt werden können.

1. Geben Sie eine formale Definition des resultierenden Automaten an.
2. Konstruieren Sie einen Zwei-Stapel-PDA, mit dessen Hilfe die Sprache

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

erkannt wird.

1. Ohne Beschränkung der Allgemeinheit verwenden beide Stapel das gleiche Stapelalphabet. ☞ Nur die δ-Funktion muss modifiziert werden. Jeder Zustand mit leerem Band und zwei leeren Kellern wird als akzeptierender Endzustand gewertet; explizite Akzeptanzzustände werden nicht verwendet. Die Signatur der Übergangsfunktion ist

$$\delta : Q \times \Sigma_\epsilon \times \Gamma \times \Gamma \to \mathcal{P}_\mathrm{f}(Q \times \Gamma^* \times \Gamma^*) \text{ (nicht-deterministisch)},$$
$$\delta : Q \times \Sigma_\epsilon \times \underbrace{\Gamma \times \Gamma}_{\substack{\text{Oberstes Zeichen auf} \\ \text{Stapel 1 bzw. Stapel 2}}} \to Q \times \underbrace{\Gamma^* \times \Gamma^*}_{\substack{\text{Neuer Inhalt für} \\ \text{Stapel 1 bzw. 2}}} \text{ (deterministisch)}.$$

   Bei jedem Übergang wird das oberste Zeichen von beiden Stapeln entfernt und die Zeichenkette auf der rechten Seite der Regel mit auf den Stapel gelegt. Das am weitesten links stehenden Zeichen kommt dabei obenauf.

2. Konstruktionsidee: Alle »a«-Zeichen lesen und auf beide Stapel pushen, dann »b«-Zeichen lesen und »a« von Stapel 1 poppen, dann »c«-Zeichen lesen und »a« von Stapel 2 poppen.

   Zur Vereinfachung wird im Folgenden $\delta(q, a, A, B) \ni (q', x_1, x_2)$ geschrieben als $qaAB \to q'x_1, x_2$ (das Trennzeichen »,« auf der rechten Seite ist notwendig, um eindeutig zwischen den beiden Stapeln unterscheiden zu können). Die Regeln sind dann gegeben durch:

   - Zeichen »a« lesen, »A« auf beiden Stapeln aufbauen:

$$q_0 a\#\# \to q_0 A, A$$
$$q_0 aAA \to q_0 AA, AA$$
$$q_0 \epsilon\#\# \to q_0 \epsilon, \epsilon$$

   - Zeichen »b« lesen, »A« vom ersten Stapel abbauen:

$$q_0 bAA \to q_1 \epsilon, A$$
$$q_1 bAA \to q_1 \epsilon, A$$

   - Zeichen »c« lesen, »A« vom zweiten Stapel abbauen:

$$q_1 c\#A \to q_2 \#, A$$
$$q_2 c\#A \to q_2 \#, \epsilon$$
$$q_2 \epsilon\#\# \to q_2 \epsilon, \epsilon$$

5.41 Aufgabe: Turing-Maschinen

Geben Sie jeweils eine Turing-Maschine für folgende Probleme an:

1. Invertieren einer Binärzahl: Alle Ziffern sollen invertiert werden, d.h. die Maschine berechnet die Zifferntransformation $0 \to 1$ und $1 \to 0$.
2. Berechnen der Funktion $x \to x + 2$. Arbeiten Sie im Binärsystem.
3. Berechnen der Funktion $x \to x + k$. *Hinweis:* Verallgemeinern Sie Ihre obige Überlegung.

Achten Sie auf eine formal vollständige Definition, und demonstrieren Sie die Arbeitsweise Ihrer Maschinen jeweils anhand eines Beispiels.

1. Die Maschine arbeitet die Eingabe von links nach rechts ab und invertiert jedes einzelne Zeichen. Dies wird durch die Regeln

$$\delta(q_0, 0) = (q_0, 1, \mathrm{R})$$
$$\delta(q_0, 1) = (q_0, 0, \mathrm{R})$$
$$\delta(q_0, \square) = (q_f, \square, \mathrm{L})$$

   erledigt. Formal ist die Maschine damit durch das 6-Tupel

$$M = (\{q_0, q_\mathrm{f}, \{0, 1\}, \{0, 1, \square\}, \delta, q_0, \square, \{q_\mathrm{f}\}\})$$

   gegeben. Nach Ende des Rechenvorgangs steht der Schreib/Lesekopf auf dem letzten Zeichen der Eingabe (trivialerweise könnte man den Kopf natürlich wieder auf das erste Zeichen zurückbewegen, was nützlich ist, wenn mehrere Maschinen konkateniert werden, wie aus den nächsten beiden Aufgaben ersichtlich ist).

2. Als Basis wird die in der Vorlesung besprochene Maschine verwendet, die die Transformation $x \to x + 1$ berechnet, also 1 zu einer beliebigen (positiven) Zahl in Binärdarstellung addiert. Die Maschine $M = (\{q_0, q_1, q_2, q_f\}, \{0, 1\}, \Sigma \cup \{\square\}, \delta, q_0, \{q_\mathrm{f}\})$ besitzt die in Tabelle 5.8 gezeigten Übergänge.

Q \ Σ	0	1	$\square$
q_0	$(q_0, 0, \mathrm{R})$	$(q_0, 1, \mathrm{R})$	$(q_1, \square, \mathrm{L})$
q_1	$(q_2, 1, \mathrm{L})$	$(q_1, 0, \mathrm{L})$	$(q_\mathrm{f}, 1, \mathrm{N})$
q_2	$(q_2, 0, \mathrm{L})$	$(q_2, 1, \mathrm{L})$	$(q_\mathrm{f}, \square, \mathrm{R})$
q_f	/	/	/

Tabelle 5.8: Transitionen der Turing-Maschine zur Berechnung von $x + 1$.

   Um die Funktion $x \to x + 2$ zu berechnen, wird die Maschine zweimal hintereinander ausgeführt. Der Endzustand q_f wird dazu durch den Zustand q_3 ersetzt, der analog zu q_0 definiert ist; entsprechend werden q_4 und q_5 eingeführt, die zu q_1 und q_2 äquivalent sind. Dies führt zur Maschine $M' = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_f\}, \{0, 1\}, \Sigma \cup \{\square\}, \delta, q_0, \{q_\mathrm{f}\})$, die die in Tabelle 5.9 gezeigten Transitionen besitzt.

3. Zur Berechnung der Funktion $x \to x + k$ wird die Maschine $x \to x + 1$ k-fach hintereinander ausgeführt. Die Zustände $\{q_0, q_1, q_2\}$ werden entsprechend k-fach benötigt. Dies kann für $i \in [1, k]$ allgemein wie in Tabelle 5.10 gezeigt definiert werden. Der Zustand q_{3k+3} ist der

Q \ Σ	0	1	□
q_0	$(q_0, 0, \mathrm{R})$	$(q_0, 1, \mathrm{R})$	$(q_1, \square, \mathrm{L})$
q_1	$(q_2, 1, \mathrm{L})$	$(q_1, 0, \mathrm{L})$	$(q_3, 1, \mathrm{N})$
q_2	$(q_2, 0, \mathrm{L})$	$(q_2, 1, \mathrm{L})$	$(q_3, \square, \mathrm{R})$
q_3	$(q_3, 0, \mathrm{R})$	$(q_3, 1, \mathrm{R})$	$(q_4, \square, \mathrm{L})$
q_4	$(q_5, 1, \mathrm{L})$	$(q_4, 0, \mathrm{L})$	$(q_\mathrm{f}, 1, \mathrm{N})$
q_5	$(q_5, 0, \mathrm{L})$	$(q_5, 1, \mathrm{L})$	$(q_\mathrm{f}, \square, \mathrm{R})$
q_f	/	/	/

Tabelle 5.9: Transitionen der Turing–Maschine zur Berechnung von $x + 2$.

Q \ Σ	0	1	□
q_{3i}	$(q_{3i}, 0, \mathrm{R})$	$(q_{3i}, 1, \mathrm{R})$	$(q_{3i+1}, \square, \mathrm{L})$
q_{3i+1}	$(q_{3i+2}, 1, \mathrm{L})$	$(q_{3i+1}, 0, \mathrm{L})$	$(q_{3i+3}, 1, \mathrm{N})$
q_{3i+2}	$(q_{3i+2}, 0, \mathrm{L})$	$(q_{3i+2}, 1, \mathrm{L})$	$(q_{3i+3}, \square, \mathrm{R})$

Tabelle 5.10: Transitionen der Turing–Maschine zur Berechnung von $x + k$.

akzeptierende Endzustand und enthält keine weiteren Übergänge. Die Maschine ist damit formal durch das 6-Tupel

$$M'' = (\{q_0, \ldots, q_{3i+3}\}, \{0, 1\}, \Sigma \cup \{\square\}, \delta, q_0, \{q_{3i+3}\})$$

gegeben.

5.42 Aufgabe: Robustheit von Turing-Maschinen

Turing-Maschinen sind wie in der Vorlesung diskutiert ein sehr robustes Konzept, dessen Berechnungskraft sich unter vielen Modifikationen nicht verändert. Beispielsweise ist bekannt, dass eine Turing-Maschine mit einem einseitig unendlichen Band gleichmächtig zur Standardmaschine mit beidseitig unendlichem Band ist. Sie sollen in dieser Aufgabe unter Verwendung dieser Aussage zeigen, dass ein PDA mit zwei Stapeln (2-PDA) gleichmächtig zu einer Turing-Maschine ist.

1. Argumentieren Sie (unter Verwendung der Church-Turing-These), warum eine Turing-Maschine einen 2-PDA simulieren kann.
2. Zeichnen und beschreiben Sie ein Konzept, wie in der umgekehrten Richtung die beiden Stapel des 2-PDAs zur Simulation des einseitig unendlichen Bandes verwendet werden können. *Hinweis:* Bilden Sie den Bandteil links und rechts des Schreib-Lesekopfes geeignet auf die Stapel ab, und überlegen Sie, wie Übergänge und Grenzfälle zu behandeln sind.
3. Geben Sie eine formale Beschreibung Ihrer Konstruktion an.

1. Die Simulation eines PDA mit zwei Stapeln ist intuitiv berechenbar, also kann die Rechnung nach der Church-Turing-These mit einer Turingmaschine durchgeführt werden. Dies bedeutet, dass eine TM einen 2-PDA simulieren kann.
2. Um eine Turing-Maschine durch einen 2-PDA zu simulieren, wird das einseitig unendliche Band in zwei konzeptionelle Hälften zerteilt. Der

Anteil links vom SL-Kopf wird auf dem ersten Stapel repräsentiert; der Anteil rechts davon auf dem zweiten Stapel. Kopfbewegungen werden simuliert, indem Zeichen zwischen den Stapeln verschoben werden. Die Zustandsübergänge folgen den Übergängen der Turing-Maschine. Wenn die TM einen akzeptierenden Endzustand erreicht, akzeptiert auch der 2-PDA. Folgende Möglichkeiten für Übergänge werden betrachtet, solange sich der LS-Kopf innerhalb des Eingabewortes bewegt (wir gehen trivialerweise davon aus, dass sich das Anfangswort auf dem Band der TM bereits auf dem zweiten Stapel befindet und das erste Zeichen oben steht):

- Bei einer Rechtsbewegung wird das oberste Zeichen vom zweiten Stapels gepopt und das in der Übergangsregel geschriebene Zeichen auf den ersten Stapel gepusht.
- Bei einer Linksbewegung wird das oberste Zeichen vom ersten Stapel gepopt. Das oberste Zeichen des zweiten Stapels wird durch das geschriebene Zeichen ersetzt; das vom ersten Stapel gepopte Zeichen wird obenauf gelegt.
- Bei einem Nullübergang wird das oberste Zeichen des zweiten Stapels entsprechend der Transitionsregel ersetzt.

Die passende Übergangsregel wird anhand des Zustands des 2-PDAs und des obersten Zeichens im zweiten Stapel identifiziert. Die Regeln müssen analog zu den Regeln der TM angegeben werden.

Wenn die Maschine über das Eingabewort nach rechts hinausläuft, ist der zweite Stapel leer. Ein Blank-Symbol wird auf den linken Stapel gepusht, um das einseitig unendliche Band zu simulieren.

3. Der 2-PDA ist formal gegeben durch das 6-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, wobei Q, Σ, Γ, q_0 und $F \subset Q$ die gleiche Bedeutung wie bei einem normalen PDA besitzen (beide Stapel verwenden oBdA das gleiche Alphabet Γ). Die Übergangsfunktion δ besitzt die Signatur

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^2 \to Q \times \Gamma^* \times \Gamma^*$$

Bei jedem Übergang werden die obersten Elemente Γ von den Stapeln gepusht und durch eine (potentiell leere) Zeichenkette aus Γ^* ersetzt, von denen das erste Zeichen zuoberst steht.

Hinweis: Das Bandalphabet Σ wird nur verwendet, um den ursprünglichen Bandinhalt einzulesen; anschließend arbeitet die Maschine ausschließlich auf den Stapeln und verwendet ϵ-Übergänge auf dem Band.

5.43 *Aufgabe: Turing-Maschine ohne Neutralbewegung*

Zeigen Sie, dass die Standard-Turingmaschine durch eine Turing-Maschine mit nur zwei Kopfbewegungen $\{L, R\}$ ersetzt werden kann.

Die zentrale Konstruktionsidee ist, die Neutralbewegung durch eine Linksbewegung zu ersetzen, die von einer Rechtsbewegung gefolgt wird, bei der der Bandinhalt invariant bleibt. Um die Rechnung der Maschine nicht abzuändern, wird ein Zwischenzustand eingeführt, der normalerweise nicht erreichbar ist. Formal wird eine Übergangsfunktion der Form

$$\delta(q, x) = (q', x', N) \tag{5.16}$$

mit $q, q' \in Q$ und $x, x' \in \Sigma$ durch die Regeln

$$\delta(q, x) = (q_{\text{null}}, x', L)$$
$$\delta(q_{\text{null}}, y) = (q', y, R)\ \forall y \in \Gamma$$

mit $q_{\text{null}} \notin Q$ ersetzt. Die Zustandsmenge der Maschine wird erweitert zu $Q \rightarrow Q \cup \{q_{\text{null}}\}$. Dies wird für alle Übergangsfunktionen der Form (5.16) wiederholt, wobei zu beachten ist, dass jedesmal ein *neuer* Zwischenzustand q_{null} eingeführt werden muss.

5.44 Aufgabe: Mehrband-Turingmaschine

Geben Sie eine *formal sauber definierte* Mehrband-Turingmaschine an, die die Transformation $x_1 x_2 \ldots x_n \rightarrow x_1 x_1 x_2 x_2 \ldots x_n x_n$, berechnet, wobei $x_i \in \{0, 1\}$.

Die Maschine M ist formal gegeben durch das Tupel $M = \{Q, \Sigma, \Gamma, \delta, \{q_f\}\}$, wobei

$$Q = \{q_0, q_1, q_f\}$$
$$\Sigma = \{0, 1\}$$
$$\Gamma = \Sigma \cup \{\square\}$$

δ besitzt die Signatur

$$\delta : Q \times \Gamma \times \Gamma \rightarrow Q \times \underbrace{\Gamma \times \Gamma}_{\substack{\text{Zweifach} \\ \text{vorhanden}}} \times \underbrace{\{L, N, R\}^2}_{\substack{\text{Zweifach} \\ \text{vorhanden}}}$$

und ist in Tabelle 5.11 definiert.

Hinweis: Für alle anderen Zeichenkombinationen aus Γ^2 sind die Übergänge nicht definiert, auch wenn dies nicht explizit in der Tabelle aufgeführt ist.

Tabelle 5.11: Übergänge für die Mehrband-Turingmaschine aus Aufgabe 5.44

Q \ Γ^2	0,□	1,□	□, □
q_0	q_1,0,0,N,R	q_1,1,1,N,R	q_f, □, □,N
q_1	q_0,0,0,R,R	q_0,1,1,R,R	-
q_f	-	-	-

5.45 Aufgabe: Turing-Maschine

Hinweis: Die Aufgabe orientiert sich am Beispiel in Sipser S. 174/175.

Geben Sie eine *informelle, aber klare* Beschreibung einer Turing-Maschine an, die folgende Sprache entscheidet:

$$L_1 \equiv \{a^i b^j c^k \mid i, j, k \in \mathbb{N} \wedge i \cdot j = k\}$$

Bestimmen Sie zusätzlich Band- und Zeitkomplexität der resultierenden Maschine.

Strategie der Turing-Maschin: Für jedes Zeichen »a« wird ein Teilstring der »c«s markiert, der die Länge des »b«-Teilstrings besitzt. Die Multiplikation $|a^i| \cdot |b^j| = |c^{i \cdot j}|$ wird also durch eine Sequenz von Additionen

$$\underbrace{|b^j| + |b^j| + \cdots + |b^j|}_{i\text{–fach}}$$

ersetzt. Dazu geht die Maschine folgendermaßen vor:

1. Eingabe von links nach rechts durchlaufen, um sicherzustellen, dass sie die Form $a^+b^+c^+$ besitzt (notwendig, da M die Sprache *entscheiden* soll – Eingaben mit inkorrekter Struktur müssen explizit zurückgewiesen werden). Wenn Wort nicht in der geforderten Form: ablehnen.

2. Kopf ans linke Eingabeende zurücksetzen bis zum ersten Buchstaben »'a«.

3. »a« durchstreichen.

4. Bis zum ersten »b« nach rechts fahren. Zwischen »b«s und »c«s hin- und herfahren und jeweils ein Paar wegstreichen, bis keine »c«s mehr vorhanden sind. Ablehnen, wenn alle »c«s durchgestrichen wurden, es aber noch »b«s gibt.

5. Durchgestrichene »b«s wieder durch normale »b«s ersetzen. Zurück zum ersten »a« fahren.

   - Wenn kein »a« mehr vorhanden: Überprüfen, ob alle »c«s durchgestrichen wurden; falls ja akzeptieren.
   - Wenn noch »a«s vorhanden sind: Weiter bei Schritt 3.

Die Bandkomplexität $S(M)$ ist $i + j + k + 1 = \mathcal{O}(i) + \mathcal{O}(j) + \mathcal{O}(i \cdot j) = \mathcal{O}(i \cdot j)$ (rechtes Blankzeichen wird benötigt, um das Ende des »c«-Zeichenstrings zu erkennen)

Bezüglich Zeitkomplexität sind folgende Aktionen der Maschine zu berücksichtigen:

1. Anfangsprüfung: $2(i + j + k)$ (Hin- und Herlaufen auf dem Wort)

2. Vorwärtslauf von »a« nach »b«: $i + j$.

3. Innerer Durchlauf (pro »b« ein »c« abstreichen): $|b| \cdot 2(|b| + |c|) = 2j^2 + 2jk$. Für die Position innerhalb des »c«-Teilstrings ist der schlechteste Fall berücksichtigt, der Eintritt, wenn fast alles »c«s abgestrichen sind.

4. Rücklauf zum Start: $i + j$.

Der Zyklus 2,3,4 muss für jeden Buchstaben im Teilstring der »a«s wiederholt werden, als i-fach:

$$\begin{aligned} i \times [(i + j) + (2j^2 + 2jk) + (i + j)] &= i \times [2(i + j) + (2j^2 + 2jk)] \\ &= i \times [\mathcal{O}(j^2) + \mathcal{O}(jk)] \\ &= \mathcal{O}(ij^2) + \mathcal{O}(\underbrace{ijk}_{=i^2 j^2}) = \mathcal{O}(i^2 j^2). \end{aligned}$$

5.46 Aufgabe: Write-Once-Turing-Maschine

Nehmen Sie an, das Band einer Turing-Maschine sei durch eine DVD-R mit unendlicher Kapazität gegeben — jede Bandzelle darf deshalb nur ein einziges Mal beschrieben werden. Die ursprüngliche Eingabe befindet sich zu Beginn wie üblich am Anfang des Bandes, und die betroffenen Zellen dürfen ebenso wie die leeren Zellen einmal beschrieben werden.

1. Zeigen Sie, dass jede beliebige Turing-Maschine M durch eine Maschine M' mit den beschriebenen Beschränkungen simuliert werden kann.

   Hinweis: Erlauben Sie in einem vereinfachenden ersten Schritt das zweifache Beschreiben jeder Bandzelle, und leiten Sie daraus die gewünschte Maschine ab. Es empfiehlt sich, dei unendliche Länge des Bandes durch häufiges Umkopieren der Eingabe auszunutzen.

2. Argumentieren Sie, warum beide Berechnungsmodelle gleich mächtig sind.

1. Um eine allgemeine Turing-Maschin auf auf einem Band zu simulieren, bei dem jede Bandzelle nur zweimal beschrieben werden darf, gehen wir folgendermaßen vor:

   - Schritt 1: Kopie des Eingabewortes anlegen; jedes kopierte Zeichen markieren. Beim Kopieren wird die Wirkung der Transitionsfunktion der Turingmaschine nachhvollzogen.

     Die Position, an der sich der Kopf in der Kopie befindet, wird mit einem besonderen Zeichen (beispielsweise â statt a) ausgezeichnet. Die Schritte werden wiederholt, bis ein akzeptierender oder ablehnender Endzustand erreicht wird.

     Jedes Zeichen wird bei dieser Vorgehensweise zweimal beschrieben: Einmal beim Kopieren, einmal beim Durchstreichen (notwendig, um die Information zu besitzen, was bereits kopiert wurde).

   - Schritt 2: Jedes Zeichen auf zwei Zellen aufteilen: Eine für das Zeichen selbst, eine für Information »kopiert« oder »nicht kopiert« ☞ Jede Zelle muss nur einmal beschrieben werden.

2. Jede TM M kann wie gezeigt durch das eingeschränkte Modell simuliert werden. Da jede TM in eingeschränktem Modell zugleich eine reguläre TM ist ☞ Die Modelle sind gleich berechnungsmächtig.

5.47 Aufgabe: Oszillierende Grammatiken

Gegen Sie eine Phrasenstrukturgrammatik an, aus der ein Wort abgeleitet werden kann, dessen Länge im Verlauf der Ableitung zwischen zwei Werten oszilliert. Demonstrieren Sie Ihre Konstruktion anhand eines Beispiels. Warum ist es nicht möglich, eine kontextsensitive Grammatik mit diesen Eigenschaften anzugeben?

Wir betrachten folgende Produktionen der Grammatik G:

$$\begin{aligned}
\langle S\rangle &\to \langle A\rangle\langle B\rangle\langle C\rangle \\
\langle B\rangle &\to \langle A\rangle\langle B\rangle\langle C\rangle \mid \mathtt{b} \\
\langle A\rangle &\to \mathtt{a} \\
\langle C\rangle &\to \mathtt{c} \\
\langle A\rangle\langle B\rangle\langle C\rangle &\to \langle B\rangle
\end{aligned}$$

Aus dem Startsymbol $\langle S\rangle$ kann das Wort »abc« über die Kette

$$\begin{aligned}\langle S\rangle &\Rightarrow \langle A\rangle\langle B\rangle\langle C\rangle \Rightarrow \mathtt{a}\langle B\rangle\mathtt{c}\\ &\Rightarrow \mathtt{a}\langle A\rangle\langle B\rangle\langle C\rangle\mathtt{c} \Rightarrow \mathtt{a}\langle B\rangle\mathtt{c}\\ &\Rightarrow \mathtt{a}\langle A\rangle\langle B\rangle\langle C\rangle\mathtt{c} \Rightarrow \ldots \Rightarrow \mathtt{a}\langle B\rangle\mathtt{c} \Rightarrow \mathtt{abc}\end{aligned}$$

abgeleitet werden. Die Länge der Ableitungskette oszilliert zwischen $|\mathtt{a}\langle A\rangle\langle B\rangle\langle C\rangle\mathtt{c}| = 3$ und $|\mathtt{a}\langle B\rangle\mathtt{c}| = 5$. Dies ist mit einer kontextsensitiven $\mathcal{L}_1$-Sprache nicht möglich, da keine Verkürzung des Wortes während der Ableitung realisierbar ist ☞ Keine Oszillation möglich, da die Länge des abgeleiteten Wortes nur wachsen kann.

Achtung: Es muss sichergestellt werden, dass aus dem Startsymbol ein konkretes Wort abgeleitet werden kann, das nur aus Terminalsymbolen besteht. Ansonsten ist die Aufgabe nicht erfüllt.

5.48 Aufgabe: Zahlenmanipulation

Gegeben sei die Dezimalzahl $x = 12345$. Führen Sie folgende Transformationen unter Verwendung der Rechenoperationen $\mod$, $\div$, $\times$ und $+$ durch:

- $x \to 123456$
- $x \to 2345$
- $x \to a2345b$, wobei $a, b \in \{1, \ldots, 9\}$
- $x \to 1234$

Die Operationen werden folgendermaßen durchgeführt:

1. $12345 \to 12345 \times 10 + 6 = 123456$.
2. $12345 \to 12345 \mod 10000 = 2345$.
3. $12345 \to (12345 \mod 10000 + a \times 10^4) \times 10 + b$.
4. $12345 \to 12345 \div 10 = 1234$.

5.49 Aufgabe: Vollständige Induktion

Zeigen Sie mittels vollständiger Induktion nach n, dass

$$\sum_{k=0}^{n} q^k = \frac{q^{n+1} - 1}{q - 1}$$

für $q \in \mathbb{R}, n \in \mathbb{N} \cup 0, q \neq 1$ gilt.

Wir lösen die Induktionsaufgabe nach dem Standardmuster für arithmetische Beweise. Betrachte zunächst den Induktionsanfang $n = 0$:

$$\sum_{k=0}^{n} q^k = \sum_{k=0}^{0} q^k = q^0 = 1$$

$$\frac{q^{n+1}-1}{q-1} = \frac{q^1-1}{q-1} = 1 \checkmark$$

Für den Induktionsschritt $n \to n+1$ rechnet man:

$$\begin{aligned}\sum_{k=0}^{n+1} q^k &= q^{n+1} + \sum_{k=0}^{n} q^k \overset{\text{IV}}{=} q^{n+1} + \frac{q^{n+1}-1}{q-1} \\ &= \frac{(q-1)q^{n+1} + q^{n+1} - 1}{q-1} = \frac{q^{n+2} - q^{n+1} + q^{n+1} - 1}{q-1} \\ &= \frac{q^{n+2}-1}{q-1} \checkmark\end{aligned}$$

5.50 *Aufgabe: Zeichenketten spiegeln*

Gegeben sei die über einem Alphabet $\Sigma = \{a, b\}$ induktiv definierte Funktion $\text{mir} : \Sigma^* \to \Sigma^*$:

$$\begin{aligned}\text{mir}(\epsilon) &= \epsilon \\ \text{mir}(\vec{w}c) &= c\,\text{mir}(\vec{w})\end{aligned}$$

wobei $\vec{w} \in \Sigma^*$ und $c \in \Sigma$.

1. Geben Sie ein Wort $\vec{x} \in \Sigma^*, |\vec{x}| \geq 5$ an und berechnen Sie die Wirkung der Funktion
2. Zeigen Sie, dass die Funktion mir(·) Turing-berechenbar ist. Argumentieren Sie auf Basis einer Ein-Band-Turingmaschine.
3. Geben Sie Band- und Zeitkomplexität Ihres Algorithmus an.

1. Die Funktion mir mit Signatur $\Sigma^* \to \Sigma^*$ wird auf das Wort $\vec{x}$ = »abaab« angewendet, indem im jedem Schritt die passende Möglichkeit aus der Definition von mir ausgesucht und angewendet wird:

$$\begin{aligned}\text{mir}\underbrace{(\text{abaab})}_{\vec{w}=\text{abaa},c=\text{b}} &= \text{b mir}\underbrace{(\text{abaa})}_{\vec{w}=\text{aba},c=\text{a}} = \text{ba mir(aba)} \\ &= \text{baa mir(ab)} = \text{baab mir}\underbrace{(\text{a})}_{\vec{w}=\epsilon,c=\text{a}} \\ &= \text{baaba}\ \underbrace{\text{mir}(\epsilon)}_{=\epsilon} = \text{baaba}\epsilon = \text{baaba}\end{aligned}$$

   Die Funktion dreht das bearbeitete Wort also um (*mirror*).

2. Strategie zur Konstruktion der Turing-Maschine: Das Ausgangswort steht als Eingabe auf dem Band und wird dort belassen; das umgedrehte Wort wird buchstabenweise rechts davon auf das Band geschrieben.

   (a) Zunächst muss das Wortende mit dem Symbol $\# \notin \Sigma$ markiert werden:

   $$\cdots \square\text{abaab}\square \cdots \Rightarrow \cdots \square\text{abaab}\#\square \cdots$$

   (b) Das am weitesten rechts stehende Symbol *links von* # lesen, markieren und dann auf die erste Position *rechts von* # schreiben, die noch nicht mit einem Zeichen aus Σ belegt ist (also auf die erste Bandposition mit einem Blank-Symbol).

(c) Schritt b) wiederholen, solange noch Zeichen des Eingabewortes nicht markiert sind.

(d) Alles Zeichen links von # markiert ☞ Fertig.

Nachdem die Funktion durch einer Turing-Maschine ausgeführt werden kann, ist sie offenbar Turing-berechenbar.

3. Die Bandkomplexität ist $2n + 1 = \mathcal{O}(n)$, nachdem das Wort zweimal auf dem Band vorkommt und ein Trennzeichen benötigt wird.

   Die Zeitkomplexität analysiert man, indem man die Worst-Case-Komplexität der Elementarschritte berechnet:

   - Schritt a): $n + 1 = \mathcal{O}(n)$ Bewegungen, um von Anfang zum Ende des Wortes zu laufen, das Wortende zu markieren und zum letzten Buchstaben des Wortes zurückzukehren.
   - Schritt b): Markieren und Kopieren des letzten Buchstabens ☞ $2 \times 2 + 1 = 5$ Schritte. Markieren und Kopieren des vorletzten Buchstabens ☞ $2 \times 4 + 1 = 9$ Schritte. Am meisten Aufwand (Worst Case) entsteht, wenn der kopierte Buchstabe am Anfang des Wortes steht ☞ $2 \times 2n + 1 \approx 4n + 1 = \mathcal{O}(n)$.

     Es müssen n Buchstaben kopiert werden: Aufwand $\leq n \times (4n + 1) = 4n^2 + n = \mathcal{O}(n^2)$.
   - ☞ Gesamtaufwand: $\mathcal{O}(n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$.

5.51 Aufgabe: Laufzeit von Turing-Maschinen

Konstruieren Sie je eine Turing-Maschine mit Worst-Case-Laufzeit n, n^2 und 2^n, wobei n die Länge der Eingabe $|\vec{w}|$ beschreibt. Erläutern Sie Ihre Konstruktionen, und geben Sie ebenfalls die Bandkomplexität an. Eine *informelle, aber dennoch klare Beschreibung* der Maschinen reicht aus.

Die Maschinen werden wie folgt analysiert:

- Zeitkomplexität n: Von links nach rechts über ein Eingabewort laufen. Die Bandkomplexität ist ebenfalls n.
- n^2: Maschine aus Aufgabe 3 verwenden. Die Bandkomplexität beträgt $2n + 1 = \mathcal{O}(n)$.
- Zeitkomplexität 2^n: Maschine verwenden, die alle Zahlen in einem n-Bit-String aufzählt. Für n Bits existieren 2^n Belegungen ☞ Zeitkomplexität $\mathcal{O}(2^n)$, Bandkomplexität zur Darstellung der Zahl $\mathcal{O}(n)$.